MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 959 November 1987

Synchronizable Series Expressions: Part II: Overview of the Theory and Implementation

by

Richard C. Waters

Abstract

The benefits of programming in a functional style are well known. In particular, algorithms that are expressed as compositions of functions operating on series/vectors/streams of data elements are much easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, many programmers hesitate to use series expressions, because they are typically implemented very inefficiently—the prime source of inefficiency being the creation of intermediate series objects.

A restricted class of series expressions, *obviously synchronizable series expressions*, is defined which can be evaluated very efficiently. At the cost of introducing restrictions which place modest limits on the series expressions which can be written, the restrictions guarantee that the creation of intermediate series objects is never necessary. This makes it possible to automatically convert obviously synchronizable series expressions into highly efficient loops using straightforward algorithms.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

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

Contents

Acknowledgments. Both concept of synchronizable series expressions and this report have benefited from the assistance of a number of people. In particular, C. Rich, A. Meyer, Y. Feldman, D. Chapman, and P. Anagnostopoulos made suggestions which led to a number of very significant improvements in the clarity and power of obviously synchronizable series expressions.

1. Theoretical Overview

The advantages (with respect to conciseness, readability, verifiability, and maintainability) of programs written in a functional style are well known. An example of the clarity of the functional style is provided by the Common Lisp [18] function **sum-sqrts** shown below. This function computes the sum of the square roots of the positive elements of a vector.

```
(defun sum-sqrts (v)
  (reduce #'+ (map 'vector #'sqrt (remove-if-not #'plusp v))))
(sum-sqrts #(4 -4 -16 9)) ⇒ 5
```

The key conceptual feature of **sum-sqrts** is that it makes use of intermediate aggregate data structures to represent the positive elements of the vector and their squares. The use of intermediate aggregates makes it possible to use functional composition to express a wide variety of computations which are usually represented as loops. In various languages, such intermediate quantities can be represented in many different ways—e.g., as lists [18], vectors [15,16,18], sequences [3,18], streams [8,13,31], and flows [17]. At a suitably abstract level, all of these data structures are the same—each of them represents an ordered, linear series of elements. To avoid confusion between the general concept and its various implementations, the term *series* is used here to refer to any data structure having these properties.

Series functions (i.e., functions that operate on series) divide naturally into three classes. *Enumerators* produce series without consuming any. *Reducers* consume series without producing any. *Transducers* compute series from series. In the example above, **reduce** is a general purpose reducer which repetitively applies a function in order to combine the elements of a series together in an accumulator; **map** is a general purpose transducer which produces a new series by applying a function to every element of the input (the indicator **'vector** specifies the type of the output series); and **remove-if-not** is a general purpose transducer which produces a restricted series containing the elements of the input which satisfy a predicate.

Most programming languages support series of one form or another. However, with the notable exception of APL [15], series expressions are not used nearly as often as they could be. An important reason for the lack of use of series expressions is that, as typically implemented, they are extremely inefficient. Since alternate algorithms (e.g., using loops) can often compute the same result much more efficiently, the overhead engendered by using series expressions is quite properly regarded as unacceptable in many situations.

The automatic elimination of intermediate series. The primary source of inefficiency when evaluating series expressions is the creation of intermediate series objects. This requires a significant amount of space overhead per element (to store them) and time overhead per element (to access them). The key to evaluating series expressions efficiently is the realization that it is often possible to transform a series expression into a form where the creation of intermediate series is eliminated. For example, the series expression in **sum-sqrts** can be transformed into the expression in the function **sum-sqrts-one-step**.

In the latter function, the desired value is computed directly from the input vector without creating any intermediate series. (The keyword parameter **:initial-value** specifies an initial accumulator value to be used by **reduce**. The need to use this parameter in **sum-sqrts-one-step** but not in **sum-sqrts** is an idiosyncrasy of **reduce** which is not important in the current context.)

```
(defun sum-sqrts-one-step (v)
  (reduce #'(lambda (result x)
              (if (plusp x) (+ result (sqrt x)) result))
          v :initial-value 0))
```

In **sum-sqrts-one-step**, the three operations to be performed (selecting positive elements, computing square roots, and computing the sum) are performed incrementally in parallel rather then serially. For example, instead of taking a vector of numbers and creating a new vector of square roots, the roots are computed individually as needed. Although the individual elements of the intermediate series are computed, these elements are used as soon as they are created and do not have to be stored in an aggregate data structure. This saves both storage space and accessing time.

After the creation of intermediate series has been eliminated, a further increase in efficiency can be obtained by open coding the resulting expression as a loop. The kind of code which results is illustrated by the program **sum-sqrts-loop** below.

```
(defun sum-sqrts-loop (v)
  (prog (element last index sum)
        (setq index 0)
        (setq last (length v))
        (setq sum 0)
      L (if (not (< index last)) (return sum))
        (setq element (aref v index))
        (if (plusp element) (setq sum (+ (sqrt element) sum)))
        (incf index)
        (go L)))
```

The pragmatic importance of the transformations above is illustrated by the following timing comparisons. Using the Symbolics Lisp Machine, each of the programs above was compiled and then run given the input vector **#(4 -4 -16 9)**. The table shows a significant reduction in running time. In addition, the transformed programs do not use up any non-stack memory space. Looked at from a broader perspective, **sum-sqrts** requires even more running time than shown in the table, because time is eventually required in order to collect the garbage it creates.

Program	Running Time	Garbage Created
sum-sqrts	3.0 milliseconds	8 words
sum-sqrts-one-step	.4 milliseconds	0 words
sum-sqrts-loop	.3 milliseconds	0 words

(In truth, it should be stated that the example above was chosen so that the running time comparison would be dramatic. If processing were done in terms of lists instead of vectors, the overhead would be less and the total speed up would be reduced to a factor

of 3. If processing were done on a long list instead of a short one, the percentage of time devoted to useful computation would be greater and the speed up would be reduced further to a factor of 2. However, using a long input would make the memory allocation comparison even more dramatic. In any event, although a factor of 2 is less dramatic than a factor of 10, it is still significant.)

Many researchers have investigated the automatic elimination of intermediate series (with and without the additional step of transforming series expressions into loops). Some APL compilers [6,7,12] can significantly reduce the number of intermediate arrays required. Wadler's Listless Transformer [19,20] can reduce the number of intermediate lists required during the evaluation of expressions in a Lisp-like language. Bellegarde's transformation system [4,5] can simplify FP [3] expressions reducing the number of intermediate sequences required. The algorithms described by Goldberg and Paige [11] can be used to reduce the number of intermediate data streams required when evaluating data base queries.

Some intermediate series cannot be eliminated. Unfortunately, there is a fundamental problem with the automatic elimination of intermediate series—it is not possible to eliminate every intermediate series. There are two kinds of problems which can block the elimination of an intermediate series. The first problem revolves around individual functions. In order for an intermediate series to be eliminated, it must be possible to create the elements of the series one at a time and use them one at a time in the order they are created. This requirement fundamentally conflicts with the way some functions operate. For example, consider the function `sort`. This function has to have all of the elements of its input simultaneously available before it can start producing its output. As a result, an intermediate series which is passed to `sort` cannot ever be eliminated.

The second kind of problem which can block the elimination of an intermediate series concerns the way functions are connected by data flow. As an example of this, consider the function `normalize` below. This function normalizes a vector by removing all of the non-positive elements and dividing each element by the largest element. The intermediate series `w` cannot be eliminated, because the division process cannot begin until after the value `biggest` has been determined and this value cannot be determined until after all of the elements of `w` are known. The elements in `w` have to be saved in some aggregate structure when computing `biggest` so that they can be used a second time when computing the output vector.

```
(defun normalize (v)
  (let* ((w (remove-if-not #'plusp v))
         (biggest (reduce #'max w)))
    (map 'vector #'(lambda (x) (/ x biggest)) w)))

(normalize #(2 -2 4)) ⇒ #(1/2 1)
```

(It is often possible to eliminate a recalcitrant intermediate series by changing the algorithm being employed. For example, in the function `normalize`, `biggest` could be computed directly from `v` instead of from `w` which would enable `w` to be eliminated. However, such algorithmic changes are beyond the scope of the current discussion. As in the other work cited above, it is assumed that the choice of algorithm should be left to the

programmer and should not be altered by an automatic intermediate series elimination process.)

Since the languages they operate on all allow functions like `sort` and expressions like the one in `normalize`, each of the compilers and transformation systems above only supports the partial elimination of intermediate series. A key task faced by each of these systems is deciding which intermediate series to eliminate. This task is difficult, because deciding to eliminate one intermediate series can make it impossible to eliminate other intermediate series. Goldberg and Paige have shown (see [11]) that, even given a number of simplifying restrictions on the form of a series expression, making an optimal choice of which intermediate series to eliminate is NP-hard.

Although the difficulty of deciding which series to eliminate is a significant problem associated with the systems above, it is not the most serious problem. A greater problem is that the operation of these systems is inscrutable enough that there is no easy way for a programmer to look at a given series expression and determine whether or not all of the intermediate series in it will be eliminated. As a result, programmers are still reluctant to use series expressions, because they cannot depend on these expressions being evaluated efficiently. (The actual creation of just one intermediate series when evaluating a series expression typically leads to unacceptable inefficiency.)

Restrictions guaranteeing the elimination of intermediate series. A solution to the problems engendered by intermediate series which cannot be eliminated is to restrict the kinds of series expressions which are allowed so that the elimination of every intermediate series is guaranteed. This allows programmers to write series expressions without worrying about inefficiency.

The use of restrictions is illustrated by the high level business data processing languages Hibol [17] and Model [16]. The key idea behind of each of these languages is that programs can be written very clearly and compactly if series expressions are used in lieu of loops. Each language supports a restricted class of series expressions which outlaws functions like `sort`. Although this does not rule out the problem exemplified by the function `normalize`, it greatly facilitates the compilation of these languages into efficient code.

However, the use of restrictions has a price—it reduces the range of algorithms which can be conveniently expressed as series expressions. If the restrictions are too severe, much of the value of using series expressions can be lost. For example, as discussed in [24], the restrictions imposed by Hibol and Model are so severe that relatively little of the expressiveness of series expressions remains and the languages are usable only in the context of business data processing. (The design of these languages was motivated by business data processing concerns and no particular attempt was made to support a wide class of series expressions.)

Lazy evaluation and coroutines. A different perspective on the problem of evaluating series expressions efficiently can be obtained by looking at lazy evaluation [10]. The main focus of lazy evaluation is not on eliminating intermediate series but rather on a separate issue. When evaluated serially, series expressions often call for the computation of large numbers of series elements which are not used by the rest of the expression. The demand-driven nature of lazy evaluation insures that an intermediate series element will

not be computed unless it (or some element computed after it) is actually required by a later computational step. This can be extremely beneficial.

However, in straightforward situations such as `sum-sqrts` where every intermediate series element is used, general purpose lazy evaluation is less efficient than ordinary serial evaluation. The problem is that lazy evaluation introduces a new kind of overhead, *scheduling overhead*, without eliminating intermediate series unless they are totally unnecessary.

Scheduling overhead is required in order to determine what to evaluate when. The scheduling enhances the likelihood that an element which is computed can be used immediately in further computation. However, by itself, it does nothing to insure that the element will not be required again at some later time (as in `normalize`). Therefore, just as in serial evaluation, the elements which are computed have to be buffered in aggregate data structures.

The above problems notwithstanding, lazy evaluation embodies two important ideas which are crucial for the efficient evaluation of series expressions. The first idea is the concept of *simulated parallel evaluation*. This approach changes the details of the way an expression is evaluated without changing the basic algorithm being employed. The second idea is using demand driven processing in order to avoid the computation of unnecessary series elements.

The transformation of `sum-sqrts` into `sum-sqrts-one-step` can be viewed as a compile-time application of lazy evaluation. However, the key to the efficiency of the result is that two special conditions are satisfied. First, each use of each intermediate element occurs immediately after the element is initially computed. Second, very little run-time scheduling overhead is required. It is possible to determine at compile time more or less exactly what must be computed when.

Yet another perspective on the problem of evaluating series expressions efficiently can be obtained by looking at coroutines. Using coroutines, series expressions can be expressed as networks of communicating parallel processes [13]. If parallel hardware is used to support these networks, very high efficiency can be obtained. However, when simulated on a serial machine, coroutines have the same fundamental limitations as lazy evaluation. In particular, additional scheduling overhead is introduced, and if no restrictions are placed on the coroutine networks which are allowed, aggregate data structures have to be used to buffer series elements which are transferred across the links in the network.

The above notwithstanding, coroutines embody a third idea which is crucial for the efficient evaluation of series expressions. Coroutines typically communicate via streams (as opposed to other kinds of series data structures) and coroutines are typically required to be *preorder* functions. (A series function is preorder iff it can be evaluated incrementally in such a way that elements of the series output (if any) and each series input are accessed one at a time in ascending order starting with the first element.) This rules out problematical functions like `sort`.

Tree-based restrictions. While the issue of restrictions is implicit in much of the work described above, restrictions have received relatively little direct attention. For example, rather than being explicitly stated, the restrictions imposed by Hibol and Model are merely implicit in the fact that only a small number of specific series functions are

permitted. Similarly, although restrictions (in the form of a compendium of special cases) can be inferred from the situations in which the various compilers and transformation systems above can successfully eliminate intermediate series, these restrictions are in no way explicit.

Essentially the only research to date which directly addresses the question of restrictions is the work of Wadler [21]. His work can be rephrased in terms of the following tree-based restrictions. If every series expression is a tree (i.e., each intermediate value is used in only one place) and every series function is preorder, then every intermediate series can always be eliminated.

Unfortunately, the practical value of the tree-based restrictions is limited, because they are much more restrictive than necessary. In particular, it is often possible to eliminate all of the intermediate series from an expression even if some of the intermediate values are used in several places. As a result, from the point of view of readability and efficiency, it is unreasonable to require that an intermediate value which is used in n places must always be computed n times.

Obviously Synchronizable Series Expressions

To be of significant practical benefit, a set of restrictions must satisfy three conflicting goals: *efficiency*, *permissiveness*, and *obviousness*. It is not good enough for the restrictions to merely guarantee efficient evaluation. They must also permit a usefully large set of series expressions. Further, it must be relatively easy for programmers (and compilers) to check whether or not the restrictions are being obeyed.

It is not possible to completely satisfy these three goals simultaneously. Nor, in all probability, is it possible to develop a set of restrictions which is a provably optimal compromise between the goals. However, satisfying the goals can be approached as an engineering problem which requires one trade-off to be balanced against another.

The primary contribution of the work presented here is a set of restrictions which is a particularly tasteful compromise between the goals. The remainder of this section presents these restrictions and an abbreviated account of the theory underlying them. A longer document is in preparation which gives formal definitions for the terms used and proves many aspects of the necessity and sufficiency of the restrictions.

Synchronizability. The paramount goal is efficiency. The intention is that the series expressions permitted by the restrictions should be so efficient that efficiency considerations will not enter into the decision of whether or not to use series expressions in a given situation.

As discussed above, the key source of inefficiency is the creation of intermediate series data objects. In order to match the efficiency of loops, it is important that permissible series expressions be evaluated in such a way that every intermediate series is *eliminated*. (An intermediate series b is eliminated by an evaluation method iff each individual element of b is transferred directly from the function which computes it to the functions which use it without the need for any intermediate storage.)

The requirement that intermediate series should be eliminated is the single most important driving force behind the restrictions. At first glance, it might seem inappropriate to place so much emphasis on a goal which, on the face of it, only addresses one aspect

of efficiency. However, in the case of series expressions, eliminating the creation of intermediate series promotes time efficiency as well as space efficiency. This is in fortuitous contrast to the more typical situation where storage efficiency can only be gained at the expense of time efficiency.

Every intermediate series in a series expression can be eliminated only if the expression is *synchronizable*. (A series expression is synchronizable iff the series functions in it can be evaluated incrementally using simulated parallel evaluation in such a way that, for each function f which computes an intermediate series b, the following condition holds. For each element b_i in b, every use of b_i occurs before f resumes execution after computing b_i.)

In order for an expression to be synchronizable, it must be the case that series elements are always created and consumed one at a time. In addition, elements must be consumed in the same order they are created—i.e., the processing order at the source and destination of each data flow arc must be the same. If one wanted to be maximally permissive, one could allow different data flow arcs to be associated with different processing orders. However, this would make it relatively hard to check whether or not an expression was synchronizable. It would also lead to significant complications when attempting to evaluate synchronizable series expressions.

Preorder functions. It turns out that a good compromise between permissiveness and obviousness can be obtained by requiring that every series function be preorder. This restriction is easy to check because it does not require global analysis of an expression. In addition, although the restriction rules out large numbers of series expressions which would otherwise be permitted, most of the expressions which are ruled out are of relatively little pragmatic value. This is true because most commonly used series functions can be straightforwardly and efficiently implemented as preorder functions.

The basic trade-off applied above underlies several of the restrictions presented here. Restrictions are chosen so that, given a series expression, it will be *obvious* whether or not the restrictions are satisfied. At the same time, every attempt is made to insure that as many pragmatically useful series expressions as possible are permitted.

The effect of the preorder restriction is more complex than it might appear. Given any series function, it is always possible to define a function computing the same results which operates in a preorder fashion. (At the least, one can merely read the input elements in preorder into a buffer, compute the output elements storing them in another buffer (while this is being done, reading and writing of elements can occur in whatever order is convenient), and then write the output elements in preorder.) Unfortunately, using buffering in this way defeats the whole purpose of trying to eliminate intermediate series.

Fortunately, essentially every enumerator, every reducer, and almost every transducer can be implemented as a preorder function without introducing internal buffering. The only Common Lisp sequence functions which cannot be efficiently supported are `sort` and `reverse`. As a result, the limits implied by the preorder restriction are really quite mild.

Problems caused by parallel data flow paths. If outputs are allowed to be used in more than one place, then it is possible to create parallel data flow paths. (Two data flow paths are parallel if they both originate on the same function and both terminate

on the same function.) Figure 1.1 shows a prototypical example of parallel data flow paths. The data flow path consisting solely of the data flow arc $\delta 1$ from the output of f to the upper input of h is parallel to the data flow path consisting of the data flow arcs $\delta 2$ and $\delta 3$.

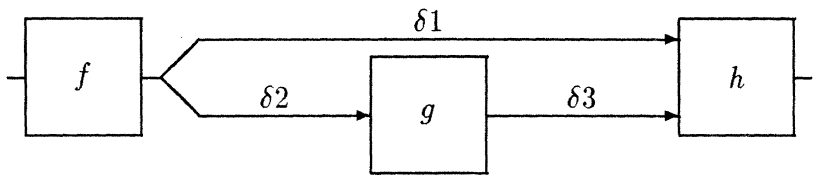

Figure 1.1: Parallel data flow paths in an expression.

If a series expression contains parallel data flow paths, then the expression as a whole cannot be synchronized unless the processing on these paths can be synchronized. There are three prototypical situations in which this is not possible. Each of these situations can be illustrated using an expression analogous to Figure 1.1.

The first situation arises if one of two parallel data flow paths contains a non-series data flow arc while the other does not. For example, suppose that f computes a series value while g does not, as in the function **normalize** used as an illustration above and shown again below. (In **normalize**, f is **remove-if-not**, g is **reduce**, and h is the **map** operation.) The fact that **reduce** cannot produce a value until after it has read all of the elements of its input forces the processing on the two data flow paths out of synchronization. As a result, the intermediate series **w** must be saved between the first and second times it is used and therefore cannot be eliminated.

```
(defun normalize (v)
  (let* ((w (remove-if-not #'plusp v))
         (biggest (reduce #'max w)))
    (map 'vector #'(lambda (x) (/ x biggest)) w)))
```

The second situation arises when one of two parallel data flow paths passes through a transducer where the processing of the relevant input is not synchronized with the processing of the output. Consider the function **moving-average** which computes a vector of moving averages by removing the elements which are not positive and averaging each remaining element with the following element. (In **moving-average**, f is **remove-if-not**, g is **subseq**, and h is the **map** operation.) The function **subseq** is inherently unsynchronized in the way it computes its output from its input. In particular, as used here, the first element of the output cannot be determined until after the second element of the input is available. This means that elements of **w** have to be buffered up on the way from **remove-if-not** to **map**. At any given moment, two elements of **w** always have to be stored (the two elements which are being averaged). As a result, **w** cannot be totally eliminated.

```
(defun moving-average (v)
  (let ((w (remove-if-not #'plusp v)))
    (map 'vector #'(lambda (x y) (/ (+ x y) 2)) w (subseq w 1))))

(moving-average #(2 -4 6 4 8)) ⇒ #(4 5 6)
```

The third situation arises when two parallel data flow paths terminate on a function where the processing of the two relevant inputs is not synchronized. Consider the function **double** which takes a vector, selects the positive elements in it, and creates an output vector by concatenating two copies of the selected elements end to end. (In **double**, f is **remove-if-not**, g is the identity function, and h is **concatenate**.) The function **concatenate** is inherently unsynchronized in the way it uses its inputs. It uses all of the elements of the first input (creating the first part of the output) before using any of the elements of the second input. As a result, the entire intermediate series **w** has to be stored after it is used the first time so that it can be used the second time.

```
(defun double (v)
  (let ((w (remove-if-not #'plusp v)))
    (concatenate 'vector w w)))

(double #(2 -2 3)) ⇒ #(2 3 2 3)
```

The tree-based restrictions avoid the problems above by forbidding parallel data flow paths. However, it is important to realize that many useful series expressions are synchronizable even though they contain parallel data flow paths. For example, the function **average-square** below has three parallel paths between **remove-if-not** and **/** involving both series and non-series data flow and yet every intermediate series can be eliminated as shown in the program **average-square-loop**.

```
(defun average-square (v)
  (let ((w (remove-if-not #'plusp v)))
    (/ (reduce #'+ (map 'vector '* w w)) (length w))))

(average-square #(2 -2 4)) ⇒ 10

(defun average-square-loop (v)
  (prog (element last index sum-squares count)
        (setq index 0)
        (setq last (length v))
        (setq sum-squares 0)
        (setq count 0)
      L (if (not (< index last)) (return (/ sum-squares count)))
        (setq element (aref v index))
        (when (plusp element)
          (incf count)
          (setq sum-squares (+ sum-squares (* element element))))
        (incf index)
        (go L)))
```

The fundamental thing which differentiates **average-square** from the other examples above is that the processing on each pair of parallel paths is synchronizable. It is possible to insure that this will always be the case by essentially forbidding the three problem situations above.

Isolation of non-series data flow. The first problem situation can be ruled out by requiring that every non-series data flow be isolated. (A data flow arc δ in an expression X is isolated iff it is possible to partition the functions in X into two parts Y and $\overline{Y}$ in such a way that: δ goes from Y to $\overline{Y}$, there is no series data flow from Y to $\overline{Y}$, and

there is no data flow from $\overline{Y}$ to Y.) If every non-series data flow arc in an expression is isolated, then if a data flow path contains a non-series data flow arc, every parallel data flow path must also contain a non-series data flow arc.

For example, in the program **normalize**, there is no way to partition the program graph so that the output of **reduce** crosses the boundary without the boundary also being crossed by the output of **remove-if-not**. In contrast, it is easy to partition the program average-square so that the output of **reduce** is the only data flow which crosses the boundary. The boundary is placed to that **/** and **length** are on one side and everything else is on the other.

The isolation restriction above can be used as the basis for a divide and conquer approach to the synchronous evaluation of series expressions. If a permissible series expression contains a non-series data flow arc, then it is always possible to divide the expression into two parts so that all of the data flow arcs between the parts are non-series arcs. As a result, the expression as a whole can be evaluated synchronously, by evaluating the first part synchronously and then evaluating the second part synchronously using the non-series values computed by the first part. This reduces the problem of synchronously evaluating series expressions to the problem of synchronously evaluating series expressions where every data flow is a series data flow.

The non-series isolation restriction places limits on what series expressions can be written in an interesting way. In theory, the restriction does not rule out any computations, because any series expression can be converted into an expression which obeys the restriction by duplicating subexpressions in order to eliminate problematical parallel data flow paths. (From this perspective, the restriction can be looked at as requiring programmers to state explicitly where they wish code copying to occur rather than having the intermediate series elimination process automatically introduce code copying.)

However, in practice, the restriction places relatively strong limits on the way non-series values can be used. In particular, it leans toward expressions where non-series values are the end result of the computation rather than intermediate values. This is a very common situation, but as can be seen in the function **normalize**, it is not the only situation.

An indirect consequence of the isolation restriction is that it limits the functions which it is useful to use. For example, in most of the places where one would typically use **reduce**, a final value is being computed. Therefore, as a pragmatic matter, the non-series isolation restriction does not significantly inhibit the use of **reduce**. In contrast, consider the function **elt** which is used to extract an individual element from a series. This function is typically used in the midst of an expression rather than to compute a final value. Inasmuch as this is the case, the restriction effectively prohibits the use of **elt**. Put another way, the restrictions are intended to apply to series expressions where series are computed and consumed as complete units, rather than to expressions which process individual series elements in complex orders.

Synchronous ports. In order to rule out the other two problem situations, one has to develop a vocabulary for talking about the extent to which the processing at a pair of ports is synchronized. In this regard, it is important to note that many common series functions are inherently synchronized in the way they operate. In particular, consider the function **map**.

A key feature of **map** is that the ith output element is computed solely form the ith input elements. As a result, it is easy to evaluate **map** so that the inputs and outputs are processed in lock step. To do this, processing proceeds by reading the first element of each input, writing the first element of the output, reading the second element of each input, writing the second element of the output, and so on.

For **map**, evaluation in lock step is not just possible, it is mandatory if high efficiency is desired. The problem is that any other evaluation pattern requires the use of extra internal storage space within **map**. (The ith output cannot be written before the ith inputs are read and if later inputs are read before the ith output is written, these inputs have to be stored until they are used later.)

It turns out that **map** is by far the most commonly used series function. In addition, there are several other common series functions which inherently require the same kind of processing. As a result, it is not unreasonable (and essentially mandatory) to promulgate restrictions which militate in favor of the kind of lock step processing described above. To formalize the notion of lock step processing, the following definitions are introduced.

Each preorder series function is assumed to have a canonical processing pattern associated with it. This pattern divides the function's evaluation up into a sequence of intervals. It is expected that the processing pattern will be data dependent in that it is different for different sets of input values.

A series port of a preorder function is *synchronous* iff the ith element of the series read (or written) through the port is always accessed only in the ith interval of the canonical processing pattern for the function. (Synchronous processing is defined for individual ports rather than entire functions in order to allow for functions where only some of the ports are synchronous.)

It is easy to select canonical processing patterns for **map** such that every port is synchronous. In contrast, it is not possible for the input and output of `remove-if-not` to both be synchronous. The problem is that as soon as an input element is omitted from the output, the output processing falls permanently out of step with the input processing.

The importance of synchronous processing can be summarized as follows. If an input and output of a preorder function are both synchronous, then the processing at these ports is tightly synchronized and cannot lead to the second problem situation discussed above. Similarly, if two inputs of a preorder function are both synchronous, then they cannot lead to the third problem situation above.

Isolation of non-synchronous ports. A port which is non-synchronous is unlikely to be synchronized with any other port. Such a port should not be allowed to appear in the middle or at the end of a pair of parallel data flow paths. Non-synchronous ports can be kept out of parallel data flow paths by requiring that they be *isolated*. (An output p in an expression X is isolated iff X can be partitioned into two parts Y and $\overline{Y}$ such that: every data flow originating on p goes from Y to $\overline{Y}$, every other data flow from Y to $\overline{Y}$ is a non-series data flow, and there is no data flow from $\overline{Y}$ to Y. An input q in an expression X is isolated iff X can be partitioned into two parts Y and $\overline{Y}$ such that: the data flow terminating on q goes from Y to $\overline{Y}$, every other data flow from Y to $\overline{Y}$ is a non-series data flow, and there is no data flow from $\overline{Y}$ to Y.)

Like the preorder restriction, the non-synchronous isolation restriction is a compromise between permissiveness and obviousness. The restriction is stronger than it needs

to be. In particular, it is possible for a non-synchronous port to be in the middle of a parallel data flow path if the other parallel data flow path contains one or more non-synchronous ports which together have the same net desynchronization. However, this is of relatively little value and would be difficult for programmers to check and difficult to support efficiently when using simulated parallel evaluation.

The non-synchronous isolation restriction continues the divide and conquer approach of the non-series data flow isolation restriction. Once partitioning based on non-series data flow has been applied, one is left with subexpressions where every data flow carries a series value. If such a subexpression of a permissible series expression contains a non-synchronous port, then it is always possible to divide the subexpression into two parts so that all of the data flow between the parts originates (or terminates) on the port in question. As a result, the expression as a whole can be evaluated synchronously by evaluating the two parts synchronously in isolation from each other and using a simplified form of lazy evaluation in order to decide which subexpression to evaluate when. The lazy evaluation is simplified because the scheduling method is very simple. The first part needs to be evaluated when (and only when) the second part needs to read a new value computed by it.

Together, the two isolation restrictions make it possible to reduce the problem of synchronously evaluating series expressions to the problem of synchronously evaluating series expressions where every data flow is a series data flow connecting synchronous ports. When this is the case, every function in the expression is analogous to **map**. No matter what the pattern of data flow is, global synchronization can be achieved by evaluating everything in lock step.

It is interesting to consider the exact way in which the non-synchronous isolation restriction places limits on what series expressions can be written. As with the non-series isolation restriction, any expression can be modified so as to satisfy the restriction by duplicating subexpressions. Nevertheless, the restriction definitely encourages the use of functions that have synchronous ports. As a result, it is important to realize that most common series functions are inherently synchronous. To start with, essentially every enumerator and reducer can easily be implemented so that all of its ports are synchronous. This is also true of approximately half of all common transducers. The exceptions include the function **remove-if-not**. One must take care in the way they are used in an expression.

On-line ports. An important subsidiary benefit of the restrictions outlined above is that they make it possible to keep most kinds of run-time scheduling overhead to a very low level. However, there is one kind of scheduling overhead which remains—dealing with the fact that the whole expression might not terminate at the same time. In the program **sum-sqrts-loop**, there is no problem in this regard. There is only one termination test and the whole program stops as soon as the termination test succeeds. To understand why things work out this way, it is necessary to look more closely at the function **map**.

In addition to being inherently synchronous in the way it operates, **map** has a key additional property. As soon as any input runs out of elements, **map** immediately stops writing output elements. This property is formalized by saying that every series input of **map** is a *passive terminator*. (A series input port q of a preorder function f is a passive terminator iff f satisfies the following property. Consider the canonical processing pattern

for f and suppose that f tries to use the ith element of the series being read through q during the jth interval of the processing pattern. If this element is beyond the end of the series, then it must be the case that f immediately terminates without producing any output elements during the jth processing interval.) This definition is phrased in terms of individual ports in order to allow for functions where some inputs are passive terminators while others are not. If all of the series inputs of a function are passive terminators, then the function as a whole is said to be a passive terminator.

Far from being an unusual situation, passive termination is very common. A wide variety of functions other than `map` have series inputs which are passive terminators. For instance, even though the input of `remove-if-not` is not synchronous, it is a passive terminator.

The term *on-line* is used to refer to inputs which are both synchronous and passive terminators. (As a matter of convenience, synchronous outputs are referred to as on-line as well. Ports which are either not synchronous or not passive terminators are referred to as *off-line*.) Preorder functions all of whose ports are on-line are themselves traditionally referred to as on-line [1].

As an example of an input which is synchronous and yet not a passive terminator, consider the function `concatenate`. When the first input of `concatenate` runs out of elements, the function does not terminate, but rather continues on, reading elements from the second input. As a result, the first input fails to be a passive terminator. Nevertheless, the first input of `concatenate` is synchronous, since the ith input element maps directly into the ith output element.

Isolation of off-line ports. Returning to the function `sum-sqrts`, the fact that every series function used in `sum-sqrts` is a passive terminator gives `sum-sqrts` the following very convenient, all or nothing termination property.

```
(defun sum-sqrts (v)
  (reduce #'+ (map 'vector #'sqrt (remove-if-not #'plusp v))))
```

Suppose that lazy evaluation is being used to compute the result of `sum-sqrts` and consider what happens at the moment when `remove-if-not` first tries to access an element which is beyond the end of `v`. At this moment, it must be the case that `remove-if-not` is being run to produce an element `map` is waiting to use so that it can produce a value `reduce` is waiting to use. Since `v` has just run out of elements, `remove-if-not` cannot produce any more elements which means that `map` cannot read any more elements which means that `map` cannot produce any more elements which means that `reduce` cannot read any more elements which means that the current value in the accumulator being used by `reduce` is the final value which should be returned by `reduce`. Put another way, as soon as `v` runs out of elements all of the computation can immediately stop. There is no need to consult the functions in the expression individually and no possibility that some functions will have to continue running after others have stopped.

As an example of the complexity which can be introduced by a port which is not a passive terminator, consider the function `split` below. This function takes a vector and returns a vector where all of the positive elements are in the front.

```
(defun split (v)
  (concatenate 'vector (remove-if-not #'plusp v)
                       (remove-if #'plusp v)))
(split #(1 -2 3 -4)) ⇒ #(1 3 -2 -4)
```

Suppose that lazy evaluation is being used to compute the result of **split** and consider what happens at the moment when **remove-if-not** first tries to access an element which is beyond the end of **v**. At this moment, **concatenate** will have finished creating the first part of its output, but will not have begun using the elements of its second input in order to construct the latter part of its output. As a result, when **remove-if-not** terminates, the computation involving **remove-if** and **concatenate** must continue. Only after **remove-if** terminates can **concatenate** terminate.

The fact that the series expression in **split** does not stop all at once, but rather only one part at a time, introduces a certain amount of run-time scheduling overhead. However, this overhead is relatively small as long as any non-passive inputs are isolated (as in **split**). Following the divide and conquer approach discussed above, the control over partial termination can be provided as part of the simplified lazy evaluation which decides which subexpression should be evaluated when.

In order to insure that things will always be this straightforward, the non-synchronous isolation restriction is strengthened to require that every off-line port must be isolated. Pragmatically speaking, this is a mild restriction, because it is not clear that there are any useful OSS functions other than **concatenate** which have inputs that are non-passive and yet synchronous.

On-line subexpressions. If an OSS expression obeys the revised isolation restrictions above, then it can be repeatedly partitioned until all of the data flow in each subexpression goes from an on-line output to an on-line input. The subexpressions which remain after partitioning are referred to as *on-line subexpressions.*

Data flow paths between termination points and outputs. In actuality, the all or nothing termination property of **sum-sqrts** does not stem solely from the fact that all the functions in the expression are passive terminators. To see this, consider the function **weighted-squares** below. This function takes in a vector of values and a vector of weights. It returns a list of two vectors: the squares of the values and the squares multiplied by the weights. Even though all of the functions in **weighted-squares** are passive terminators, the expression as a whole is not guaranteed to terminate in an all or none way.

```
(defun weighted-squares (values weights)
  (let* ((squares (map 'vector #'* values values))
         (weighted-squares (map 'vector #'* squares weights)))
    (list squares weighted-squares)))
(weighted-squares #(1 2 3) #(10 20)) ⇒ (#(1 4 9) #(10 80))
```

If **weighted-squares** is called with two vectors where **weights** is shorter than **values**, **squares** will be the same length as **values**, and **weighted-squares** will be the same length as **weights**. This is problematical, because when **weights** runs out of elements, the

evaluation of `squares` must continue even though the computation of `weighted-squares` must stop. This partial termination introduces significant run-time scheduling overhead.

The key difference between `sum-sqrts` and `weighted-squares` is the relation of the *termination points* to the outputs. (The concept of a termination point is defined relative to the on-line subexpressions which result from partitioning based on the isolation restrictions above. A termination point is a function within an on-line subexpression which can, by itself, cause termination. The primary reason for a function being a termination point is that it reads a series which is computed by a different on-line subexpression. The length of the series read controls the termination of the function.)

Given the way the partitioning is performed, every function in an on-line subexpression which is not a termination point must be a passive terminator which receives all of its series inputs from other functions in the same on-line subexpression. Each of these functions must terminate as soon as any function feeding an OSS value to them terminates. As a result, if within an on-line subexpression, there is a data flow path from each termination point to each output, every function computing an output of the subexpression must immediate terminate as soon as any of the termination points terminates. (The problem in `weighted-squares` is that there is no data flow path from `weights` to `squares` which is why `squares` can be longer than `weights`.)

Early termination. A final wrinkle revolves around the notion of *early termination*. This term is introduced in order to refer to inputs which cause termination strictly more easily than required by the definition of a passive terminator. If any of the inputs of a function is an early terminator, then the function as a whole is said to be an early terminator. In addition, a function that does not have any series inputs is an early terminator iff it is capable of terminating. That is to say, enumerators which produce bounded outputs are early terminators.

As an example of early termination, consider a function which reads in a series and returns all of the elements of that series up to but not including the first element which is zero. The input of this function is an early terminator because the function can terminate before the input runs out of elements. However, the input is also a passive terminator, because the function will immediately terminate if the input does run out of elements.

In an on-line subexpression, early terminators are also considered to be termination points. Enumerators act like series inputs. Other early terminators are capable of prematurely stopping the computation in an expression.

Given this extended definition, all of the functions in an on-line subexpression which are not termination points must be purely passive terminators. This means that none of them can terminate until after some other function in the subexpression has terminated.

As a result, if within an on-line subexpression, there is a data flow path from each termination point to each output, every function computing an output of the subexpression must immediate terminate as soon as any function in the subexpression terminates. This implies that the on-line subexpression has the same all or nothing termination property as `sum-sqrts`.

Requiring all or nothing termination. It turns out that there actually are not very many situations where one would want to write a series expression like the one in `weighted-squares`. As a result, it is of significant pragmatic benefit for the restrictions

to outlaw this kind of expression altogether by requiring that within each on-line subexpression of a permissible series expression there must be a data flow path from each termination point to each output. Most common series expressions trivially obey this restriction, because they only compute a single value and the entire expression contributes to the computation of this value.

Together with the isolation restrictions, the restriction above insures that when the divide and conquer approach outlined above is applied, the indivisible subexpressions which remain will all have the all or nothing termination property possessed by `sum-sqrts`. The only place where partial termination has to be supported is when isolated expressions communicate through non-passive inputs. This introduces only a very small amount of run-time scheduling overhead.

(Returning for a moment to a consideration of `weighted-squares`, the programmer might well have intended that `v` and `weights` should always have the same length. If this is the case, then the termination problems discussed above will not arise. Using the OSS function `Tcotruncate` (see [27]), it is easy to write `weighted-squares` in a way that makes this intention clear and that does not violate any of the restrictions proposed above.)

Straight-line computation. An issue which is implicit in the discussion above is exactly what is meant by the term "expression". In the interest of efficient simulated parallel evaluation, it turns out to be important to restrict series expressions to being *straight-line* computations. (A computation is straight-line if it does not contain any control constructs such as loops or conditionals.)

From the point of view of programming languages such as Lisp, outlawing conditionals in series expressions may seem overly restrictive. However, many (if not most) conditionals one might want to include in a series expression are either embedded in a non-series function which is mapped over series elements or can easily be replaced by selection operations such as `remove-if-not`.

Defining a new series data type. Any reasonable set of restrictions is forced to prohibit some useful series expressions which are traditionally permitted. Although there are important reasons for prohibiting these expressions, it would not be reasonable to ban these from a programming language altogether. As a result, it would not be reasonable to apply the restrictions to a preexisting series data type. Rather, a new type should be defined. This allows programmers to benefit from the restrictions when they choose to follow them, without being prevented from using vectors, sequences, and streams in standard ways.

Defining a new data type has the added virtue of making it possible to pursue the elimination of series to its logical extreme. If instances of this new data type are prohibited from being used as components of data structures, then it is possible to guarantee that no storage will ever be required for any instance of this new data type.

In addition to saving space, the restriction that instances of the new series data type cannot be contained in other data structures promotes obviousness and simplicity in general. (The languages Hibol and Model show that this restriction can be relaxed by allowing series to contain series. However, they also show that this can only be done at the cost of significant user-visible complexity and that, as a practical matter, there is relatively little to be gained.)

Static identifiability. Much of the discussion above revolves around the question of how one can guarantee that it will be possible to transform permissible series expressions so that every intermediate series is eliminated. However, it leaves open the question of exactly how this transformation will be performed. In the interest of overall efficiency, it is critical that this transformation be applied at compile-time rather than run-time. In order for this to be the case, every series function and variable must be *statically identifiable.* (A series variable is statically identifiable if it is possible to tell, before evaluation begins, whether or not the variable holds a series. A series function is statically identifiable if it is possible to tell, before evaluation begins, for each input (and the output) whether or not a series will be read (or written). In addition, it must be possible to tell exactly what computation will be performed.)

With regard to series variables the restriction above is a weakened form of strong typing. However, with regard to series functions, the restriction is more stringent than strong typing. Rather than merely requiring that the type of every series function be identified at compile time, it requires that the exact computation to be performed be known. This rules out the possibility of using series functions which are passed as parameters or otherwise computed at run-time.

The restrictions. The discussion above can be summarized in the form of the following set of restrictions. Let *obviously synchronizable series* be a new series data type (hereinafter referred to as OSS series). A function is an OSS function iff it either takes in an OSS series or produces one. A variable is an OSS variable iff it holds an OSS series. An expression X is an OSS expression iff (1) X is a syntactic unit in the language (e.g., an expression or a statement); (2) the outermost part of X is an OSS operation (e.g., an OSS function or a form that binds an OSS variable); and (3) X does not take in or produce OSS series. (i.e., X is complete in the sense that is is not merely a subexpression of some larger OSS expression.)

The definitions above define the term OSS expression syntactically without placing any limits on what can be written. The following seven restrictions guarantee that every OSS expression can be evaluated efficiently.

1) OSS series must not be contained in data structures.
2) OSS functions and variables must be statically identifiable.
3) OSS functions must be preorder.
4) OSS expressions must be straight-line programs.
5) Non-OSS data flows must be isolated.
6) Off-line OSS ports must be isolated.
7) Within each on-line subexpression of an OSS expression, there must be a data flow path from each termination point to each output.

White lies and other simplifications. In the interest of clarity and conciseness, the presentation above relies on a number of simplifications. For example, functions are assumed to have only one output. (The restrictions are worded so that they cover the case of multiple outputs.) This issue and a number of other issues (which it is hoped

have passed by without troubling the reader) will be discussed in detail in a subsequent document.

Language Issues

The concept of OSS expressions can be used as the basis for a programming language preprocessor which takes expressions like the one in the function `sum-sqrts` and automatically transforms them into loops like the one in `sum-sqrts-loop`. As described at length in [27], the OSS macro package adds this support to Common Lisp. However, the concept of OSS expressions is not limited in any way to Lisp. Similar support could be added to essentially any programming language.

Supporting OSS expressions in harmony with a language. In order to add support for OSS expressions to a programming language, one has to do three things. First, an OSS series data type and a set of predefined OSS functions have to be added into the language. Second, provisions have to be made for seeing that the restrictions are obeyed. Third, a preprocessor has to be implemented which converts OSS expressions into a form where they can be efficiently executed.

The key user-visible part of introducing support for OSS expressions is the introduction of the OSS data type and predefined OSS functions. For this extension to fit in harmoniously, it needs to be done in a way which is consistent with the syntax of the language. In general, if a language allows the definition of new data types and functions with functional arguments, then OSS expressions can be supported without making any syntactic extensions to the language itself. (Otherwise, some apparent syntactic extensions will be necessary. However, since OSS expressions are supported by a preprocessor, actual syntactic extensions are never required.)

The OSS data type can be defined in analogy with vectors. When defining this new data type, one should be sure to avoid making the mistake of specifying the length of an OSS series as part of its type. (Since OSS series do not require any physical storage this would limit the ability of OSS functions to operate on OSS series of arbitrary length without having any counterbalancing benefits.)

Once an OSS series data type has been defined, OSS variables, functions and expressions can be written using exactly the same syntax as ordinary variables, functions and expressions.

Predefined functions. Just as a set of basic functions and operations must be provided for any other data type, a set of basic OSS functions must be provided. To be most useful, these functions should include a number of specific higher-order functions. The list of predefined functions in [27] is a useful guide to the kinds of functions which can and should be provided when supporting OSS expressions.

As discussed in Section 2, the predefined functions provided by the OSS macro package combine almost all of the functionality of the Common Lisp sequence functions [18], the Loop macro [9], and the vector operations of APL [15]. They also include most of the functionality of Barstow's stream operations [8]. In addition, the macro package supports a few complex higher-order functions which are not supported by any preexisting system.

One way to look at OSS expressions in general is that they make it possible to introduce 90% of the expressive power of the vector operations of APL into a standard programming

language without introducing an inscrutable syntax or incurring run-time overhead.

Another way to view OSS expressions is that they are a logical continuation of the trend in programming language design toward supporting the reuse of loop fragments. (This is, in fact, the perspective from which they were originally developed.) From this point of view, the concept of an enumerator extends the approach taken by iterators in CLU [14] and generators in Alphard [29].

Implicit mapping. As supported by the OSS macro package, OSS expression include one feature which is essentially orthogonal to the restrictions discussed above. Whenever an ordinary Lisp function is applied to an OSS series, it is automatically mapped over the elements of the OSS series. For example, in the expression below, the function **sqrt** is mapped over the OSS series of numbers created by **Evector**.

```
(Rsum (sqrt (Evector #(4 16))))
  ≡ (Rsum (TmapF #'sqrt (Evector #(4 16)))) ⇒ 6
```

This concept is borrowed from APL, and due to the ubiquitous nature of mapping is extremely useful in OSS expressions just as it is in APL. However, implicit mapping runs somewhat counter to many programming languages. The problem is that implicit mapping allows programmers to write programs which appear to violate strong typing. This is only an apparent violation since the preprocessor produces output which does obey strong typing. Nevertheless, introducing implicit mapping goes beyond the simple idea of merely adding a new data type and a few functions.

Enforcing the restrictions. It turns out that enforcing the OSS restrictions is a straightforward matter. To start with, it is relatively easy to arrange things so that it is impossible for a programmer to construct an OSS expression which violates any of the first four restrictions. Consider the OSS macro package as a specific example. The OSS macro package does not provide any functions which can cause an OSS series to become part of a data structure. The requirement that OSS variables and functions must be statically identifiable is supported by two special forms **letS** and **defunS**. (This would require no additional support in a language that had strong typing.) Finally, there is no way to create an OSS function which is not preorder or an OSS expression which is not a straight-line program.

Implicit mapping plays an important role in the enforcement of the first and fourth restrictions, by providing an alternate interpretation for expressions which might appear to violate one of these restrictions. For example, one might think that the function below would return a cons cell containing a series. However, this is not the case. Implicit mapping causes the **cons** operation to be applied to the individual elements of the series rather than to the series as a whole. As a result, the function returns an OSS series of conses rather than a cons of an OSS series.

```
(defunS map-cons (n)
  (cons (Eup 1 :to n) nil))
(map-cons 3) ⇒ [(1) (2) (3)]
```

The isolation restrictions and the requirement that within each on-line subexpression, there must be a data flow path from each termination point to each output are the

only restrictions that a programmer is capable of violating. However, it is easy for the OSS macro package to check and see that these restrictions are satisfied by every OSS expression. The best approach for programmers to take is to simply write OSS expressions without worrying about these restrictions and then fix the expressions in the unlikely event that the restrictions are violated. In particular, note that an expression which does not contain any `letS` variables cannot violate any of these restrictions.

Benefits. The benefit of OSS expressions is that they retain most of the advantages of functional programming using series, while eliminating the costs. However, given the restrictions described above, the question naturally arises as to whether OSS expressions are applicable in a wide enough range of situations to be of real pragmatic benefit.

An informal study [22] was undertaken of the kinds of loops programmers actually write. This study suggests that approximately 80% of the loops programmers write are constructed by combining a few common kinds of looping algorithms. The OSS macro package is designed so that all of these algorithms can be represented as OSS functions. As a result, it appears that approximately 80% of loops can be trivially rewritten as OSS expressions. Many more can be converted to this form with only minor modification.

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

2. Comparisons

The following sections compare and contrast the functionality supported by OSS expressions in general, and the OSS macro package (see [27]) in particular, with the functionality supported by the Common Lisp sequence functions, the Loop macro, and APL. In each case, detailed examples are given illustrating the way each individual aspect is supported. (It is assumed throughout that the reader has read [27].)

There are two ways in which these sections can be used. On the one hand, they are useful for showing how OSS expressions measure up to the standards offered by other things. On the other hand, for people who happen to have a good understanding of either sequence functions, the Loop macro, or APL, the comparisons are a useful way to increase their understanding of OSS expressions.

Sequence Functions

This section shows how the predefined OSS functions capture the functionality of the Common Lisp sequence functions. Although not strictly necessary, a good understanding of the sequence functions as described in Chapter 14 of [18] will contribute significantly to an understanding of the discussion below which follows the outline of that chapter.

The OSS functions support essentially all of operations of the sequence functions except **reverse**, **nreverse**, **sort**, and **stable-sort**. These functions are ruled out because there is no way to implement them efficiently as preorder functions (see Section 1). (In a similar vein, the keyword **:from-end** is in general not supported.) To apply these operations to an OSS series, reduce the series to a list or vector, apply the function in question, and then enumerate the result.

An additional group of sequence functions are not relevant to OSS series. Since OSS series do not have a physical existence, it is not possible to side-effect them. As a result, there is no need to have side-effect variants of OSS functions. This eliminates the need for OSS functions analogous to **delete**, **delete-duplicates**, **fill**, and **nsubstitute**. Further, **setf** cannot be applied to OSS functions. The lack of side-effects also eliminates the need for an OSS function analogous to **copy-seq**. Since side-effects are not possible, there is no need to copy an OSS series to protect it from modification.

Increased orthogonality. Many of the sequence functions exist as entire families of related versions. Some of these versions are indicated by the suffixes **-if** and **-if-not**. Other versions are indicated by the use of a variety of keyword parameters. The corresponding OSS functions need only have two versions: one which does not take a functional argument and one (ending in "F") which does. In order to understand why OSS functions do not need to have so many versions, one has to understand why the sequence functions need to have so many versions.

The need for multiple versions of sequence functions stems from efficiency considerations. For example, many sequence functions support the keywords **:from** and **:end** which can be used to specify a subsequence of the input the function in question operates on. As illustrated below, these keywords are totally redundant in meaning with the sequence function **subseq**. However, the keyword form is more efficient, because it eliminates the

creation of an intermediate series.

```
(reduce #'+ '(0 1 2 3 4) :start 2 :end 4)
  ≡ (reduce #'+ (subseq '(0 1 2 3 4) 2 4)) ⇒ 5
```

Since the OSS macro package eliminates every intermediate series, there is no need to complicate **ReduceF** by including the functionality of **subseq**. This functionality is supported by the separate OSS function **Tsubseries**. Keeping the two functionalities separate allows them to be used with maximal clarity and generality.

A second reason why multiple versions of sequence functions is desirable stems from the fact that literal **lambda** expressions are cumbersome to specify. The **-if-not** sequence function versions, the **:test-not** keyword, and the **:key** keyword are helpful, because they reduce the number of **lambda** expressions which have to be written. As illustrated below, they do this by increasing the number of situations where preexisting functions can be used as functional arguments.

```
(count-if-not #'plusp '((1) (-2) (3)) :key #'car)
  ≡ (count-if #'(lambda (x) (not (plusp (car x))))
              '((1) (-2) (3))) ⇒ 1
```

When using OSS functions, implicit mapping makes it possible to avoid **lambda** expressions almost entirely in any case. One simply maps the desired test and key over the OSS series in question. As above, the use of separate operations would lead to inefficiency when using sequence functions, but does not cause any problem when using OSS functions.

```
(Rlength (Tselect (not (plusp (car [(1) (-2) (3)])))))) ⇒ 1
```

Function by function comparison. The operation performed by **elt** is provided by **Rnth**. However, **setf** cannot be used with **Rnth**. Also, as discussed in Section 1, the non-OSS data flow isolation restriction makes **Rnth** less useful than **elt**. (The notation "≡" is used rather loosely here in that the left hand form is applied to a sequence while the right hand form is applied to an OSS series.)

```
(elt '(a b c) 1) ⇒ b ≡ (Rnth 1 [a b c]) ⇒ b
```

The operation performed by **subseq** is provided by **Tsubseries**. However, **setf** cannot be used with **Tsubseries** or **Tselect**.

```
(subseq '(a b c d) 1 3) ⇒ (b c)
  ≡ (Tsubseries [a b c d] 1 3) ⇒ [b c]
```

As discussed above, the operation performed by **copy-seq** is not supported by an OSS function, because it is not useful in the context of OSS expressions.

The operation performed by **length** is provided by **Rlength**.

```
(length '(a b c)) ⇒ 3 ≡ (Rlength [a b c]) ⇒ 3
```

The operations performed by **reverse** and **nreverse** cannot be supported, because they are not preorder.

The operation performed by **make-sequence** is provided by **Tsubseries** and **Eoss** except that no type argument is required.

```
(make-sequence 'list 4 :initial-element T) ⇒ (T T T T)
  ≡ (Tsubseries (Eoss :R T) 0 4) ⇒ [T T T T]
```

The operation performed by **concatenate** is provided by **Tconcatenate**. The only difference is that no result-type parameter is necessary, and at least two series inputs are required. Given that copying and type conversion of OSS series is never necessary, allowing only a single input would not be useful.

```
(concatenate 'list '(a b) '(c)) ⇒ (a b c)
  ≡ (Tconcatenate [a b] [c]) ⇒ [a b c]
```

The operation performed by **map** is provided by the function **TmapF** and, if the mapped function is quoted, by implicit mapping. The only difference is that no result-type parameter is necessary, there need not be any input series, and macros can be mapped. Since infinite OSS series are supported, it is not necessary to require that there must always be a series input. Also since OSS series are never physically produced, there is never any need to specify that one should not be produced. As with **map**, it is guaranteed that the function being mapped will be applied to the series elements in order.

```
(map 'list #'+ '(1 2) '(3 4 5)) ⇒ (4 6)
  ≡ (TmapF #'+ [1 2] [3 4 5]) ⇒ [4 6]
  ≡ (prognS (+ [1 2] [3 4 5])) ⇒ [4 6]

(map 'list (symbol-function '+) '(1 2) '(3 4 5)) ⇒ (4 6)
  ≡ (TmapF (symbol-function '+) [1 2] [3 4 5]) ⇒ [4 6]
```

The operations performed by **some**, **every**, **notany**, and **notevery** are provided by implicit mapping and the functions **Ror** and **Rand**.

```
(some #'plusp '(1 -2 3)) ≡ (Ror (plusp [1 -2 3])) ⇒ T
(every #'plusp '(1 -2 3)) ≡ (Rand (plusp [1 -2 3])) ⇒ nil
(notany #'plusp '(1 -2 3)) ≡ (not (Ror (plusp [1 -2 3]))) ⇒ nil
(notevery #'plusp '(1 -2 3)) ≡ (not (Rand (plusp [1 -2 3]))) ⇒ T
```

The operation performed by **reduce** is provided by **ReduceF** and **Tsubseries**. However, there are two differences. First, the **:from-end** keyword is not supported, since it calls for non-preorder processing. Second, the **:initial-value** input is mandatory. This can be partially avoided by using **Rlast** and **TscanF**. However, there is no way to obtain the behavior whereby **reduce** calls the reduction function with zero arguments when the input is of zero length. This was judged to be more confusing than useful.

The operation performed by **fill** is a destructive operation and therefore is not provided by an OSS function. However, a non-destructive version of this operation can be obtained using **if** and **Tmask**. In addition, **alterS** in conjunction with **Elist**, **Evector**, and **Esequence** can be used to fill lists and vectors.

```
(fill '(0 1 2 3) 'x :start 1 :end 3) ⇒ (0 x x 3)
  ≡ (prognS (if (Tmask (Eup 1 :below 3))
                (Eoss :R 'x)
                [0 1 2 3])) ⇒ [0 x x 3]

(fill '(0 1 2 3) 'x :start 1 :end 3) ⇒ (0 x x 3)
  ≡ (let ((l '(0 1 2 3)))
       (alterS (Tsubseries (Elist l) 1 3) (Eoss :R 'x))
       l) ⇒ (0 x x 3)
```

The operation performed by **replace** is a destructive operation and therefore is not provided by an OSS function. However, **alterS** in conjunction with **Elist**, **Evector**, or **Esequence** can be used to replace elements in a list or vector.

```
(replace #(0 1 2 3) '(a b c)
         :start1 1 :end1 3 :start2 0 :end2 2) ⇒ #(0 a b 3)
  ≡ (let ((v #(0 1 2 3)))
       (alterS (Evector v (Eup 1 :below 3)) (Tsubseries [a b c] 0 2))
       v) ⇒ #(0 a b 3)
```

The operations performed by **remove**, **remove-if**, and **remove-if-not** are provided by **Tselect** or **TselectF**, **Tsubseries** or **Tmask**, and **Tlatch**. The only fundamental difference is that the **:from-end** keyword is not supported, because it calls for non-preorder processing. In the general case, things are complicated. However, in simple situations they are simple. The operations performed by **delete**, **delete-if**, and **delete-if-not** are not provided by OSS functions since they are destructive.

```
(remove -2 '(1 -2 3 -2)) ⇒ (1 3)
  ≡ (letS ((x [1 -2 3 -2]))
       (Tselect (not (eql -2 x)) x)) ⇒ [1 3]

(remove-if #'minusp '(1 -2 3 -2)) ⇒ (1 3)
  ≡ (letS ((x [1 -2 3 -2]))
       (Tselect (not (minusp x)) x)) ⇒ [1 3]

(remove-if-not #'minusp '(1 -2 3 -2)) ⇒ (-2 -2)
  ≡ (letS ((x [1 -2 3 -2]))
       (Tselect (minusp x) x)) ⇒ [-2 -2]

(remove-if #'minusp '((-1) (-2) (3) (-2) (1) (-4) (-3) (2))
           :start 1 :end 6
           :count 2 :key #'car) ⇒ '((-1) (3) (1) (-4) (-3) (2))
  ≡ (letS ((e (Tsubseries [(-1) (-2) (3) (-2) (1) (-4) (-3) (2)] 1 6)))
       (Tselect (not (Tlatch (minusp (car e)) :after 2))))
           ⇒ [(-1) (3) (1) (-4) (-3) (2)]
```

The operation performed by **remove-duplicates** is provided by **Tremove-duplicates**. However, **Tremove-duplicates** always operates in the way **remove-duplicates** operates when the **:from-end** keyword is specified. Operating in any other way calls for non-preorder processing. In addition, there is no easy way to obtain the effect of the **:start** and **:end** keywords. The operation performed by **delete-duplicates** is not provided since it is destructive.

```
(remove-duplicates '(a b a d) :from-end T) ⇒ (a b d)
  ≡ (Tremove-duplicates [a b a d]) ⇒ [a b d]

(remove-duplicates '((a 0) (a 1) (b 3) (a 2) (a 3))
                   :test #'eql :key #'car
                   :from-end T) ⇒ '((a 0) (b 3))
  ≡ (Tremove-duplicates [(a 0) (a 1) (b 3) (a 2) (a 3)]
      #'(lambda (x y)
          (eql (car x) (car y)))) ⇒ [(a 0) (b 3)]
```

The operations performed by **substitute**, **substitute-if**, and **substitute-if-not** are provided by **if**, **Tmask**, and **Tlatch**. The only fundamental difference is that the **:from-end** keyword is not supported, because it calls for non-preorder processing. The operations performed by **nsubstitute**, **nsubstitute-if**, and **nsubstitute-if-not** are not provided by OSS functions since they are destructive.

```
(substitute 0 -2 '(1 -2 3 -2)) ⇒ (1 0 3 0)
  ≡ (letS ((x [1 -2 3 -2]))
      (if (eql -2 x) 0 x)) ⇒ [1 0 3 0]

(substitute-if 0 #'minusp '(1 -2 3 -2)) ⇒ (1 0 3 0)
  ≡ (letS ((x [1 -2 3 -2]))
      (if (minusp x) 0 x)) ⇒ [1 0 3 0]

(substitute-if-not 0 #'minusp '(1 -2 3 -2)) ⇒ (0 -2 0 -2)
  ≡ (letS ((x [1 -2 3 -2]))
      (if (not (minusp x)) 0 x)) ⇒ [0 -2 0 -2]

(substitute-if 0 #'minusp '((-1) (-2) (3) (-2))
               :start 1 :end 3
               :count 2 :key #'car) ⇒ ((-1) 0 (3) (-2))
  ≡ (letS ((x [(-1) (-2) (3) (-2)]))
      (if (Tlatch (and (Tmask (Eup 1 :below 3))
                       (minusp (car x)))
                  :after 2)
          0
          x)) ⇒ [(-1) 0 (3) (-2)]
```

The operations performed by **find**, **find-if**, and **find-if-not** are provided by **Tselect** or **TselectF**, **Tsubseries** or **Tmask**, and **Rfirst** or **Rlast**. Unlike most of the functions above, the **:from-end** keyword is supported, because it can be supported tolerably efficiently in a preorder way.

```
(find 'b '((a 1) (b 3) (a 3) (b 4)) :key #'car) ⇒ (b 3)
  ≡ (letS ((x [(a 1) (b 3) (a 3) (b 4)]))
      (Rfirst (Tselect (eql 'b (car x)) x))) ⇒ (b 3)

(find-if #'minusp '(1 -2 3 -3)) ⇒ -2
  ≡ (letS ((x [1 -2 3 -2]))
      (Rfirst (Tselect (minusp x) x))) ⇒ -2

(find-if-not #'minusp '(1 -2 3 -2) :from-end T) ⇒ 3
  ≡ (letS ((x [1 -2 3 -2]))
      (Rlast (Tselect (not (minusp x)) x))) ⇒ 3

(find-if #'minusp '((-1) (-2) (3) (-5) (1))
         :start 1 :end 4 :key #'car) ⇒ (-2)
  ≡ (letS ((x (Tsubseries [(-1) (-2) (3) (-5) (1)] 1 4)))
      (Rfirst (Tselect (minusp (car x)) x))) ⇒ (-2)
```

The operations performed by `position`, `position-if`, and `position-if-not` are provided by `Tpositions`, `Tsubseries` or `Tmask`, and `Rfirst` or `Rlast`. As with the `find` functions, the `:from-end` keyword is supported. As illustrated below, things are much the same as with the `find` functions except that `Tpositions` is used instead of `Tselect`. In addition, if a subseries is specified, then `Tmask` (or a numerical adjustment) has to be used in order to get the position relative to the correct origin.

```
(position 'b '((a 1) (b 3) (a 3)) :key #'car) ⇒ 1
  ≡ (Rfirst (Tpositions (eql 'b (car [(a 1) (b 3) (a 3)])))) ⇒ 1

(position-if #'minusp '(1 -2 3 -3)) ⇒ 1
  ≡ (Rfirst (Tpositions (minusp [1 -2 3 -2]))) ⇒ 1

(position-if-not #'minusp '(1 -2 3 -2) :from-end T) ⇒ 2
  ≡ (Rlast (Tpositions (not (minusp [1 -2 3 -2])))) ⇒ 2

(position-if #'minusp '((-1) (-2) (3))
             :start 1 :end 3 :key #'car) ⇒ 1
  ≡ (Rfirst (Tpositions (and (Tmask (Eup 1 :below 3))
                             (minusp (car [(-1) (-2) (3)]))))) ⇒ 1
  ≡ (+ 1 (Rfirst
           (Tpositions
             (minusp (car (Tsubseries [(-1) (-2) (3)] 1 3)))))) ⇒ 1
```

The operations performed by `count`, `count-if`, and `count-if-not` are provided by `Rlength` and `Tsubseries` or `Tmask`. As illustrated below, things are much the same as with the `find` functions except that `Rlength` is used instead of `Rfirst`.

```
(count 'b '((a 1) (b 3) (b 4)) :key #'car) ⇒ 2
  ≡ (Rlength (Tselect (eql 'b (car [(a 1) (b 3) (b 4)])))) ⇒ 2

(count-if #'minusp '(1 -2 3 -3)) ⇒ 2
  ≡ (Rlength (Tselect (minusp [1 -2 3 -2]))) ⇒ -2

(count-if-not #'minusp '(1 -2 3 -2)) ⇒ 2
  ≡ (Rlength (Tselect (not (minusp [1 -2 3 -2])))) ⇒ 2

(count-if #'minusp '((-1) (-2) (3) (-5) (1))
          :start 1 :end 4 :key #'car) ⇒ 2
  ≡ (Rlength
      (Tselect
        (minusp
          (car (Tsubseries [(-1) (-2) (3) (-5) (1)] 1 4))))) ⇒ 2
```

The operations performed by `mismatch` and `search` could be provided as off-line OSS functions. However, they seem to be too specialized to be of general use. To perform these operations, construct a physical series and use the standard functions. As mentioned above, the functions `sort` and `stable-sort` cannot be supported since the are inherently non-preorder.

The operation performed by `merge` is provided by `Tmerge`, except that `Tmerge` is not destructive and no result type argument is required.

```
(merge 'list '(1 3 5) '(2 6) #'<) ⇒ (1 2 3 5 6)
  ≡ (Tmerge [1 3 5] [2 6] #'<) ⇒ [1 2 3 5 6]

(merge 'list '((1) (3) (5)) '((2) (6))
       #'< :key #'car) ⇒ ((1) (2) (3) (5) (6))
  ≡ (Tmerge [(1) (3) (5)] [(2) (6)]
            #'(lambda (x y)
                (< (car x) (car y)))) ⇒ [(1) (2) (3) (5) (6)]
```

As can be seen above, most of the sequence operations are supported by OSS functions. In addition, the OSS functions support a number of additional operations. For one thing, the whole concept of an enumerator is more or less absent from the sequence functions and the sequence functions support very few specific reducers. (In Common Lisp the equivalent of enumerators exist in the guise of forms like **dotimes** and **dosymbols**. However, is no practical way to apply sequence functions to the values enumerated by these forms.) In contrast, the OSS functions contain a wide range of specific enumerators and reducers.

The OSS functions also support a number of complex higher-order functions such as **TscanF** and **TsplitF** which are not supported by the sequence functions. In addition, the concepts of implicit mapping and alterability go beyond the scope of the sequence functions.

The OSS functions are also stylistically quite different. In particular, composition of functions is used wherever possible in lieu of keyword arguments. This allows the individual functions to be simpler and makes things more functional in appearance. In addition, it allows maximal freedom for programmers to do exactly what they want to do exactly when they want to do it.

Taken individually, OSS functions are no more efficient than sequence functions. However, when combined with each other in expressions they are significantly more efficient.

There are two key places where sequence functions are more powerful than OSS functions. First, there is no limit to the sequence functions which can be defined. In particular, it is possible to define a function which takes in a sequence and operates on its elements in any arbitrary order. In contrast, OSS functions must be preorder. However, as shown by the examples above, most of the predefined sequence functions are preorder. This suggests that the preorder restriction is not unduly limiting. Nevertheless, it is a significant limitation. Second, there is no limit to the way sequence functions can be combined together. In contrast, the isolation restrictions limit the way OSS functions can be combined.

The Loop Macro

This section shows how the predefined OSS functions capture the functionality of the Loop macro. Although not strictly necessary, a good understanding of the Loop macro [9] as described on pages 537–563 in Volume 2A of [31] will contribute significantly to an understanding of the discussion below which follows the outline of the description cited above.

Uses of the Loop macro have the following basic form where the *iteration-clauses* are analogous to OSS functions and the *body* is mapped over the values enumerator-like clauses produce. The most obvious difference between Loop and OSS expressions is that loop iteration clauses are rendered in stylized English while OSS expressions use standard functional notation. As discussed below, the functionality of every Loop iteration clause is supported by an OSS function. However, a few of the complex ways in which the clauses can be combined are not supported.

```
(loop iteration-clauses
   do body)
```

Using **for** (or **as**), iteration clauses can bind values to variables either sequentially or in parallel via the keyword **and**. This functionality is supported by **letS** and **letS***. The effect of data type specifications in a clause are provided by declarations in a **letS**. (Note that using the Loop macro, it is not possible to use the value of an enumerator without binding it to a variable. In contrast, OSS expressions make it possible to simply put an enumerator where it is used.)

```
(loop for x integer from 1 to 4 collect x)
  ≡ (letS* ((x (Eup 1 :to 4)))
       (declare (type integer x))
     (Rlist x)) ⇒ (1 2 3 4)
```

However, the distinction between sequential and parallel binding in Loop is different from the distinction between **letS*** and **letS**. This is so, because Loop variables are not thought of as containing series, but rather as containing individual values as in a **do**. As a result, "in parallel" really means "using values from the previous iteration". This effect can be obtained in an OSS expression by using **Tprevious** as shown below.

```
(loop for x from 1 to 4 and for y = 0 then (1- x)
      collect (list x y))
  ≡ (letS* ((x (Eup 1 :to 4))
            (y (Tprevious (1- x) 0)))
     (Rlist (list x y))) ⇒ ((1 0) (2 0) (3 1) (4 2))
```

The operation performed by **repeat** is provided by **Eup** and **Tcotruncate**. All that has to be done is to cotruncate the series being operated on with a series of the required length.

```
(loop repeat 4 for x from 1 to 10 collect x)
  ≡ (Rlist (Tcotruncate (Eup 1 :to 10) (Eup :length 4))) ⇒ (1 2 3 4)
```

The operations performed by **from**, **to**, **by**, **downto**, **below**, **above**, **downfrom**, and **upfrom** are provided by **Eup** and **Edown**.

```
(loop for x from 1 to 7 by 2 collect x)
  ≡ (Rlist (Eup 1 :to 7 :by 2)) ⇒ (1 3 5 7)

(loop for x from 1 downto -7 by 2 collect x)
  ≡ (Rlist (Edown 1 :to -7 :by 2)) ⇒ (1 -1 -3 -5 -7)

(loop for x upfrom 1 by 2 and for y from 1 to 3 collect (list x y))
  ≡ (Rlist (list (Eup 1 :by 2) (Eup 1 :to 3))) ⇒ ((1 1) (3 2) (5 3))
```

The simple forms of the operations performed by **in** and **on** are provided by **Elist** and **Esublists**. The complex forms of these operations are provided by **EnumerateF**.

```
(loop for x on '(a b c) collect x)
  ≡ (Rlist (Esublists '(a b c))) ⇒ ((a b c) (b c) (c))

(loop for x in '(a b c) collect x)
  ≡ (Rlist (Elist '(a b c))) ⇒ (a b c)

(loop for x in '(a b c) by #'cddr collect x)
  ≡ (Rlist (car (EnumerateF '(a b c) #'cddr #'endp))) ⇒ (a c)
```

The operation performed by = is provided by **TmapF**, **mapS**, or usually more conveniently by implicit mapping. If the **then** keyword is added, things are more complex. Often the only free variable in the expression after the **then** is the same variable which is bound to the result of the clause. When this is the case, **EnumerateF** can be used. Alternately, if an **and** keyword also appears, then **Tprevious** can be used as shown above. If the expression is complex and there is no **and**, then there is no simple way to get the all-but-the-first-iteration effect. In addition, there is nothing in OSS expressions corresponding to the difference between = ... **then** and **first** ... **then**. These difficulties stem from the different points of view held by the Loop macro and the OSS macro package about what a variable is.

```
(loop for x from 1 to 3 for y = (1+ x) collect (list x y))
  ≡ (letS ((x (Eup 1 :to 3)))
      (Rlist (list x (1+ x)))) ⇒ ((1 2) (2 3) (3 4))
(loop for x from 1 to 3 for y = 1 then (* 2 y)
      collect (list x y))
  ≡ (Rlist (list (Eup 1 :to 3)
                 (EnumerateF 1
                   #'(lambda (y) (* 2 y))))) ⇒ ((1 1) (2 2) (3 4))
```

The kind of variables created by **with** can be created as non-OSS variables in a **letS***. However, these variables cannot be assigned to, so you have to think about things from a different, more functional, perspective.

```
(loop for x from 0 to 4
      with (one four)
      with three = "three"
      do (setq four x) (setq one "one")
      finally (return (values one three four)))
  ≡ (letS* ((x (Eup :to 4))
            (one "one")
            (three "three")
            (four (Rlast x)))
      (valS one three four)) ⇒ "one" "three" 4
```

The effect of **initially** and **finally** can be obtained in several ways. When defining a new OSS function **prologS** and **epilogS** can be used in a **defun-primitiveS**. This represents exactly the same functionality. In an expression, a **funcallS** of a **lambda-primitiveS** can be used in order to get the exact effect. However, things can often be arranged so that things happen initially and finally in a natural way. The natural elimination of a need for **finally** is shown in the last example. (It is never necessary to use **return** in an OSS expression.) The elimination of an **initially** by using **Eoss** (or better by just putting the operation in question in front of the OSS expression) is shown below. It should be realized that due to the different points of view taken by the Loop macro and the OSS macro package, there is little reason to think of the concepts of initially and finally when writing an OSS expression.

```
(loop for x from 1 to 4
      initially (princ "lets count... ")
      do (princ x))
  ≡ (prognS (Eoss :R (princ "lets count... ")) (princ (Eup 1 :to 4)))
  ≡ (princ "lets count... ") (prognS (princ (Eup 1 :to 4)))
;;The output "lets count... 1234" is produced
```

The various accumulating clauses are provided by reducers. The effect of the `into` keyword can be obtained by binding the result of a reducer to an OSS variable. A feature of the Loop macro which is emphatically not supported by any OSS function is the concept of combining several different accumulators into the same result. Using an OSS expression, it is necessary to define a special reducer or merge the series or something of that order.

```
(loop for x in '(1 2 3 6)
      count t into count-var
      sum x into sum-var
      finally (return (/ sum-var count-var)))
  ≡ (letS* ((x (Elist '(1 2 3 6)))
            (count-var (Rlength x))
            (sum-var (Rsum x)))
     (/ sum-var count-var)) ⇒ 3

(loop for x in '(a b) for y in '((1 2) (3 4))
      collect x
      append y)
  ≡ (Rappend (list* (Elist '(a b))
                    (Elist '((1 2) (3 4))))) ⇒ (a 1 2 b 3 4)
```

The operations performed by `collect`, `nconc`, and `append` are provided by `Rlist`, `Rnconc`, and `Rappend` respectively. The operations performed by `sum`, `maximize`, and `minimize` are provided by `Rsum`, `Rmax` and `Rmin` respectively.

```
(loop for x in '((1 2) (3 4)) nconc x)
  ≡ (Rnconc (Elist '((1 2) (3 4)))) ⇒ (1 2 3 4)

(loop for x in '(1 3 2) sum x)
  ≡ (Rsum (Elist '(1 2 3))) ⇒ 6

(loop for x in '(1 3 2) maximize x)
  ≡ (Rmax (Elist '(1 2 3))) ⇒ 3
```

The operation performed by `count` is more interesting. In general, it is provided by `Rlength` of `Tselect`. However, things have to be arranged so that the expression being counted contains a reference to the series of elements being counted. (Observe the way `count` is handled in the averaging example above.) In the Loop macro, this is not always necessary, because count is controlled by the implicit control environment it appears in. In OSS expressions, there is no notion of control environments—everything depends on data flow.

```
(loop for x in '(1 -3 2) count (plusp x))
  ≡ (Rlength (Tselect (plusp (Elist '(1 -3 2))))) ⇒ 2
```

The effect of `until` and `while` can typically be obtained by using `EnumerateF`. The primitive operation performed by `loop-finish` is provided by `terminateS`.

```
(loop for x = '(a b c) then (cdr x) until (null x) collect (car x))
  ≡ (Rlist (car (EnumerateF '(a b c) #'cdr #'null))) ⇒ (a b c)
```

The operations performed by `always`, `never`, and `thereis` are provided by `Rand`, `not` of `Ror`, and `Rfirst` of `Tselect` respectively.

```
(loop for x in '(1 -3 2) always (plusp x))
  ≡ (Rand (plusp (Elist '(1 -3 2)))) ⇒ nil

(loop for x in '(nil (nil c) (a b)) thereis (car x))
  ≡ (Rfirst (Tselect (car (Elist '(nil (nil c) (a b)))))) ⇒ a
```

Conditionalization of operations is handled in OSS expressions in a completely different way than in the Loop macro. Instead of directly effecting the control flow, series are restricted using `Tselect` or `Tsplit`. Given the very different approaches, it is hard to make simple comparisons. The following examples illustrate the two approaches.

```
(loop for i from 1 to 10
      when (zerop (mod i 3)) collect i)
  ≡ (letS* ((i (Eup 1 :to 10))
            (i-divisible-by-3 (Tselect (zerop (mod i 3)) i)))
      (Rlist i-divisible-by-3)) ⇒ (3 6 9)

(loop for i from 1 to 10
      when (zerop (mod i 3))
        do (prin1 i)
        and when (zerop (mod i 2))
              collect i)
  ≡ (letS* ((i (Eup 1 :to 10))
            (i-divisible-by-3 (Tselect (zerop (mod i 3)) i)))
      (prin1 i-divisible-by-3)
      (Rlist (Tselect (zerop (mod i-divisible-by-3 2))
                      i-divisible-by-3))) ⇒ (6)
;;The output "369" printed.

(loop for i from 1 to 9
      if (oddp i)
         collect i into odd-numbers
      else collect i into even-numbers
      finally (return odd-numbers even-numbers))
  ≡ (letS* ((i (Eup 1 :to 9))
            ((odd-i even-i) (TsplitF i #'oddp))
            (odd-numbers (Rlist odd-i))
            (even-numbers (Rlist even-i)))
      (valS odd-numbers even-numbers)) ⇒ (1 3 5 7 9) (2 4 6 8)
```

The functionality of the **return** and **named** keywords is not directly supported. However, this effect can be obtained by wrapping a (named) **block** around an OSS expression and returning from the block. No support is provided for destructuring, since destructuring is not a standard part of Common Lisp. Synonyms for OSS functions can be defined using **defunS**.

For the most part, the operations performed by predefined Loop iteration paths are provided by predefined OSS enumerators. In particular, **being the hash-elements** is supported by **Ehash**, **being the array-elements** is supported by **Evector**, and **being the local-interned-symbols** is supported by **Esymbols**.

```
(loop for x being the array-elements of #(a b c) collect x)
  ≡ (Rlist (Evector #(a b c))) ⇒ (a b c)
```

However, heap iteration is not supported since heaps are not supported by Common Lisp. Further, there is no support for the iteration performed by **interned-symbols**, although it could be supported easily enough. In addition, the functionality of **using** is emphatically not supported by OSS expressions. Rather, OSS functions go to considerable lengths to hide their internal states so that other functions cannot disturb their operation.

The functionality of **define-loop-sequence-path** can be obtained straightforwardly using **defunS**, **Tuntil**, and a nested syntax where complex indexing can be specified by **Eup** or any other OSS function which produces a series of indices.

```
(define-loop-sequence-path (array-element array-elements)
  aref length)
  ≡ (defunS Earray (v &optional (i (Eup)))
      (declare (type oss i))
      (aref v (Tuntil (not (< i (length v))) i)))
```

The functionality of **define-loop-path** is provided in a quite different way by **defunS**. The major difference is that the Loop macro requires the user to explicitly deal with the details of the way loops are constructed, while OSS expressions do not. The following shows how the iteration path used as an example on page 561 of Volume 2A of [31] could be rendered using **defunS**. In [31] the example requires 31 lines of code not counting the comments. (The subprimitive form **defun-primitiveS**, see Section 3, defines OSS functions in a way which is much more closely analogous to **define-loop-path**. However, it is still much simpler.)

```
(defunS Estring-chars (s)
  (aref s (Eup 0 :to (1- (length s)))))
```

As can be seen above, most of the operations supported by the Loop macro are supported by OSS functions. In addition, the OSS functions support a number of additional operations. For one thing, the whole concept of non-mapping transducers is more or less absent from the Loop macro. In contrast, the OSS functions contain a wide range of powerful transducers. In addition, they contain many more specific enumerators and reducers.

The code produced by OSS expressions is no more efficient than the code produced by the Loop macro. However, OSS expressions are more concise. In particular, they are functional rather than based on a pseudo-English syntax. A particular advantage of OSS expressions is that it is much easier to define new OSS functions than it is to define new Loop clauses.

In contrast to the sequence functions, the Loop macro is not fundamentally more powerful than OSS expressions. The user is effectively limited to preorder operations when creating new Loop operations and there are, in effect, significant limitations analogous to the isolation restrictions placed on the way Loop operations can be combined. This is not surprising in light of the fact that the two systems end up producing quite similar code in quite similar ways (see Section 3).

APL

This section shows how the predefined OSS functions capture the functionality of the APL vector operators. Although not strictly necessary, a good understanding of APL (see for example [15]) will contribute significantly to an understanding of the discussion below.

Since OSS series correspond solely to arrays of rank one, no support is given for any operator which either must take in or must produce an array of rank greater than one (i.e., ravel, outer product, matrix inverse, matrix divide, lamination). Also, no support is given for non-preorder operations (i.e., reshape, indexing, index-of, grade up, grade down, transpose, reversal, rotation).

The APL concept of the extension of scalar functions to vectors corresponds exactly to implicit mapping. It is as ubiquitous and useful in OSS expressions as in APL expressions.

The index generator (count) operation is provided more generally by **Eup**. When restricted to vectors, the shape operation is provided by **Rlength**. The catenate operation is performed by **Tconcatenate**.

```
ι4 ⇒ 1 2 3 4 ≡ (Eup 1 :to 4) ⇒ [1 2 3 4]
ρ7 8 9 ⇒ 3 ≡ (Rlength [7 8 9]) ⇒ 3
1 2 3,4 5 ⇒ 1 2 3 4 5 ≡ (Tconcatenate [1 2 3] [4 5]) ⇒ [1 2 3 4 5]
```

The membership operation is provided by **Ror** and the implicit mapping of an equality test as shown below. (In APL, boolean values are represented as integers with 1 for true and 0 for false.)

```
2∈1 2 3 ⇒ 1 ≡ (Ror (= 2 [1 2 3])) ⇒ T
```

The take and drop operations (without the introduction of padding) are both provided by **Tsubseries**. However, they are only supported for positive numbers (i.e., taking from the front and dropping from the front). (Efficient support for taking and dropping from the end is incompatible with preorder processing.)

```
2↑1 2 3 4 5 ⇒ 1 2 ≡ (Tsubseries [1 2 3 4 5] 0 2) ⇒ [1 2]
2↓1 2 3 4 5 ⇒ 3 4 5 ≡ (Tsubseries [1 2 3 4 5] 2) ⇒ [3 4 5]
```

The APL concept of reduction is supported in a more general way by **ReduceF**. In APL, reduction can only be applied to the primitive dyadic scalar functions and cannot be applied to user defined functions. As a result, in APL, reduction only corresponds to a specific set of reducers rather than to a true combinator. In contrast, **ReduceF** can be used with any binary function. An advantage of the APL approach is that users do not have to specify initial values to use in reductions, because APL has built-in knowledge of what initializations are appropriate. In OSS expressions, this advantage can be obtained by using predefined reducers (e.g., **Rsum**, **Rmax**, **Rmin**) rather than **ReduceF**.

```
+/1 2 3 ⇒ 6 ≡ (ReduceF 0 #'+ [1 2 3]) ≡ (Rsum [1 2 3]) ⇒ 6
⌈/1 2 3 ⇒ 3 ≡ (Rmax [1 2 3]) ⇒ 3
⌊/1 2 3 ⇒ 1 ≡ (Rmin [1 2 3]) ⇒ 1
```

The APL concept of scanning is supported by **TscanF**. As is the case with **ReduceF**, **TscanF** is fundamentally more general. However, there are no predefined OSS functions corresponding to the APL operations which can be scanned.

```
+\1 2 3 ⇒ 1 3 6 ≡ (TscanF 0 #'+ [1 2 3]) ⇒ [1 3 6]
```

The inner product operation is straightforwardly supported by **ReduceF** and implicit mapping. However, in OSS expressions this concept is impoverished since all of the arguments must be one dimensional.

```
1 2 3+.*3 4 5 ⇒ 26 ≡ (ReduceF 0 #'+ (* [1 2 3] [3 4 5])) ⇒ 26
```

The operations of compression and expansion are provided by **Tselect** and **Texpand**. (In APL, compression and expansion make use of the binary forms of the same operator symbols which are used to indicate reduction and scanning.)

```
1 0 1 0/1 2 3 4 ⇒ 1 3
  ≡ (Tselect [T nil T nil] [1 2 3 4]) ⇒ [1 3]
1 0 1 0\1 3 ⇒ 1 0 3 0
  ≡ (Texpand [T nil T nil] [1 3] 0) ⇒ [1 0 3 0]
```

The operations of encoding and decoding numbers into mixed radix notations are an interesting case. As defined in APL, they require postorder processing and therefore cannot be supported by OSS functions. However, if they were redefined so that the least significant digit was stored first, then they could easily be supported. This is a good example of the kind of change which sometimes has to be made to facilitate the use of OSS expressions.

When compared solely with the built-in vector operations in APL, OSS expressions can be seen to be more powerful. However, the comparison with APL points up several of the limitations of OSS expressions. To start with, APL is like the sequence functions in that there is no limit to the new vector functions which a user can define and there is no limit to the way these functions can be combined. In addition, unlike the sequence functions or the Loop macro, APL shows the power which can be obtained by operating on multidimensional data. An interesting aspect of this is that while relatively few useful, non-preorder functions spring to mind when thinking about one dimensional data, it is much easier to think of such functions when operating on multi-dimensional data.

As with the sequence functions, individual OSS functions are no more efficient than individual APL operations. However, OSS expressions are a great deal more efficient than APL expressions. Further, it almost goes without saying that the most striking difference between OSS expressions and APL is that OSS expressions use standard functional notation while APL uses a host of concise, but cryptic operators.

3. Algorithms

Once OSS expressions have been syntactically hosted in a language, the only thing which remains to be done is to implement a preprocessor which supports their efficient evaluation. The restrictions underlying OSS expressions guarantee that such a preprocessor can be implemented in three parts: a *parser* which locates OSS expressions, an implicit mapper which determines what subexpressions should be mapped, and a *transformer* which converts the OSS expressions into highly efficient loops. Below, the OSS macro package [27] is used as a concrete vehicle for describing the algorithms required.

Parsing. The characteristics of Lisp make the parser component in the OSS macro package particularly easy to implement. In fact, the standard Common Lisp macro facilities can be used to locate OSS expressions. Supporting implicit mapping requires explicit parsing code to be written. However, this task is simplified by the fact that Lisp programs are represented internally in a parse-tree-like form. The only real complexity stems from the fact that the OSS macro package must have an understanding of the special forms supported by Common Lisp. This is a problem which is faced by many complex macro packages.

In a language other than Lisp, the parsing task would be inherently more complex. However, the same basic approach of extending the standard parser for the language could be used.

Implicit mapping. The implicit mapper takes a parsed OSS expression and turns it into a data flow graph which shows the way the various functions are connected. Both nesting of subexpressions and the variables bound by `letS` lead to links in this graph. Based on an inspection of the graph, implicit mapping is introduced whenever a non-OSS function receives an OSS input.

Transformation. The operation of the transformer component is illustrated in Figure 3.1. The OSS expression in the function `oss-sum-sqrts` is transformed into the loop shown in the function `oss-sum-sqrts-loop`. The readability of the transformed code is reduced by the fact that it contains a number of internally generated variables. However, although the code is a bit unusual in some ways, it is quite efficient. As a result, assuming that both functions are compiled, `oss-sum-sqrts` is just as efficient as the function `sum-sqrts-loop` used as an example at the beginning of Section 1. (If `oss-sum-sqrts` is evaluated interpretively, it is quite slow since the OSS macro package has to be called in order to transform the loop as shown.)

The only significant problem with the code produced by the OSS macro package is that it often uses more variables than strictly necessary (e.g., `#:out-4`). However, this problem need not lead to inefficiency during execution as long as a compiler capable of simple optimizations is available.

The transformation process operates in several steps. In the first step, the data flow graph produced by the implicit mapping step is broken into on-line subexpressions using the divide and conquer strategy discussed in conjunction with the non-series isolation restriction and the off-line isolation restriction (see Section 1). As part of this process, the transformer checks that the expression obeys the isolation restrictions and the re-

```
    (defun oss-sum-sqrts (v)
      (Rsum (sqrt (TselectF #'plusp (Evector v)))))

    (defun oss-sum-sqrts-loop
      (let (#:element-7 #:index-9 #:last-8 #:out-4 #:sum-2)
2          (declare (type integer #:index-9 #:last-8)
3,5                 (type number #:out-4 #:sum-2))
2        (tagbody (setq #:index-9 -1)
1,2              (setq #:last-8 (length v))
5                (setq #:sum-2 0)
2           #:L-1 (incf #:index-9)
2                 (if (not (< #:index-9 #:last-8)) (go oss:END))
1,2               (setq #:element-7 (aref v #:index-9))
3                 (if (not (plusp #:element-7)) (go #:L-1))
4                 (setq #:out-4 (sqrt #:element-7))
5                 (setq #:sum-2 (+ #:sum-2 #:out-4))
                  (go #:L-1)
           oss:END)
5       #:sum-2))

(1) -- non-oss evaluation of v -----------------------
     outputs: (out)
       auxes: (out)
      prolog: ((setq out v))

(2) -- Evector ---------------------------------------
      inputs: (vector)
     outputs: (element)
       auxes: (last index)
       decls: ((type integer last index) (type vector vector))
   alterable: ((element (aref vector index)))
      prolog: ((setq index -1) (setq last (length vector)))
        body: ((incf index)
               (if (not (< index last)) (terminateS))
               (setq element (aref vector index)))

(3) -- TselectF of #'plusp ---------------------------
      inputs: (numbers)
     outputs: (numbers)
        body: (L (next-inS numbers) (if (not (plusp numbers)) (go L)))

(4) -- implicit mapping of sqrt ----------------------
      inputs: (in)
     outputs: (out)
        body: ((setq out (sqrt in)))

(5) -- Rsum ------------------------------------------
      inputs: (numbers)
     outputs: (sum)
       auxes: (sum)
       decls: ((type number numbers sum))
      prolog: ((setq sum 0))
        body: ((setq sum (+ sum numbers)))
```

Figure 3.1: Transforming OSS expressions into loops.

quirement that within each on-line subexpression, there must be a data flow path from each termination point to each output.

Once partitioning is complete, the functions in each on-line subexpression are combined together into a single operation. These operations are then combined together based on the data flow between the on-line subexpressions.

To support the combination process, each OSS function is represented as a fragment consisting of several parts including:

`inputs`: A list of input variables.
`outputs`: A list of output variables.
`auxes`: A list of auxiliary variables.
`decls`: A list of declarations.
`alterable`: A list of specifications as to which outputs are alterable.
`prolog`: A list of statements which are executed before the computation starts.
`body`: A list of statements which are repetitively executed.
`epilog`: A list of epilog statements which are executed after the loop terminates.

(As discussed in the next section, these pieces of information can be specified directly by using `lambda-primitiveS`.) The bottom part of Figure 3.1 shows the fragments which represent the five parts of the OSS expression in `oss-sum-sqrts`. (Fragments typically have several parts which are empty lists. In the interest of brevity, these parts are omitted in the figure.)

The loop in `oss-sum-sqrts-loop` is created by combining the five fragments shown in Figure 3.1. The numbers in the left hand margin of `oss-sum-sqrts-loop` indicate which fragment each line of the loop comes from. Three different combination algorithms are used: one corresponding to data flow between on-line ports (i.e., within on-line subexpressions), one corresponding to data flow touching off-line ports (i.e., between on-line subexpressions), and one corresponding to non-OSS data flow.

The algorithm for on-line data flow is illustrated by the combination of the implicit mapping of `sqrt` with `Rsum`. When two functions are connected by an OSS data flow between on-line ports, the functions are combined by simply concatenating the eight parts of the corresponding fragments. In addition, all of the variables and top level labels in the two fragments are renamed using new internally generated names so that there will be no possibility of name conflicts. The data flow between the functions is implemented by renaming the input variable of the destination so that it is the same as the output variable of the source. The on-line combination algorithm is essentially an application of the standard compiler optimization technique of loop fusion [2].

The algorithm for off-line data flow is illustrated by the combination of `Evector` with `TselectF` of `plusp`. When two functions are connected by an OSS series data flow terminating on an off-line input, the fragment representing the destination function contains an instance of `next-inS` specifying when elements of the input should be computed. The two fragments are combined exactly as in the on-line combination algorithm except that the body of the source fragment is substituted in place of the call on `next-inS` (described in the next section), rather than being concatenated with the body of the destination fragment.

The algorithm for non-OSS data flow is illustrated by the combination of **Evector** with its input v. Non-OSS expressions such as **v** are converted into fragments which have a prolog, but no body or epilog. Given this, the algorithm for non-OSS data flow is identical to the algorithm for on-line data flow. In addition, whenever a fragment (such as the one corresponding to v) consists merely of assigning one variable to another, simplifications are applied to the combined result in order to eliminate unnecessary variables.

Once all of the fragments representing the functions in an OSS expression have been combined into a single fragment, this fragment is converted into a loop as shown below.

```
(let (auxes)
    (declare decls)
  (tagbody prolog
       #:L body
           (go #:L)
   oss:END epilog)
  (values outputs))
```

Six of the parts of the fragment appear directly in the loop. The other three are handled as follows. The combination process eliminates all of the inputs. The information about alterability is discarded. In addition to the above, instances of **terminateS** (described in the next section) are converted into branches to the label **oss:END**. The net result of all this is returned as the macro expansion of the OSS expression as a whole.

As can be seen from the description above, the transformer is based on very simple algorithms. In the OSS macro package, the transformer is implemented using approximately 10 pages of Common Lisp code. Further, in contrast to the parser, the transformer is essentially programming language independent. Therefore, there is no reason why the transformer would not be just as simple in any language environment.

The OSS macro package as a whole (and the transformer component in particular) is descended from an earlier macro package called LetS [23,24]. LetS is similar to the OSS macro package in many ways, however, it is less powerful and less clear in its focus, because it is based on an unnecessarily strict set of ad hoc restrictions. (A system intermediate between LetS and the OSS macro package as presented in this report is described in [26].) LetS (and its transformer component) are in turn descended from ideas developed in the context of the Programmer's Apprentice project [22,25].

The same basic approach to representing and combining series functions was independently developed by Wile [28]. However, he does not explicitly address the question of restrictions and his approach does not guarantee that every intermediate series can be eliminated.

A quite similar approach is also used internally by the Loop macro [9]. However, Loop is externally very different from the OSS macro package. In particular, it uses an idiosyncratic Algol-like syntax rather than representing computations as compositions of functions operating on series. In addition, it does not support as wide a range of operations and does not make it easy for users to define new series operations.

Subprimitives

The OSS macro package provides a special primitive form **lambda-primitiveS** which can be used to specify off-line OSS functions and other complex kinds of OSS functions.

This form has a very restricted syntax, but makes it possible to define a very wide variety of OSS functions. It is oriented toward power rather than foolproof ease of use and is intended for the advanced user.

- **lambda-primitiveS** *input-list output-list aux-list* {*decl*}* **&body** *expr-list*

The first three parts of a **lambda-primitiveS** are lists of variables. The *input-list* specifies the names of the inputs of the OSS function being specified. The *output-list* specifies the outputs of the function being specified. The *aux-list* specifies internal state variables which are used by the computation in the OSS function. Each of these lists must be a list of variable names. Each of the names on the output-list must also be on the *input-list* or the *aux-list*. The outputs of the OSS function are the values of the output variables, rather than being the value of the last expression in the body.

There may be zero or more declarations just as in a **lambdaS**. Every input and output variable which is to carry an OSS value must be declared using **type oss**. It must either be the case that all of the output variables are OSS variables, or none of them is. An aux variable cannot carry an OSS value unless it is also an output. None of the input, aux, or output variables can be declared special. The input variables cannot be assigned to. The aux variables and output variables must be assigned to.

The *expr-list* specifies the computation to be performed. The key aspect of these expressions is that they are not OSS expressions. Rather, they are non-OSS expressions which define one cycle of operation of the OSS function as follows. The non-OSS inputs are available before any computation begins. Each time a new element is available in the OSS inputs, the expressions in the body of the **lambda-primitiveS** are run once. While this is going on, the OSS input variables have these elements as their values. After the running is completed, the current values of the OSS output variables are written out as the next element of the OSS outputs. If there are non-OSS outputs then they are not written out until after the OSS function terminates after processing all of the elements in the inputs. As a trivial example, the following shows an OSS function equivalent to mapping the function **car**. (This is intended merely as an illustration of **lambda-primitiveS**; the operation could be defined more easily using **TmapF**.)

```
(funcallS #'(lambda-primitiveS (x) (y) (y)
                (declare (type oss x y))
              (setq y (car x)))
          [(a) (b)]) ⇒ [a b]
```

The key aspect of the *expr-list* is that it can contain instances of the special forms described below. In addition, the *expr-list* can contain labels. These labels are local to the **lambda-primitiveS** and can be branched to from other places in the *expr-list*. However, the only way these labels can be used is in a **go**. Further, the OSS macro package assumes that if **L** is a label in the *expr-list*, every instance of **(go L)** anywhere in the *expr-list* refers to this label.

- **prologS &rest** *expr-list*

This form specifies a number of expressions which should only be evaluated once before the containing **lambda-primitiveS** begins evaluation. (The form can only appear

at top level in the *expr-list* of a **lambda-primitiveS**.) The following shows how **prologS** could be used to specify an OSS function analogous to **Rsum**. (This is intended merely as an illustration of **lambda-primitiveS**; the operation could be defined more easily using **ReduceF**.)

```
(funcallS #'(lambda-primitiveS (numbers) (number) (number)
                (declare (type oss numbers))
              (prologS (setq number 0))
              (setq number (+ number numbers)))
            [1 2 3]) ⇒ 6
```

A feature of **lambda-primitiveS** which is not highlighted in the discussion above is that the auxiliary variables make it possible for **lambda-primitiveS** to support the notion of having one or more internal state variables. (Using the predefined OSS functions, the capability is only available indirectly by using **EnumerateF**, **ReduceF**, or **TscanF**.) The following shows how **lambda-primitiveS** could be used to define an OSS function **GenerateF** which is the same as **EnumerateF** except that it does not take a test argument. (This is an essential example of using **lambda-primitiveS**.)

```
(defun-primitiveS GenerateF (init fn) (items) (state items)
    (declare (type oss items))
  (prologS (setq state init))
  (setq items state)
  (setq state (funcall fn state)))
(generateF '(a b) #'cdr) ⇒ [(a b) (a) () ...]
```

In general, new higher-order OSS functions can be defined by using **funcall** nested in a **lambda-primitiveS** as shown above. The OSS macro package takes special steps to insure that efficient loop code will be produced corresponding to such a **funcall** and that quoted macro names can be used as a value for an input like **fn**.

- **epilogS &rest** *expr-list*

  This form specifies a number of expressions which should only be evaluated once after the containing **lambda-primitiveS** finishes evaluation. (The form can only appear at top level in the *expr-list* of a **lambda-primitiveS**.) The following shows how **epilogS** could be used to specify an OSS function analogous to **Rlist**. (This is intended merely as an illustration of **lambda-primitiveS**; the operation could be defined more easily by applying **nreverse** to the output of a **ReduceF**.)

```
(funcallS #'(lambda-primitiveS (items) (list) (list)
                (declare (type oss items))
              (prologS (setq list nil))
              (setq list (cons items list))
              (epilogS (setq list (nreverse list))))
            [a b c]) ⇒ (a b c)
```

- **terminateS**

  This form specifies that the containing **lambda-primitiveS** should terminate its evaluation. (The form can only appear in the *expr-list* of a **lambda-primitiveS**. However, it

need not be at top level.) The use of **terminateS** in a **lambda-primitiveS** implies that the OSS function being specified is an early terminator. The following shows how **terminateS** could be used to specify an OSS function analogous to **Elist**. The aux variable **state** is needed, because it is not possible to assign to an input variable. (This is intended merely as an illustration of **lambda-primitiveS**; the operation could be defined more easily using **EnumerateF**.)

```
(funcallS #'(lambda-primitiveS (list) (items) (state items)
              (declare (type oss items))
            (prologS (setq state list))
            (if (null state) (terminateS))
            (setq items (car state))
            (setq state (cdr state)))
          '(a b c)) ⇒ [a b c]
```

- **next-inS** *var* **&rest** *expr-list*

This form specifies that *var* is an off-line input of the containing **lambda-primitiveS**. (The form can only appear at top level in the *expr-list* of a **lambda-primitiveS**.) The *var* must be on the *input-list* and must be declared to be an OSS variable. If a given input variable appears in a **next-inS**, it can only appear in a single **next-inS**. If it does not appear in a **next-inS**, it is an on-line input.

If an input variable appears in a **next-inS**, then the position of the **next-inS** indicates the point at which the elements of the corresponding series become available. All other uses of the variable must be after this point. Each time the **next-inS** is executed, a new series element is read from the input. The *expr-list* specifies what should be done if the OSS series in the variable runs out of elements. (It is the programmer's responsibility to insure that the **next-inS** is never again executed after the series has run out of values. If this happens, arbitrarily bad errors can occur.) If no *expr-list* is specified, then the OSS function being defined will terminate when the series runs out. If an *expr-list* is specified which does not cause termination, then the OSS function will not be a passive terminator. The following shows how **next-inS** could be used to specify an OSS function analogous to a two argument version of **Tconcatenate**. (This is an essential example of **lambda-primitiveS**.)

```
(funcallS #'(lambda-primitiveS (Nitems1 Nitems2) (items) (items done)
              (declare (type oss Nitems1 Nitems2 items))
            (prologS (setq done nil))
            (if done (go D))
            (next-inS Nitems1 (setq done T) (go D))
            (setq items Nitems1)
            (go F)
          D (next-inS Nitems2)
            (setq items Nitems2)
          F)
        [a b c] [d e]) ⇒ [a b c d e]
```

- **next-outS** *var*

This form specifies that *var* is an off-line output of the containing **lambda-primitiveS**. (The form can only appear at top level in the *expr-list* of a **lambda-primitiveS**.) The *var*

must be on the *output-list* and must be declared to be an OSS variable. If a given OSS output variable appears in a **next-outS**, it can only appear in a single **next-outS**. If it does not appear in a **next-outS**, it is an on-line output.

If an output variable appears in a **next-outS**, then the position of the **next-outS** indicates the point at which the elements of the corresponding series become available for output. A value must be assigned to the variable before this point. Each time the **next-inS** is executed, a new series element is written into the output. The following shows how **next-outS** could be used to specify an OSS function analogous to a two argument version of **TsplitF**. (This is an essential example of **lambda-primitiveS**.)

```
(funcallS #'(lambda-primitiveS (items pred)
                (Nitems1 Nitems2) (Nitems1 Nitems2)
              (declare (type oss items Nitems1 Nitems2))
            (if (not (funcall pred items)) (go D))
            (setq Nitems1 items)
            (next-outS Nitems1)
            (go F)
          D (setq Nitems2 items)
            (next-outS Nitems2)
          F)
        [1 -2 3 -4] #'minusp) ⇒ [-2 -4] [1 3]
```

- **wrapS** *function*

This form specifies a function which places a wrapper around the loop corresponding to the entire OSS expression containing the OSS function begin defined. (The form can only appear at top level in the *expr-list* of a **lambda-primitiveS**.) The argument *function* must be a quoted non-OSS function which is prepared to take in a Lisp expression and return a Lisp expression.

A complex situation arises if the wrapping form binds any variables. The OSS function being defined can refer to these variables by just using their names free in the function. However, the OSS macro package will know nothing about these variables. In particular, it will do nothing to avoid name clashes if the same wrapping form is used twice or if a name clashes with some other variable in the expression. Therefore, **gensym** variables should be used for any bound variables. Also, no guarantees are made about the nesting order of wrapping forms, so one form cannot refer to the variables bound by another. The following shows how **wrapS** could be used when specifying a simplified form of **Rfile**. In this function, the **wrapS** is used to surround the loop corresponding to the containing OSS expression with a **with-open-file**. (This is an essential example of **lambda-primitiveS**.)

```
(defmacro Rfile (name items)
  (let ((file (gensym)))
    `(funcallS #'(lambda-primitiveS (items) (result) (result)
                   (declare (type oss items))
                 (wrapS #'(lambda (body)
                            (list 'with-open-file
                                  '(,file ,name :direction :output)
                                  body)))
                 (print items ,file)
                 (epilogS (setq result T)))
              ,items)))
```

- `alterableS` *var form*

This form specifies that the output *var* can be altered by using `alterS`. (The form can only appear at top level in the *expr-list* of a `lambda-primitiveS`.) The *var* must be on the output list. The *form* can refer to variables on the aux and output lists. However, due to the way the OSS macro package is implemented, it cannot refer to variables on the input list. The *form* must be such that evaluating (`setf` *form new-value*) will change the underlying value copied to *var* to *new-value* without changing the value of *var* itself.

The following shows how `alterableS` could be used to specify the alterability of the output of an OSS function analogous to `Elist`. (This is an essential example of `lambda-primitiveS`.) Note that evaluating `(setf (car parent) new)` changes an element in `list` without changing the value stored in `items`.

```
(let ((l '(a b)))
  (alterS
   (funcallS #'(lambda-primitiveS (list) (items) (state parent items)
                 (declare (type oss items))
               (prologS (setq state list))
               (if (null state) (terminateS))
               (setq parent state)
               (setq items (car state))
               (setq state (cdr state))
               (alterableS items (car parent)))
             l)
   nil)
  l) ⇒ (nil nil)
```

Another aspect of the interaction between `lambda-primitiveS` and alterability is illustrated by the following definition. This definition shows how to define an OSS function analogous to `TselectF` of `#'plusp`. A key aspect of this definition is the fact that the output variable is on the input list. Whenever this is the case, alterability of the output is inherited from alterability of the input. This inheritability applies to the transducers `Tuntil`, `TuntilF`, `Tcotruncate`, `Tsubseries`, `Tremove-duplicates`, `Tselect`, and `TselectF`, because they are all defined with output variables which come directly from input variables.

```
(let ((l '(1 -2 3)))
  (alterS (funcallS #'(lambda-primitiveS (Nnumbers) (Nnumbers) ()
                        (declare (type oss Nnumbers))
                      L (next-inS Nnumbers)
                        (if (not (plusp Nnumbers)) (go L)))
                    (Elist l))
          nil)
  l) ⇒ (nil -2 nil)
```

The forms `prologS`, `epilogS`, `wrapS`, and `alterableS` can appear in any order in the *expr-list* of a `lambda-primitiveS`. However, the order has relatively little to do with when they will be executed. The times at which they will be executed are defined for each construct. The relative order of the different constructs makes no difference. For example, if a `lambda-primitiveS` contains two `prologS` forms, they will be evaluated in the order

they appear. In contrast, if a **lambda-primitiveS** contains a **prologS** and an **epilogS**, it does not matter what relative order they appear in.

- **defun-primitiveS** *name input-list output-list aux-list* {*doc*} {*decl*}* **&body** *expr-list*

  The form **defun-primitiveS** bears the same relationship to **lambda-primitiveS** that **defunS** bears to **lambdaS**. It can be used to define named OSS functions using the facilities of **lambda-primitiveS**.

```
(defun-primitiveS Tplusp (Nnumbers) (Nnumbers) ()
    "Selects the positive elements of an OSS series of numbers"
    (declare (type oss Nnumbers))
  L (next-inS Nnumbers)
    (if (not (plusp Nnumbers)) (go L)))
(Tplusp [1 -2 3]) ⇒ [1 3]
```

4. Bibliography

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

[2] F. Allen and J. Cocke, "A Catalogue of Optimizing Transformations" in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice Hall, 1971.

[3] J. Backus, "Can Programming be Liberated from the Von Neuman Style? A Functional Style and Its Algebra of Programs", *Comm. of the ACM*, 21(8), Aug. 1978.

[4] F. Bellegarde, "Rewriting Systems on FP Expressions That Reduce the Number of Sequences They Yield", *Proc. ACM Symp. on Lisp and Functional Programming*, Aug. 1984.

[5] F. Bellegarde, "Convergent Term Rewriting Systems Can Be Used for Program Transformation", *Proc. Workshop on Programs as Data Objects*, Springer-Verlag LNCS 217, H. Ganziger and N.D. Jones (eds), 1985

[6] T. Budd, *An APL Compiler*, Univ. of Arizona, TR 81-17, Oct. 1981.

[7] T. Budd, "An APL Compiler for a Vector Processor", *ACM Trans. on Programming Languages and Systems*, 6(3), July 1984.

[8] D. Barstow, "Automatic Programming for Streams", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence*, Aug. 1985.

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

[10] D. Friedman and D. Wise, *CONS Should Not Evaluate Its Arguments*, Indiana Tech. Rep. 44, Nov. 1975.

[11] A. Goldberg and R. Paige, *Stream Processing*, Rutgers report LCSR-TR-46, Aug. 1983.

[12] L. Guibas and D. Wyatt, "Compilation and Delayed Evaluation in APL", *Proc. 1978 ACM Conf. on the Principles of Programming Languages*, Sept. 1978.

[13] G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes", *Proc. 1977 IFIP congress*, North-Holland, Amsterdam, 1977.

[14] B. Liskov, *et. al.*, *CLU Reference Manual*, Lecture Notes in Computer Science, 114, G. Goos and J. Hartmanis eds., Springer-Verlag, New York, 1981.

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

[16] N. Prywes, A. Pnueli, and S. Shastry, "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development", *ACM Trans. on Programming Languages and Systems*, 1(2), Oct. 1979.

[17] G. Ruth, S. Alter, and W. Martin, *A Very High Level Language for Business Data Processing*, MIT/LCS/TR-254, 1981.

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

[19] P. Wadler, "Applicative Languages, Program Transformation, and List Operators", *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, Oct. 1981.

[20] P. Wadler, "Listlessness is Better than Laziness; Lazy Evaluation and Garbage Collection at Compile-Time", *Proc. ACM Symp. on Lisp and Functional Programming*, Aug. 1984.

[21] P. Wadler, "Listlessness is Better than Laziness II: Composing Listless Functions", *Proc. workshop on Programs as Data Objects*, Springer-Verlag LNCS 217, H. Ganziger and N. Jones (eds), 1985

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

[23] R. Waters, *LetS: an Expressional Loop Notation*, MIT/AIM-680a, Oct. 1982.

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

[25] R. Waters, "The Programmer's Apprentice: A Session With KBEmacs", *IEEE Trans. on Software Engineering*, 11(11), Nov. 1985.

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

[27] R. Waters, *Synchronizable Series Expressions: Part I: Reference Manual for the OSS Macro Package*, MIT/AIM-958, November 1987

[28] D. Wile, *Generator Expressions*, USC Information Sciences Institute Technical Report ISI/RR-83-116, 1983.

[29] W. Wulf, R. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng.*, 2(4):253–265, December 1976.

[30] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, February 1983.

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

5. Error Messages Concerning Subprimitives

To facilitate the debugging of OSS expressions, this section discusses the various error messages which can be issued by the OSS macro package when processing the subprimitive functions described in Section 3. This section assumes that the reader is familiar with the basic format of the error messages produced by the OSS macro package (see Section 4 of [27] which documents the errors numbered 1–20).

21 **Lambda-primitiveS used in inappropriate context:** *call*

This error message is issued if a **lambda-primitiveS** ends up (after macro expansion of the surrounding code) being used in any context other than as the quoted first argument of a **funcallS**.

22.1 **PrologS used in inappropriate context:** *call*
22.2 **EpilogS used in inappropriate context:** *call*
22.3 **Next-inS used in inappropriate context:** *call*
22.4 **Next-outS used in inappropriate context:** *call*
22.5 **WrapS used in inappropriate context:** *call*
22.6 **AlterableS used in inappropriate context:** *call*

These errors are issued whenever one of the specified forms appears anywhere other than at the top level in a **lambda-primitiveS** or **defun-primitiveS**.

23.1 **Bad lambda-primitiveS input variable:** *var.*
23.2 **Bad lambda-primitiveS output variable:** *var.*
23.3 **Bad lambda-primitiveS aux variable:** *var.*

These errors are issued when a **lambda-primitiveS** input, output, or auxiliary variable is malformed. In particular, they are issued if the variable, is not a symbol, is **T** or **nil**, is a symbol in the keyword package, or is an **&-keyword**. In addition, it is an error if an auxiliary variable is on the input list, or if an output variable is not on the input or auxiliary list.

24 **Malformed next-inS call:** *call*

This error is issued if the arguments to a **next-inS** are anything other than an OSS variable from the input list followed by zero or more expressions. It is also issued if there is more than one **next-inS** for the same input variable.

25 **Malformed next-outS call:** *call*

This error is issued if the arguments to a **next-outS** are anything other than single OSS variable from the output list. It is also issued if there is more than one **next-outS** for the same output variable.

26 **Malformed wrapS call:** *call*

This error is issued if the argument of **wrapS** is anything other than a quoted function.

27 **Malformed alterableS call:** *call*

This error is issued if the arguments of an **alterableS** are anything other than a variable in the output list followed by an expression which does not contain any of the variables on the input list. It is also issued if there is more than one **alterableS** for the same output variable.

6. Index of Subprimitives

This section is an index and concise summary of the subprimitive forms described in this document. Each entry shows the inputs of the form, the page where a detailed description can be found, and a one line description.