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 argument 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 [5] 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
(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
  (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 4). In Figure 4, 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.

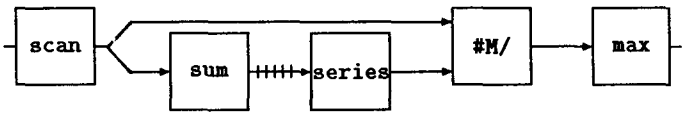

Figure 4: A constraint cycle.

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 4 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 the constraint cycle associated with `scan`, `collect-sum`, `series`, and `#M/` 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 ordered multisets 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 jth 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 in this example, 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 not withstanding, 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. (Like a list, it is also possible

for a series to be side effected in such a way that changes propagate from one use to another. However, this is not the typical way they are used.)

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 jth 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))
```

- `#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 `#(...)`, 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* ((v '#((1 2 3) (3 4 5)))
       (series-of-lists (scan 'vector v))
       (series-of-series (#Mscan series-of-lists)))
  (collect (mapping ((integers series-of-series))
             (collect-sum integers))))  ; Warning
  ⇒ (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 ((v '#((1 2 3) (3 4 5)))
      (series-of-lists (scan 'vector v)))
  (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 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 nil) ⇒ #Z() and #Z()
(scan-alist '((a . 1) () (a . 3) (b . 2)))
  ⇒ #Z(a b) and #Z(1 2)
```

- 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 ... 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 $r_1 \ldots r_m$, 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` $r_1 \ldots r_m$`))`.
The *step* must be of type
`(function (`$r_1 \ldots r_m$`) (values` $r_1 \ldots r_m$`))`.
The *test* (if present) must be of type
`(function (`$r_1 \ldots r_m$`) T)`.
The elements of the *results-i* are computed as follows:

`(values` *results-*$1_0 \ldots$ *results-*m_0`) = (funcall` *init*`)`
`(values` *results-*$1_j \ldots$ *results-*m_j`)`
`= (funcall` *step results-*$1_{(j-1)} \ldots$ *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-*$1_j \ldots$*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 which *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 12/3)) ⇒ #Z(0 4) and #Z(1/4 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.

```
(let ((v '#((1 2 3) (3 4 5)))
  (collect (mapping ((l (scan 'vector v)))
             (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 consists 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(a b) #Z()) ⇒ #Z() and #Z()
(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)
```

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` $r_1 \ldots r_m$`))`.

The *function* must be of type

`(function (`$r_1 \ldots r_m\ s_1 \ldots s_n$`)`
`(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 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 with which to 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))
```

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

This function is a 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)) ⇒ #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)
(expand #Z(nil T nil T T) #Z(a b c))
  ⇒ #Z(nil a nil b c)
```

• `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 jth element of *bools-1* is non-null. In this case, the jth element of *items* will be placed in *items-1*. On the other hand, if the jth 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 jth element of every boolean series is `nil`, then the jth 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 jth 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) #'<) ⇒ #Z(1 3 4 5 7 9)
(mingle #Z(1 7 3 9) #Z(4 5) #'<) ⇒ #Z(1 4 5 7 3 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-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)))))
           (cons prop val))) ⇒ ((a . 2) (b . 5))
```

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 nth 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 nth 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 with a↦3 and b↦2>
(collect-hash #Z(a b) #Z())
  ⇒ <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* ⇒ *number*

Computes the maximum of the elements in *numbers*. These elements must be non-complex numbers, but they need not be integers. The value `nil` is returned if *numbers* has length zero.

```
(collect-max #Z()) ⇒ nil
(collect-max #Z(2 1 4 3)) ⇒ 4
(collect-max #Z(1.2 1.1 1.4 1.3)) ⇒ 1.4
```

- `collect-min` *numbers* ⇒ *number*

Computes the minimum of the elements in *numbers*. These elements must be non-complex numbers, but they need not be integers. The value `nil` is returned if *numbers* has length zero.

```
(collect-min #Z()) ⇒ nil
(collect-min #Z(2 1 4 3)) ⇒ 1
(collect-min #Z(1.2 1.1 1.4 1.3)) ⇒ 1.1
```

- `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.

```
(defun 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 of 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 cannot necessarily be determined 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` 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)
```

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`

This form takes in two series. The destination series is altered so that its elements have the values specified in *items*. More importantly, in the manner of `setf`, the data structure that underlies the destination series is altered so that if the series were to be regenerated, the new values would be observed. This alteration process stops as soon as either input runs out of elements. The function `alter` always returns `nil`.

Consider the example below. Each negative element in a list is replaced with its square. 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 1 -2 3 4 -5 6))
       (x (choose-if #'minusp (scan data))))
  (alter x (#M* x x))
  data)
  ⇒ (1 4 3 4 25 6)
```

Like `setf`, `alter` cannot be applied to just any destination. Rather, `alter` can only be applied to series that are *alterable*. `scan`, `scan-alist`, `scan-multiple`, `scan-plist`, and `scan-lists-of-lists-fringe` produce alterable series. However, the alterability of the output of `scan-lists-of-lists-fringe` is incomplete. If `scan-lists-of-lists-fringe` is applied to an object that is a leaf, altering the output series does not change the object.

In general, the output of a transducer is alterable as long as the elements of the output come directly from the elements of an input that is alterable. In particular, the outputs of `choose`, `choose-if`, `split`, `split-if`, `cotruncate`, `until`, `until-if`, and `subseries` are alterable as long as the corresponding inputs are alterable.

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 jth element of the output series, the alteration is performed by applying *alter-fn* to the new value as its first argument and the jth 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)
```

Generators and Gatherers

Generators and gatherers are yet another way of processing ordered multi-sets. 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 antagonistic. 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, ordered multi-set 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. 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. 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 function 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
```

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 these 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 collects 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 execution of the body must be terminated using the form terminate-producing. 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)
             (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))
  (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
```

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 [6].)

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

⇓

```
(let ((#:index-9 0) (#:limit-8 0) (#:sum-2 0) (x 0))
    (declare (type fixnum #:index-9 #:limit-8)
             (type number #:sum-2)
             (type integer x))
  (tagbody (setq #:index-9 -1)
           (setq #:limit-8 (length v))
           (setq #:sum-2 0)
     #:L-1 (incf #:index-9)
           (if (not (< #:index-9 #:limit-8))
               (go series::end))
           (setq x (aref v #:index-9))
           (setq #:sum-2 (+ #:sum-2 x))
           (go #:L-1)
   series::end)
  #:sum-2)
```

- `*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.