MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1082 December 1989

# Optimization of Series Expressions: Part I: User's Manual for the Series Macro Package

by

Richard C. Waters

## Abstract

The benefits of programming in a functional style are well known. In particular, algorithms that are expressed as compositions of functions operating on series/vectors/streams of data elements are much easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, many programmers hesitate to use series expressions. In part, this is due to the fact that series expressions are typically implemented very inefficiently.

A Common Lisp macro package (called Series) has been implemented that can evaluate a wide class of series expressions very efficiently by transforming them into iterative loops. When using this class of series expressions, programmers can obtain the advantages of expressing computations as series expressions without incurring any run-time overhead.



This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens corporation, in part by the Microelectronics and Computer Technology Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-88-K-0487.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the NYNEX Corporation, of the Siemens corporation, of the Microelectronics and Computer Technology Corporation, or of the Department of Defense.

# Contents




**Acknowledgments.** The Series macro package has benefited from the suggestions of a number of people. In particular, A. Meyer, C. Perdue, C. Rich, D. Wile, Y. Feldman, D. Chapman, Y. Tan, and P. Anagnostopoulos made suggestions that led to significant improvements.

# 1. All You Need To Know to Get Started

This first section describes everything you need to know to start using the Series macro package. It then presents a detailed example. The remaining sections are a comprehensive reference manual. They describe the functions and macros supported by the Series macro package in full detail. A companion paper [17] gives an overview of the theory underlying the macro package and compares the macro package with related systems.

Series combine aspects of sequences (vectors and lists), streams, and loops. Like sequences, series represent one-dimensional totally-ordered collections of elements. In addition, the series functions have the same flavor as the sequence functions—namely, they operate on whole series, rather than extracting elements to be processed by other functions. For instance, the series expression below computes the sum of the positive elements in a list.

```
(collect-sum (choose-if #'plusp (scan '(1 -2 3 -4)))) ⇒ 4
```

Like streams, series can represent unbounded collections of elements and are supported by lazy evaluation: The $i$th element of a series is not computed until it is needed. For instance, the series expression below returns a list of the first five even natural numbers and their sum. The call on **scan-range** returns a series of all the even natural numbers. However, since no elements beyond the first five are ever used, no elements beyond the first five are ever computed.

```
(let ((x (subseries (scan-range :from 0 :by 2) 0 5)))
  (values (collect x) (collect-sum x))) ⇒ (0 2 4 6 8) and 20
```

Like sequences and unlike streams, the act of accessing the elements of a series does not alter the series. For instance, both users of **x** above receive the same elements.

In a loop, a one-dimensional totally-ordered collection of elements can be represented by the successive values of a variable. This is extremely efficient, because it avoids the need to store the elements as a group in any kind of data structure. In most situations, series expressions achieve this same high level of efficiency, because they are automatically transformed into loops before being evaluated or compiled. For instance, the first expression above is transformed into a loop like the following.

```
(let ((sum 0))
  (dolist (i '(1 -2  3 -4) sum)
    (if (plusp i) (setq sum (+ sum i))))) ⇒ 4
```

A wide variety of algorithms can be expressed clearly and succinctly using series expressions. In particular, most of the loops programmers typically write can be replaced by series expressions that are much easier to understand and modify, and just as efficient. From this perspective, the key feature of series is that they are supported by a rich set of functions. These functions more or less correspond to the union of the operations provided by the sequence functions, the `loop` clauses, and the vector operations of APL.

Unfortunately, some series expressions cannot be transformed into loops. This matters because, while transformable series expressions are much more efficient than equivalent expressions involving sequences or streams, non-transformable series expressions are much less efficient. Whenever a problem comes up that blocks the transformation of a series expression, a warning message is issued. Based on the information in the message, it is usually easy to provide an efficient fix for the problem.

Fortunately, most series expressions can be transformed into loops. In particular, pure expressions (ones that do not store series in variables) can always be transformed. As a result, the best approach for programmers to take is to simply write series expressions without worrying about transformability. When problems come up, they can be ignored (since they cannot lead to the computation of incorrect results) or dealt with on an individual basis.

**The series data type.** The Series macro package supports the series data type and a suite of functions operating on this data type. Series are self-evaluating objects. In analogy with `#(`*items*`)`, the `#` macro character syntax `#Z(`*items*`)` is provided for writing literal series. This same syntax is used when series are printed. If `*print-length*` is not `nil`, then long (or unbounded) series are abbreviated using "...", as in the second example below.

```
#Z(a (b c) d) ⇒ #Z(a (b c) d)
#Z(a b . #1=(c d . #1#)) ⇒ #Z(a b c d c d ...)
```

**Predefined series functions.** The heart of the Series macro package is a set of several dozen functions that operate on series. (See Section 8 for a quick summary.) These functions divide naturally into three classes. *Scanners* produce series without consuming any. *Transducers* compute series from series. *Collectors* consume series without producing any.

Predefined scanners include: `series` which creates an unbounded series indefinitely repeating a given value, `scan` which enumerates the elements in an object of type sequence, `scan-range` which enumerates the integers in a range, and `scan-plist` which creates a series of the indicators in a property list along with a second series containing the corresponding values. The first argument of `scan` specifies the type of sequence to be scanned. If omitted, the type defaults to `list`.

```
(series 'a) ⇒ #Z(a a a ...)
(scan '(a b c)) ⇒ #Z(a b c)
(scan 'vector '#(a b c)) ⇒ #Z(a b c)
(scan-range :from 1 :upto 3) ⇒ #Z(1 2 3)
(scan-plist '(a 1 b 2)) ⇒ #Z(a b) and #Z(1 2)
```

Predefined transducers include: `positions` which returns the positions of the non-null elements in a series and `choose` which selects the elements of its second argument that correspond to non-null elements of its first argument.

```
(positions #Z(a nil b c nil nil)) ⇒ #Z(0 2 3)
(choose #Z(nil T T nil) #Z(1 2 3 4)) ⇒ #Z(2 3)
```

Predefined collectors include: `collect` which combines the elements of a series into a sequence, `collect-sum` which adds up the elements of a series, `collect-length` which computes the length of a series, and `collect-first` which returns the first element of a series. The first argument of `collect` specifies the type of the sequence to be produced. If omitted, the type defaults to `list`.

```
(collect #Z(a b c)) ⇒ (a b c)
(collect 'simple-vector #Z(1 2 3)) ⇒ #(1 2 3)
(collect-sum #Z(1 2 3)) ⇒ 6
(collect-length #Z(a b c)) ⇒ 3
(collect-first #Z(a b c)) ⇒ a
```

**Higher-Order series functions.** The Series macro package provides a number of higher-order functions, which support general classes of series operations. For example, the function (`map-fn` *type function items*) supports the generic transduction operation of mapping a function over a series. The *type* argument specifies the type of the elements in the series being created. (When efficient compilation is desired, it is important to use an informative *type* such as `integer` rather than an uninformative *type* such as `T`.) Each element of the output is computed by applying *function* to the corresponding element of *items*.

```
(map-fn 'integer #'sqrt #Z(4 9 16)) ⇒ #Z(2 3 4)
```

Scanning is supported by (`scan-fn` *type init step test*). The *type* argument specifies the type of the elements in the series being created. The function *init* is called to obtain the first element of the output. Subsequent elements are obtained by applying the function *step* to the previous element. The series consists of the elements up to, but not including, the first element for which the function *test* returns non-null.

```
(scan-fn 'integer #'(lambda () 3) #'1- #'minusp) ⇒ #Z(3 2 1 0)
```

Collecting (accumulating) is supported by (`collect-fn` *type init function items*). The elements of the series *items* are combined together using *function*. The quantity returned by *init* is used as an initial seed value for the accumulation. The *type* argument specifies the type of the summary value returned.

```
(collect-fn 'integer #'(lambda () 3) #'+ #Z(1 2 3)) ⇒ 9
```

**Convenient support for mapping.** Mapping is by far the most commonly used series operation. In cognizance of this fact, the Series macro package provides three mechanisms that make it easy to express particular kinds of mapping. The `#` macro character syntax `#M`*f* converts a function *f* into a transducer that maps *f*.

```
(#Msqrt #Z(4 16)) ≡ (map-fn T #'sqrt #Z(4 16)) ⇒ #Z(2 4)
```

The form `mapping` can be used to specify the mapping of a complex expression over one or more series without having to write a literal `lambda` expression. For example,

```
(mapping ((x (scan '(2 -2 3))))
  (expt (abs x) 3)) ⇒ #Z(8 8 27)
```

is the same as

```
(map-fn T #'(lambda (x) (expt (abs x) 3))
        (scan '(2 -2 3))) ⇒ #Z(8 8 27)
```

The form `iterate` is the same as `mapping` except that the value `nil` is always returned.

```
(iterate ((x (scan '(2 -2 3))))
  (if (plusp x) (prin1 x))) ⇒ nil <after printing "23">
```

To a first approximation, `iterate` and `mapping` differ in the same way as `mapc` and `mapcar`. In particular, like `mapc`, `iterate` is intended to be used in situations where the body is being evaluated for side effect rather than for its result. However, due to the lazy evaluation semantics of series, the difference between `iterate` and `mapping` is more than just a question of efficiency. If `mapping` is used in a situation where the output is not used, no computation is performed, because series elements are not computed until they are used.

**User-defined series functions.** As shown by the definitions of simplified versions of `collect-sum` and `mapping` below, the standard Lisp forms `defun` and `defmacro` can be used to define new series functions. However, when a series function is defined with `defun`, the Series macro package is not capable of optimizing series expressions containing this new function unless the declaration `optimizable-series-function` is specified in the `defun`. This declaration is not required when using `defmacro`.

```
(defun simple-collect-sum (numbers)
    (declare (optimizable-series-function))
  (collect-fn 'number #'(lambda () 0) #'+ numbers))

(defmacro simple-mapping (var-value-pair-list &body body)
  (let* ((pairs (scan var-value-pair-list))
         (arg-list (collect (#Mcar pairs)))
         (value-list (collect (#Mcadr pairs))))
    `(map-fn T #'(lambda ,arg-list ,@ body) ,@ value-list)))
```

**Benefits.** The advantage of series expressions is that they retain most of the virtues of loop-free, functional programming, while eliminating most of the costs. However, given the fact that optimization is not always possible, the question naturally arises as to whether optimization is possible in a wide enough range of situations to be of real pragmatic benefit.

An informal study [10] was undertaken of the kinds of loops programmers actually write. This study suggests that approximately 80% of the loops programmers write are constructed by combining a few common kinds of looping algorithms in a few simple ways. The Series macro package is designed so that all of these loops can be trivially expressed as optimizable series expressions. Many more loops can be expressed as optimizable series expressions with only minor modification.

Moreover, the benefits of using series expressions go beyond replacing individual loops. A major shift toward using series expressions would be a significant change in the way programming is done. At the current time, most programs contain one or more loops and most of the interesting computation in these programs occurs in these loops. This is quite unfortunate, since loops are generally acknowledged to be one of the hardest things to understand in any program. If series expressions were used whenever possible, most programs would not contain any loops. This would be a major step forward in conciseness, readability, verifiability, and maintainability.

## Example

The following example shows what it is like to use series expressions in a realistic programming context. The example consists of two parts: a pair of functions that convert between sets represented as lists and sets represented as bits packed into an integer and a graph algorithm that uses the integer representation of sets.

**Bit sets.** Small sets can be represented very efficiently as binary integers where each 1 bit in the integer represents an element in the set. Here, sets represented in as binary integers are referred to as *bit sets*.

Common Lisp provides a number of bitwise operations on integers, which can be used to manipulate bit sets. In particular, `logior` computes the union of two bit sets while `logand` computes their intersection.

The functions in Figure 1.1 convert between sets represented as lists and bit sets. To perform this conversion, a mapping has to be established between bit positions and potential set elements. This mapping is specified by a *universe*. A universe is a list of elements. If a bit set integer $b$ is associated with a universe $u$, then the $i$th element in $u$ is in the set represented by $b$ if and only if the $i$th bit in $b$ is 1. For example, given the universe `(a b c d e)`, the integer `#b01011` represents the set {`a`,`b`,`d`}. (By Common Lisp convention, the 0th bit in an integer is the least significant bit.)

Given a bit set and its associated universe, the function `bset->list` converts the bit set into a set represented as a list of its elements. It does this by scanning the elements in the universe along with their positions and constructing a list of the elements which correspond to 1s in the integer representing the bit set. (When no `:upto` argument is

```
(defun bset->list (bset universe)
  (collect (choose (#Mlogbitp (scan-range :from 0) (series bset))
                   (scan universe))))

(defun list->bset (items universe)
  (collect-fn 'integer #'(lambda () 0) #'logior
    (mapping ((item (scan items)))
      (ash 1 (bit-position item universe)))))

(defun bit-position (item universe)
  (or (collect-first (positions (#Meq (series item) (scan universe))))
      (1- (length (nconc universe (list item))))))
```

Figure 1.1: Converting between lists and bit sets.

```
(defun collect-logior (bsets)
    (declare (optimizable-series-function))
  (collect-fn 'integer #'(lambda () 0) #'logior bsets))

(defun collect-logand (bsets)
    (declare (optimizable-series-function))
  (collect-fn 'integer #'(lambda () -1) #'logand bsets))
```

Figure 1.2: Operations on series of bit sets.

supplied, `scan-range` counts up forever.)

The function `list->bset` converts a set represented as a list of its elements into a bit set. Its second argument is the universe that is to be associated with the bit set created. For each element of the list, the function `bit-position` is called to determine which bit position should be set to 1. The function `ash` is used to create an integer with the correct bit set to 1. The function `collect-fn` is used to combine the integers corresponding to the individual elements together into a bit set corresponding to the list.

The function `bit-position` takes an item and a universe and returns the bit position corresponding to the item. The function operates in one of two ways depending on whether or not the item is in the universe. The first line of the function contains a series expression that determines the position of the item in the universe. If the item is not in the universe, the expression returns `nil`. (The function `collect-first` returns `nil` if it is passed a series of length zero.)

If the item is not in the universe, the second line of the function adds the item onto the end of the universe and returns its position. The extension of the universe is done by side effect so that it will be permanently recorded in the universe.

Figure 1.2 shows the definition of two collectors that operate on series of bit sets. The first function computes the union of a series of bit sets, while the second computes their intersection.

**Live variable analysis.** As an illustration of the way bit sets might be used, consider the following. Suppose that in a compiler, program code is being represented as blocks of straight-line code connected by possibly cyclic control flow. The top part of Figure 1.3 shows the data structure that represents a block of code. Each block has several pieces of information associated with it. Two of these pieces of information are the blocks that can branch to the block in question and the blocks it can branch to. A program is represented as a list of blocks that point to each other through these fields.

In addition to control flow information, each structure contains information about the way variables are accessed. In particular, it records the variables that are written by the block and the variables that are used by the block (i.e., either read without being written or read before they are written). An additional field (computed by the function `determine-live` discussed below) records the variables that are *live* at the end of the block. (A variable is live if it has to be saved, because it can potentially be used by a following block.) Finally, there is a temporary data field, which is used by functions (such as `determine-live`) that perform computations involved with the blocks.

The remainder of Figure 1.3 shows the function `determine-live` which, given a pro-

```
(defstruct (block (:conc-name nil))
  predecessors ;Blocks that can branch to this one.
  successors   ;Blocks this one can branch to.
  written      ;Variables written in the block.
  used         ;Variables read before written in the block.
  live         ;Variables that must be available at exit.
  temp)        ;Temporary storage location.

(defun determine-live (program-graph)
  (let ((universe (list nil)))
    (convert-to-bsets program-graph universe)
    (perform-relaxation program-graph)
    (convert-from-bsets program-graph universe))
  program-graph)

(defstruct (temp-bsets (:conc-name bset-))
  used written live)

(defun convert-to-bsets (program-graph universe)
  (iterate ((block (scan program-graph)))
    (setf (temp block)
          (make-temp-bsets
            :used (list->bset (used block) universe)
            :written (list->bset (written block) universe)
            :live 0))))

(defun perform-relaxation (program-graph)
  (let ((to-do program-graph))
    (loop
      (when (null to-do) (return (values)))
      (let* ((block (pop to-do))
             (estimate (live-estimate block)))
        (when (not (= estimate (bset-live (temp block))))
          (setf (bset-live (temp block)) estimate)
          (iterate ((prev (scan (predecessors block))))
            (pushnew prev to-do)))))))

(defun live-estimate (block)
  (collect-logior
    (mapping ((next (scan (successors block))))
      (logior (bset-used (temp next))
              (logandc2 (bset-live (temp next))
                        (bset-written (temp next)))))))

(defun convert-from-bsets (program-graph universe)
  (iterate ((block (scan program-graph)))
    (setf (live block)
          (bset->list (bset-live (temp block)) universe))
    (setf (temp block) nil)))
```

Figure 1.3: Live variable analysis.

gram represented as a list of blocks, determines the variables that are live in each block. To perform this computation efficiently, the function uses bit sets. The function operates in three steps. The first step (`convert-to-bsets`) looks at each block and sets up an auxiliary data structure containing bit set representations for the written variables, the used variables, and an initial guess that there are no live variables. This auxiliary structure is defined by the third form in Figure 1.3 and is stored in the `temp` field of the block. The integer 0 represents an empty bit set.

The second step (`perform-relaxation`) determines which variables are live. This is done by relaxation. The initial guess that there are no live variables in any block is successively improved until the correct answer is obtained.

The third step (`convert-from-bsets`) operates in the reverse of the first step. Each block is inspected and the bit set representation of the live variables is converted into a list, which is stored in the live field of the block.

On each cycle of the loop in `perform-relaxation`, a block is examined to determine whether its live set has to be changed. To do this (see the function `live-estimate`), the successors of the block are inspected. Each successor needs to have available to it the variables it uses, plus the variables that are supposed to be live after it, minus the variables it writes. (The function `logandc2` takes the difference of two bit sets.) A new estimate of the total set of variables needed by the successors as a group is computed by using `collect-logior`.

If this new estimate is different from the current estimate of what variables are live, then the estimate is changed. In addition, if the estimate is changed, `perform-relaxation` has to make sure that all of the predecessors of the current block will be examined to see if the new estimate for the current block requires that their live estimates be changed. This is done by adding each predecessor onto the list `to-do` unless it is already there. As soon as the estimates of liveness stop changing, the computation stops.

**Summary.** The function `determine-live` is a particularly good example of the way series expressions are intended to be used in two ways. First, series expressions are used in a number of places to express computations which would otherwise be expressed less clearly as loops or less efficiently as sequence function expressions. Second, the main relaxation algorithm is expressed as a loop. This is done, because neither optimizable series expressions (nor Common Lisp sequence function expressions) lend themselves to expressing the relaxation algorithm. This highlights the fact that series expressions are not intended to render iterative programs entirely obsolete, but rather to provide a greatly improved method for expressing the vast majority of loops.

## Setting Up the Series Macro Package

The Series macro package was originally developed under version 7 of the Symbolics Lisp Machine software [19]. However, it is written in standard Common Lisp and has been tested in several different versions of Common Lisp. To use the Series macro package, the file containing it has to be loaded. At the MIT AI Laboratory, XP resides in the file `"b:>lmlib>s.lisp"`. Compiled versions exist for Symbolics and Lucid (Sun) Common Lisp.

The source for the Series macro package can be obtained over the INTERNET by using

FTP. Connection should be made to the TRIX.AI.MIT.EDU machine. Login as 'anonymous' and copy the files shown below. It is advisable to run the tests in **stest.lisp** after compiling the Series macro package for the first time on a new system. A comment at the beginning of the file describes how to run the tests.

files on TRIX.AI.MIT.EDU, (INTERNET number 128.52.32.6)

| | |
|---|---|
| **/com/ftp/pub/series/s.lisp** | source code |
| **/com/ftp/pub/series/stest.lisp** | tests |
| **/com/ftp/pub/series/sdoc.txt** | brief documentation |

As the series macro package is being made available free of charge, it is being distributed *as is*, with no warranty of any kind either expressed or implied including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, and further including no warranty as to conformity with this manual or any other literature that may be issued from time to time. In addition, if you wish to use the Series macro package for anything other than your own experimental use, you will have to get a license from MIT. Information about obtaining a non-exclusive, royalty-free license can be obtained by sending a message to "**dick@ai.mit.edu**".

The functions and forms discussed in this manual are defined in the package "**series**". To make these names easily accessible, you must *use* the package "**series**". The most convenient way to do this is to call the function **series::install**, which also sets up some additional features of the series macro package. The examples in this manual assume that the form **(series::install)** has been evaluated.

- **series::install &key (:pkg *package*) (:macro T) (:shadow T) (:remove nil)** ⇒ **T**

  Calling this function sets up Series for use in the package **:pkg**. The argument **:pkg** can either be a package, a package name, or a symbol whose name is the name of a package. It defaults to the current package.

  The package "**series**" is used in **:pkg**. If **:macro** is not **nil**, the **#** macro character syntax **#Z** and **#M** is set up. If **:shadow** is not **nil**, the symbols **series::let**, **series::let***, **series::multiple-value-bind**, **series::funcall**, and **series::defun** are shadowing imported into **:pkg**. These forms are identical to their standard counterparts, except that they support various features of the Series macro package. When shadowing is not done, you have to explicitly use **series::let**, **series::let***, and **series::multiple-value-bind** when binding series in an expression you want optimized; **series::funcall** when funcalling a series function you want optimized; and **series::defun** when defining a series function with the declaration **optimizable-series-function**.

  If **:remove** is not **nil**, the effects of having previously installed the Series macro package are undone. In particular, the package is unused and any shadowing is undone. However, any changes to the readtable are left in place.

# 2. Basic Reference Manual

This section (and the following two sections) are organized around descriptions of the various functions and macros supported by the Series macro package. Each description begins with a header showing the arguments and results of the function or macro. For easy reference, the headers are duplicated in Section 8. In Section 8, the headers are in alphabetical order and show the page where the full description can be found.

In a reference manual like this one, it is advantageous to describe each construct separately and completely. However, this inevitably leads to presentation problems, because everything is related to everything else. Therefore, one cannot avoid referring to things that have not yet been discussed. The reader is encouraged to skip around in the document and to realize that more than one reading will probably be necessary to gain a complete understanding of the Series macro package.

## Restrictions and Definitions of Terms

Series expressions are transformed into loops by pipelining them—the computation is converted from a form where entire series are computed one after the other to a form where the series are incrementally computed in parallel. In the resulting loop, each individual element is computed just once, used, and then discarded before the next element is computed. For this pipelining to be possible, four restrictions have to be satisfied. Before looking at these restrictions, it is useful to consider a related issue.

**All series functions are preorder functions.** The composition of two series functions cannot be pipelined unless the destination function consumes series elements in the same order that the source function produces them. Taken together, the series functions guarantee that this will always be true, because they all follow the same fixed processing order. In particular, they are all *preorder* functions—they process the elements of their series inputs and outputs in ascending order starting with the first element. Further, while it is easy for users to define new series functions, it is impossible to define ones that are not preorder.

It turns out that most series operations can easily be implemented in a preorder fashion, (the only notable exceptions being reversal and sorting). As a result, little is lost by outlawing non-preorder functions. If some non-preorder operation has to be applied to a series, the series can be converted into a list or vector and the operation applied to this new data structure. (This is inefficient, but no less efficient than what would be required if non-preorder series functions were supported.)

**Series expressions.** Before discussing the restrictions on series expressions, it is useful to define precisely what is meant by the term *series expression*.

Loosely speaking, a series function is a function that consumes or returns a series. However, the Series macro package is not capable of looking at an arbitrary function and determining whether or not it consumes or returns a series. To deal with this, series functions are precisely defined as being a function that consumes or returns a series and is either: (1) described in this manual, (2) defined using the declaration `optimizable-series-function`, (3) a literal `lambda` expression appearing as the first ar-

gument of a `funcall`, or (4) a macro that expands into an expression involving (1), (2), or (3). Everything else is treated as not being a series function no matter what kind of data objects it consumes or returns.

A series expression is an expression composed of series functions. However, beyond this, the definition of the term 'series expression' is semantic rather than syntactic in nature. Given a program, imagine it converted from Lisp code into a data flow graph. In a data flow graph, functions are represented as boxes, and both control flow and data flow are represented as arrows between the boxes. Data flow constructs such as `let` and `setq` are converted into patterns of data flow arcs. Control constructs such as `if` and `loop` are converted into patterns of control flow arcs. For example, the expression in the program below is converted into a graph with a chain of seven nodes corresponding to the seven function calls.

```
(defun expression-examp (data)
  (abs (collect-sum (scan (cdr (collect-last (choose (scan data)))))))))
```

A series expression is a subgraph of the data flow graph for a program that contains a group of interacting series functions. More specifically, given a call $f$ on a series function, the series expression $E$ containing it is defined as follows. $E$ contains $f$. Every function using a series created by a function in $E$ is in $E$. Every function computing a series used by a function in $E$ is in $E$. Finally, suppose that two functions $g$ and $h$ are in $E$ and that there is a data flow path consisting of series and/or non-series data flow arcs from $g$ to $h$. Every function touched by this path (be it a series function or not) is in $E$.

In the example program above, there are two series expressions: one corresponding to `(collect-sum (scan ...))` and the other to `(collect-last (choose (scan ...)))`. Optimization is applied to each series expression. The non-series parts of the Lisp code (e.g., the calls on `abs` and `cdr`) are left as-is and are evaluated/compiled in the normal way. While series functions and non-series functions can freely coexist in a piece of code, they are rigidly partitioned from each other when optimization is applied.

**Static analyzability.** For optimization to be possible, Series expressions have to be statically analyzable. As with most other optimization processes, a series expression cannot be transformed into a loop at compile time, unless it can be determined at compile time exactly what computation is being performed. This places a number of relatively minor limits on what can be written. To start with, the definition of a series function must appear before its first use. In addition, when using a series function that takes keyword arguments, the keywords themselves have to be constants rather than being the values of expressions.

Whenever there is a failure of static analyzability, a warning message is issued and the containing series expression is left unoptimized. The various limits imposed by the static analyzability restriction are described in Section 7 in conjunction with the associated warning messages.

**Locality of series.** For optimization to be possible, every series created within a series expression must be used solely inside the expression. (If a series is transmitted outside of the expression that creates it, it has to be physically represented as a whole. This is incompatible with the transformations required to pipeline the creating expression.)

To avoid this problem, series must not be returned as the results of series expressions, assigned to free variables, assigned to special variables, or stored in data structures. Further, optimization is blocked if a series is passed as an argument to an ordinary Lisp function. Series can only be passed to the series functions defined in this manual and to new series functions defined using the declaration `optimizable-series-function`.

**Straight-line computation.** For optimization to be possible, series expressions must correspond to straight-line computations. That is to say, the data flow graph corresponding to the series expression cannot contain any conditional branches or loops. (Complex control flow is incompatible with pipelining.) Optimization is possible in the presence of standard straight-line forms such as `progn`, `funcall`, `setq`, `lambda`, `let`, `let*`, and `multiple-value-bind` as long as none of the variables bound are special. There is also no problem with macros as long as they expand into series functions and straight-line forms. However, optimization is blocked by forms that specify complex control flow (i.e., conditionals `if`, `cond`, etc., looping constructs `loop`, `do`, etc., or branching constructs `tagbody`, `go`, `catch`, etc.).

In the first example below, optimization is blocked, because the `if` form is inside of the series expression. However, in the second example, optimization is possible, because although the `if` feeds data to the series expression, it is not inside the corresponding subgraph. The two expressions produce the same value, however, the second one is much more efficient, because it can be transformed into a loop.

```
(collect (if flag (scan x) (scan y)))    ; Warning 20 signaled.
(collect (scan (if flag x y)))
```

An obvious direction of future research with regard to the Series macro package is applying optimization to series expressions containing control flow constructs. There is little doubt that simple conditionals such as `if` and `cond` could be handled. However, it is not clear whether more complex constructs could be handled in a reasonable way.

**Constraint cycles.** Even if a series expression satisfies all of the restrictions above, it may still not be possible to transform the expression into a loop. The sole remaining problem is that if a series is used in two places, the two uses may place incompatible constraints on the times at which series elements should be computed.

The series expression below shows a situation where this problem arises. The expression creates a series `x` of the elements in a list. It then creates a normalized series by dividing each element of `x` by the sum of the elements in `x`. Finally, the expression returns the maximum of the normalized elements.

```
(let ((x (scan '(1 2 5 2))))              ; Warning 21 signaled.
  (collect-max (#M/ x (series (collect-sum x))))) ⇒ 1/2
```

The two uses of `x` in the expression place contradictory constraints on the way pipelined evaluation must proceed. The function `collect-sum` requires that all of the elements of `x` be produced before the sum can be returned and `series` requires that its input be available before it can start producing its output. However, `#M/` requires that the first element of `x` be available at the same time as the first element of the output of `series`. For pipelining to work, this implies that the first element of the output of `series`

(and therefore the output of **collect-sum**) must be available before the second element of **x** is computed. Unfortunately, this is impossible.

The essence of the inconsistency above is the cycle of constraints used in the argument. This in turn stems from a cycle in the data flow graph underlying the expression (see Figure 2.1). In Figure 2.1, function calls are represented by boxes and data flow is represented by arrows. Simple arrows indicate the flow of series values and cross hatched arrows indicate the flow of non-series values.

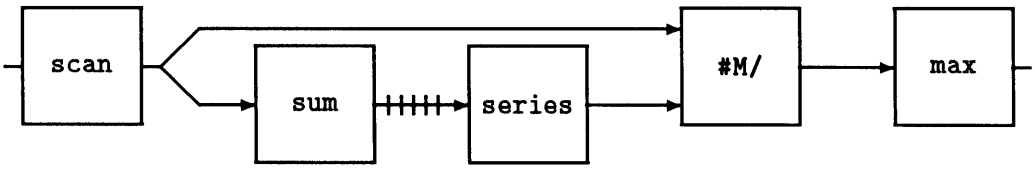


Figure 2.1: A constraint cycle.

Given a data flow graph corresponding to a series expression, a *constraint cycle* is a closed loop of data flow arcs that can be traversed in such a way that each arc is traversed exactly once and no non-series arc is traversed backwards. (Series data flow arcs can be traversed in either direction.) A constraint cycle is said to *pass through* an input or output port when exactly one of the arcs in the cycle touches the port. In Figure 2.1 the data flow arcs touching **scan**, **sum**, **series**, and **#M/** form a constraint cycle. Note that if the output of **scan** were not a series, this loop would not be a constraint cycle, because there would be no valid way to traverse it. Also note that while the constraint cycle passes through all the other ports it touches, it does not pass through the output of **scan**.

Whenever a constraint cycle passes through a non-series output, an argument analogous to the one above can be constructed and therefore pipelining is impossible. When this situation arises, a warning message is issued identifying the problematical port and the cycle passing through it. For instance, the warning triggered by the example above states that a constraint cycle passes through the non-series output of **collect-sum**.

Given this kind of detailed information, it is easy to alleviate the problem. To start with, every cycle must contain at least one function that has two series data flows leaving it. At worst, the cycle can be broken by duplicating this function (and any functions computing series used by it). For instance, the example above can be rewritten as shown below.

```
(let ((x (scan '(1 2 5 2)))
      (sum (collect-sum (scan '(1 2 5 2)))))
  (collect-max (#M/ x (series sum)))) ⇒ 1/2
```

It would be easy enough to automatically apply code copying to break problematical constraint cycles. However, this is not done for two reasons. First, there is considerable virtue in maintaining the property that each function in a series expression turns into one piece of computation in the loop produced. Users can be confident that series expressions that look simple and efficient actually are simple and efficient. Second, with a little

creativity, constraint problems can often be resolved in ways that are much more efficient than copying code. In the example above, the conflict can be eliminated efficiently by interchanging the operation of computing the maximum with the operation of normalizing an element.

```
(let ((x (scan '(1 2 5 2))))
  (/ (collect-max x) (collect-sum x))) ⇒ 1/2
```

The restriction that optimizable series expressions cannot contain constraint cycles that pass through non-series outputs places limits on the qualitative character of optimizable series expressions. In particular, optimizable series expressions all have the general form of creating some number of series using scanners, computing various intermediate series using transducers, and then computing one or more summary results using collectors. The output of a collector cannot be used in the intermediate computation unless it is the output of a separate subexpression.

It is worthy of note that the last expression above fixes the constraint conflict by moving the non-series output out of the cycle, rather than by breaking the cycle. This illustrates the fact that constraint cycles that do not pass through non-series outputs do not necessarily cause problems. Such constraint cycles cause problems only if they pass through *off-line* ports.

**On-line and off-line.** A series input port or series output port of a series function is on-line if and only if it is processed in lock step with all the other on-line ports as follows: The initial element of each on-line input is read, then the initial element of each on-line output is written, then the second element of each on-line input is read, then the second element of each on-line output is written, and so on. Ports that are not on-line are off-line. If all the series ports of a function are on-line, the function is said to be on-line; otherwise, it is off-line. (The above extends the standard definition of the term 'on-line' (see [1]) so that it applies to individual ports as well as whole functions.)

The prototypical example of an on-line series function is **map-fn**. Each time it reads an input element, it applies the mapped function to it and writes an output element. In contrast, the function **positions** is not on-line. Since null input elements do not lead to output elements, it is not possible for **positions** to write an output element every time it reads an input element.

For every series function, the documentation below specifies which ports are on-line and which are off-line. In this regard, it is interesting to note that every function that has only one series port (i.e., scanners with only one output and collectors with only one input) are trivially on-line. The only series functions that have off-line ports are transducers.

If all of the ports a cycle passes through are on-line, the lock step processing of these ports guarantees that there cannot be any conflicts between the constraints associated with the cycle. However, passing through an off-line port leads to the same kinds of problems as passing through a non-series output.

In summary, the fourth and final restriction is that: for optimization to be possible, a series expression cannot contain a constraint cycle that passes through a non-series output or an off-line port. Whenever this restriction is violated, a warning message is issued.

Violations can be fixed either by breaking the cycle or restructuring the computation so that the offending port is removed from the cycle.

## Series

The Series macro package adds support for a new data type called *series* to Common Lisp. Series are similar to lists or vectors in that they are one-dimensional totally-ordered collections of elements and similar operations can be applied to them. However, series are also closely related to streams, both because they can contain an unbounded number of elements and because they are supported using lazy evaluation semantics. In particular, the $j$th element of a series is not computed until it is actually used (if ever). As a concrete example of the lazy evaluation semantics of series, consider the following.

```
(setq x 0) ⇒ 0
(collect-first (map-fn T #'(lambda (a) (incf x) (* 3 a))
                           (scan-range :from 1 :upto 10))) ⇒ 3
x ⇒ 1
```

The call on **scan-range** creates a series of ten elements. The **map-fn** creates another series of ten elements computed from this series. However, **collect-first** only uses the first element of its input. Since the result of **map-fn** is not used anywhere else, only the first element of this series is computed. As a result, the function being mapped is only applied once and **x** is only incremented once. In the absence of side effects, there is typically no need to think about the lazy evaluation nature of the support for series. However, when side effects are involved, this has to be kept in mind.

The above notwithstanding, it is typically better to think of series as being like lists rather than streams in most situations. The reason for this is that there is a critical difference between series and streams. Consider the code below. If **x** contains a stream, the function **g** will only see the elements of **x** that are not used by **f**. That is to say, if **f** reads the first ten elements of **x**, these elements are gone and the first element seen by **g** will be the eleventh.

```
(let ((x ...))
  (f x)
  (g x)
  ...)
```

In contrast, suppose that **x** contains a list. The mere act of looking at the elements of a list does not alter the list. As a result, both **f** and **g** see all the elements of **x**. This situation is exactly the same when **x** is a series. If a series is used in several places, all of the elements of the series are available in each place.

For the convenience of the reader, this documentation uses the following two orthographic conventions with regard to series. First, the notation $S_j$ is used to designate the $j$th element of the series S. As in a list or vector, the first element of a series has the subscript 0. Second, plural nouns (e.g., *items*, *numbers*) are used to represent series inputs and outputs of functions, while singular nouns (e.g., *item*, *number*) are used to indicate non-series inputs and outputs.

- **series &optional** (*type* **T**)

  This type specifier can be used to declare that something is a series value. The *type* argument specifies the type of the items in the series.

```
(let ((x (scan '(1 2 3))))
    (declare (type (series integer) x))
  (collect-sum x)) ⇒ 6
```

- **series** *item-1 ... item-n* ⇒ *items*

  An unbounded series is created that endlessly repeats the values of the *item-i*. As shown in the last example below, the function **series** is often used to create what is in effect a constant value to be passed into a series input of a series function. (Like lists, the same name is used for the name of the type specifier and the name of the primary constructor function.)

```
(series 'b 'c) ⇒ #Z(b c b c b c ...)
(series 1) ⇒ #Z(1 1 1 1 1 1 ...)
(#Mlist (series 'a) (scan '(1 2 3))) ⇒ #Z((a 1) (a 2) (a 3))
```

- **make-series** *item-1 ... item-n* ⇒ *items*

  A bounded series of length *n* is created containing the *items-i*.

```
(make-series 'b 'c) ⇒ #Z(b c)
(make-series ...) ≡ (scan (list ...))
```

- **#Z** (*item-1 ... item-n*) ⇒ *items*

  The **#** macro character syntax **#Z** is used to specify a literal series. It must be followed by a list of items. A series is created that contains these items. As in **#(...)** (and in contrast to **make-series**), the *item-i* are implicitly quoted. Unlike **#(...)**, which turns directly into a data object when read in, instances of **#Z(...)** turn into function calls and therefore should not be quoted. To activate the syntax **#Z** for input, you must call **(series::install :macro T)**. However, whether or not this is done, the **#Z** syntax is used for printing series.

```
#Z(a b c) ⇒ #Z(a b c)
#Z(...) ≡ (scan '(...))
```

**Series of Series.** It is possible to create a series whose elements are themselves series. For instance, given a vector of lists of integers, the expression below creates a series of series of integers. It then creates a list of the sums of these integers.

```
(let* ((series-of-lists (scan 'vector '#((1 2 3) (3 4 5))))
       (series-of-series (#Mscan series-of-lists)))
  (collect (mapping ((integers series-of-series))
             (collect-sum integers))))  ; Warning 28 signaled.
  ⇒ (6 12)
```

It should be possible to optimize the expression above creating a pair of nested loops. However, the Series macro package is not capable of optimizing series of series. Rather, the expression above triggers a warning message (because there is data flow from the assumed non-series value `integers` to the series input of `collect-sum`). Only the outermost level of the series of series is optimized.

An obvious direction of future research with regard to the Series macro package is applying optimization to series of series. However, it is not obvious whether the pragmatic benefits would be worth the effort involved. For instance, full optimization can be obtained in the example above, by merely writing it in the form shown below. The key difference is that the inner loop is completely contained in the body of the `mapping`.

```
(let ((series-of-lists (scan 'vector '#((1 2 3) (3 4 5)))))
  (collect (mapping ((list series-of-lists))
             (collect-sum (scan list))))) ⇒ (6 12)
```

## Scanners

Scanners create series outputs based on non-series inputs. There are two basic kinds of scanners: ones that create a series based on some formula (e.g., scanning a range of integers) and ones that create a series containing the elements of an aggregate data structure (e.g., scanning the elements of a list).

- `scan` {*type*} *sequence* ⇒ *elements*

Creates a series containing the successive elements of *sequence*. If *sequence* is a list, then it must be a proper list ending in `nil`. The *type* argument specifies the type of sequence to be scanned. This type must be a (not necessarily proper) subtype of `sequence`. If omitted, the type defaults to `list`. Scanning is significantly more efficient if it can be determined at compile time whether the type is a subtype of `list` or `vector`.

```
(scan '()) ⇒ #Z()
(scan '(a b c)) ⇒ #Z(a b c)
(scan 'string "BAR") ⇒ #Z(#\B #\A #\R)
(scan '(simple-vector integer 3) '#(1 2 3)) ⇒ #Z(1 2 3)
```

- `scan-multiple` *type sequence-1 ... sequence-n* ⇒ *elements-1 ... elements-n*

Several sequences can be scanned at once by using several calls on `scan`. Each call on `scan` will test to see when its sequence input runs out of elements and execution will stop as soon as any of the sequences are exhausted. Although very robust, this approach to scanning can be a significant source of inefficiency. In situations where it is known in advance which sequence is the shortest, `scan-multiple` can be used to obtain the same results more rapidly.

The function `scan-multiple` is similar to `scan` except that two or more sequences can be scanned at once. If there are $n$ sequence inputs, `scan-multiple` returns $n$ series containing the elements of these sequences. It must be the case that none of the sequence inputs is shorter than the first sequence. All of the output series are the same length as the first input sequence. Extra elements in the other input sequences are ignored.

Using **scan-multiple** is more efficient than using multiple instances of **scan**, because **scan-multiple** only has to check for the first input running out of elements.

If *type* is of the form **(values** $s_1$ ... $s_n$**)**, then there must be $n$ sequence inputs and *sequence-i* must have type $s_i$. Otherwise there can be any number of sequence inputs each of which must have type *type*.

```
(multiple-value-bind (data weights)
    (scan-multiple 'list '(1 6 3 2 8) '(2 3 3 3 2))
  (collect (map-fn T #'* data weights))) ⇒ (2 18 9 6 16)
```

- **scan-range &key (:start 0) (:by 1) (:type 'number)**
  **:upto :below :downto :above :length** ⇒ *numbers*

Creates a series of numbers starting with **:start** (default integer **0**) and counting up by **:by** (default integer **1**). The **:type** argument (which defaults to **number**) specifies the type of numbers produced and must be a subtype of **number**. The arguments **:start** and **:by** must be of type *type*.

The last five arguments specify the kind of end test to be used. If **:upto** is specified, counting continues only so long as the numbers generated are less than or equal to **:upto**. If **:below** is specified, counting continues only so long as the numbers generated are less than **:below**. If **:downto** is specified, counting continues only so long as the numbers generated are greater than or equal to **:downto**. If **:above** is specified, counting continues only so long as the numbers generated are greater than **:above**. If **:length** is specified, the series created has length **:length**. (It must be the case that **:length** is a non-negative integer.) If none of the termination arguments are specified, the output has unbounded length. If more than one termination argument is specified, it is an error.

```
(scan-range) ⇒ #Z(0 1 2 3 4 ...)
(scan-range :upto 4) ⇒ #Z(0 1 2 3 4)
(scan-range :from 1 :below 4) ⇒ #Z(1 2 3)
(scan-range :by -3 :downto -4) ⇒ #Z(0 -3)
(scan-range :from 1 :above -4 :by -1) ⇒ #Z(1 0 -1 -2 -3)
(scan-range :from 1.5 :by .1 :length 3 :type 'float) ⇒ #Z(1.5 1.6 1.7)
```

- **scan-sublists** *list* ⇒ *sublists*

Creates a series containing the successive sublists of *list*, which must be a proper list ending in **nil**.

```
(scan-sublists '(a b c)) ⇒ #Z((a b c) (b c) (c))
```

- **scan-alist** *alist* **&optional** (*test* **#'eql**) ⇒ *keys values*

Scans the entries in an association list, returning two series containing keys and their associated values. The first element of *keys* is the key in the first entry in *alist*, the first element of *values* is the value in the first entry, and so on. The *alist* must be a proper list ending in **nil** and each entry in *alist* must be a cons cell or **nil**. Like **assoc**, **scan-alist** skips entries that are **nil** and entries that have the same key as an earlier entry. The *test* argument (default **eql**) is used to determine when two keys are the same.

```
(scan-alist '((a . 1) () (a . 3) (b . 2))) ⇒ #Z(a b) and #Z(1 2)
(scan-alist nil) ⇒ #Z() and #Z()
```

- **scan-plist** *plist* ⇒ *indicators values*

Scans the entries in a property list, returning two series containing indicators and their associated values. The first element of *indicators* is the first indicator in *plist*, the first element of *values* is the associated value, and so on. The *plist* argument must be a proper list of even length ending in **nil**. In analogy with the way **get** works, if an indicator appears more than once in *plist*, it (and its value) will only be enumerated the first time it appears.

```
(scan-plist '(a 1 a 3 b 2)) ⇒ #Z(a b) and #Z(1 2)
(scan-plist nil) ⇒ #Z() and #Z()
```

- **scan-hash** *table* ⇒ *keys values*

Scans the entries in a hash table, returning two series containing keys and their associated values. The first element of *keys* is the key of the first entry, the first element of *values* is the value in the first entry, and so on. (There are no guarantees as to the order in which entries will be scanned.)

```
(let ((h (make-hash-table)))
  (setf (gethash 'color h) 'brown)
  (setf (gethash 'name h) 'fred)
  (scan-hash h)) ⇒ #Z(name color) and #Z(fred brown)
```

- **scan-lists-of-lists** *lists-of-lists* **&optional** *leaf-test* ⇒ *nodes*

The argument *lists-of-lists* is viewed as an n-ary tree where each internal node is a non-empty list and the elements of the list are the children of the node. A node is considered to be a leaf if it is an atom or if it satisfies the predicate *leaf-test* (if present). (The predicate can count on only being applied to conses.)

The function **scan-lists-of-lists** creates a series containing all of the nodes in *lists-of-lists*. The nodes are enumerated in preorder (i.e., first the root is output, then the nodes in the first child of the root are enumerated in full, then the nodes in the second child of the root are enumerated in full, etc.).

The function **scan-lists-of-lists** does not assume that the node lists end in **nil**; however, it ignores any non-list cdrs. (This behavior increases the utility of **scan-lists-of-lists** when it is used to scan Lisp code.) However, **scan-lists-of-lists** assumes that *lists-of-lists* is a tree as opposed to a more general graph. If some node in the input has more than one parent, then this node (and its descendants) are enumerated more than once. If the input is cyclic, the output series is unbounded in length.

```
(scan-lists-of-lists 'c) ⇒ #Z(c)
(scan-lists-of-lists '((c) nil)) ⇒ #Z(((c) nil) (c) c nil)
(scan-lists-of-lists '((c) nil) #'(lambda (e) (atom (car e))))
  ⇒ #Z(((c) nil) (c) nil)
```

- **scan-lists-of-lists-fringe** *lists-of-lists* **&optional** *leaf-test* ⇒ *leaves*

This is the same as **scan-lists-of-lists** except that it only scans the leaves of the tree, skipping all internal nodes. Note that **nil** is treated as a leaf, rather than as an internal node with no children.

```
(scan-lists-of-lists-fringe 'c) ⇒ #Z(c)
(scan-lists-of-lists-fringe '((c) nil)) ⇒ #Z(c nil)
(scan-lists-of-lists-fringe '((c) nil) #'(lambda (e) (atom (car e))))
  ⇒ #Z((c) nil)
```

• **scan-symbols &optional** (*package* ***package***) ⇒ *symbols*

Creates a series, in no particular order, and possibly containing duplicates, of the symbols accessible in *package* (which defaults to the current package).

```
(scan-symbols) ⇒ #Z(foo bar baz ... zot) <in some order>
```

• **scan-file** *file-name* **&optional** (*reader* **#'read**) ⇒ *items*

Opens the file named by the string *file-name* and applies the function *reader* to it repeatedly until the end of the file is reached. The function *reader* must accept the standard input-function arguments *input-stream*, *eof-error-p*, and *eof-value* as its arguments. (For instance, *reader* can be **read**, **read-preserving-white-space**, **read-line**, or **read-char**.) If omitted, *reader* defaults to **read**. The function **scan-file** returns a series of the values returned by *reader*, up to but not including the value returned when the end of file is reached. The file is correctly closed, even if an abort occurs. As the basis for the examples below, suppose that the file **"test.lisp"** contains "**(A) 1**".

```
(scan-file "test.lisp") ⇒ #Z((a) 1)
(scan-file "test.lisp" #'read-char) ⇒ #Z(#\( #\A #\) #\space #\1)
```

• **scan-fn** *type init step* **&optional** *test* ⇒ *results-1* ... *results-m*

The higher-order function **scan-fn** supports the generic concept of scanning. The *type* is a type specifier. The **values** construct can be used to indicate multiple types; however, *type* cannot indicate zero types. If *type* indicates $m$ types $r_1$ ... $r_m$, then **scan-fn** returns $m$ series where *results-i* has the type (**series** $r_i$). The arguments *init*, *step*, and *test* are functions.

The *init* must be of type (**function** () (**values** $r_1$ ... $r_m$)).

The *step* must be of type (**function** ($r_1$ ... $r_m$) (**values** $r_1$ ... $r_m$)).

The *test* (if present) must be of type (**function** ($r_1$ ... $r_m$) **T**).

The elements of the *results-i* are computed as follows:

(**values** *results-*$1_0$ ... *results-*$m_0$) = (**funcall** *init*)

(**values** *results-*$1_j$ ... *results-*$m_j$) = (**funcall** *step results-*$1_{(j-1)}$ ... *results-*$m_{(j-1)}$)

The outputs all have the same length. If there is no *test*, the outputs have unbounded length. If there is a *test*, the outputs consist of the elements up to but not including, the first elements for which the following is not **nil**. It is guaranteed that *step* will not be applied to the elements that pass the test.

(**funcall** *test results-*$1_j$ ... *results-*$m_j$)

If *init*, *step*, or *test* have side effects, they can count on being called in the order indicated by the equations above, with *test* called just before *step* on each cycle. However, due to the lazy evaluation nature of series, these functions will not be called until their outputs are actually used (if ever). In addition, no assumptions can be made about the relative order of evaluation of these calls with regard to execution in other parts of a given series expression.

```
(scan-fn 'list #'(lambda () '(a b c d)) #'cddr #'null)
  ⇒ #Z((a b c d) (c d))

(scan-fn T  #'(lambda () '(a b c d)) #'cddr)
  ⇒ #Z((a b c d) (c d) nil nil ...)

(let ((list '(a b c)))
  (scan-fn '(values T list)
           #'(lambda () (values (car list) list))
           #'(lambda (element list) (declare (ignore element))
               (values (cadr list) (cdr list)))
           #'(lambda (element list) (declare (ignore element))
               (null list))))
  ⇒ #Z(a b c) and #Z((a b c) (a b) (c))
```

If there is no *test*, then each time an element is output, the function *step* is applied to it. Therefore, it is important that other factors in an expression cause termination before **scan-fn** computes an element that *step* cannot be applied to. In this regard, it is interesting that the following equivalence is almost, but not quite true. The difference is that including the *test* argument in the call on **scan-fn** guarantees that *step* will not be applied to the element which fails *test*, while the expression using **until-if** guarantees that it will.

```
(scan-fn T init step test) ≠ (until-if test (scan-fn T init step))
```

• **scan-fn-inclusive** *type init step test* ⇒ *results-1 ... results-m*

The higher-order function **scan-fn-inclusive** is the same as **scan-fn** except that the first set of elements for which *test* is true is included in the output. As with **scan-fn**, it is guaranteed that *step* will not be applied to the elements for which *test* is true.

```
(scan-fn-inclusive 'list #'(lambda () '(a b c d)) #'cddr #'null)
  ⇒ #Z((a b c d) (c d) ())
```

## Mapping

By far the most common kind of series operation is mapping. In cognizance of this fact, four different ways are provided for specifying mapping.

• **map-fn** *type function sources-1 ... sources-n* ⇒ *results-1 ... results-m*

The higher-order function **map-fn** supports the generic concept of mapping. The *type* is a type specifier, which specifies the type of value(s) returned by *function*. The **values** construct can be used to indicate multiple types; however, *type* cannot indicate

zero values. If *type* indicates $m$ types $r_1 \ldots r_m$, then **map-fn** returns $m$ series where *results-i* has the type **(series** $r_i$**)**. The argument *function* is a function. The remaining arguments (if any) are all series. Suppose that *sources-i* has the type **(series** $s_i$**)**.

The *function* must be of type **(function (**$s_1 \ldots s_n$**) (values** $r_1 \ldots r_m$**))**.

The length of each output is the same as the length of the shortest input. If there are no bounded series inputs, the outputs are unbounded. The elements of the *results-i* are the results of applying *function* to the corresponding elements of the *sources-i*.

**(values** *results-*$1_j$ *...* *results-*$m_j$**) = (funcall** *function sources-*$1_j$ *...* *sources-*$n_j$**)**

If *function* has side effects, it can count on being called first on the *sources-*$i_0$, then on the *sources-*$i_1$, and so on. However, due to the lazy evaluation nature of series, *function* will not be called on any group of input elements until the result is actually used (if ever). In addition, no assumptions can be made about the relative order of evaluation of these calls with regard to execution in other parts of a given series expression.

```
(map-fn 'integer #'+ #Z(1 2 3) #Z(4 5)) ⇒ #Z(5 7)
(map-fn T #'gensym) ⇒ #Z(#:G003 #:G004 #:G005 ...)
(map-fn '(values integer rational) #'floor #Z(1/4 9/5 12/3))
  ⇒ #Z(0 1 4) and #Z(1/4 4/5 0)
```

The function **map-fn** can be used to specify any kind of mapping operation. However, in practice, it can be cumbersome to use. Three shorthand forms are provided, which are more convenient in particular common situations.

- **#M** *function* ⇒ *series-function*

Often one wants to map a given named function over one or more series producing a series of the resulting values. This can be done succinctly by using the **#** macro character syntax **#M**. This readmacro converts a non-series function into a series function by using mapping. All but the first value returned by *function* are ignored. The form **#M***function* can only be used in the function position of a list. To activate the syntax **#M**, you must call **(series::install :macro T)**.

```
(#Mf x y) ≡ (map-fn T #'f x y)
(collect (#Mcar (scan '((a) (b) (c))))) ⇒ (a b c)
```

- **mapping** *var-value-pair-list* **&body** *body* ⇒ *items*

The syntax **#M***function* is only helpful when the computation to be mapped is a named function. The form **mapping** is helpful in situations where a more complex computation needs to be mapped. The syntax of **mapping** is analogous to **let**. The *var-value-pair-list* specifies zero or more variables that are bound to successive values of series. The value parts of the pairs must all return series. The *body* is treated as the body of a **lambda** expression that is mapped over the series values. A series of the first values returned by this **lambda** expression is returned as the result of **mapping**. Any kind of declaration can be used at the beginning of the *body*; however it should be noted that the variables in the var-value pairs contain series elements, not series.

```
(mapping ((x r) (y s)) ...) ≡ (map-fn T #'(lambda (x y) ...) r s)
(mapping ((x (scan '(2 -2 3))))
    (declare (fixnum x))
  (expt (abs x) 3)) ⇒ #Z(8 8 27)
```

The form **mapping** supports a special syntax that facilitates the use of series functions that return multiple values. Instead of being a symbol, the variable part of a var-value pair can be a list of symbols. This list is treated the same way as the first argument to **multiple-value-bind**.

```
(mapping (((i v) (scan-plist '(a 1 b 2))))
  (list i v)) ⇒ #Z((a 1) (b 2))
```

- **iterate** *var-value-pair-list* **&body** *body* ⇒ **nil**

The form **iterate** is identical to **mapping** except that the value **nil** is always returned.

```
(iterate ...) ≡ (progn (collect-last (mapping ...)) nil)
(let ((item (scan '((1) (-2) (3)))))
  (iterate ((x (#Mcar item)))
    (if (plusp x) (prin1 x)))) ⇒ nil <after printing "13">
```

To a first approximation, **iterate** and **mapping** differ in the same way as **mapc** and **mapcar**. In particular, like **mapc**, **iterate** is intended to be used in situations where the *body* is being evaluated for side effect rather than for its result. However, due to the lazy evaluation semantics of series, the difference between **iterate** and **mapping** is more than just a question of efficiency.

If **mapcar** is used in a situation where the output is not used, time is wasted unnecessarily creating the output list. However, if **mapping** is used in a situation where the output is not used, no computation is performed, because series elements are not computed until they are used. Thus **iterate** can be thought of as a declaration that the indicated computation is to be performed even though the output is not used.

```
(let ((item (scan '((1) (-2) (3)))))
  (mapping ((x (#Mcar item)))
    (if (plusp x) (prin1 x)))
  nil) ⇒ nil <without printing any output>
```

An important use of the forms **mapping** and **iterate** is to create series expressions corresponding to nested loops. For instance, the following expression takes a vector of lists and produces a list of the sums of the elements in these lists. When optimization is applied, the series expression in the **mapping** body becomes a nested loop.

```
(collect (mapping ((l (scan 'vector '#((1 2 3) (3 4 5)))))
           (collect-sum (scan l)))) ⇒ (6 12)
```

## Truncation

The functions below support the concept of producing a bounded series as opposed to an unbounded one.

- `until` *bools items-1 ... items-n* ⇒ *initial-items-1 ... initial-items-n*

Truncates one or more series of elements based on a series of boolean values. The outputs consist of the elements of the inputs up to, but not including, the first element which corresponds to a non-null element of *bools*. That is to say, *initial-items-*$i_j$=*items-*$i_j$ and if the first non-null value in *bools* is the $m$th, each output has length $m$. (The effect of including the $m$th element in the output can be obtained by using `previous` as shown in the last example below.) In addition, the outputs terminate as soon as any input runs out of elements even if a non-null element of *bools* has not been encountered.

```
(until #Z(nil nil T nil T) #Z(1 2 -3 4 -5)) ⇒ #Z(1 2)
(until #Z(nil nil T nil T) #Z(1 2) #Z(a b c)) ⇒ #Z(1 2) and #Z(a b)
(until (series nil) (scan-range)) ⇒ #Z(0 1 2 ...)
(until #Z(nil nil T nil T) (scan-range)) ⇒ #Z(0 1)
(let ((x #Z(1 2 -3 4 -5)))
  (until (previous (#Mminusp x)) x)) ⇒ #Z(1 2 -3)
```

- `until-if` *pred items-1 ... items-n* ⇒ *initial-items-1 ... initial-items-n*

This function is the same as `until` except that it takes a functional argument instead of a series of boolean values. The function *pred* is mapped over *items-1* to obtain a series of boolean values that control the truncation. The basic relationship between `until-if` and `until` is shown in the last example below.

```
(until-if #'minusp #Z(1 2 -3 4 -5)) ⇒ #Z(1 2)
(until-if #'minusp #Z(1 2) #Z(a b c)) ⇒ #Z(1 2) and #Z(a b)
(until-if #'minusp (scan-range)) ⇒ #Z(0 1 2 ...)
(until-if #'pred items) ≡ (let ((v items)) (until (#Mpred v) v))
```

- `cotruncate` *items-1 ... items-n* ⇒ *initial-items-1 ... initial-items-n*

The inputs and outputs are all series and the number of outputs is the same as the number of inputs. Further, the elements of the outputs are exactly the same as the elements of the inputs. However, the outputs are truncated so that they are all the same length as the shortest input.

```
(cotruncate #Z(1 2 -3 4 -5) #Z(10)) ⇒ #Z(1) and #Z(10)
(cotruncate (scan-range) #Z(a b)) ⇒ #Z(0 1) and #Z(a b)
(cotruncate #Z(a b) #Z()) ⇒ #Z() and #Z()
(cotruncate ...) ≡ (until (series nil) ...)
```

## Other On-Line Transducers

Transducers compute series from series and form the heart of most series expressions. The ubiquitous transduction operations of mapping and truncating are described above. This section presents the other predefined transducers that are on-line.

- **previous** *items* **&optional** (*default* **nil**) (*amount* **1**) ⇒ *shifted-items*

  Creates a series that is shifted right *amount* elements. The input *amount* must be a positive integer. The shifting is done by inserting *amount* copies of *default* before *items* and discarding *amount* elements from the end of *items*. The output is always the same length as the input.

```
(previous #Z(a b c)) ⇒ #Z(nil a b)
(previous #Z(a b c) 'z) ⇒ #Z(z a b)
(previous #Z(a b c) 'z 2) ⇒ #Z(z z a)
(previous #Z()) ⇒ #Z()
```

  The word *previous* is used as the name for this function, because the function is typically used to access previous values of a series. An example of **previous** used in this way is shown in conjunction with **until** above. To insert some amount of stuff in front of a series without losing any of the elements off the end, use **catenate**.

- **latch** *items* **&key :after :before :pre :post** ⇒ *masked-items*

  This function acts like a *latch* electronic circuit component. Each input element causes the creation of a corresponding output element. After a specified number of non-null input elements have been encountered, the latch is triggered and the output mode is permanently changed.

  The **:after** and **:before** arguments specify the latch point. The latch point is just after the **:after**-th non-null element in *items* or just before the **:before**-th non-null element. If neither **:after** nor **:before** is specified, an **:after** of 1 is assumed. If both are specified, it is an error.

  If a **:pre** is specified, every element prior to the latch point is replaced by this value. If a **:post** is specified, this value is used to replace every element after the latch point. If neither is specified, a **:post** of **nil** is assumed.

```
(latch #Z(nil c nil d e)) ⇒ #Z(nil c nil nil nil)
(latch #Z(nil c nil d e) :before 2 :pre 'z) ⇒ #Z(z z z d e)
(latch #Z(nil c nil d e) :before 2 :post T) ⇒ #Z(nil c nil T T)
```

- **collecting-fn** *type init function sources-1 ... sources-n* ⇒ *results-1 ... results-m*

  The higher-order function **collecting-fn** supports the generic concept of an on-line transducer with internal state. The *type* is a type specifier, which specifies the type of value(s) returned by *function*. The **values** construct can be used to indicate multiple types; however, *type* cannot indicate zero types. If *type* indicates $m$ types $r_1 \ldots r_m$, then **collecting-fn** returns $m$ series where *result-i* has the type **(series** $r_i$**)**. The arguments *init* and *function* are functions. The remaining arguments (if any) are all series. Suppose that *sources-i* has the type **(series** $s_i$**)**.

The *init* must be of type `(function () (values` $r_1$ ... $r_m$`))`.
The *function* must be of type
`(function (`$r_1$ ... $r_m$ $s_1$ ... $s_n$`) (values` $r_1$ ... $r_m$`))`.
The length of each output is the same as the length of the shortest input. If there are no bounded series inputs, the outputs are unbounded. The elements of the *results-i* are computed as follows:

`(values` *results-*$1_0$... *results-*$m_0$`) =`
`  (multiple-value-call` *function* `(funcall` *init*`)` *sources-*$1_0$ ... *sources-*$n_0$`)`
`(values` *results-*$1_j$... *results-*$m_j$`) =`
`  (funcall` *function results-*$1_{(j-1)}$ ... *results-*$m_{(j-1)}$ *sources-*$1_j$ ... *sources-*$n_j$`)`

If *init* and/or *function* have side effects, they can count on being called in the order indicated by the equations above. However, due to the lazy evaluation nature of series, these functions will not be called until their outputs are actually used (if ever). In addition, no assumptions can be made about the relative order of evaluation of these calls with regard to execution in other parts of a given series expression.

```
(collecting-fn T #'(lambda () 0) #'+ #Z(1 2 3)) ⇒ #Z(1 3 6)
(collecting-fn T #'(lambda () 5) #'+ #Z(1 2 3)) ⇒ #Z(6 8 11)
(collecting-fn T #'(lambda () 0) #'+ #Z(1 2) #Z(4 5)) ⇒ #Z(5 12)
(collecting-fn '(values integer integer)
               #'(lambda () (values 0 1))
               #'(lambda (sum prod x) (values (+ sum x) (* prod x)))
               #Z(1 2 3))
  ⇒ #Z(1 3 6) and #Z(1 2 6)
```

It is important to remember that when computing the first elements of the output, *function* is called with the values returned by *init* preceding the first elements of the series inputs. The order of arguments to `collecting-fn` is chosen to highlight this fact.

```
(collecting-fn T #'(lambda () nil) #'cons #Z(a b))
  ⇒ #Z((nil . a) ((nil . a) . b))
(collecting-fn T #'(lambda () nil) #'(lambda (l x) (cons x l)) #Z(a b))
  ⇒ #Z((a) (b a))
```

The first of the six examples above shows the most common way `collecting-fn` is used. In this usage, *function* takes two arguments returning one and the value returned by *init* is a left identity of *function*. In this situation, `results-`$1_0$`=sources-`$1_0$. Sometimes, this behavior is desired even in situations where *function* does not have a left identity. This can be achieved by using an auxiliary flag as shown below. This example computes a running maximum. The auxiliary flag is used to differentiate the first element of the input from the rest.

```
(defun collecting-max (numbers)
    (declare (optimizable-series-function))
  (values
    (collecting-fn '(values number T)
                   #'(lambda () (values 0 T))
                   #'(lambda (max first? x)
                       (values (if first? x (max max x)) nil))
                   numbers)))
(collecting-max #Z(9 4 25 6)) ⇒ #Z(9 9 25 25)
```

The use of an auxiliary flag is not particularly efficient. As a result, it is usually better to use a left identity of *function* when possible. The only exception to this is that if *function* is expensive to compute, using a flag may promote efficiency by eliminating one execution of *function*.

## Choosing and Expanding

Choosing and its inverse are particularly important kinds of off-line transducers. (Underlining is used to indicate series inputs and outputs that are off-line.)

- `choose` *bools* `&optional` *items* ⇒ *chosen-items*

Chooses elements from a series based on a boolean series. The off-line output consists of the elements of *items* that correspond to non-null elements of *bools*. That is to say, the $j$th element of *items* is in the output if and only if the $j$th element of *bools* is non-null. The order of the elements in *chosen-items* is the same as the order of the elements in *items*. The output terminates as soon as either input runs out of elements. If no *items* input is specified, then the non-null elements of *bools* are themselves returned as the output of `choose`.

```
(choose #Z(T nil T nil) #Z(a b c d)) ⇒ #Z(a c)
(choose #Z(a nil b nil)) ⇒ #Z(a b)
(choose #Z(nil nil) #Z(a b)) ⇒ #Z()
```

(An interesting aspect of `choose` is that the output series is off-line rather than having the two input series be off-line. This is done in recognition of the fact that the two input series are always in synchrony with each other; and having only one off-line port allows more flexibility then having two off-line ports.)

One might want to select elements out of a series based on their positions in the series rather than on boolean values. This can be done using `mask` as shown below.

```
(choose (mask #Z(0 2)) #Z(a b c d)) ⇒ #Z(a c)
(choose (#Mnot (mask #Z(0 2))) (scan-range)) ⇒ #Z(1 3 4 5 ...)
```

A key feature of `choose` in particular, and many off-line transducers in general, is illustrated by the expression below. In this expression, the `choose` causes the first `scan` to get out of phase with the second `scan`. As a result, it is important to think of series expressions as passing around series objects rather than as abbreviations for loops where things are always happening in lock step. The latter point of view might lead to the idea that the output of the expression below would be `((a 1) (c 2) (d 4))`.

```
(let ((tag (scan '(a b c d e)))
      (x (scan '(1 -2 2 4 -5))))
  (collect (#Mlist tag (choose (#Mplusp x) x)))) ⇒ ((a 1) (b 2) (c 4))
```

- `choose-if` *pred* *items* ⇒ *chosen-items*

This function is the same as `choose`, except that it maps the non-series function *pred* over *items* to obtain a series of boolean values which control the choosing. In addition, the input is off-line rather than the output. (It turns out that this allows for better optimization in some situations.) The logical relationship between `choose` and `choose-if` is shown in the last example below.

```
(choose-if #'plusp #Z(-1 2 -3 4)) ⇒ #Z(2 4)
(choose-if #'identity #Z(a nil nil b nil)) ⇒ #Z(a b)
(choose-if #'pred items) ≡ (let ((v items)) (choose (#Mpred v) v))
```

- **spread** *gaps items* **&optional** (*default* **nil**) ⇒ *expanded-items*

This function is a quasi-inverse of **choose**. The off-line output contains the elements of *items* spread out by interspersing them with copies of *default*. If the $i$th element of *gaps* is $n$ (a non-negative integer), then the $i$th element of *items* is preceded by $n$ copies of *default*. The output stops as soon as either input runs out of elements.

```
(spread #Z(1 1) #Z(2 4) -1) ⇒ #Z(-1 2 -1 4)
(spread #Z(0 2 4) #Z(a b)) ⇒ #Z(a nil nil b)
(spread #Z(1) #Z(a b)) ⇒ #Z(nil a)
```

- **expand** *bools items* **&optional** (*default* **nil**) ⇒ *expanded-items*

This function is second kind of quasi-inverse of **choose**. The output contains the elements of the off-line input *items* spread out into the positions specified by the non-null elements in *bools*—i.e., the $j$th element of *items* is in the position occupied by the $j$th non-null element in *bools*. The other positions in the output are occupied by *default*. The output stops as soon as *bools* runs out of elements or a non-null element in *bools* is encountered for which there is no corresponding element in *items*.

```
(expand #Z(nil T nil T T) #Z(a b c)) ⇒ #Z(nil a nil b c)
(expand #Z(nil T nil T T) #Z(a)) ⇒ #Z(nil a nil)
(expand #Z(nil T) #Z(a b c) 'z) ⇒ #Z(z a)
(expand #Z(nil T nil T T) #Z()) ⇒ #Z(nil)
```

- **split** *items bools-1 ... bools-n* ⇒ *items-1 ... items-n items-n+1*

This function is similar to **choose** except that instead of producing one restricted output, it partitions the input series between two or more outputs. This makes it possible to use both the chosen items and the non-chosen items in later computations.

If there are $n$ boolean inputs then there are $n+1$ outputs, all of which are off-line. Each input element is placed in exactly one output series. Suppose that the $j$th element of *bools-1* is non-null. In this case, the $j$th element of *items* will be placed in *items-1*. On the other hand, if the $j$th element of *bools-1* is **nil**, the second boolean input (if any) is consulted to see whether the input element should be placed in the second output or in a later output. (As in a **cond**, each time a boolean element is **nil**, the next boolean series is consulted.) If the $j$th element of every boolean series is **nil**, then the $j$th element of *items* is placed in *items-n+1*.

```
(split #Z(-1 -2 3 4) #Z(T T T T)) ⇒ #Z(-1 -2 3 4) and #Z()
(split #Z(-1 -2 3 4) #Z(T T nil nil)) ⇒ #Z(-1 -2) and #Z(3 4)
(split #Z(-1 -2 3 4) #Z(T T nil nil) #Z(nil T nil T))
  ⇒ #Z(-1 -2) and #Z(4) and #Z(3)
```

- **split-if** *items pred-1 ... pred-n* ⇒ *items-1 ... items-m items-n+1*

This function is the same as **split**, except that it takes predicates as arguments rather than boolean series. The predicates are applied to the elements of *items* to create boolean values. The relationship between **split-if** and **split** is almost but not exactly as shown below.

```
(split-if items #'f #'g) ≠ (let ((v items)) (split v (#Mf v) (#Mg v)))
```

The reason that the equivalence above does not quite hold is that, as in a `cond`, the predicates are not applied to individual elements of *items* unless the resulting value is needed to determine which output series the element should be placed in (e.g., if the first predicate returns non-null when given the $j$th element of items, the second predicate will not be called). This promotes efficiency and allows earlier predicates to act as guards for later predicates.

```
(split-if #Z(1.3 3 2.7 4) #'floatp) ⇒ #Z(1.3 2.7) and #Z(3 4)
(split-if #Z(1.3 3 2.7 4) #'floatp #'evenp)
  ⇒ #Z(1.3 2.7) and #Z(4) and #Z(3)
```

## Other Off-Line Transducers

This section describes a number of off-line transducers. (Underlining is used to indicate series inputs and outputs that are off-line.)

- `catenate` *items-1* ... *items-n* ⇒ *items*

  Creates a series by concatenating together two or more off-line input series. The length of the output is the sum of the lengths of the inputs.

```
(catenate #Z(b c) #Z() #Z(d)) ⇒ #Z(b c d)
(catenate #Z() #Z()) ⇒ #Z()
```

- `subseries` *items start* `&optional` *below* ⇒ *selected-items*

  Creates a series containing a subseries of the elements in the off-line input *items* from *start* up to, but not including, *below*. If *below* is greater than the length of *items*, output nevertheless stops as soon as the input runs out of elements. If *below* is not specified, the output continues all the way to the end of *items*. Both of the arguments *start* and *below* must be non-negative integers.

```
(subseries #Z(a b c d) 1) ⇒ #Z(b c d)
(subseries #Z(a b c d) 1 3) ⇒ #Z(b c)
(collect (subseries (scan list) x y)) ≡ (subseq list x y)
```

- `positions` *bools* ⇒ *indices*

  Returns a series of the indices of the non-null elements in the off-line input *bools*.

```
(positions #Z(T nil T 44)) ⇒ #Z(0 2 3)
(positions #Z(nil nil nil)) ⇒ #Z()
```

- `mask` *monotonic-indices* ⇒ *bools*

  This function is a quasi-inverse of `positions`. The off-line input *monotonic-indices* must be a strictly increasing series of non-negative integers. The output, which is always unbounded, contains `T` in the positions specified by *monotonic-indices* and `nil` everywhere else.

```
(mask #Z()) ⇒ #Z(nil nil ...)
(mask #Z(0 2 3)) ⇒ #Z(T nil T T nil nil ...)
(mask (positions #Z(nil a nil b nil))) ⇒ #Z(nil T nil T nil nil ...)
```

- **mingle** *items-1 items-2 comparator* ⇒ *items*

The output series contains the elements of the two off-line input series. The elements of *items-1* appear in the same order that they are read in. Similarly, the elements of *items-2* appear in the same order that they are read in. However, the elements from the two inputs are stably intermixed under the control of the *comparator*.

The *comparator* must accept two arguments and return non-null if and only if its first argument is strictly less than its second argument (in some appropriate sense). At each step, the *comparator* is used to compare the current elements in the two series. If the current element from *items-2* is strictly less than the current element from *items-1*, the current element is removed from *items-2* and transferred to the output. Otherwise, the next output element comes from *items-1*. (If, as in the first example below, the elements of the individual input series are ordered with respect to *comparator*, then the result will also be ordered with respect to *comparator*.)

```
(mingle #Z(1 3 7 9) #Z(4 5 8) #'<) ⇒ #Z(1 3 4 5 7 8 9)
(mingle #Z(1 7 3 9) #Z(4 5 8) #'<) ⇒ #Z(1 4 5 7 3 8 9)
```

- **chunk** *m* {*n*} *items* ⇒ *items-1* ... *items-m*

This function has the effect of breaking the off-line input series *items* into (possibly overlapping) chunks of width *m*. Successive chunks are displaced *n* elements to the right, in the manner of a moving window. The inputs *m* and *n* must both be positive integers. The input *n* is optional and defaults to 1. For uses of **chunk** to be transformed into loops, the arguments *m* and *n* must be constants.

The function **chunk** produces *m* output series. The *i*th chunk is composed of the *i*th elements of the *m* outputs. Suppose that the length of *items* is *l*. The length of each output is $\lfloor 1 + (l-m)/n \rfloor$. The outputs are computed as follows: $items\text{-}k_j = items_{(j*n+k-1)}$, $j$ counting from zero and $k$ counting from one.

Note that if $l < m$, there will be no output elements and if $l-m$ is not a multiple of $n$, the last few input elements will not appear in the output. If $m \geq n$, one can guarantee that the last chunk will contain the last element of *items* be catenating $n-1$ copies of an appropriate padding value to the end of *items*.

The first example below shows **chunk** used to compute a moving average. The second example shows **chunk** used to convert a property list into an association list.

```
(mapping (((xi xi+1 xi+2) (chunk 3 #Z(1 5 3 4 5 6))))
  (/ (+ xi xi+1 xi+2) 3)) ⇒ #Z(3 4 4 5)

(collect (mapping (((prop val) (chunk 2 2 (scan '(a 2 b 5 c 8)))))
           (cons prop val))) ⇒ ((a . 2) (b . 5) (c . 8))
```

## Collectors

Collectors produce non-series outputs based on series inputs. There are two basic kinds of collectors: ones that combine the elements of series together into aggregate data structures (e.g., into a list) and ones that compute some summary value from these elements (e.g., the sum).

- `collect-last` *items* `&optional` (*default* `nil`) ⇒ *item*

  Returns the last element of *items*. If *items* is of zero length, *default* is returned.

```
(collect-last #Z(a b c)) ⇒ c
(collect-last #Z() 'z) ⇒ z
```

- `collect-first` *items* `&optional` (*default* `nil`) ⇒ *item*

  Returns the first element of *items*. If *items* is of zero length, *default* is returned. The function `collect-first` only reads the first element of *items*. This means that none of the other elements will be computed, unless they are needed for some other purpose.

```
(collect-first #Z(a b c)) ⇒ a
(collect-first #Z() 'z) ⇒ z
```

- `collect-nth` *n items* `&optional` (*default* `nil`) ⇒ *item*

  Returns the *n*th element of *items*. If *n* is greater than or equal to the length of *items*, *default* is returned. The function `collect-nth` does not read past the *n*th element of *items*.

```
(collect-nth 1 #Z(a b c)) ⇒ b
(collect-nth 1 #Z() 'z) ⇒ z
```

- `collect` {*type*} *items* ⇒ *sequence*

  Creates a sequence containing the elements of *items*. The *type* argument specifies the type of sequence to be created. This type must be a proper subtype of `sequence`. It omitted, *type* defaults to `list`. If the *type* specifies an explicit length (i.e., of a vector), *items* must be short enough to fit in the space allowed. Any extra space is left uninitialized.

```
(collect #Z()) ⇒ ()
(collect #Z(a b c)) ⇒ (a b c)
(collect 'string #Z(#\B #\A #\R)) ⇒ "BAR"
(collect (#Mf (scan x) (scan y))) ≡ (mapcar #'f x y)
```

  Collecting is significantly more efficient if it can be determined at compile time whether the type is a subtype of `list` or `vector`. For vectors, further efficiency is obtained if the length of the vector is also specified as part of the type and known at compile time.

```
(collect '(vector * 3) #Z(1 2 3)) ⇒ #(1 2 3)
```

In addition to subtypes of sequence, the *type* can be specified to be **bag**. If this is the case, a list is produced with no guarantees as to the order of the elements. All other types specify that the order of the elements in the sequence created must be the same as their order in the input series. An unordered output is acceptable in many situations and is significantly more efficient than collecting into an ordered list.

```
(collect 'bag #Z(a b c)) ⇒ (c a b) <in some order>
(collect 'bag #Z()) ⇒ ()
```

- **collect-append** {*type*} *sequences* ⇒ *sequence*

Given a series of sequences, **collect-append** returns a new sequence by concatenating these sequences together in order. The *type* is a type specifier indicating the type of sequence created and must be a proper subtype of **sequence**. If *type* is omitted, it defaults to **list**. It must be possible for every element of every sequence in the input series to be an element of a sequence of type *type*. The result does not share any structure with the sequences in the input.

```
(collect-append #Z()) ⇒ ()
(collect-append #Z((a b) nil (c d))) ⇒ (a b c d)
(collect-append 'string #Z("A " "big " "cat.")) ⇒ "A big cat."
```

- **collect-nconc** *lists* ⇒ *list*

This function **nconc**s the elements of the series *lists* together in order and returns the result. This is the same as **collect-append** except that the input must be a series of lists, the output is always a list, the concatenation is done rapidly by destructively modifying the input elements, and therefore the output shares all of its structure with the input elements.

```
(collect-nconc #Z()) ⇒ ()
(collect-nconc #Z((a b) nil (c d))) ⇒ (a b c d)
(collect-nconc (#Mf (scan x) (scan y))) ≡ (mapcan #'f x y)
```

- **collect-alist** *keys values* ⇒ *alist*

Creates an association list containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. Following the order of the inputs, each key/value pair is entered into the association list being created so that it overrides all earlier associations.

```
(collect-alist #Z(a b) #Z()) ⇒ ()
(collect-alist #Z(a b) #Z(1 2)) ⇒ ((b . 2) (a . 1))
(collect-alist #Z(a b a) #Z(1 2 3)) ⇒ ((a . 3) (b . 2) (a . 1))
```

- **collect-plist** *indicators values* ⇒ *plist*

Creates a property list containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. Following the order of the inputs, each key/value pair is entered into the property list being created so that it overrides all earlier associations.

```
(collect-plist #Z(a b) #Z()) ⇒ ()
(collect-plist #Z(a b a) #Z(1 2 3)) ⇒ (a 3 b 2 a 1)
```

- **collect-hash** *keys values* **&rest** *option-plist* ⇒ *table*

  Creates a hash table containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. Following the order of the inputs, each key/value pair is entered into the hash table being created so that it overrides all earlier associations. The *option-plist* can contain any options acceptable to **make-hash-table**.

```
(collect-hash #Z(a b a) #Z(1 2 3))
  ⇒ #<hash-table 3764432> <a hash table containing a↦3 and  b↦2>
(collect-hash #Z(a b) #Z())
  ⇒ #<hash-table 3764464> <an empty hash table>
```

- **collect-file** *file-name items* **&optional** (*printer* **#'print**) ⇒ **T**

  Creates a file named *file-name* and writes the elements of the series *items* into it using the function **printer**. The function *printer* must accept two inputs: an object and an output stream. (For instance, *printer* can be **print**, **prin1**, **princ**, **pprint**, **write-char**, **write-string**, or **write-line**.) If omitted, *printer* defaults to **print**. The value **T** is always returned. The file is correctly closed, even if an abort occurs.

```
(collect-file "test.lisp" #Z((a) (1 2) T) #'prin1)
  ⇒ T <after writing "(A)(1 2)T" into the file>
```

- **collect-length** *items* ⇒ *number*

  Returns the number of elements in *items*.

```
(collect-length #Z()) ⇒ 0
(collect-length #Z(a b c)) ⇒ 3
```

- **collect-sum** *numbers* **&optional** (*type* **'number**) ⇒ *number*

  Computes the sum of the elements in *numbers*. These elements must be numbers, but they need not be integers. The *type* is a type specifier that indicates the type of sum to be created. If there are no elements in the input, a zero (of the appropriate type) is returned.

```
(collect-sum #Z() 'complex) ⇒ #C(0 0)
(collect-sum #Z(1 2 3) 'integer) ⇒ 6
(collect-sum #Z(1.1 1.2 1.3)) ⇒ 3.6
```

- **collect-max** *numbers* **&optional** *items* (*default* **nil**) ⇒ *item*

  Returns the element of *items* that corresponds to the maximum element of *numbers*. If *items* is omitted, then the maximum element of *numbers* is itself returned. The elements of *numbers* must be non-complex numbers, but they need not be integers. Processing stops as soon as either *numbers* or *items* runs out of elements. The value *default* is returned if either *numbers* or *items* has length zero.

```
(collect-max #Z(2 1 4 3) #Z()) ⇒ nil
(collect-max #Z() #Z() 0) ⇒ 0
(collect-max #Z(2 1 4 3)) ⇒ 4
(collect-max #Z(1.2 1.1 1.4 1.3) #Z(a b c d)) ⇒ c
```

- `collect-min` *numbers* `&optional` *items* (*default* `nil`) ⇒ *item*

  Returns the element of *items* that corresponds to the minimum element of *numbers*. If *items* is omitted, then the minimum element of *numbers* is itself returned. The elements of *numbers* must be non-complex numbers, but they need not be integers. Processing stops as soon as either *numbers* or *items* runs out of elements. The value *default* is returned if either *numbers* or *items* has length zero.

```
(collect-min #Z(2 1 4 3) #Z()) ⇒ nil
(collect-min #Z() #Z() 0) ⇒ 0
(collect-min #Z(2 1 4 3)) ⇒ 1
(collect-min #Z(1.2 1.1 1.4 1.3) #Z(a b c d)) ⇒ b
```

- `collect-and` *bools* ⇒ *bool*

  Computes the `and` of the elements in *bools*. As with the function `and`, `nil` is returned if any element of *bools* is `nil`. Otherwise, the last element of *bools* is returned. The value `T` is returned if *bools* has length zero. If a value of `nil` is encountered, `collect-and` immediately stops reading elements from *bools*.

```
(collect-and #Z()) ⇒ T
(collect-and #Z(a b c)) ⇒ c
(collect-and #Z(a nil c)) ⇒ nil
(collect-and (#Mpred (scan x) (scan y))) ≡ (every #'pred x y)
```

- `collect-or` *bools* ⇒ *bool*

  Computes the `or` of the elements in *bools*. As with the function `or`, `nil` is returned if every element of *bools* is `nil`. Otherwise, the first non-null element of *bools* is returned. The value `nil` is returned if *bools* has length zero. If a non-null value is encountered, `collect-or` immediately stops reading elements from *bools*.

```
(collect-or #Z()) ⇒ nil
(collect-or #Z(a b c)) ⇒ a
(collect-or #Z(a nil c)) ⇒ a
(collect-or (#Mpred (scan x) (scan y))) ≡ (some #'pred x y)
```

- `collect-fn` *type init function sources-1 ... sources-n* ⇒ *result-1 ... result-m*

  The higher-order function `collect-fn` is used to create collectors. It is identical to `collecting-fn` except that rather than returning series of values, it only returns the last element of each series. If the series that would be returned by `collecting-fn` given the same arguments have zero length, then the values returned by *init* are returned directly as the output of `collect-fn`.

```
(collect-fn 'integer #'(lambda () 0) #'+ #Z()) ⇒ 0
(collect-fn 'integer #'(lambda () 0) #'+ #Z(1 2 3)) ⇒ 6
(collect-fn T #'(lambda () init) #'f s)
   ≡ (let ((v init))
       (collect-last (collecting-fn T #'(lambda () v) #'f s) v))
```

As shown in the last example above, the *init* input to `collect-fn` does double duty, acting both as the *init* input to `collecting-fn` and as the *default* input to `collect-last`. To specify a default value that is different from the initial value, use `collect-last` and `collecting-fn` directly. This is shown in the following definition of a simplified form of `collect-max`.

```
(defun simple-collect-max (numbers)
    (declare (optimizable-series-function))
  (collect-last
    (collecting-fn '(values integer T)
                   #'(lambda () (values 0 T))
                   #'(lambda (max first? x)
                       (values (if first? x (max max x)) nil))
                   numbers)
    nil))
```

If the series inputs of `collect-fn` are unbounded, then `collect-fn` will not terminate. This is a property shared by all the predefined collectors, except `collect-first`, `collect-nth`, `collect-and` and `collect-or`.

## Defining New Series Functions

An important aspect of the Series macro package is that it is easy for programmers to define new series functions and macros. The standard Lisp defining forms `defun` and `defmacro` can be used to define new series operations.

However, when a series function is defined with `defun`, the Series macro package is not capable of optimizing a series expression containing this new function unless the declaration `optimizable-series-function` is specified in the `defun` and the `defun` appears before the expression in question. The declaration `optimizable-series-function` is not required when using `defmacro`.

- `optimizable-series-function &optional (`*n*` 1)`

The only place the declaration specifier `optimizable-series-function` is allowed to appear is in a declaration immediately inside a `defun`. It indicates that the function being defined is a series function that needs to be analyzed so that it can be optimized when it appears in series expressions. (A warning is issued if the function being defined neither takes a series as input nor produces a series as output.)

For optimization to be possible, there are some limitations on the form of the containing `defun`. The `lambda` list cannot contain any keywords other than `&optional`. It is erroneous if a default value for an optional argument refers to the values of other arguments. There cannot be any declarations in the body of the `defun` other than `ignore` and `type` declarations. In particular, none of the arguments can be declared to be `special`.

A final limitation is that the number of values returned by the function being defined must be a constant and this constant must be known to the Series macro package at the time the definition is initially processed. The argument *n* (default 1) to the declaration `optimizable-series-expression` specifies the number of values returned by the function being defined. (This is something that it would not be possible to reliably determine by local analysis.)

```
(defun collect-product (numbers)
    (declare (optimizable-series-function))
  (collect-fn 'number #'(lambda () 1) #'* numbers))
```

It may seem unduly restrictive that one can only use the keyword **&optional** when using **defun** to define an optimizable series function. However, this is not much of a problem, because **defmacro** can be used in situations where other keywords are desired. For example, **catenate** could be defined in terms of a more primitive series function **catenate2** (see page 44) as follows.

```
(defmacro catenate (items-1 items-2 &rest items-i)
  (if (null items-i) `(catenate2 ,items-1 ,items-2)
      `(catenate2 ,items-1 (catenate ,items-2 ,@ items-i))))
```

Using **defmacro** directly also makes it possible to define new higher-order series functions. For example, a series function analogous to the sequence function **substitute-if** could be defined as follows.

```
(defmacro substitute-if-series (newitem test items)
  `(let ((newitem ,newitem) (test ,test) (items ,items))
     (mapping ((item items))
       (if (funcall test item) newitem item))))

(substitute-if-series 3 #'minusp #Z(1 -1 2 -3)) ⇒ #Z(1 3 2 3)
```

# 3. Advanced Features

The various macros and functions discussed in Section 2 form a consistent whole and are all that most programmers will ever need to use. This section presents a number of useful but less often used features of the Series macro package.

## Alteration of Values

The transformations introduced by the Series macro package are inherently antagonistic to the transformations introduced by the macro **setf**. As a result, series function calls are not allowed to be used as destinations of **setf**. However, the Series macro package supports a related concept that is actually more powerful than **setf**.

- **alter** *destinations items* ⇒ **nil**

**alter** modifies the data structure underlying the series *destinations* so that, if the series were to be regenerated, the values in the series *items* would be obtained. The alteration process stops as soon as either input runs out of elements. The value **nil** is always returned. Note that while the data structure underlying *destinations* is modified, the series itself is not modified.

Consider the example below. The variable **x** initially contains the series of elements in the list **data** (i.e., **#Z(a b c)**). When this is collected, it yields the list **(a b c)**. The use of **alter** changes the first two elements of the list **data** so that it becomes **(1 2 c)**. However, the series **x** remains unchanged. The function **alter** is more powerful than **setf**, because it can be applied to a variable that holds a value, rather than having to be directly applied to the function call that produces the value. This makes it convenient to use the old value when deciding what the new value should be.

```
(let* ((data (list 'a 'b 'c))
       (x (scan data)))
  (values (collect x)
          (alter x (scan '(1 2)))
          data
          (collect x)))
 ⇒ (a b c) and nil and (1 2 c) and (a b c)
```

Like **setf**, **alter** cannot be applied to just any destination. Rather, **alter** can only be applied to series that are *alterable*. As shown in Figure 3.1, many scanners produce alterable series, however, many do not.

Two facts should be kept in mind about the alterability of the series produced by **scan-lists-of-lists-fringe**. First, altering a leaf will have no effect on the leaves enumerated. In particular, if a leaf is altered into a list of lists, the leaves of this subtree will not get enumerated. Second, if the entire input happens to be a leaf and gets altered, this will have no side effect on the input as a whole.

Figure 3.1 also shows that a number of transducers produce series that are alterable as long as the corresponding input series is alterable. In general, this is true as long as the elements of the output come directly from elements of an input. For example, the following alters a segment of a list.

```
(let ((data (list 'a 'b 'c 'd 'e)))
  (alter (subseries (scan data) 1 3) (scan-range))
  data) ⇒ (a 0 1 d e)
```

- `to-alter` *items alter-fn other-items-1 ... other-items-n* ⇒ *alterable-items*

Alterable series are created by using this function. The function `to-alter` takes a series and returns an alterable series containing the same elements. The elements of the output are taken directly from *items*. The input *alter-fn* is a function. The other inputs are all series, each of which must be at least as long as *items*. If there are $n$ inputs *other-items-i*, *alter-fn* must accept $n+1$ inputs.

If an attempt is made to alter the $j$th element of the output series, the alteration is performed by applying *alter-fn* to the new value as its first argument and the $j$th elements of the *other-items-i* as the remaining arguments. As an example, consider the following definition of a series function that scans the elements of a list. Alteration is performed by changing cons cells in the list being scanned.

```
(defun scan-list (list)
    (declare (optimizable-series-function))
  (let ((sublists (scan-sublists list)))
    (to-alter (#Mcar sublists)
              #'(lambda (new parent) (setf (car parent) new))
              sublists)))

(let* ((data (list 1 -1 2 -2))
       (x (scan-list data)))
  (alter (choose (#Mminusp x) x) (series 0))
  data) ⇒ (1 0 2 0)
```

`choose` — Output alterable if *items* input alterable.
`choose-if` — Output alterable if *items* input alterable.
`cotruncate` — Each output alterable if corresponding input alterable.
`scan` — Output alterable.
`scan-alist` — Both outputs alterable.
`scan-lists-of-lists-fringe` — Output alterable.
`scan-multiple` — All outputs alterable.
`scan-plist` — Both outputs alterable.
`split` — All outputs alterable if *items* input alterable.
`split-if` — All outputs alterable if *items* input alterable.
`subseries` — Output alterable if *items* input alterable.
`until` — Each output alterable if corresponding input alterable.
`until-if` — Each output alterable if corresponding input alterable.

Figure 3.1: Series functions producing alterable series.

## Generators and Gatherers

Generators and gatherers are yet another way of processing one-dimensional totally-ordered collections of elements. They were originally proposed by C. Perdue and P. Curtis as an alternative to series. However, it has since been realized that the two concepts are actually synergistically supportive, rather than competitive. As a result, generators and gatherers have been included as an integral part of the Series macro package.

**Generators.** A generator is similar to a series in that it represents a potentially unbounded, one-dimensional totally-ordered collection of elements and is supported by lazy evaluation. However, generators follow the semantics of streams more closely than series do. In particular, the fundamental operation available for generators is `next-in`, which gets the next element from a series by side effect. (No such operation is available for series.) If a generator is used in two places, the second use will only see the elements that are not read by the first use.

There is a close relationship between a generator and a series of the elements it produces. In particular, any series can be converted into a generator. As a result, all of the scanner functions used for creating series can be used to create generators as well and there is no need to have a separate set of functions for creating generators.

- `next-in` *generator* `&body` *action-list* ⇒ *item*

Reads the next element out of a generator. As with streams, the element is removed by side effect and will therefore not be seen anywhere else that elements are read from the generator in question.

The *action-list* specifies what should be done when *generator* runs out of elements. When the generator runs out of values, the *action-list* is evaluated and the value of the final form in it is returned as the value of the call on `next-in`.

If the *action-list* is empty, it is an error for the generator to run out of elements. It is erroneous (with unpredictable results) to apply `next-in` to a generator a second time after the generator runs out of elements.

- `generator` *series* ⇒ *generator*

Given a series, this function creates a generator containing the same elements. As an example of the use of generators consider the following.

```
(let ((x (generator (scan '(1 2 3 4)))))
  (loop (print (next-in x (return T)))
        (print (next-in x (return nil)))
        (princ ",")))
 ⇒ T <after printing "12,34,">
```

**Gatherers.** A gather is the inverse of a generator—i.e., it is analogous to an output stream rather than an input stream. An unbounded number of elements can be put into a gatherer one at a time. In a manner similar to a collector, the gatherer combines the elements based on some formula. The resulting value can be obtained at any time.

There is a close relationship between a gatherer and a collector function that combines elements in the same way. In particular, any one-input one-output collector can be

converted into a gatherer. As a result, all of the collectors used for computing summary results from series can be used to create gatherers and there is no need to have a separate set of functions for creating gatherers.

- `next-out` *gatherer item* ⇒ `nil`

  Writes a value into a gatherer. This is done be side effect in such a way that the value is seen from the perspective of every use of the gatherer in question. The value `nil` is always returned.

- `result-of` *gatherer* ⇒ *result*

  Retrieves the net result from a gatherer. This can be done at any time. However, it is erroneous (with unpredictable results) to apply `result-of` twice to the same gatherer, or to apply `next-out` to a gather once `result-of` has been applied.

- `gatherer` *collector* ⇒ *gatherer*

  The *collector* input must be a one input collector function. The *collector* input can be of the form `#'(lambda ...)`. (This is necessary when utilizing a predefined collector that takes more than one argument.) The function `gatherer` returns a gatherer that performs the same internal computation as the collector. As an example of the use of gatherers, consider the following.

```
(let ((x (gatherer #'collect))
      (y (gatherer #'(lambda (x) (collect-sum (choose-if #'oddp x))))))
  (dotimes (i 4)
    (next-out x i)
    (next-out y i)
    (if (evenp i) (next-out x (* i 10))))
  (values (result-of x) (result-of y))) ⇒ (0 0 1 2 20 3) and 4
```

- `gathering` *var-collector-pair-list* `&body` *body* ⇒ *result-1* ... *result-n*

  The *var-collector-pair-list* must be a list of pairs, where the first element of each pair is a symbol. The second element of each pair must be a form that, when prefixed with `#'` is acceptable as an argument to `gatherer`. The body can be any Lisp expression. Typically it will contain calls on `next-out`.

  Gathering operates as follows. Each variable in the *var-collector-pair-list* is bound to a gatherer produced by applying `gatherer` to the corresponding collector in the *var-collector-pair-list*. The *body* is then run until it terminates. The `gathering` form returns $n$ values where $n$ is the length of the *var-collector-pair-list*. Each value is the `result-of` the corresponding gatherer. For instance,

```
(gathering ((x collect)
            (y collect-sum))
  (dotimes (i 3)
    (next-out y i)
    (if (evenp i) (next-out x (* i 10))))) ⇒ (0 20) and 3
```

is equivalent to

```
(let ((x (gatherer #'collect))
      (y (gatherer #'collect-sum)))
  (dotimes (i 3)
    (next-out y i)
    (if (evenp i) (next-out x (* i 10))))
  (values (result-of x) (result-of y))) ⇒ (0 20) and 3
```

**Optimization of generators and gatherers.** A key idea behind generators and gatherers is that they can be implemented simply and elegantly as closures and that these closures can be compiled very efficiently if certain conditions are met.

First, the compiler must support an optimization P. Curtis calls "`let` eversion" in addition to the optimization methods presented in [7]. If a closure is created and used entirely within a limited lexical scope, the scopes of any bound variables nested in the closure can be enlarged (everted) to enclose all the uses of the closure. This allows the variables to be allocated on the stack rather than the heap.

Second, for a generator/gatherer closure to be compiled efficiently, it must be possible to determine at compile time exactly what closure is involved and exactly what the scope of use of the closure is. There are several aspects to this. The expression creating the generator/gatherer cannot refer to a free series variable. The generator/gatherer must be stored in a local variable. This variable must only be used in calls of `next-in`, `next-out`, and `result-of`, and not inside of a closure. In particular, the generator/gatherer cannot be stored in a data structure, stored in a special variable, or returned as a result value.

All of the examples above satisfy these restrictions. Further, as long as the collectors do not refer to free series variables, every use of `gathering` satisfies these restrictions.

The Series macro package includes an implementation of generators and gatherers. However, it does not support optimizations of the kind discussed in [7]. Thus, in general, there is no guarantee that uses of generators and gatherers will be optimized. However, it is guaranteed that uses of `gathering` will always be optimized and that generators and gatherers will be optimized when used in conjunction with `producing` (see below).

## Defining New Off-Line Series Functions

The following primitive form can be used to define any preorder series operation.

- `producing` *output-list input-list* `&body` *body* ⇒ *output-1 ... output-n*

Computes and returns a group of series and non-series outputs given a group of series and non-series inputs. The key feature of `producing` is that some or all of the series inputs and outputs can be processed in an off-line way. To support this, the processing in the body is performed from the perspective of generators and gatherers. Each series input is converted to a generator before being used in the body. Each series output is associated with a gatherer in the body.

The *output-list* has the same syntax as the binding list of a `let`. The names of the variables must be distinct from each other and from the names of the variables in the `input-list`. If there are $n$ variables in the *output-list*, then `producing` computes $n$ outputs. There must be at least one output variable. The variables act as the names for the outputs and can be used in either of two ways. First, if an output variable has

a value associated with it in the *output-list*, then the variable is treated as holding a non-series value. The variable is initialized to the indicated value and can be used in any way desired in the body. The eventual output value is whatever value is in the variable when the execution of the body terminates. Second, if an output variable does not have a value associated with it in the *output-list*, the variable is given as its value a gatherer that accumulates elements. The only valid way to use the variable in the body is in calls on `next-out`. The output returned is a series containing these elements. If the body never terminates, this series is unbounded.

The *input-list* also has the same syntax as the binding list of a `let`. The names of these variables must be distinct from each other and the names of the variables in the *output-list*. The values can be series or non-series. If the value is not explicitly specified, it defaults to `nil`. The variables act logically both as inputs and state variables and can be used in one of two ways. First, if an input variable is associated with a non-series value, then it is given this value before the evaluation of the body begins and can be used in any way desired in the body. Second, if an input variable is associated with a series, then the variable is given a generator corresponding to this series as its initial value. The only valid way to use the variable in the body is in calls on `next-in`.

Declarations can be included at the start of the body. However, the only declarations allowed are `ignore` declarations, type declarations, and `propagate-alterability` declarations (see below). In particular, it is an error for any of the input or output variables to be special.

In conception, the body can contain arbitrary Lisp expressions. After the appropriate generators and gatherers have been set up, the body is executed until it terminates. At that time the final values of the non-series output variables are returned as results of the `producing` form. The series outputs are returned one element at a time as they are produced. (Following the lazy evaluation semantics of series, the evaluation of the body is delayed so that individual series elements are not computed until they are actually used.) If the body never terminates, the series outputs (if any) are unbounded in length and the non-series outputs (if any) are never produced.

Although easy to understand, this view of what can happen in the body presents severe difficulties when optimizing (and even when evaluating) series expressions that contain calls on `producing`. As a result, several limitations are imposed on the form of the body to simplify the processing required.

The first limitation is that, exclusive of any declarations, the body must have the form `(loop (tagbody ...))`. The following example shows how `producing` could be used to implement a scanner creating an unbounded series of integers.

```
(defun scan-integers ()
    (declare (optimizable-series-function))
  (producing (nums) ((num -1))
      (declare (integer num) (type (series integer) nums))
    (loop
      (tagbody
        (setq num (1+ num))
        (next-out nums num)))))

(scan-integers) ⇒ #Z(0 1 2 3 4 ...)
```

The second limitation is that the form **terminate-producing** must be used to terminate the execution of the body. Any other method of terminating the body (e.g., with **return**) is an error. The following example shows how **producing** could be used to implement a simplified version of **collect-sum**. The function **terminate-producing** is used to stop the computation when **numbers** runs out of elements.

```
(defun simple-collect-sum (numbers)
    (declare (optimizable-series-function))
  (producing ((sum 0)) ((numbers numbers) num)
    (loop
      (tagbody
        (setq num (next-in numbers (terminate-producing)))
        (setq sum (+ sum num))))))

(simple-collect-sum #Z(1 2 3)) ⇒ 6
```

The third limitation is that calls on **next-out** associated with output variables must appear at top level in the **tagbody** in the body. They cannot be nested in other forms. In addition, an output variable can be the destination of at most one call on **next-out** and if it is the destination of a **next-out**, it cannot be used in any other way.

If the call on **next-out** for a given output appears in the final part of the **tagbody** in the body, after everything other than other calls on **next-out**, then the output is an on-line output—a new value is written on every cycle of the body. Otherwise the output is off-line.

The following example shows how **producing** could be used to implement a simple version of **split-if** that only accepts one predicate input. Items are read in one at a time and tested. Depending on the test, they are written to one of two outputs. Note the use of labels and branches to keep the calls on **next-out** at top level. Both outputs are off-line. The **scan-integers** example above shows an on-line output.

```
(defun split-if2 (items pred)
    (declare (optimizable-series-function 2) (off-line-port 0 1))
  (producing (items-1 items-2) ((items items) item)
     (declare (propagate-alterability items items-1)
              (propagate-alterability items items-2))
    (loop
      (tagbody
          (setq item (next-in items (terminate-producing)))
          (if (not (funcall pred item)) (go D))
          (next-out items-1 item)
          (go F)
        D (next-out items-2 item)
        F))))

(split-if2 #Z(1 -2 3 -4) #'plusp) ⇒ #Z(1 3) and #Z(-2 -4)
```

The fourth limitation is that the calls on **next-in** associated with an input variable **v** must appear at top level in the **tagbody** in the body, nested in assignments of the form (**setq** *element-variable* (**next-in v** ...)). They cannot be nested in other forms. In addition, an input variable can be the source for at most one call on **next-in** and if it is the source for a **next-in**, it cannot be used in any other way.

If the call on **next-in** for a given input has as its sole termination action **(terminate-producing)** and appears in the initial part of the **tagbody** in the body, before anything other than similar calls on **next-in**, then the input is an on-line input—a new value is read on every cycle of the body. Otherwise the input is off-line.

The following example shows how **producing** could be used to implement a simple version of **catenate** that only accepts two arguments. To start with, elements are read from the first input series. When this runs out, a flag is set and reading begins from the second input. Both inputs are off-line. The **simple-collect-sum** and **split-if2** examples above have on-line inputs.

```
(defun catenate2 (items-1 items-2)
    (declare (optimizable-series-function)
             (off-line-port items-1 items-2))
  (producing (items) ((items-1 items-1) (items-2 items-2)
                      (in-2 nil) item)
    (loop
      (tagbody
          (if in-2 (go D))
          (setq item (next-in items-1 (setq in-2 T) (go D)))
          (go F)
        D (setq item (next-in items-2 (terminate-producing)))
        F (next-out items item)))))
(catenate2 #Z(1 2) #Z(3 4)) ⇒ #Z(1 2 3 4)
```

- **terminate-producing** ⇒

This form (which takes no arguments) is used to terminate the execution of (the expansion of) the macro **producing**. As with the form **go**, **terminate-producing** does not return any values, rather control immediately leaves the current context. The form **terminate-producing** is only allowed to appear in the body of **producing**.

- **propagate-alterability** *input output*

Transducers that propagate alterability from inputs to outputs (such as **choose** and **split**) can be defined using the declaration **propagate-alterability** in conjunction with **producing**. (This declaration is not valid in any other context.) The declaration **propagate-alterability** specifies that attempts to alter an element of the indicated output will be supported by altering the corresponding element of the indicated input. (The corresponding element of the input is the one most recently read at the moment when the output element is written. It must be the case that the output element is the corresponding input element.) For an example, see the definition of **split-if2** above.

**Warnings about off-line inputs and outputs.** It is possible to obtain off-line inputs and outputs without using producing. The easiest way to do this is to define a series function by combining together one or more off-line series functions. For instance, in the example below, the *items* input is off-line, because it is connected directly to the off-line input of **choose-if**.

```
(defun choose-positive (items)
    (declare (optimizable-series-function) (off-line-port items))
  (choose-if #'plusp items))
(choose-positive #Z(1 -2 3 -4)) ⇒ #Z(1 3)
```

Although it may seem surprising, it is also possible to get an off-line input or output even when all of the functions used when defining a new series function are on-line. For instance, in the example below, the input *weights* is off-line.

```
(defun weighted-sum (numbers weights)
    (declare (optimizable-series-function 2) (off-line-port weights))
  (values (collect-sum numbers) (collect-sum (#M* numbers weights))))

(weighted-sum #Z(1 2 3) #Z(3 2)) ⇒ 6 and 7
```

To see why *weights* is off-line, consider what happens when the input **numbers** is longer than the input **weights**. In this situation, the computation of the first **collect-sum** must continue even after the computation of the second **collect-sum** halts. Thus, the reading of **weights** has to stop before the reading of **numbers** stops. As a result, **weights** cannot be handled in an on-line way.

As can be seen by the examples above, it is not simple to look at a function and determine whether or not a given input is on-line. This is unfortunate, since on-line ports are significantly more useful than off-line ones. The declaration **off-line-port** is supported to allow programmers to verify that ports they think are on-line are in fact on-line. It is also worthy of note that off-line ports virtually never arise when defining scanners or reducers.

- **off-line-port** *port-spec-1 ... port-spec-n*

  The declaration specifier **off-line-port** is used to indicate the inputs and outputs of a function that are off-line. The only place this declaration is allowed is in a **defun** that also contains the declaration **optimizable-series-function**. Each *port-spec-i* must either be a symbol that is one of the inputs of the function or an integer $j$ indicating the $j$th output (counting from zero). For example, **(off-line-port x 1)** indicates that the input **x** and the second output are off-line. By default, every port that is not mentioned in an **off-line-port** declaration is assumed to be on-line. A warning is issued whenever a port's actual on-line/off-line status does not agree with its declared status. Several examples of using the declaration specifier **off-line-port** are shown on the last few pages.

In the function **weighted-sum** above, it might well have been the programmers intention that the inputs **numbers** and **weights** would always have the same length. Or failing that, it might have been his intention that any excess values of *numbers* be ignored. If that were the case, there would be no need for the function to be off-line. An on-line version could be written by using the function **cotruncate** as shown below.

```
(defun on-line-weighted-sum (numbers weights)
    (declare (optimizable-series-function 2))
  (multiple-value-bind (numbers weights) (cotruncate numbers weights)
    (values (collect-sum numbers) (collect-sum (#M* numbers weights)))))

(on-line-weighted-sum #Z(1 2 3) #Z(3 2)) ⇒ 3 and 7
```

## Primitives

Given the large number of series functions described above, two questions naturally spring to mind. First, what is the motivation behind the exact set of functions chosen? In particular, what basis is there for thinking that the functions are all useful and that there are not lots of other functions that would be just as useful? Second, are all these functions primitive, or can some be defined in terms of others?

The functions supported by the Series macro package were chosen by essentially taking the union of the preorder operations supported by the Common Lisp sequence functions [8], the MacLisp Loop macro [2], and APL [6] with the addition of a few new functions that fill obvious gaps in the resulting collection of functions. Thus, most of the functions have proven their utility in other contexts.

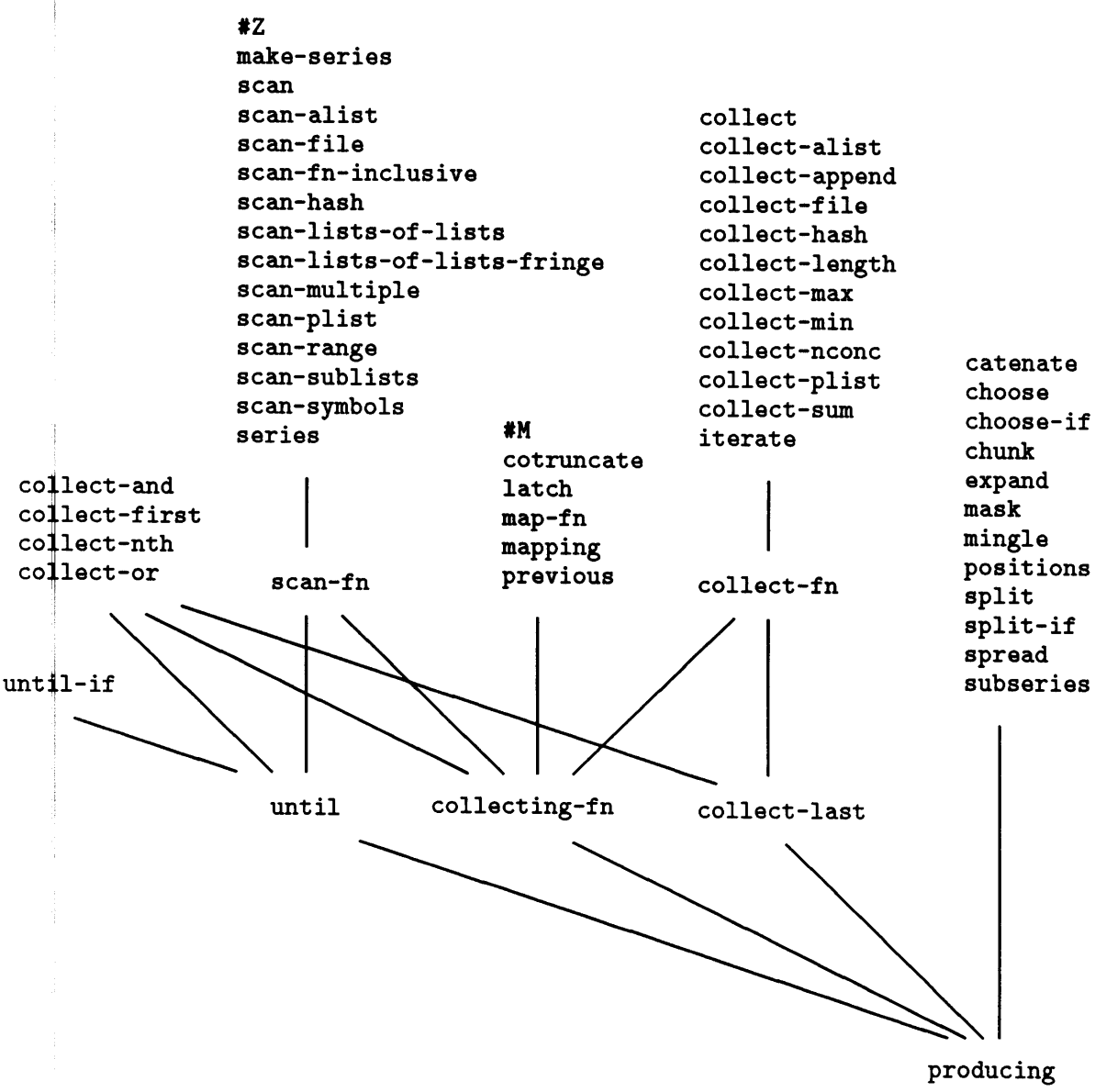


Figure 3.2: Series functions that can be defined in terms of others.

On the other side of the coin, there are undoubtedly many other useful series functions. As discussed above, comprehensive facilities are provided for defining new series functions. A key reason for presenting a wide array of predefined functions is to inspire users with thoughts of the wide variety of functions they can write for themselves.

The second question can be answered very precisely. As shown in Figure 3.2, a quite small set of basic series functions can be used to define all of the other series functions. In particular, three series functions (`until`, `collecting-fn`, and `collect-last`) can be used to implement most of the other series functions.

The function `collecting-fn` embodies the fundamental idea of series computations that utilize internal state. It can be used to define `map-fn` and other on-line transducers. (Off-line transducers typically have to be defined using `producing`.)

The function `until` embodies the fundamental idea of producing a series that is shorter than the shortest input series. As part of this, it embodies the idea of computing a bounded series from non-series inputs. Together with `collecting-fn`, `until` can be used to define `scan-fn`, which can be used to define any kind of scanner.

The function `collect-last` embodies the fundamental idea of producing a non-series value from a series. Together with `collecting-fn`, it can be used to define `collect-fn`, which can be used to define most kinds of collectors.

The collectors `collect-first`, `collect-nth`, `collect-and`, and `collect-or` are unusual in that they can sometimes produce their outputs without having to process all of the elements of their inputs. The function `until` has to be used in conjunction with `collecting-fn` and `collect-last` when defining these collectors.

Beyond what is shown in Figure 3.2, there are three other primitives that are sometimes required when defining series functions. The function `to-alter` is used to make series alterable. The form `series-element-type` is used to create `type` declarations that are based on the types of inputs. The form `encapsulated` (see below) is used to assist in the definition of the functions `scan-file`, `scan-symbols`, `scan-hash`, and `collect-file`.

It should also be noted that if the non-primitive operations in Figure 3.2 were defined by the user, many of them would have to be defined as macros rather than functions, because their argument lists are more complicated than what is allowed in a `defun` that uses the declaration `optimizable-series-function`. For example `defmacro` would have to be used when defining `catenate`, because the argument list utilizes `&rest`.

**Encapsulating forms.** Some of the features provided by Common Lisp are supported solely by encapsulating forms and are not available in any other way. The prototypical example of this is `unwind-protect`. There is no way to specify code that will always be run, even when a computation aborts, without using this form. The Series macro package provides a special primitive form that can be used to take advantage of such encapsulating forms when defining a series function.

- `encapsulated` *encapsulating-fn scanner-or-collector* ⇒ *result*

This macro specifies a function that places an encapsulating form around the computation performed by the second input. The input *encapsulating-fn* must be of the form `#'...` and must be a function that takes in a Lisp expression and wraps the appropriate encapsulating form around this expression, returning the resulting code. If optimization is possible, this function will be called with the entire loop produced as its

argument. If optimization is not possible, the encapsulation will be applied to a smaller context, which is nevertheless guaranteed to surround the computation corresponding to the *scanner-or-collector*.

The second input of **encapsulated** must be a literal call on **collect-fn**, **scan-fn**, or **scan-fn-inclusive**. If it is a call on **scan-fn**, a *test* argument must be supplied. The second argument can count on being evaluated in the scope of the encapsulating form. The results returned by the call are returned as the results of **encapsulated**. The following shows how **encapsulated** could be used to define a simplified version of **collect-file**.

```
(defmacro simple-collect-file (name items)
  (let ((file (gensym)))
    `(encapsulated
       #'(lambda (body)
           `(with-open-file (,',file ,',name :direction :output)
              ,body))
       (collect-fn T #'(lambda () T)
                   #'(lambda (state item)
                       (print item ,file)
                       state)
                   ,items))))
```

An important aspect of the example above is that the encapsulating form binds a variable (**file**) that is referred to by the expression forming the second input. However, the Series macro package does not know anything about this variable. In particular, it will do nothing to avoid name clashes if the same encapsulating form is used twice or if a name clashes with some other variable in the expression. Therefore, **gensym** should be used to create any variables bound by an encapsulating form. Further, no guarantees are made about the relative nesting order of encapsulating forms, so one form cannot refer to the variables bound by another.

The series functions **scan-file** and **collect-file** are defined using the encapsulating form **with-open-file**. In addition, the Symbolics Lisp Machine versions of the functions **scan-symbols** and **scan-hash** are defined using special encapsulating forms that are not part of standard Common Lisp. For the time being, the pure Common Lisp versions of these functions are handled in rather inefficient ways. However, the future standard version of Common Lisp is slated to contain encapsulating forms that will allow efficient, portable implementations of these functions.

## Features That Facilitate Debugging

The Series macro package supports a number of features that facilitate debugging. One example of this is the fact that the macro package tries to use the variable names that are bound by a **let** in the code produced. Since the macro package is forced to use variable renaming to implement variable scoping, it cannot guarantee that these variable names will be used. However, there is a high probability that they will. If a break occurs in the middle of a series expression, these variables can be inspected to determine what is going on. If a **let** variable holds a series, then the variable will contain the current element of the series. For example, the series expression below is transformed into the loop shown. (For a discussion of how this transformation is performed see [17].)

```
(let* ((v (get-vector user))
       (x (scan 'vector v)))
  (collect-sum x))
```

⇓

```
(let ((#:index-7 0) (#:limit-5 0) (#:sum-9 0) v x)
    (declare (type fixnum #:index-7 #:limit-5) (type number #:sum-9))
  (setq v (get-vector user))
  (setq #:index-7 -1)
  (setq #:limit-5 (length v))
  (setq #:sum-9 0)
  (tagbody
    #:L-10 (incf #:index-7)
           (if (not (< #:index-7 #:limit-5)) (go series::end))
           (setq x (aref v #:index-7))
           (setq #:sum-9 (+ #:sum-9 x))
           (go #:L-10)
   series::end)
  #:sum-9)
```

- `*last-series-loop*`

  This variable contains the loop most recently produced by the Series macro package. After evaluating (or macro-expanding) a series expression, this variable can be inspected to see the code produced.

- `*last-series-error*`

  This variable contains the most recently printed warning or error message produced by the Series macro package. The information in this variable can be useful for tracking down errors.

- `*series-expression-cache*`

  Like any macro package, the Series macro package operates based on macros—macro expansion causing series expressions to be replaced by loops. In the case of the Series macro package, the macros are relatively complex and the macro expansion process takes a significant amount of time. Fortunately, from the point of view of running compiled code, it matters very little how much overhead was involved expanding macros during the compilation process. However, the overhead can be significant when evaluating interpreted code. This is particularly true if a series expression is in an inner loop and has to be macro expanded many times.

  This problem is alleviated by maintaining a hash table of macro expansions to avoid having to process the same series expression more than once. However, using a hash table can cause two kinds of problems. First, by pointing to series expressions in otherwise obsolete pieces of code, the hash table can stymie garbage collection and use up significant amounts of memory. Second, once a series expression has been processed, it will never be reprocessed unless it is read in a second time. This speeds up execution; however, if the definition of some series function, series macro, or even any other macro that is contained in the series expression is changed, this change will not be reflected in the evaluation of the expression unless the expression is reread. (This same problem also applies in any environment where functions are being compiled before use rather than interpreted.)

The control variable `*series-expressions-cache*` normally contains the hash table used for hashing. If it is set to `T`, all cached information is forgotten and the construction of a new hash table is initiated. This causes each series expression to be reprocessed the next time it is evaluated. This can be helpful before garbage collecting or after changing the definition of some series function or macro. If the control variable is set to `nil`, the caching feature is turned off altogether and each series expression is processed every time it is evaluated.

# 4. Ugly Details

There are two areas of difficulty in the current implementation of the series macro package. The first stems from theoretical limits of the algorithms used. The second stems from the desire to provide a fully portable implementation. Maintaining portability significantly limits the amount of integration that can be achieved with any one implementation of Common Lisp.

## Theoretical Difficulties

There are a number of theoretical limitations to the algorithms underlying the Series macro package. Some of these are fundamental in that there is no reason to believe that they can ever be relaxed. Others are the subject of continuing research.

**Side effects.** It is believed that the transformations applied when a series expression is converted into a loop are correctness preserving even in the presence of side effects. In addition, since side effects (particularly in the form of input and output) are an inevitable part of programming, several steps have been taken to make the behavior of series expressions containing side effect operations as easy to understand as possible.

First, it should be realized that the order of operations within the body of a non-series `lambda` expression appearing in a series expression (or the body of a `mapping` or `iterate` form) is never changed. Therefore, there is no problem with understanding side effects as long as they are confined to a single such body.

Second, lazy evaluation is used as the foundation for the semantics of series. This means that most of the transformations applied when converting a series expression into a loop leave the order of evaluation of individual series elements unchanged. However, it also means that it can be hard to tell what this order of evaluation will be even when a series expression is not converted into a loop.

Third, the Series macro package tries to ensure that the order of evaluation will follow the apparent syntactic order of the subexpressions in an expression as closely as possible. Nevertheless, you should not depend on a series element $R_i$ being evaluated before another element $S_j$, unless data flow requires $R_i$ to be computed before $S_j$.

Fourth, off-line ports are supported by code motion. This can change the order in which subexpressions are evaluated. However, things are simplified here by ensuring that the evaluation order implied by the order of the inputs of an off-line function is preserved. To make this be the case in a new off-line function being defined, you should make the order of the calls on `next-in` in the `producing` form being used be the same as the order of the corresponding series inputs.

**Multiple values.** In an effort to make series and series functions be as much like other Lisp data objects and functions as possible, the Series macro package supports multiple values in essentially all contexts. As shown above, several of the standard series functions return multiple values. The forms `multiple-value-bind`, `multiple-value-setq`, etc. (but not `multiple-value-call`) can be used to access these values. However, there is one fundamental limitation in the way multiple values are handled.

When a series expression is transformed into a loop, all of the values computed by,

and passed to, series functions are carried by variables. As a result, it is not possible to support the standard Common Lisp feature that multiple values can coexist with single values without the programmer having to pay any attention to what is going on. Rather, the exact number of values returned by every series operation must be known at compile time.

The Series macro package has been designed so that it is always possible to determine at compile time how many values will be returned by a given call on a series function. To start with, you are required to declare the number of values a series function will return when defining the function (see the declaration **optimizable-series-function**). Beyond this, higher-order series functions such as **choose-if** or **scan-fn** either return a number of values that can be computed from the number of arguments or have a type argument that specifies how many values are returned.

A separate issue relating to multiple values is that if you are not careful, it is easy for multiple series return values to end up being off-line. The series macro package cannot make them on-line unless it knows that they are the same length and generated exactly in phase with each other. If this is your intent, you should indicate it by using the function **cotruncate** to return the values. For instance, the following example shows how a simple version of **scan-plist** could be defined.

```
(defun simple-scan-plist (plist)
    (declare (optimizable-series-function 2))
  (let ((entry (scan-fn T #'(lambda () plist) #'cddr #'null)))
    (cotruncate (#Mcar entry) (#Mcadr entry))))
```

**Declarations.** The transformation of series expressions into loops causes two problems with regard to declarations. First, the transformations drastically alter syntactic scopes. As a result, none of the variables bound in a series expression can be declared special, because it would not be possible to make sure that they were bound in the correct scope. In addition, since variable renaming sometimes has to be used, it is not even possible to guarantee that a given variable will be bound at all.

A second problem is that the transformations introduce new variables. For example, when the expression **(collect-sum (scan 'vector v))** is transformed into a loop, four variables are introduced: one acts as an index into the vector, one holds the length of the vector, one holds individual elements of the vector, and one holds the evolving sum. Since none of these variables exists in the original code, none of them can be given declarations in the normal way. This makes it difficult to satisfy the general desiderata that, while type declarations are never required in Lisp, it should be possible to provide a type declaration for every variable. The Series macro package has been designed to ensure that this is in fact possible (though sometimes awkward).

The Series macro package creates appropriate declarations for the variables it introduces, using three mechanisms. First, many variables have a specific type associated with them that is inherent in the way they are used. For example, the index used when scanning a vector is inevitably a fixnum. Second, some variables have types associated with them that follow directly from the types of other variables. For example, the internal variable used to hold the output of **previous** must have the same type as the variable holding the input. These types can be automatically determined by the Series macro

package. Third, other variables have types that must be directly (or indirectly) specified by the user. As discussed below, a variety of methods are provided for specifying what these types are. The situations in which the various methods have to be used are summarized in Figure 4.1. Note that if type information is specified for the scanners in an expression, most transducers and collectors can generate appropriate declarations without any further information from the user.

Normal declarations for explicit variables:
**iterate**, **mapping**, **producing**, **defun**, **let**, etc.

Special type arguments:
**collect**, **collect-append**, **collect-fn**, **collect-sum**, **collecting-fn**, **map-fn**, **scan**, **scan-fn**, **scan-fn-inclusive**, **scan-multiple**, and **scan-range**.

The form **the** required to specify output type:
**#M**, **#Z**, **catenate**, **latch**, **scan-alist**, **scan-file**, **scan-lists-of-lists**, **scan-lists-of-lists-fringe**, **scan-hash**, **scan-plist**, and **series**.

No user action required, all types known a priori:
**collect-alist**, **collect-file**, **collect-hash**, **collect-length**, **collect-nconc**, **collect-plist**, **mask**, **positions**, **scan-sublists**, and **scan-symbols**.

No user action required, types propagated from types of inputs:
**choose**, **choose-if**, **chunk**, **collect-and**, **collect-first**, **collect-last**, **collect-max**, **collect-min**, **collect-nth**, **collect-or**, **cotruncate**, **expand**, **mingle**, **previous**, **split**, **split-if**, **spread**, **subseries**, **until**, and **until-if**.

Figure 4.1: Methods for specifying type information in series expressions.

In situations where explicit variables contain series, the declaration (**series** *type*) can be used to declare the type of the variable. This declaration will carry over when series expressions are converted into loops.

Several series functions have explicit type arguments. The higher order functions **scan-fn**, **collect-fn**, etc. have type arguments that specify the number and kind of outputs produced by their functional arguments. These also specify the types of the internal state variables required. The functions **scan**, **scan-multiple**, **collect**, and **collect-append** have type arguments that specify the type of sequence involved. A type such as (**vector integer**) as opposed to merely **vector** also specifies the type of the variable holding the elements returned by the scanners. It is permissible to use the form (**list** *type*) to specify the type of elements in a list. The functions **scan-range** and **collect-sum** have type arguments that specify the type of value returned.

The form **the** can be used to specify the type of the output of a series function—e.g., (**the** (**series float**) (**scan-file f**)). As shown in Figure 4.1, there are a number of scanners for which this is the only way to specify the type of the variable containing the output. This is also needed in conjunction with the transducers **latch** and **catenate**, and the read macros **#M** and **#Z**. In each case, the variable holding the output elements is given the type **T**, unless more specific information is provided by the user.

The form **the** has another important use in series expressions. Whenever a non-series

argument to a series function is an expression (as opposed to a constant or a variable), a variable is introduced to hold its value. In order for this variable to have an informative type declaration, the argument expression must be wrapped in a **the** form specifying the type of the result. (It should be noted, that the variables introduced in this situation are used in a very simple way—they are assigned in one place and then used in one other place. A quality compiler can produce good code in this situation without needing any declarations.)

Many of the variables introduced by series functions have types that are known in advance. For example, the output of **scan-sublists** is always a series of lists. Similarly, the output of **positions** is always a series of fixnums. Many other series functions have internal variables whose types are known a priori.

A number of transducers and collectors have the property that the type of internal variables and/or the type of the variable holding the output can be determined by looking at the type of the variable holding the input. For example, the elements output by **previous** are the very same elements that are in the input. Therefore, they must have the same type. The Series macro package automatically generates a declaration for the output variable that gives it the same type as the input.

The functions **chunk**, **previous**, **mingle**, **collect-first**, **collect-nth**, and **collect-last** all have the character that the output comes directly from the input. Some of these functions (e.g., **previous** and **collect-last**) take a default argument that sometimes becomes part of the output. It is assumed that the default value will have the same type as the elements of the main input.

A further set of functions (**until**, **choose**, **split**, **subseries**, etc.) have the property that the outputs come from the inputs in such a direct way that it is not necessary to have an output variable separate from the input variable. In these cases, there are no new variables that need to be declared.

Other functions (e.g., **expand**, **collect-max**, **collect-and**, etc.) have output values that primarily come from the the input, but occasionally take on other values (e.g., **nil**). Declarations are automatically created reflecting this fact.

- **series-element-type** *variable* ⇒ *type*

This special form can be used to specify the propagation of type information when a new series function is defined. The *variable* argument must be a variable carrying a series value (e.g., a series argument of a series function). As the last act of creating optimized code for a series expression, the series macro package replaces *every* use of **series-element-type** with the type of the elements in the corresponding series.

The following example shows how the function **collect-last** could be defined. The use of **series-element-type** ensures that the internal variable used to hold the last value will have the correct type.

```
(defun collect-last (items &optional (default nil))
    (declare (optimizable-series-function))
  (collect-fn '(series-element-type items)
              #'(lambda () default)
              #'(lambda (old new) new)
              items))
```

When optimized code is produced for the following series expression, the use of `series-element-type` in `collect-last` is replaced with the type `integer`.

```
(collect-last (scan '(vector integer) v))
```

## Lack of Integration With Common Lisp

One of the great benefits of Lisp is its ability to support complex language extensions with macro packages. No other language would allow a portable implementation of Series to be integrated into the language anywhere near as well. Nevertheless, there are a number of unfortunate limitations on the amount of integration that can be achieved.

**The optimization of series expressions should be a compiler extension.** The most logical way to support the optimization of series expressions would be as a compiler extension. This is what would be done if optimizable series expressions were included directly as part of Common Lisp. Unfortunately, their is no portable way to define such an extension.

In the absence of extending the compiler, the only portable way to support optimized series expressions is as a macro package. For the most part, this works very well. However, it leads to a few problems that would not exist if direct compiler support were provided.

First, many of the series functions actually have to be macros. In particular, while the scanners and transducers really are functions, all of the collectors are macros. This is necessary so that the Series macro package will be activated when a potentially optimizable series expression begins.

In general, the fact that collectors are macros need not concern the user. However, the fact that they are macros does limit their use in some ways. For example, they cannot be used as functional arguments to functions like `mapcar`. In addition, collectors have to be defined before the first time they are used, even if optimization is not applied.

Second, a number of standard Common Lisp special forms and functions (`let`, `let*`, `funcall`, `multiple-value-bind`, and `defun`) have to be shadowed by macros. Again, this is necessary to ensure that the Series macro package will be activated whenever a potentially optimizable series expressions begins. This change is almost totally transparent and works remarkably well. However, it is inherently dangerous to shadow things that are this fundamental to Lisp. For one thing, `funcall` is changed from a function to a macro. Although this is unlikely to be a problem, it is not a correctness preserving change.

A third problem concerns the generation of appropriate initial values. In Common Lisp there is no way to bind a variable without giving it an initial value. In general this is not a problem. However, when the Series macro package wishes to bind the variables it creates, this means that it has to generate appropriate initial values for them. This is done heuristically under the assumption that any variable that is not some kind of number can safely be bound to `nil`.

**Weak support for code analysis.** Common Lisp is quite deficient in its support for the kind of code analysis that has to be done by a complex macro package. Common Lisp provides the function `macroexpand` for dealing with macros and has gratifyingly few special forms in comparison with earlier Lisps. However, it provides little other support

for code analysis. This places a number of limits on what the Series macro package can do.

First, there are a number of complex special forms that cannot be used in a series expression. These include **compiler-let**, **flet**, **labels**, **macrolet**, and any non-standard special forms supported by a given Common Lisp implementation.

Second, the standard Lisp forms **let**, **let***, **multiple-value-bind**, **progn**, **prog1**, **funcall**, **setq**, **multiple-value-call**, **multiple-value-prog1**, **multiple-value-setq**, and **lambda** are all handled. However, for optimization to be possible, some restrictions have to be followed. The only declarations that can be used inside **let**, **let***, and **multiple-value-bind** are type declarations and **ignore** declarations. In particular, none of the variables involved can be declared or proclaimed to be **special**. When using a **lambda** representing a series function that is to be optimized, no "**&**" keywords can be used in the argument list. Further, the only declarations allowed are **type** declarations and **ignore** declarations.

Third, there is no standard Common Lisp method that a macro can use for finding out about **proclaim**ed declarations or declarations that are specified in the scope surrounding the macro call. As a result, the Series macro package cannot make use of a declaration, unless it is syntactically embedded in a series expression.

The future standard version of Common Lisp is slated to contain features that address several of the problems above.

# 5. Bibliography


[1] A. Aho, J. Hopcraft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[2] G. Burke and D. Moon, *Loop Iteration Macro*, MIT/LCS/TM-169, July 1980.

[3] J. Orwant, *Support of Obviously Synchronizable Series Expressions in Pascal*, MIT/AI/WP-312, September 1988.

[4] C. Perdue, W. van Melle, and P. Curtis, Various unpublished manuscripts describing generators and gatherers, 1988.

[5] C. Perdue & R. Waters, "Generators and Gatherers", in *Common Lisp: the Language*, Second Edition, G. Steele jr., Digital Press, Maynard MA, 1989.

[6] R. Polivka and S. Pakin, *APL: The Language and Its Usage*, Prentice-Hall, Englewood Cliffs NJ, 1975.

[7] G. Steele Jr., *Rabbit a Compiler for Scheme*, MIT/AI/TR-474, May 1978.

[8] G. Steele Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[9] R. Waters, *Automatic Analysis of the Logical Structure of Programs*, MIT/AI/TR-492, December 1978.

[10] R. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[11] R. Waters, *LetS: an Expressional Loop Notation*, MIT/AIM-680, Oct. 1982. (Revised February 1983.)

[12] R. Waters, "Expressional Loops", *Proc. 1984 ACM Conf. on the Principles of Programming Languages*, Jan. 1984.

[13] R. Waters, "Efficient Interpretation of Synchronizable Series Expressions" *Proc. ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, ACM SIGPLAN Notices*, 22(7):74–85, July 1987.

[14] R. Waters, *Obviously Synchronizable Series Expressions: Part I: User's Manual for the OSS Macro Package*, MIT/AIM-958, November 1987. (Revised March 1988.)

[15] R. Waters, *Obviously Synchronizable Series Expressions: Part II: Overview of the Theory and Implementation*, MIT/AIM-959, November 1987. (Revised March 1988.)

[16] R. Waters, "Using Obviously Synchronizable Series Expressions Instead of Loops", *Proc. 1988 International Conference on Computer Languages*, 338–346, Miami FL, IEEE Computer Society press, October 1988.

[17] R. Waters, *Optimization of Series Expressions: Part II: Overview of the Theory and Implementation*, MIT/AIM-1083, December 1989.

[18] R. Waters, "Series", in *Common Lisp: the Language*, Second Edition, G. Steele jr., Digital Press, Maynard MA, 1989.

[19] *Lisp Machine Documentation for Genera 7.0*, Symbolics, Cambridge MA, 1986.

# 6. Historical Note

The Series macro package is the last in a line of five related systems developed by the author during the past 12 years. The first of these systems was a macro called `over` (mnemonic for mapping *over* a sequence of values). This macro was written in MacLisp in 1977 and was loosely inspired by looping constructs in Algol-like languages. However, it went well beyond those constructs by supporting a variety of scanners and several collectors (but no transducers other than mapping). In addition, `over` supported powerful pattern matching facilities á là Snobol for decomposing lists.

Except for its pattern matching facilities, `over` was semantically (although not syntactically) quite similar to the `loop` macro [2], which was independently developed at around the same time. However, unlike `loop`, which has been widely distributed and has become the standard in Lisp for this kind of iteration facility, `over` was only used by the author.

The central ideas behind the Series macro package were developed in 1977–78 as part of a PhD thesis on program understanding [9, 10]. It was observed that loops are typically constructed out of standard fragments of looping behavior and that loops can be viewed as functional compositions of operations on *temporal sequences* of values (sequences spread out in time). A program was implement that could analyze loops in terms of fragments communicating via temporal sequences. In addition, although not implemented until much later, an algorithm was designed for the reverse process of *temporal composition.*

Starting in 1979, work began on a macro called `TC` (for Temporal Composition) that made use of the insights described above. This work started out as simple extensions to `over`, but gradually grew to become more general, more powerful, and more focused purely on looping (due to the elimination of most of the pattern matching facilities).

The work on `TC` culminated in the implementation of a MacLisp/LispMachine macro package called `LetS` [11], which was released for general use at the MIT AI Laboratory in October of 1982. The central macro in this package was named `LetS` (for `let` Sequence). However, although named after the form `let`, the semantics of `LetS` was actually somewhat of a cross between the pure semantics of `let` and the semantics of the macro `mapping` described in this document.

The key advance of `LetS` in comparison to things like `over` and `loop` was that it introduced transducers as full fledged components and allowed operations to be combined by simple functional composition rather than special syntax. In addition, it made extensive use of implicit mapping á là APL.

In comparison with Series, `LetS` had three key defects. First, it only supported on-line transducers. (This limit was ameliorated somewhat by introducing a set of special conventions implemented using flag variables that allowed various filtering operations to be supported in an on-line way.) Second, in hindsight, the theory behind `LetS` was rather confused. There was no clear understanding of the distinction between on-line and off-line or how important this concept is. In addition, there was no clear understanding of what has to happen when one part of an expression wants to terminate before other parts are ready to terminate. It was merely stated as a 'feature' that everything stops as soon as anything wants to stop. Third, things were stated half from a loop-like perspective

and half from a functional composition perspective.

A key aspect of the ideas underlying LetS (and the the Series macro package) is that they have nothing to do with the language Lisp *per se*. In early 1984, the ideas behind LetS were presented at the POPL conference [12] in the context of a proposed language extension for Ada. (At this time there was also a general shift in vocabulary from talking about sequences to series, because Common Lisp adopted the term sequence with a different meaning.)

The feedback gained from the presentation at POPL lead to a flurry of ideas for ways to improve LetS. After a delay of several years caused by the need to pursue other areas of research, and a desire to actually prove the correctness of the newly developed ideas, a clear theory emerged for when series expressions can and cannot be transformed into loops (see [15, 17]). Based on this new theory, a new Common Lisp macro package called OSS [13, 14, 15] was implemented. This macro package was released for general use at MIT in late 1987 and distributed by DEC in conjunction with there implementation of Common Lisp. (The name OSS stands for Obviously Synchronizable Series. As discussed in [15], this term is derived from the underlying theory.)

To show the utility of the basic ideas to languages other than Lisp, a demonstration implementation of OSS was produced as an extension to Pascal, during 1988 (see [3, 16]).

In addition to a clearer, better theory, expressed as a set of restrictions on how the macro could be used, and a more purely functional viewpoint, the key advantage of OSS over LetS was that it supported a variety of off-line transducers. In addition, it moved strongly toward supporting constructs that were semantically identical to standard Lisp constructs, rather than just similar. For example, while there was still a form called LetS, this form was now almost exactly the same as let.

One problem with OSS was that, while the system was presented purely from the perspective of composing functions operating on series, series were not first class data objects. Rather, a series could only be used in situations where the surrounding expression could be completely transformed into a loop. Another problem was that although the theory was expressed clearly as a set of restrictions, these restrictions were far from simple. In particular, a restriction was introduced to guarantee that as soon as any part of a series expression was ready to terminate, the whole expression would be ready to terminate. Unfortunately, although the restriction was important for giving OSS a firm basis in theory, the detailed statement of the restriction could only be described as arcane.

During 1988, the iteration subcommittee of the Common Lisp standardization committee (X3J13) held a series of meetings. The goal of these meetings was to decide on a set of improved iteration facilities to be included in Common Lisp. Three proposals were put forward. The first proposal was a slightly cleaned up version of the MacLisp loop macro [2]. Given the wide user community of loop, it was decided that it had to be included in the Common Lisp standard.

The second proposal was OSS. From the beginning, it was clear that there was considerable resistance to OSS, stemming primarily from its complexity. However, large amounts of very useful criticism were obtained. This led to a renewed burst of implementation activity, which culminated in the Series macro package described in this document. Unfortunately, even though it was agreed that the Series macro package answered most of the detailed objections of the subcommittee, it was decided that the Series macro package

was simply too new for inclusion in the Common Lisp standard. However, Guy Steele decided to include a description of the Series macro package in the new edition of his Common Lisp book [18].

The third proposal was developed by C. Perdue and P. Curtis during 1988 and revolved around what they called *generators* and *gatherers*. After considerable discussion, they became persuaded that the generators/gatherers proposal was actually quite similar to the Series macro package. In particular, although the underlying data structures were fundamentally different, most of the functions in the two proposals were redundant. As reflected in this document, a joint proposal was produced that preserved each of the underlying data structures while eliminating all redundancy. Unfortunately, as with Series, it was decided that generators and gatherers were simply too new for inclusion in the Common Lisp standard. However, Guy Steele again decided to include a description in the new edition of his Common Lisp book [5].

There are several key differences between Series and OSS. First, special forms such as `LetS` have been removed in favor of simply using the standard forms. Second, a wholesale change of function names has been introduced to bring things closer in line with standard Common Lisp naming conventions. Third, Series have been made first class objects. This change forced the abandonment of implicit mapping, but is more in line with the basic spirit of Lisp. Fourth, the basic theory has been simplified, most notably by getting rid of the restriction guaranteeing that as soon as any part of a series expression is ready to terminate, the whole expression will be ready to terminate. To do this, the macro package was extended to handle partial termination correctly. Fifth, generators and gatherers are included as an integral part of the Series macro package.

# 7. Warning and Error Messages

To facilitate the debugging of series expressions, this section discusses the way the Series macro package issues warning and error messages, with particular attention to warnings about problems that block optimization. There are three classes of messages. Restriction violation messages identify problems that make it impossible to optimize an expression but do not block its correct evaluation. Warning messages identify minor problems that do not block optimization or evaluation. Error messages describe problems that make it impossible to optimize or evaluate an expression.

All of the messages are printed in the following format. The type of message is indicated by the opening phrase. (The format is shown as it appears on the Symbolics Lisp machine with `*pretty-print*` set to `T`. The format may differ somewhat in other implementations of Common Lisp.)

> {`Restriction violation`|`Warning`|`Error`} *id-number* `in series expression:`
> *containing series expression*
> *detailed message*

For example, consider the following warning message.

```
Error 66 in series expression:
(LET (((X Y) (SCAN-PLIST X)))
  (COLLECT-ALIST X Y))
Malformed binding pair (((X Y) (SCAN-PLIST X))).
```

The first line of each message specifies an identifying number. For restriction violations, this number is useful for looking up further information in the documentation below. The next part of the message shows the complete series expression that contains the problem. This makes it easier to locate the problem in a program. (This part of the print out is omitted if an error occurs after optimization has already been abandoned.) The remainder of the message describes the particular problem in detail. (The variable `*last-series-error*` contains a list of the information that was used to print the most recent restriction violation, warning, or error message.)

Restriction violation messages have identifiers from 1–29. They indicate that the Series macro package has detected a problem that prevents optimization, but which does not prevent correct (non-optimized) evaluation. In response to these problems, the Series macro package abandons any attempt to optimize the series expression in question. This results in a significant loss of efficiency, but does not lead to incorrect results. Each restriction violation is discussed in detail in the following subsection.

Warnings have identifiers from 30–59. They indicate that the Series macro package has located a feature that is often associated with there being a bug in an expression, but which of itself does not block optimization or evaluation.

Error messages have identifiers from 60–89. They indicate conditions that make it impossible to evaluate a series expression correctly. Each time an error is detected, evaluation/compilation is abandoned altogether and no more problems will be detected. It should be noted that many other kinds of errors in series expressions (e.g., using the wrong number of arguments to a series function) are detected directly by the Lisp system and do not have special Series macro package error messages associated with them.

## Restriction Violations

Each message description below begins with a header line containing a schematic rendition of the message. Italics is used to indicate pieces of specific information that are inserted in the message. The number of the message is shown in the left margin at the beginning of the header. For easy reference, the messages are described in numerical order.

- `*suppress-series-warnings*` default value `nil`

If this control variable is set to other than its default value of `nil`, restriction violations still block optimization, however, no warnings are printed.

**Situations that block static analysis.** The following warnings are triggered by situations that block the compile-time analysis of a series expression.

1 `The declaration` *declaration* `blocks optimization.`

This indicates that a declaration has been encountered (e.g., in a `let`, `let*`, `lambda`, `producing`, or `optimizable-series-function defun`) that prevents optimization either because it is beyond the capabilities of the Series macro package to analyze (e.g., `ftype`, `inline`) or which blocks the optimization transformations (e.g., `special`). For example, evaluating the expression below

```
(let ((x (scan '(1 2 3))))  ; Warning 1 signaled.
  (declare (special x))
  (collect-sum x)) ⇒ 6
```

yields the restriction violation message:

```
Restriction violation 1 in series expression:
(LET ((X (SCAN '(1 2 3))))
  (DECLARE (SPECIAL X))
  (COLLECT-SUM X))
The declaration (SPECIAL X) blocks optimization.
```

2 `Non-quoted type` *type*.

This indicates that the type argument to a `scan-fn`, `scan-fn-inclusive`, `map-fn`, `collect-fn`, `collecting-fn`, or `collect-sum` call fails to be a type specifier constant (i.e., `T`, `nil`, or a quoted type specifier list). This prevents static analysis of the series function call. (The type arguments of other series functions, e.g., `scan` and `collect`, do not have to be constants.) For example, evaluating

```
(let ((x '(values)))
  (collect-fn x #'(lambda () 0) #'+ (scan-range :upto 4)))
```

generates the restriction violation message:

```
Restriction violation 2 in series expression:
(COLLECT-FN X #'(LAMBDA NIL 0) #'+ (SCAN-RANGE :UPTO 4))
Non quoted type X.
```

Specific errors are issued if any type argument fails to evaluate to an appropriate type. For instance, the restriction violation message above is immediately followed by the error message below.

```
Error 62
The type (VALUES) specified where at least one value required.
```

3 **M argument** *m* **to CHUNK fails to be a positive integer.**
4 **N argument** *n* **to CHUNK fails to be a positive integer.**

For optimizations to be possible, the *m* and *n* arguments of `chunk` must be positive integer constants. (Errors `63` and `64` are signalled if these arguments do not evaluate to positive integers.)

5 **ALTER applied to a series that is not known at compile time:** *form*.

For optimization to be possible, the *destinations* argument of `alter` must be a series that can be identified at compile time. (Error `65` is signalled if *destinations* does not evaluate to an alterable series.)

6 **The form** *function* **not allowed in series expressions.**

In general, the Series macro package has a sufficient understanding of special forms to handle them correctly when they appear in series expressions. However, it does not handle the forms `compiler-let`, `flet`, `labels`, `macrolet`, or `multiple-value-call`. The forms `compiler-let` and `macrolet` would not be that hard to handle, however it does not seem worth the effort. The forms `flet` and `labels` would be hard to handle, because the Series macro package does not preserve binding scopes and therefore does not have any obvious place to put them in the code it produces. (All four forms can be used by simply wrapping them around entire series expressions rather than putting them in the expressions.)

The form `multiple-value-call` is impractical to support, because it is in general not possible at compile time to determine how many values will be returned by a function. The form `multiple-value-bind` can be used to avoid the need for `multiple-value-call` in most situations.

7 **VALUES returns multiple series:** *form*.

The form `values` can be used anywhere to return multiple non-series values. However, if it is used to return multiple series values it can only be used as the last form of the body of an `optimizable-series-function` `defun`. In other contexts, the Series macro package is not capable of correctly analyzing the data flow involved. (Also not that, as discussed on page 52, using `values` in any context to return multiple series values is quite likely to causes problems by making the series off-line. It is better to use `cotruncate` whenever possible.)

**Situations that allow a series to escape from an expression.** The following warnings are triggered by situations that allow series values to escape from the confines of a single series expression. This blocks optimization because a physical series object has to be created to represent any such series.

10 **Series value returned by** *form*.

This warning indicates that a syntactically complete series expression returns a series as its value.

11 **series value assigned to free variable** *form*.

This indicates that a series value is stored in a variable that is bound outside of the scope of the series expression that creates it.

12 **Series var** *variable* **referenced in nested non-series LAMBDA.**

This indicates that a series value is passed directly into a **lambda** that is an argument of a function like **scan-fn**. This forces a physical series object to be created.

13 **Series to non-series data flow from:** *call* **to:** *call*.

As illustrated below, this warning is issued whenever data flow connects a series output to a non-series input. This indicates that arbitrary computation is being applied to it. A physical series has to be created because the Series macro package cannot analyze the way in which the series is being used. In particular, it may well escape out of the scope of the series expression. For instance, in the following example a series is being stored in a data structure.

```
Restriction violation 13 in series expression:
(LET ((ITEMS (SCAN DATA)))
  (RPLACA X ITEMS)
  (COLLECT ITEMS))
Series value carried to non-series input by data flow from:
(SCAN DATA)
to:
(RPLACA X ITEMS)
```

The message prints out two pieces of code to indicate the source and destination of the data flow in question. The outermost part of the first piece of code shows the function that creates the value in question. The outermost function in the second piece of code shows the function that receives the value. (Entire subexpressions are printed in order to make it easier to locate the functions in question within the series expression as a whole.) If nesting of expressions is used to implement the data flow, then the first piece of code will be part of the second one. (See the example of warning 21 below.)

14 **The optimizable series function input** *variable* **used as a series value by** *form* **and as a non-series value by** *form*.

This indicates that an argument to an **optimizable-series-function defun** is used in two ways that indicate both that it must be a series and that a physical series must be created so that it can be processed arbitrarily.

**Non-straight-line code.** The following warning is issued when a series expression fails to be a straight line computation.

20 **Not straight-line code** *form*.

The warning prints the part of the series expression as a whole that contains the looping or branching form that causes the problem. For example:

```
Restriction violation 20 in series expression:
(LET ((X (SCAN '(1 2 3))))
  (COLLECT-SUM (IF (EVENP (COLLECT-LENGTH X)) X (#M1+ X))))
not straight line code
(IF (EVENP (COLLECT-LENGTH X)) X (#M1+ X))
```

**Problematical constraint cycles.** The most complex warning messages concern constraint cycles that block optimization (see page 12).

```
21 A constraint cycle passes through the non-series output
at the start of the data flow from: call to: call.
22 A constraint cycle passes through the off-line output
at the start of the data flow from: call to: call.
23 A constraint cycle passes through the off-line input
at the end of the data flow from: call to: call.
```

Whenever a constraint cycle passes through a non-series or off-line port, one of these warning messages is used to indicate the problematical port. Like warning 13, the message prints out two pieces of code that indicate a data flow that ends (or starts) on the port in question.

```
Restriction violation 21 in series expression:
(LET ((X (#MSQRT (SCAN 'VECTOR V))))
  (COLLECT-MAX (#M/ X (SERIES (COLLECT-LENGTH X)))))
A constraint cycle passes through the non-series output at the start
of the data flow from:
(COLLECT-LENGTH X)
to:
(SERIES (COLLECT-LENGTH X))
```

It is always possible to eliminate this kind of problem by code copying and/or minor rearrangement of the expression. For example, you could duplicate the computation of the series of square roots of the elements of `v`.

```
(let ((x (#Msqrt (scan 'vector v)))
      (length (collect-length (#msqrt (scan 'vector v)))))
  (collect-max (#M/ x (series length))))
```

However, with a little thought, you can fix things much more efficiently. In particular, the computation of the maximum can be distributed over the computation of the division.

```
(let ((x (#Msqrt (scan 'vector v))))
  (/ (collect-max x) (collect-length x)))
```

Alternatively, the length can be computed without computing the square roots.

```
(let ((x (#Msqrt (scan 'vector v))))
  (collect-max (#M/ x (series (length v)))))
```

**Related warnings.** The following two warnings do not indicate restriction violations. However, they are described here because they are also controlled by the variable `*suppress-series-warnings*`.

28 `Non-series to series data flow from:` *call* `to:` *call*.

This warning is issued when a value that comes from outside of the series expression as a whole is used in such a way that it is clear that it must be a series. This indicates that the value must be a physical series object that escaped from some other series expression (which therefore could not have been optimized). However, the fact that the series is a physical one does not block the optimization of the series expression that uses it. (It should be noted that, like restriction violation 13, the reported type conflict may just be a symptom of some algorithmic problem in the expression.)

29 `Non-terminating series expression:` *expression*.

On the theory that non-terminating loops are seldom desired, the Series macro package checks each loop constructed to see whether it appears capable of terminating. If not, the warning above is issued. The expression printed may well be only a subexpression of the series expression being processed.

A warning message is issued instead of an error message, because the expression may in fact be capable of terminating or the expression might not be intended to terminate. For instance, the warning is useful in the first example below, but not in the second. The form `compiler-let` can be used to bind the control variable `*suppress-series-warnings*` to `T` around such an expression.

```
(collect (series 4))          ; Warning 29 signaled

(block bar                    ; Warning 29 signaled
  (iterate ((x (scan-range :by 10)))
    (if (> x 15) (return-from bar x)))) ⇒ 20

(compiler-let ((*suppress-series-warnings* T))
  (block bar
    (iterate ((x (scan-range :by 10)))
      (if (> x 15) (return-from bar x))))) ⇒ 20
```

# 8. Index of Functions

This section is an index and concise summary of the functions, macros, and variables supported by the Series macro package. The first line of each entry shows the inputs and outputs of the function. The second line shows the page where complete documentation can be found and gives a brief one line description. In the first line of each entry, the following documentation conventions are used.

A plural name (e.g., items) indicates that an input or output is a series.
Underlining indicates that an input or output is off-line.
† indicates that an output is alterable.
‡ indicates that an output is alterable if the corresponding input is alterable.

**#M** *function* ⇒ *series-function*
p. 22 # macro character syntax, creates a series operation that maps *function*.

**#Z** (*item-1* ... *item-n*) ⇒ *items*
p. 16 # macro character syntax, specifies a literal series.

**alter** *destinations items* ⇒ **nil**
p. 37 Alters the values in *destinations* to be *items*.

**catenate** *items-1* ... *items-n* ⇒ *items*
p. 29 Concatenates two or more series end to end.

**choose** *bools* **&optional** *items* ⇒ *chosen-items*‡
p. 27 Selects the elements of *items* corresponding to non-null elements of *bools*.

**choose-if** *pred items* ⇒ *chosen-items*‡
p. 27 Selects the elements of *items* for which *pred* is non-null.

**chunk** *m* {*n*} *items* ⇒ *items-1* ... *items-m*
p. 30 Breaks a series up into chunks of width *m*.

**collect** {*type*} *items* ⇒ *sequence*
p. 31 Combines the elements of a series into the indicated type of sequence.

**collect-alist** *keys values* ⇒ *alist*
p. 32 Combines a series of keys and a series of values together into an alist.

**collect-and** *bools* ⇒ *bool*
p. 34 Computes the **and** of the elements of *bools*.

**collect-append** {*type*} *sequences* ⇒ *sequence*
p. 32 Appends a series of sequences into a single sequence.

**collect-file** *file-name items* **&optional** (*printer* **#'print**) ⇒ **T**
p. 33 Prints the elements of *items* into a file.

**collect-first** *items* **&optional** (*default* **nil**) ⇒ *item*
p. 31 Returns the first element of *items*.

**collect-fn** *type init function sources-1* ... *sources-n* ⇒ *result-1* ... *result-m*
p. 34 Computes a cumulative value by applying *function* to series elements.

**collect-hash** *keys values* **&rest** *option-plist* ⇒ *table*
p. 33 Combines a series of keys and a series of values together into a hash table.

**collect-last** *items* **&optional** (*default* **nil**) ⇒ *item*
p. 31 Returns the last element of *items*.