MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 680a

February 1983

LetS

An Expressional Loop Notation

by

Richard C. Waters

ABSTRACT

Many loops can be more easily understood and manipulated if they are viewed as being built up out of operations on sequences of values. A notation is introduced which makes this viewpoint explicit. Using it, loops can be represented as compositions of functions operating on sequences of values. A library of standard sequence functions is provided along with facilities for defining additional ones.

The notation is not intended to be applicable to every kind of loop. Rather, it has been simplified wherever possible so that straightforward loops can be represented extremely easily. The expressional form of the notation makes it possible to construct and modify such loops rapidly and accurately. The implementation of the notation does not actually use sequences but rather compiles loop expressions into iterative loop code. As a result, using the notation leads to no reduction in run time efficiency.

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 Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, or the United States Government.

Introduction

This paper presents an expressional loop notation based on the ideas described in [16,17]. The notation makes it possible to represent loops as compositions of functions applied to sequences of values. The principal benefit of the notation is that it brings the powerful metaphor of expressions and decomposability to bear on the domain of loops. Wherever this metaphor can be applied, it makes algorithms much easier to construct, understand, and modify.

The paper is divided into five parts. The first part discusses what it means to view a loop as an expression composed of functions operating on sequences of values. It then presents the major features of the notation in terms of the expressional metaphor. It concludes with a large example which shows the way the notation is intended to be used.

The most straightforward way to implement the expressional notation would be to simply implement sequences as concrete data objects and then operate on them with ordinary functions. Unfortunately, this approach entails unacceptable time and space overheads which would render the notation impractical to use. In order to provide for efficient execution, the notation has been carefully designed so that a macro preprocessor can convert loop expressions into highly efficient iterative loop code. This conversion process is summarized in the second part of the paper, and discussed in detail in Appendix A.

The iterative loops which result from the conversion process operate on the sequences in a loop expression in parallel, computing each one an element at a time. As is discussed in the third part of the paper, this element at a time metaphor is made an explicit part of the expressional loop notation for two reasons. First, the user must be aware of the execution order which will be used when a loop is evaluated in order to be able to understand the results of side-effect producing operations such as input/output. Second, the element at a time metaphor is in itself a very convenient way to think about many loops. Its explicit introduction into the notation increases the range of loops which can be conveniently represented.

The third part of the paper concludes with a discussion of the limits of the applicability of the notation. The expressional loop notation is not intended to be applicable to every kind of loop. Rather, it is designed to make it particularly easy to represent and manipulate the kind of straightforward loops which appear most commonly in programs. By focusing on the main concept and resisting the temptation to add embellishments, the notation is rendered semantically clean, and easy to understand.

The expressional loop notation has been implemented as a LispMachine [18]/MacLisp [9] macro package LETS ("*let ess*"). (Note that several of the macros described in this paper end in the letter "S". This "S" stands for "sequence", and in all cases it is pronounced separately.) This paper discusses the notation in the context of this particular implementation and the examples are all couched in terms of Lisp. However, none of the basic concepts behind the notation have anything to do with the Lisp language *per se.*

The fourth part of the paper summarizes the basic features of the notation and argues that the notation could be implemented as a logical extension to almost any language. As a specific example, introduction of the notation as an extension to the language Ada [1] is discussed.

The fifth and final part of the paper presents a comprehensive comparison between the expressional notation and other looping constructs. The concept of expressional loops presented here was motivated by observing regularities in the kinds of straightforward loops which appear in programs most often [16]. Over the years, many language designers have also noticed various aspects of these regularities and therefore many of the key features of the expressional notation appear in one form or another in currently existing looping constructs. The constructs which are most similar appear in the languages APL [10], Hibol [13], and Model [11]. The advantage of the notation presented here is that it distills these concepts into a semantically complete whole which is easy to understand, easy to execute efficiently, and easy to add as an extension to current languages.

I - The Expressional Metaphor

The key property of expressions which makes them particularly easy to construct, manipulate, and understand is *decomposability*. Given an expression, it is easy to decompose it into separate parts each of which (in the absence of side-effects) can be completely understood in isolation from all of the other parts. Further, the behavior of the expression as a whole is merely the composition of the behaviors of its parts.

Consider the expression "(SIN (SQRT X))". Its two parts can be understood in isolation. For example, you can understand what the SQRT does (i.e., compute the square root of its input) without having to think about where its input comes from, where its output will be used, or about anything else that is going on in the expression. The only interaction between the two functions is the data flow between them. In order to understand what the expression as a whole does, (i.e., compute the sine of the square root of its input) you merely have to compose your understandings of the two functions.

Viewing Loops as Expressions Involving Sequences

In order to represent loops as expressions, the concepts of *sequences* and *sequence functions* which operate on them are introduced. In this context, all other data structures are referred to as *unitary*. A sequence is an ordered (possibly infinite) one dimensional series of slots containing unitary data objects. A sequence function is a function which produces one or more sequences as outputs and/or consumes one or more sequences as inputs. Loops are represented as expressions built out of sequence function applications.

For reasons of efficiency, sequences are not represented as actual data structures at run time. Rather, expressions involving sequences are compiled into iterative loops in which the existence of the sequences is only implicit. This is analogous to the way in which many program constructs are handled by compilers. For example, references to components of a record structure in a program typically appear to pass indirectly through the structure as a whole. However, for efficiency, such references are generally compiled into direct accesses on the components as if they were atomic objects. The existence of the structure as an identifiable unit is only implicit in the compiled code.

Sequences and sequence functions exist as explanatory devices. The point is that thinking of loops as compositions of functions operating on sequences makes them easier to understand. The fact that the compiled form is very different is in general of no import. (The third part of this paper discusses situations where the user does have to be cognizant of the compiled form.)

Consider the program SUM-POSITIVE-EXPRESSIONAL below. Its body is a sequence expression which sums up the positive elements of a one dimensional array. Given an array containing <0 1 -1 2 -2> the program would produce the result 3.

```
(defun sum-positive-expressional (vector)
  (Rsum (Fpositive (Evector vector))))
```

The sequence function EVECTOR ("*ee vector*") takes in a one dimensional array and enumerates a sequence of the data items in the array (e.g., producing the sequence [0 1 -1 2 -2]). (Note that most of the names of the built-in sequence functions begin with prefix letters. These letters indicate the type of operation performed by the sequence function. The letter "E" stands for enumerate, "G" stands for generate, "F" stands for filter, and "R" stands for reduce. In each case, these prefix letters are pronounced separately.)

The sequence function FPOSITIVE ("*ef positive*") takes in a sequence and filters it producing a sequence containing only the positive elements in the input sequence (e.g., producing [_ 1 _ 2 _]). In order to make it possible to compile a filter efficiently, its action is encoded by leaving some of the slots in the output sequence be empty (symbolized by "_") rather than by creating a sequence of reduced length. In order to make this work, everything is set up so that empty slots are ignored in subsequent computations.

The sequence function `RSUM` ("*ar sum*") takes in a sequence of integers and reduces it to a unitary object containing their sum (e.g., 3). The sequence expression above is easy to understand because the actions of the sequence functions can be understood in isolation from each other, and the action of the expression as a whole (i.e., to sum the positive elements of a vector) is simply the composition of these actions. Further, it is as easy to modify as any other expression.

Simple Examples of Sequence Functions

This section presents a number of built-in sequence functions which are used in examples in the rest of this paper. The complete set of built-in sequence functions provided as part of the `LETS` macro package is presented in Appendix B. There are three basic kinds of sequence functions: *unitary→sequence*, *sequence→unitary*, and *sequence→sequence*. The most common kind of *unitary→sequence* function takes some aggregate data object and creates a sequence of its components.

`Elist` *list*

Takes in a list and creates a sequence of its elements.

e.g., `(Elist '(1 2 3)) => [1 2 3]`

`Esublists` *list*

Takes in a list and creates a sequence of its successive sublists.

e.g., `(Esublists '(1 2 3)) => [(1 2 3) (2 3) (3)]`

`Evector` *vector* `&optional (`*first* `0) (`*last* `(1- (array-length` *vector*`)))`

Takes in a one dimensional array and creates a sequence of its elements.

e.g., `(Evector <1 2 3>) => [1 2 3]`

`Efile` *file-name*

Creates a sequence of values by reading all of the objects out of a file.

e.g., `(Efile "data.lisp") => [1 2 3]`
`if the file "data.lisp" contains "1 2 3 "`

Another family of *unitary→sequence* functions computes a sequence of values according to some formula:

`Erange` *first last* `&optional (`*step-size* `1)`

Creates a sequence of integers by counting from *first* to *last* by the positive increment *step-size*.

e.g., `(Erange 4 8 2) => [4 6 8]`

`Gsequence` *object*

Generates an infinite sequence all of whose elements are *object*.

e.g., `(Gsequence 'A) => [A A A ...]`

The most common kind of *sequence→unitary* function takes in a sequence and combines the elements in it together into an aggregate data structure.

`Rlist` `&sequence` *sequence*

`CONS`es the non-empty values in a sequence into a list.

e.g., `(Rlist [1 2 _ 3]) => (1 2 3)`

Rvector *vector* &sequence *sequence* &unitary &optional (*first* 0) (*last* (1- (array-length *vector*)))

Stores the non-empty values in a sequence into successive slots of a one dimensional array.

e.g., (Rvector <A B C D> [1 2 _ 3]) => <1 2 3 D>

Rfile *file-name* &sequence *sequence*

Writes the non-empty values in a sequence into the indicated file.

e.g., (Rfile "data.lisp" [1 2 _ 3]) => T

"<cr>1 <cr>2 <cr>3 " is printed in "data.lisp"

Another kind of *sequence→unitary* function computes a summary value based on a sequence.

Rsum &sequence *integers*

Computes the sum of the non-empty integer values in a sequence.

e.g., (Rsum [1 2 _ 3]) => 6

Rcount &sequence *sequence*

Counts the number of non-empty slots in a sequence.

e.g., (Rcount [A B _ C]) => 3

Rlast &sequence *sequence* &unitary &optional (*default* NIL)

Returns the last non-empty element (if any) of the sequence as its value; otherwise returns *default*.

e.g., (Rlast [A B C _]) => C

Sequence→sequence functions take in a sequence of values and compute some related sequence. They tend to be much more idiosyncratic than other kinds of sequence functions and very few are predefined.

Fpositive &sequence *numbers*

Selects the positive non-empty elements of a sequence of integers.

e.g., (Fpositive [-1 0 _ 1]) => [_ _ _ 1]

The programs below give a number of examples of loops built up out of the sequence functions described above. COPY-LIST copies a list by enumerating the items in the list and then CONSing them up into a new list. LAST enumerates all of the sublists in a list and then returns the last one. SUM-FIRST-N adds up the first N integers by enumerating these integers and then summing the resulting sequence of values.

```
(defun copy-list (list)
  (Rlist (Elist list)))

(defun last (list)
  (Rlast (Esublists list)))

(defun sum-first-n (n)
  (Rsum (Erange 1 n)))
```

FILE-LENGTH computes the number of objects in a file by enumerating them and then counting the items in this sequence. DUMP-VECTOR prints the elements of a vector into a file. ZERO-VECTOR initializes a vector by setting the elements to zero. It uses GSEQUENCE in order to generate a sequence of zeros to use.

```
(defun file-length (file-name)
  (Rcount (Efile file-name)))

(defun dump-vector (file-name vector)
  (Rfile file-name (Evector vector)))

(defun zero-vector (vector)
  (Rvector vector (Gsequence 0)))
```

MapS

The most fundamental and often used sequence function is MAPS which is a generalization of the Lisp function MAP.

mapS *function* &rest &sequence *args*

Creates a sequence by applying *function* to the elements of one or more input sequences.
e.g., (mapS #'+ [1 _ 3] [4 5 6 7]) => [5 _ 9]

The number of sequences provided must be compatible with the number of arguments required by *function.* The nth element of the output sequence is computed by applying *function* to the nth elements of the input sequences. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. The length of the output sequence is the same as the length of the shortest input sequence.

The use of MAPS is illustrated by the following two functions. The program PAIRWISE-MAX takes in two lists and creates a list where each element is the maximum of the corresponding elements in the two input lists. The program TIMES-N multiplies every element in a list by a parameter N. (Note that if a functional argument to a sequence function is a quoted LAMBDA expression, then it is compiled inline. As a result, it can refer to local variables (such as N) without having to declare them special.)

```
(defun pairwise-max (list1 list2)
  (Rlist (mapS #'max (Elist list1) (Elist list2))))

(defun times-n (list n)
  (Rlist (mapS #'(lambda (x) (* x n)) (Elist list))))
```

The use of MAPS in loop expressions is so common that a syntactic sugaring has been introduced to make this easier. Whenever a unitary expression is applied to sequences or appears in an environment where a sequence value is expected then the entire expression down to, but not including, any components which create sequences is separated out as a LAMBDA expression and MAPSed. This is illustrated by the following alternate definitions for the functions used as examples above.

```
(defun pairwise-max-implicit (list1 list2)
  (Rlist (max (Elist list1) (Elist list2))))

(defun times-n-implicit (list n)
  (Rlist (* (Elist list) n)))
```

Due to the requirements of efficient execution, sequences are not actually implemented as concrete data objects, and therefore ordinary lisp functions cannot in any case be applied to them as whole entities. The implicit introduction of MAPS gives a useful meaning to expressions which would otherwise be meaningless.

LetS*

In an ordinary expression, if you want to use the value of a subexpression in two places, you have to bind this value to a variable. The prototypical way to do this in Lisp is with the macro LET. The macro LETS* (which is analogous to LET*) makes it possible to create variables which have sequences as their values. As shown below, the macro consists of a list of variable/value pairs and a body which consists of one or more loop expressions. Each initializing value must be a sequence. LETS* cannot be used to bind a variable to a unitary value. (However, using GSEQUENCE, you can bind a sequence variable to an infinite sequence of a unitary value, which will usually be sufficient.)

```
(letS* ((variable value) ...)
  loop-expression ...)
```

The expressions in the body of a LETS* can refer to the sequences bound to the sequence variables, and can result in either sequence or unitary values. The value of the last form must be unitary and is returned as the result of the LETS*. An important feature of LETS* is that all of the loop expressions in it are combined together into a single loop. This will be discussed in more detail in the section on the element at a time metaphor.

The use of LETS* is illustrated by the function SQUARE-ALIST (below) which takes in an alist of keys and integers and creates a new alist with each integer squared. Note that the value of ENTRY is used when computing (using implicit MAPSing) the value for SQUARE.

```
(defun square-alist (alist)
  (letS* ((entry (Elist alist))
          (square (* (cdr entry) (cdr entry))))
    (Rlist (list (car entry) square))))
```

The macro LETS* supports destructuring as shown in the program SQUARE-ALIST-DESTRUCTURING. This program also illustrates the fact that you can use SETQs and other assigning forms (e.g., PSETQ, MULTIPLE-VALUE, etc.) in the body of a LETS* in order to assign sequence values to sequence variables. In addition, SETQs and other assigning forms can be used to assign to any number of free (unitary) variables in the body of a LETS*. This is often useful for passing information out of a loop.

```
(defun square-alist-destructuring (alist)
  (letS* (((key . i) (Elist alist)))
    (setq i (* i i))
    (Rlist (list key i))))
```

DefunS

The macro DEFUNS makes it possible for a user to created new sequence functions which he can then use in loop expressions. These sequence functions are actually macros which are compiled inline by the LETS macro package. However, in the context of the expressional notation, they are intended to be thought of as functions just like any other function.

```
(defunS name lambda-list
  . body)
```

The macro DEFUNS is analogous to DEFUN. It has two basic parts: a lambda-list and a body. The lambda-list is a list of variable names and keywords. In addition to the standard keywords &OPTIONAL, &REST, and &AUX it supports two new keywords &UNITARY and &SEQUENCE. These two new keywords are used to specify whether a particular parameter is a sequence or an ordinary unitary object. Each of the keywords is *sticky* and specifies the type of all of the parameters which follow it until another keyword changes the type. By default, the parameters are initially unitary.

The body of a DEFUNS is the same as the body of a LETS* except that the last form is not required to yield a unitary value. The value of the last form, be it unitary or sequence, is returned as the value of the sequence function being created. Note, that DEFUNS is completely different from LETS* in that it creates a sequence function which can later be combined together with other sequence functions while LETS* creates an actual loop.

The following examples illustrate the use of DEFUNS. RALIST takes in a sequence of keys and a sequence of values and CONSes them up into an alist. Note that ELIST and EVECTOR return sequences while RALIST returns a unitary value. EVECTOR takes in two optional arguments which specify a region of the vector to enumerate.

```
(defunS Ralist (&sequence key value)
  (Rlist (cons key value)))
(defunS Elist (list)
  (car (Esublists list)))
(defunS Evector (vector &optional (start 0) (end (1- (array-length vector))))
  (aref vector (Erange start end)))
```

The following definition for MAPS is meta-circular in that it uses implicit introduction of MAPS in order to define MAPS. However, it illustrates the use of functional and &REST arguments in a sequence function. It should be noted that the LETS macro package uses special knowledge of APPLY and FUNCALL in order to compile uses of functional arguments into highly efficient inline code as long as the arguments supplied are quoted function names or LAMBDA expressions.

```
(defunS mapS (function &rest &sequence args)
  (apply function args))
```

The ability for the user to conveniently define his own named sequence functions is a particularly important part of the expressional loop notation. It makes it possible for him to extend the notation to deal with the particular data abstractions he creates. A detailed example of this is given in a later section.

Seven Basic Sequence Functions

The expressional loop notation provides seven basic sequence functions which embody the basic operations supported by the notation. Every other sequence function is defined in terms of these basic ones. The most fundamental sequence function (MAPS) has already been discussed. Three sequence functions are available for specifying computation to be performed before and after the execution of a loop.

at-start *function* &rest *args*
    Applies *function* to *args* in the initialization code before the loop begins.

at-end *function* &rest *args*
    Applies *function* to *args* in the epilog code after the loop ends.

at-unwind *function* &rest *args*
    Applies *function* to *args* in an UNWIND-PROTECT wrapped around the loop.

The key difference between AT-END and AT-UNWIND is that AT-UNWIND operations will be performed no matter how a loop is exited, while AT-END operations will not be performed if a loop is terminated in some extraordinary way (e.g., by a THROW).

The following definition of the sequence function RFILE illustrates the use of the three sequence functions above. AT-START is used to open the file before processing starts. AT-END is used to set a flag which indicates that processing has terminated normally, and to specify that the unitary value T should be returned as the value of the sequence function as a whole. AT-UNWIND is used to insure that the file will be properly closed no matter how the loop is terminated.

```
(defunS Rfile (name &sequence items &aux &unitary file (finished nil))
  (at-start #'(lambda () (without-interrupts (setq file (open name 'out)))))
  (let (prinlevel prinlength)
    (print items file))
  (at-unwind #'(lambda ()
                 (cond (finished (close file))
                       ((null file) nil)
                       ((y-or-n-p "delete partial output file")
                        (send file ':close ':abort))
                       (T (close file)))))
  (at-end #'(lambda () (setq finished T))))
```

The purpose of defining a sequence function is to group together into a single unit a standard fragment of looping algorithm. AT-START, AT-END, and AT-UNWIND make it possible to include initializing and epilog computation as part of an individual sequence function. This extends the range of loop computation fragments which can be expressed as sequence functions. For example, RFILE would be conceptually much less useful if it did not encapsulate the relatively complex actions needed in order to open and close the file in a completely reliable way into the same unit with printing out the objects.

Often, uses of AT-START and AT-END are implicit in loop expressions. For example, the unitary inputs of a sequence function must be available before a loop can begin. As a result, the calls on MAKE-ARRAY and ARRAY-LENGTH in the program COPY-VECTOR (below) are computed at the beginning of the loop even though AT-START is not used explicitly. Similarly, the unitary outputs of sequence functions are not available until after the loop terminates. As a result, the SORT in RLIST-SORTED is computed in the epilog code at the end of the loop even though AT-END is not used explicitly. It should be noted that the explicit use of AT-START and AT-END is actually very seldom necessary.

```
(defun copy-vector (vector)
  (Rvector (make-array (array-length vector)) (Evector vector)))

(defunS Rlist-sorted (&sequence symbols)
  (sort (Rlist symbols) #'alphalessp))
```

The basic idea of filtering a sequence to produce a restricted sequence is supported by the sequence function FILTERS.

filterS *function* &sequence *source* &rest *args*

Selects the elements of *source* corresponding to non-NIL values of *function* applied to the inputs.

e.g., (filterS #'> [1 _ 2 3 4] [0 0 _ 0] => [1 _ _ 4]

The elements of the output sequence of FILTERS are computed as follows. If the result of applying *function* to the nth elements of the input sequences (the *source* and *args*) is non-NIL then the nth element of the *source* is used as the nth element of the output; otherwise the nth output element is empty. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty.

Note that the output sequence is exactly the same length as the shortest input sequence; however, some of the output sequence slots may be empty. The idea of empty slots is the foundation for the notion of filtering being presented here. In order to make empty slots work correctly, all of the basic sequence functions are restricted so that the empty slots in their inputs are propagated to their outputs. The use of FILTERS is illustrated by the following definition of FPOSITIVE.

```
(defunS Fpositive (&sequence integers)
  (filterS #'plusp integers))
```

The basic capability of truncating the length of a sequence is embodied in the sequence function TERMINATES. It takes in one or more sequences and produces a sequence which is potentially shorter than any of them. Every other basic sequence function which computes sequences from sequences produces an output which is the same length as the minimum length of its inputs. As will be discussed below, the primary use for TRUNCATES is to reduce infinite sequences to finite length.

truncateS *function* &sequence *source* &rest *args*

Truncates *source* at the first point where the value of *function* applied to the inputs is non-NIL.
e.g., (truncateS #'< [1 _ 2 3] [0 0 _ 4]) => [1 _ _]
e.g., (truncateS #'> [1 _ 2 3] [0 0 _ 4]) => []

In order to compute the output of TRUNCATES, *function* is applied to successive groups of corresponding elements of the input sequences. The output sequence is composed of the elements of the *source* up to but not including the first element corresponding to a non-NIL evaluation of *function*. As with the other sequence functions, if any of the nth elements of the input sequences are empty then *function* is not applied and the nth output element is empty.

The seventh basic sequence function (PREVIOUS) makes it possible to access sequence values from the previous cycle of a loop.

previouS *init function* &sequence &rest *args*

Creates a sequence (whose first value is *init*) by applying *function* to the previous values of the inputs.
e.g., (previouS NIL #'ncons [A B C]) => [nil (A) (B)]
e.g., (previouS NIL #'ncons [_ A _ B C]) => [_ nil _ (A) (B)]

If there are no empty slots in any of the input sequences, then the first element of the output of PREVIOUS is the value *init* and the nth element of the output is computed by applying *function* to the (n-1)th elements of the input sequences. If there are empty slots then this is generalized as follows. The nth slot of the output is empty if and only if the nth slot of any input is empty. The first non-empty slot of the output contains *init*. after that each non-empty slot is computed by applying *function* to the previous group of non-empty input values. The length of the output sequence is the same as the length of the shortest input sequence. (Note that *function* is applied to the last values of the input sequences even though the result is not part of the output.)

The primary use of PREVIOUS is to define sequence functions which have feedback between cycles of a loop. Three sequence functions are defined which embody this feature. The sequence function REDUCES (shown below) is a generalized *sequence→unitary* function. It has an internal state variable. The nth value of the state is computed by applying *function* to the (n-1)th value of the state (accessed by a call on PREVIOUS which uses *seed* as an initial seed value for the state) and the nth element of the input. However, if the nth element of the input sequence is empty then *function* is not applied and the state is not changed. When the input sequence is exhausted, the final value of the state variable is returned as the (unitary) result. If there are no non-empty elements in the input sequence then the value *seed* will be returned. The use of REDUCES is illustrated by the following definition of RLIST.

```
(defunS reduceS (fn seed &sequence sequence &aux state)
  (setq state (mapS fn (previouS seed #'(lambda (x) x) state) sequence))
  (Rlast state seed))

(defunS Rlist (&sequence items)
  (nreverse (reduceS #'xcons nil items)))
```

The sequence function GENERATES is a generalized *unitary→sequence* function. It takes in a function and a seed value which is used to initialize an internal state variable. The nth value of the state is computed by applying *function* to the (n-1)th value of the state. The output sequence is composed of the successive values of the state starting with the initial value *seed*. Note that the sequence produced is infinite in extent.

```
(defunS generateS (fn seed &aux &sequence state)
  (setq state (previouS seed fn state)))

(defunS Gsequence (object)
  (generateS #'(lambda (x) x) object))
```

A loop expression which contains only a generator will never terminate because it operates on an infinite sequence. However, if a loop expression is working on several sequences some of which are finite and some of which are not, it will terminate as soon as the shortest finite sequence has been exhausted. This is discussed further in the section on termination below.

Generators are typically used in loop expressions in conjunction with finite sequences of unknown length. For example, the program DIGITS-TO-NUMBER takes in a vector of one digit numbers (ordered least significant digit first) and computes the corresponding integer (e.g., <3 2 1> becomes 123). The loop expression works with two basic sequences. It enumerates the digits in the vector. It also creates an unbounded sequence of scale factors consisting of the successive powers of ten. The result is computed by summing up the product of each digit with the appropriate scale factor. The loop terminates when the digits run out.

```
(defun digits-to-number (v)
  (Rsum (* (Evector v) (generateS #'(lambda (x) (* x 10.)) 1))))
```

The sequence function ENUMERATES combines GENERATES and TRUNCATES. It is the preferred way to define enumerators such as ELIST.

```
(defunS enumerateS (trunc-fn gen-fn seed)
  (truncateS trunc-fn (generateS gen-fn seed)))

(defunS Elist (list)
  (car (enumerateS #'null #'cdr list)))
```

Conversions and Coercions

Two sequence functions are available for converting between unitary values and sequences: GSEQUENCE which converts an object into an infinite sequence of that object, and RLAST which converts a sequence into a unitary object by taking its last element. The use of GSEQUENCE is illustrated in the program VECTOR-NCONS which fills a vector with an NCONS of NIL. Note that the sequence MAPS can also be used to create a sequence of objects by specifying the repeated execution of a function of no arguments as in the function VECTOR-NCONSES. However, MAPS is very different from GSEQUENCE in that it calls for repeated execution of the function. As a result, VECTOR-NCONSES fills each slot of the vector with a different CONS cell while VECTOR-NCONS fills each slot with the same CONS cell.

```
(defun vector-ncons (vector)
  (Rvector vector (Gsequence (ncons nil))))

(defun vector-nconses (vector)
  (Rvector vector (mapS #'(lambda () (ncons nil)))))
```

In order to make things more convenient for the user, automatic type coercions are applied between sequences and unitary values. The most important coercion has already been discussed. Whenever a unitary expression is applied to sequences or placed where a sequence value is required, MAPS is automatically introduced in order to convert it into a sequence expression. Note that GSEQUENCE is never automatically introduced and therefore VECTOR-NCONSES-IMPLICIT is equivalent to VECTOR-NCONSES, and not to VECTOR-NCONS.

```
(defun vector-nconses-implicit (vector)
  (Rvector vector (ncons nil)))
```

In the reverse direction, whenever a sequence expression is placed where a unitary value is required, RLAST is automatically introduced to convert it into a unitary value producing expression. The places where unitary values are required are the last expression in the body of a LETS* and the value of loop expressions which appear in isolation in ordinary Lisp code. Examples are shown below.

Some of the other features of the expressional notation could also be looked at as coercions, for example, the automatic introduction of AT-START and AT-END discussed in the beginning of the last section. Taken together, these coercions have no semantic import -- they do not make it possible to express anything which could not be expressed without them. However, they do make it significantly more convenient to specify many kinds of loops.

Nested Loops

Like any looping notation, the expressional notation can be used to express nested loops. Consider the program SUM-VECTORS-IN-LIST-MAPS. It takes in a list of vectors of integers (e.g., (<1 2> <3 4>)) and returns a list of the sums of these vectors (e.g., (3 7)). The outer loop enumerates the vectors of numbers in the list supplied as the input to the function as a whole. The inner loop adds up the numbers in these vectors. The outer loop then CONSes these sums up into a list to be returned.

```
(defun sum-vectors-in-list-mapS (list-of-vectors)
  (letS* ((vector (Elist list-of-vectors))
          (sum (mapS #'(lambda (l) (Rsum (Evector l))) vector)))
    (Rlist sum)))
```

In the program, MAPS is used to apply the inner loop to each list of numbers in turn. The LAMBDA used with the MAPS delineates the boundary of the inner loop. This could also be done by wrapping a LETS* around the inner loop (which would then be implicitly MAPSed) as in SUM-VECTORS-IN-LIST-LETS.

```
(defun sum-vectors-in-list-letS (list-of-vectors)
  (letS* ((vector (Elist list-of-vectors))
          (sum (letS* () (Rsum (Evector vector)))))
    (Rlist sum)))
```

Though relatively clear, both of the above programs are somewhat cumbersome in appearance. If the (RSUM (EVECTOR ...)) were in isolation, there would be no need to wrap it in either a MAPS or LETS*. One might therefore assume that one could right the program as in SUM-VECTORS-IN-LIST-BUGGY; however, this is not the case.

```
(defun sum-vectors-in-list-buggy (list-of-vectors)
  (letS* ((vector (Elist list-of-vectors))
          (sum (Rsum (Evector vector))))
    (Rlist sum)))
```

Note that, there are obvious type conflicts in the program SUM-LIST-IN-VECTORS-BUGGY. The unitary value of RSUM is assigned to the sequence variable SUM and VECTOR is used where EVECTOR expects a unitary value. An early experimental version of the LETS macro package resolved this kind of type conflict by automatically introducing loop nesting. However, experimentation indicated that this was not a good idea. The key problem is that the implicit introduction of nesting can have large unobvious effects on a loop expression. This is particularly true if the type conflicts in a loop expression are do to error rather than intent. In addition, the potential for automatic error detection is markedly reduced by the fact that almost any loop expression (no matter how erroneous) can be given some interpretation in terms of implicitly nested loops.

It should be noted in passing that in programs like SUM-COPY-OF-LIST below, there are no type conflicts and no nesting of loops. Rather, the program simply specifies that one loop (RLIST (ELIST LIST)) is to be executed in order to compute the initial value used by the second loop (RSUM (ELIST ...)).

```
(defun sum-copy-of-list (list)
  (Rsum (Elist (Rlist (Elist list)))))
```

A Large Example

To conclude the description of the features of the expressional loop notation, this section presents a large example. The example is a data abstraction which implements sets of symbols as bit vectors. The abstraction not only makes available some ordinary functions for operating on these sets, but some sequence functions as well.

Sets are represented as bits packed into a single integer. The size of the sets is limited by the number of bits in an integer (e.g., 24 bits on the LispMachine). The global variable *BSET-DOMAIN* stores the correspondence between potential set elements and bit positions. This mapping is represented by a vector of CONSes. The index of a CONS in the vector indicates the bit position which is being described. The CAR of the CONS holds the symbol which corresponds to the bit position. The CDR of the CONS holds the representation for a set which has only that one symbol in it (i.e., an integer with only the one corresponding bit on). The variable *BSET-DOMAIN* is initialized to a vector of CONSes of NIL and the appropriate single element sets. Note that the unit sets are created by a special generator which starts with an integer with a 1 in bit position 0 and then rotates this bit around from position to position.

```
(defvar *bset-domain*
  (Rvector (make-array 24) (cons nil (generateS #'(lambda (x) (rot x 1)) 1)))
  "The bset domain element mapping.")
```

The global variable *BSET-INDEX* keeps track of the largest bit position used so far. The number -1 is used to represent the fact that no bit positions have been used yet.

```
(defvar *bset-index* -1 "The largest bit position used so far.")
```

The function BSET-RESET is used to reinitialize these variables. It MAPSes over the vector in *BSET-DOMAIN* setting the CAR of each CONS cell to NIL, and sets *BSET-INDEX* to -1.

```
(defun bset-reset ()
  (letS* ((item (Evector *bset-domain*)))
    (setf (car item) nil))
  (setq *bset-index* -1))
```

The function BSET-UNITSET takes in a symbol and returns the unit set corresponding to it. It issues an error if the symbol is not representable as a unit set (i.e., if it is not in the vector *BSET-DOMAIN*). It uses ROR-FAST (which computes the OR of the items in a sequence, stopping as soon as a non-NIL item is encountered) in order to look for the symbol in *BSET-DOMAIN* returning the corresponding unit set as soon as it is found. Note that the COND in the ROR-FAST is implicitly MAPSed.

```
(defun bset-unitset (symbol)
  (or (letS* ((item (Evector *bset-domain* 0 *bset-index*)))
        (Ror-fast (cond ((eq (car item) symbol) (cdr item)))))
      (ferror "The symbol ~A is not in the bset domain" symbol)))
```

The function BSET-ADD-DOMAIN-ELEMENT takes a symbol and enters it in *BSET-DOMAIN* so that it can be used in the bit vector sets. If the symbol is not already in the domain, and if there is an available bit position, then the program increments *BSET-INDEX* and stores the symbol in the appropriate CONS cell in *BSET-DOMAIN*.

```
(defun bset-add-domain-element (symbol)
  (cond ((Ror-fast (eq symbol (car (Evector *bset-domain* 0 *bset-index*)))))
        ((> *bset-index* 22) (ferror "bset domain size exceeded"))
        (T (incf *bset-index*)
           (setf (car (aref *bset-domain* *bset-index*)) symbol))))
```

As examples of the kind of ordinary functions which would be implemented as part of the data abstraction consider the following four. The first three are examples of the operations for which the bit vector implementation is particularly efficient. Intersection, union, and the test for equality between two sets can all be implemented as single operations independent of how many symbols are in the sets operated on.

```
(defun bset-intersect (bset1 bset2)
  (logand bset1 bset2))

(defun bset-union (bset1 bset2)
  (logior bset1 bset2))

(defun bset-equal (bset1 bset2)
  (= bset1 bset2))

(defun bset-mem (symbol bset)
  (not (zerop (bset-intersect (bset-unitset symbol) bset))))
```

The next four definitions are examples of the kind of sequence functions which would be provided as part of the data abstraction. The first two implement reducers which can be used to take the intersection and union of sequences of bit vector sets. The third (EBSET) takes in a bit vector set and creates a sequence of the symbols in that set. The last (RBSET) performs the inverse operation, taking in a sequence of symbols and creating a set by taking the union of the corresponding unit sets.

```
(defunS Rbset-intersect (&sequence bset)
  (reduceS #'(lambda (x) (bset-intersect x bset)) -1))

(defunS Rbset-union (&sequence bset)
  (reduceS #'(lambda (x) (bset-union x bset)) 0))
```

```
(defunS Ebset (bset)
  (car (filterS #'(lambda (x) (not (zerop (bset-intersect (cdr x) bset))))
                (Evector *bset-domain* 0 *bset-index*))))

(defunS Rbset (&sequence symbol)
  (Rbset-union (bset-unitset symbol)))
```

The example in this section is a particularly good one in that it shows the expressional notation being used to represent a variety of loops which are small and simple. This is the application for which the notation has been specifically designed.

II - Efficient Execution

There are many different ways in which the expressional notation could be executed. The most straightforward way would be to implement sequences as normal data objects and the sequence functions as normal functions. Loop expressions could then be evaluated just like any other expressions. This direct execution approach is taken by APL [9]. At the other extreme, a compilation process can be used to convert loop expressions into ordinary iterative loops which operate in an element at a time fashion. This conversion approach is used by the languages Hibol [11,12] and Model [10].

The main advantage of direct execution is that it is easy to implement. In particular, it is very easy to see how it directly supports the expressional metaphor. The main disadvantage of direct execution is that, in comparison with ordinary iterative loops, it imposes very large time and space overheads.

The main advantage of the conversion approach is that it is capable of creating very efficient code. In fact there is no reason why there has to be any time or space overhead at all. There is however a major drawback to this approach. The notation has to be significantly restricted in order to guarantee that conversion will always be possible. (This will be discussed further in the next part of the paper.)

There has been a lot of interesting work which tries to chart a middle course between the simplicity of the direct execution approach and the efficiency of the conversion approach. One way in which this has been done is by representing sequences explicitly, but without trying to compute the elements in them until they are actually needed. This can be done explicitly through coroutines [7], or implicitly through lazy evaluation [4,6]. One unfortunate problem with this approach is that it leaves the execution order only weakly constrained. In fact the actual execution order may well depend on the actual data processed during a particular execution of a loop. As a result, it can be very difficult to predict the interaction between side-effect producing operations (e.g., input/output).

Also, although delayed evaluation is more efficient (particularly in space) than direct execution in many situations it is still much less efficient than complete conversion. Several researchers have pursued an interesting mixed mode approach which provides an interpreted implementation where sequences are represented explicitly and, in addition, provides a compiler which performs conversions to eliminate intermediate sequences whenever possible.

The premier example of this has been the work on compilers for APL [2,5]. Optimizing APL compilers attempt to locate array expressions where the arrays are being used merely as intermediate sequences, and then eliminate the actual computation of these arrays. When an expression corresponding to the kind of simple loop representable by the expressional notation is located, then it is easy to eliminate the intermediate arrays. Wadler's Listless Transformer [14] pursues a similar approach for compiling a Lisp-like language. It takes programs where finite intermediate sequences are represented as lists, and converts them into programs where these intermediate lists do not actually have to be created. The resulting programs can then be efficiently compiled by normal means.

Unfortunately, there are several inherent problems with the partial conversion approach. First, the only reason to pursue partial conversion is that the notation supports features which cannot be practically converted. Unfortunately this raises a whole new problem -- identifying what parts of what loops can be converted. In addition, steps have to be taken to interface loop expressions which have been converted with those which have not.

A second and much more serious problem is that in the presence of side-effects, conversion is not a correctness preserving process. The reason for this is that it entails a radical change in execution order from computing each sequence as a unit to processing several sequences in parallel an element at a time. To deal with this you have to refrain from converting any loop containing side-effects (including input/output).

When designing the expressional notation it was felt that the issue of efficiency could not be ignored. As a result, the notation was designed from the beginning with conversion in mind. This had three major effects on the design. First, the element at a time metaphor was introduced as an explicit part of the notation so that the user would be aware of the execution order introduced by the conversion process. Second, the notation was restricted wherever necessary in order to insure that conversion would always be possible. Third it was possible to introduce generators creating infinite sequences into the notation. These are convenient in many situations, but difficult to implement if direct execution is used. It should be noted that the languages Hibol and Model support somewhat similar notations (described in last part of the paper) which are also suitable for complete conversion.

The Compilation Process

This section gives a brief outline of the compilation process used in the Lisp implementation of the expressional loop notation. The process is described in detail in Appendix A. The following example illustrates the result of the compilation process. The iterative loop which results is composed of a number of separate pieces which are specified by the sequence functions in the loop expression.

```
        (macroexpand '(Rsum (Fpositive (Evector vector))))

yields: (prog T (sum2 element10 index6 f4 last7)
                (setq last7 (1- (array-length vector)))
                (setq index6 0)
                (setq sum2 0)
             L0 (cond ((> index6 last7) (go E0)))
                (setq element10 (aref vector index6))
                (setq f4 (plusp element10))
                (cond (f4 (setq sum2 (+ sum2 element10))))
                (setq index6 (1+ index6))
                (go L0)
             E0 (return-from T sum2))
```

When a loop expression is encountered, it is first parsed in order to locate all of the sequence functions in it. As part of the parsing process, implicit MAPS introduction and other coercion are performed. The loop expression is then compiled by combining all of the sequence functions in it together.

Internally, each sequence function is represented by a data structure specifying some initialization computation to perform before the loop begins, some inside computation to perform repetitively on each cycle of the loop, and some epilog computation to perform after the loop terminates.

A composition of two sequence functions "(A (B...))" is compiled by combining their parts together into a new compound sequence function. The resulting initialization, insides, and epilog are derived by concatenating the corresponding parts of B and A. The data flow from B to A is implemented by data flow from the inside part of B to the inside part of A in the new compound inside part.

III - The Element at a Time Metaphor

In addition to the expressional metaphor, the loop notation described here supports a second *element at a time* computational metaphor. The expressional metaphor is based on the idea that a sequence is a logical unit which is created in its entirety by one sequence function and then consumed by another sequence function. The element at a time metaphor is based on the idea that the computation involving all of the sequences in a single loop proceeds in parallel and that the loop expression is essentially describing what happens on a typical step in this process.

The element at a time metaphor is included as a basis for the proposed notation for three reasons. First, it makes the exact execution order in a loop expression explicit. This makes it possible for users to understand the interaction of side-effect producing operations. Second, the element at a time metaphor makes it possible to conveniently state the restrictions on the notation which are needed in order to guarantee that compilation into efficient code will always be possible. Third, it is an independent metaphor which is often a more convenient way to think about a loop than the expressional metaphor.

The program INVENTORY-REPORT below shows a typical example of a loop expression which is strongly based on the element at a time metaphor. The program reads in a file of inventory records and prints out a report. Each record is a list of four fields: the name of the inventory item, the quantity on hand, the minimum acceptable quantity on hand, and the unit price. For each item the report prints out its name, how many are on hand, and the valuation of these items based on the specified price. The last line of the output reports the total valuation of all of the items. In addition to the above, the report prints out a notification in front of each item which is understocked, indicating how many should be ordered.

Sample Inventory File Contents

```
("Widget"  8.    8.  20.5)
("Frob"    2.    9.  9.68)
("Thingy"  312. 40. 19.65)
("Dingus"  0.   20.  8.25)
("Whatsit" 3.    7.  5.67)
```

Resulting Printout

```
                  Inventory Report

Order?            Name      On Hand   Valuation
                  Widget       8       $164.00
Order:       7    Frob         2        $19.36
                  Thingy     312      $6130.80
Order:      20    Dingus       0         $0.00
Order:       4    Whatsit      3        $17.01

                  Total Valuation:    $6331.17
```

Looking at the loop in the program, note the use of destructuring and sequential assignment in the bound variable value pairs. In the first line of the LETS*, the sequence variable NAME is bound to a sequence of the first field of each record, the variable QUANTITY is bound to a sequence of the second field of each record, etc. The variable VALUATION is bound to a sequence of products of PRICE and QUANTITY.

```
(defun inventory-report ()
  (with-open-file (report "inventory.report" ':out)
    (format report "~10X Inventory Report~2%")
    (format report "Order?     Name      On Hand  Valuation~%")
    (letS* (((name quantity minimum price) (Efile "inventory.data"))
            (valuation (*$ price (float quantity))))
      (cond ((>= quantity minimum) (format report "~10X"))
            (T (format report "Order: ~3D" (- minimum quantity))))
      (format report "~X~10A~4D~2X~10<$~$~>~%" name quantity valuation)
      (format report "~%~10X Total Valuation:~10<$~$~>" (Rsum$ valuation)))))
```

The body of the LETS* prints the main part of the actual report. The first form prints the ordering notifications. It compares the quantity in stock with the minimum required and prints out the number to be ordered if the quantity is less than the minimum. The second form prints the main information about each inventory item. (Note that the FORMAT function is a Lisp function for creating formatted output. Like the Fortran construct it is modeled after, it is inscrutable but convenient.) Both of the first two forms in the body are implicitly MAPSed. The third form prints out the summary line at the end of the report. It is only executed once at the end of the loop because it uses the unitary output of the reducer RSUM$ (floating point sum).

Two aspects of LETS* exist primarily in order to support the element at a time metaphor. First, a key feature of LETS* is that it specifies that all of the expressions in it are to be combined into a single loop. From the point of view of the expressional metaphor this doesn't make any difference. However, it is important in order to delineate the exact group of actions which occur on a typical loop cycle.

Second, the rule for implicit introduction of MAPS is extended by that a unitary expression in a LETS* will be implicitly MAPSed even if it does not refer to any sequences. (The only time that this is not possible is if the expression refers to values which are not available until the end of the loop. In this situation the expression is executed AT-END.) This extension is made so that, as much as possible, all of the lines in a LETS* describe typical actions and not just single actions.

It is interesting to consider the way that the LETS* mixes together the expressional and element at a time metaphors. For the most part, the body specifies the computation to be performed by describing the operations to be performed on a typical inventory record (i.e., for each record get the four fields, compute the valuation, print the number to order if any, and then print the name, quantity, and valuation). However, the LETS* also uses the standard sequence functions EFILE and RSUM. Since these sequence functions operate on sequences as whole entities they make much more sense when looked at from the point of view of the expressional metaphor.

Another interesting aspect of the program INVENTORY-REPORT is that although the process of actually printing out the report (i.e., opening the file, printing some initial lines, printing a group of internal lines, printing a final line, and then closing the file) is clearly a logically identifiable loop fragment, it is not represented as a sequence function. The problem is that, unlike the simpler actions represented by RFILE, there are so many ways in which the items to be printed, and the format for printing them, can vary that there is very little constant structure which could be captured in a sequence function. Basically, the only thing which is common between different instances of this fragment is opening and closing the file which is already captured in the form WITH-OPEN-FILE.

A key aspect of LETS* is that even though the operations of actually printing the report are not represented as a sequence function, LETS* makes it possible for them to be conveniently expressed. This is done in basically the same way that it would be done in an ordinary looping notation i.e., by distributing the parts of the computation into places where they will be executed in the correct situations. Note that the sequence variables exist as real variables at run time. On a given cycle of the loop, these variables contain the individual sequence elements corresponding to that cycle. If you declare a sequence variable to be special you can refer to it outside of the lexical context of the LETS*.

It must be said that since the report production fragment is distributed throughout the loop, it is no easier to understand than it would be in an ordinary looping notation. However, the loop as a whole is more understandable because much of the computation is represented concisely in terms of sequence functions. The ability to mix computations which are not specified as sequence functions into a loop expression is another important capability which is facilitated by the element at a time metaphor.

The expressional and element-at-a-time metaphors are really very separate ideas. One could easily decide to support just one of them. For example, the language APL [10] supports much of the expressional metaphor and almost none of the element-at-a-time metaphor, while the language Hibol [13] does the opposite. Experience with the Lisp implementation of the expressional notation has shown that it is beneficial to blend these two ideas together.

Registration

The fundamental restriction which the element at a time metaphor places on the expressional loop notation is that every sequence function is required to have the property of *registration.* As discussed above, a sequence is an ordered series of slots containing values. The registration property requires that the nth element in the sequence produced by a sequence function must be computed from the nth elements of the input sequences to that function. The computation can also involve state variables internal to the sequence function and therefore can have implicit access to prior input values. However, it cannot refer to future elements in the inputs. The fact that the registration property is universally satisfied insures that it makes sense to talk about the interaction between the nth values in all of the sequences in a loop expression as *typical* because they are computed from each other.

The process of combining sequence functions together preserves registration and the only way to create new sequence functions is by combining the basic sequence functions. As a result, the fact that each of the basic sequence function is designed so that it satisfies the registration property insures that the registration property will always be satisfied.

The registration property makes loop expressions easy to understand and compile; however, it is significantly restrictive. The key limitation is that there are a number of quite logical operations on sequences which cannot be supported. In particular, operations which disturb the ordering of the slots are prohibited. For example, merging sequences, concatenating sequences, changing the order in a sequence (e.g., reversing it), etc. Such complex operations are not supported because the overhead associated with supporting them is not warranted by the rather infrequent need for them. When they are needed, other loop notations should be used.

Filters and Expressions Involving Multiple Sequences

The only sequence functions where there is any logical difficulty in satisfying the registration property are filters. It would be perfectly reasonable to say that a filter takes in a sequence and produces a shorter sequence containing only selected elements of the input sequence. From the point of view of the expressional metaphor there is nothing wrong with this definition, and there would be no difficulty in understanding a program like SUM-POSITIVE-EXPRESSIONAL based on this definition.

However, if filters produced shortened sequences, they would not satisfy the registration property. In order to satisfy this property, a filter is defined as producing a sequence which has exactly the same number of slots as its input with the selectivity of the filter encoded in the fact that some of the output slots are *empty.*

The selected values remain in the same slots as in the input sequence. In order to make this work, loop expressions are defined as simply not operating on empty slots. This can be seen in the definitions of the basic sequence functions presented above. The following general statement can be made: a given loop subexpression is only executed on those cycles of the loop when values are available for all of the sequences it refers to.

In order to appreciate the full impact of the definition of how filters operate, one must consider loop expressions involving several sequences. Consider the program LIST-EVEN-SQUARES. It takes in a list and returns a list. The output list contains an entry corresponding to each even number in the input. Each entry consists of the number followed by its square. For example when passed the argument (1 2 3 4) the program will produce the output ((2 4) (4 16)).

```
(defun list-even-squares (list)
  (letS* ((integers (Elist list)))
    (Rlist (list (filterS #'evenp integers)
                 (* integers integers)))))
```

In the program, the function LIST is implicitly MAPSed over two sequences. The first is generated by enumerating the elements in the input and selecting the even elements (e.g., [_ 2 _ 4]). The second is generated by squaring all of the elements in the list (e.g., [1 4 9 16]). The registration between the two sequences is maintained by the fact that the missing elements in the filtered sequence are represented as empty slots. The function LIST is only executed when values are available in both sequences i.e., only for even elements of the list. The output of the implicit MAPS is a sequence which has values in it corresponding to the times when LIST was executed (e.g., [_ (2 4) _ (4 16)]). When RLIST reduces this sequence to a list it ignores the empty slots.

Termination

There is one place where the expressional metaphor and the element at a time metaphor are antagonistic -- the question of termination. From the point of view of the pure expressional metaphor, termination is not really an issue any more than it is in ordinary expressions. Each sequence function is logically executed separately and as long as each one terminates in and of itself, the whole expression will terminate. However, in order to fit in with the element at a time metaphor, termination has to be treated in a somewhat more complex way which reflects the reality of the way loop expressions are compiled.

The termination of a loop expression is controlled by the length of the sequences in it. The loop expression terminates as soon as the *shortest* sequence in it is exhausted. This definition of termination is an example of *action at a distance* which makes it impossible to understand the various parts of a loop expression completely in isolation from each other. As a result, it violates the basic decomposability property of the expressional metaphor.

Consider again the program DIGITS-TO-NUMBER (reproduced below). There are several questions about the sequence functions in this program which cannot be answered completely locally. For example, although it is convenient to describe the generator as creating an infinite sequence of powers of 10, it cannot actually do that. The generator will eventually halt with an error due to arithmetic overflow unless some other sequence function terminates the loop before then. On the other hand, in order to be sure that the EVECTOR will succeed in enumerating all of the digits, one must check that no other sequence function will terminate the loop before the end of the vector of digits is reached. Because of these problems, you cannot just compose an understanding of its parts in order to understand the loop expression as a whole.

```
(defun digits-to-number (v)
  (Rsum (* (Evector v) (generateS #'(lambda (x) (* x 10.)) 1))))
```

Fortunately, there is a middle ground with regard to the property of decomposability. As discussed in [16], as long as you make no statements which depend on a specific minimum length for any sequence, any statement which is true about a sequence function in isolation will be true when it is composed with other functions in a loop expression. For example, you can say that the generator creates a sequence of powers of 10 beginning with 1. However, you cannot make any statement about whether it will or will not get arithmetic overflow in the process. Similarly, you can say that EVECTOR enumerates successive elements of a vector starting with the first one. You can even say that it will produce a sequence no longer than the vector. However, you cannot make any claim about any minimum number of vector elements which definitely will be enumerated.

Given the kind of statements you can dependably make, you can determine a great deal about a loop by using straight composition. For example, in DIGITS-TO-NUMBER it is easy to tell that the values in the sequence created by the generator correspond to successive powers of 10, and that the sequence created by the EVECTOR correspond to successive digits with the least significant digit first. In addition, it is clear that the program multiplies each digit by the appropriate power of 10 and that these results are summed up.

In order to go beyond this and make statements about termination, you must do more global reasoning. In this case, there is only one basic finite sequence involved (the one created by EVECTOR), and it clearly dominates the computation. As a result, the program clearly processes all of the digits and terminates computing the correct number.

Note that all of the basic sequence functions (with the exception of TRUNCATES) are carefully restricted so that the lengths of their output sequences (if any) are the same as the minimum of the lengths of their input sequences (if any). This is done so that truncators (and the enumerators which are built out of them) will be the only sequence functions which ever trigger termination. As a result, reasoning about termination can focus solely on the truncators and enumerators in a loop.

The two step reasoning process outlined above is usually very satisfactory for the kind of straightforward loops the expressional notation is designed to represent. In particular, the global reasoning about termination is usually not at all difficult.

Done

In addition to using the basic sequence function TRUNCATES to limit the length of a sequence, a loop can also be terminated by executing the special form DONE. Consider the program SUM-INITIAL. It takes in a list and adds up any initial group of numbers (e.g., (SUM-INITIAL '(1 2 A 4)) returns 3). The program works by enumerating the elements in the list and summing them up, but terminating the loop as soon as a non-number is encountered. The form (DONE) causes the immediately enclosing loop to terminate normally -- any AT-END loop computation which has been specified is performed, and the return value which is specified by the last line is returned (here the sum).

```
(defun sum-initial (list)
  (letS* ((x (Elist list)))
    (cond ((not (numberp x)) (done)))
    (Rsum x)))
```

DONE can also be called with an argument. In this case the loop is immediately terminated and the specified value is returned. When DONE is used in this way, it overrides the outputs specified in the last line of the LETS* and any AT-END computations are not performed. An example of this use of DONE is shown in the program FIND-POSITIVE which returns the first positive number in a list.

```
(defun find-positive (list)
  (letS* ((x (Elist list)))
    (cond ((plusp x) (done x)))))
```

The use of DONE is also illustrated by the following sequence functions. The sequence function ROR computes the OR of a sequence in the obvious way by successively ORing each value into a state variable. The first non-NIL value encountered is returned. The sequence function ROR-FAST also returns the first non-NIL value encountered; however, it causes the loop as a whole to terminate as soon as this value is found. Note the way the DONE overrides the NIL which is returned AT-END if no non-NIL items are found.

```
(defunS Ror (&sequence item)
  (reduceS #'or nil item))

(defunS Ror-fast (&sequence item)
  (cond (item (done item)))
  (at-end #'(lambda () nil)))
```

In general, ROR-FAST is more efficient than ROR; however, when you use it you must consider the effect that it will have on the rest of the loop it is used in. For example, because its operation is peremptorily terminated by the ROR-FAST, the program PRINT-LIST-OR-BUGGY neither prints out all of the elements in the list, nor prints out the summary line AT-END. In order to operate as intended, it needs to use ROR instead of ROR-FAST.

```
(defun print-list-or-buggy (list)
  (format T "~%Elements: ")
  (letS* ((x (Elist list)))
    (format T "~A " x)
    (format T "~%Their Or: ~D~%" (Ror-fast x))))
```

The output produced by (print-list-or-buggy '(NIL A NIL B))

```
Elements: NIL A
```

It should be noted that in many ways DONE clashes with the rest of the expressional notation. By causing sudden termination of a loop it creates action at a distance which violates the decomposability property of the expressional metaphor. If given an argument, it violates the integrity of the individual sequence functions by preventing the execution of AT-END computation. As shown by the example above, these problematical effects are particularly apparent in a sequence function like ROR-FAST. DONE is included in the notation because though it is semantically awkward it is simply too useful to be omitted.

You can cause the premature termination of a loop in other ways which are outside the scope of the LETS macro package. For example, you can wrap the loop in a PROG and then do a RETURN or GO from inside the loop to outside the loop. (Note that the loop expression itself is implemented by means of a PROG. In the LispMachine version (but not the MacLisp version), this PROG is named T in order to eliminate interference with user specified RETURNs.) Similarly, you can do a THROW from inside the loop to some CATCH outside the loop. An important aspect of these kinds of exits is that they do not cause normal termination of the loop. No AT-END loop computation will be run, and the return value is directly specified by the RETURN or THROW.

Side-Effects

The behavior of side-effect producing operations (such as input/output) in a loop expression can only be understood from the point of view of the element at a time metaphor. The compilation process is constrained so that the order of execution in the loop produced is exactly the same as the lexical order of expressions in the original loop expression. As a result, it is relatively easy to predict the consequences of side-effects as long as you bear in mind the fact that processing is occurring an element at a time so that the side-effect operations are interleaved and that each one is executed many times. Consider the program INVENTORY-REPORT above. The two main output statements are each executed once on each cycle of the loop. The final output statement is executed only once after the loop terminates.

Note that the requirement that every unitary expression in a LETS* be MAPSed (if possible) even if it does not refer to any sequences makes it much easier to understand side-effect operations. One might have chosen to say that an expression which neither uses nor produces sequence values should not be MAPSed since its value cannot change on different cycles of the loop. However, this would be missing the fact that if it has a side-effect (such as output to a file) this effect is probably desired on every cycle of the loop. A programmer can use AT-START in order to specify that something should only be executed once.

Most side-effects interact with the expressional notation in straightforward ways and can easily be understood as outlined above. However, there are some situations where things are not so clear.

Side-Effects and Termination

In order to understand how side-effects interact with termination, one has to be aware of exactly when termination will occur. For example, consider the program PRINT-LIST below. This function prints all of the items in a list preceding each one with an index of its position in the list. (Note that the first FORMAT is executed on each cycle of the loop even though it does not refer to any sequences.)

```
(defun print-list (list)
  (letS* ((i (generateS #'1+ 1))
          (x (Elist list)))
    (format T "~%Item ")
    (format T "~D:" i)
    (format T " ~A" x)))
```

The output produced by (print-list '(A B C)):

```
Item 1: A
Item 2: B
Item 3: C
```

There is one potential pitfall which the user must be aware of. A loop is terminated *immediately* upon discovering that one of the sequences has been exhausted. As a result of this, unless the termination test happened to be the first thing executed on that cycle of the loop, some things will get executed on that last cycle, and others will not. In particular, all and only those expressions which lexically precede the termination will be executed. For example, consider the program PRINT-LIST-BUGGY. (Note that although no sequence variables are bound, a LETS* is required in this program in order to specify that the three FORMATs should be executed in a single loop instead of in three separate loops. The LETS* also specifies that the FORMATs should be MAPSed.)

```
(defun print-list-buggy (list)
  (letS* ()
    (format T "~%Item ")
    (format T "~D:" (generateS #'1+ 1))
    (format T " ~A" (Elist list))))
```

The output produced by (print-list-buggy '(A B C)):

```
Item 1: A
Item 2: B
Item 3: C
Item 4:
```

This program does not produce the same output as PRINT-LIST. The problem is that it does not discover that the list has been exhausted until after the first two FORMATs have been executed on the last cycle of the loop. Note that this problem cannot be avoided by any straightforward change to the definition of LETS*. You could not say that nothing in a cycle will be executed if any termination is triggered because some of the computation may be necessary in order to compute whether to terminate. On the other hand, you could not say that everything will be executed on the cycle where termination occurs, because typically some (or all) of the computation after the termination test will be in error if the test is true.

The programmer is capable of exercising control over this problem because, in the loop code which is produced, everything is evaluated in the order in which it appears in the original loop expression. As a result, it is always possible for him to get the termination tests to occur at the places he wants by correctly ordering the forms in the LETS*. For example, the ELIST is merely placed before the first FORMAT in PRINT-LIST. As a result, this is not really a severe problem; however, it is one to which the user must be sensitized.

On a deeper level, the real problem with PRINT-LIST-BUGGY is that neither it (nor for that matter PRINT-LIST) makes the logical relationship between the three FORMATs explicit. The correct thing to do is to group them together into a single form as in the function PRINT-LIST-BEST.

```
(defun print-list-best (list)
  (letS* ()
    (format T "~%Item ~D: ~A" (generateS #'1+ 1) (Elist list))))
```

Side-Effects Between Sequence Functions

As mentioned above, the expressional notation attempts to maintain the property of decomposability of loop expressions whenever possible. An important feature of this is that any internal state variables of a sequence function are hidden from view and cannot be modified by SETQs, or the like, in a loop expression. Unfortunately, side-effect producing functions such as RPLACD are capable of modifying the values of state variables without having to actually refer to the variables themselves. If such side-effect functions are being used, then the programmer must take care that this kind of problem does not arise.

The problem is illustrated by the program DASH-LIST-BUGGY. The purpose of this program is to take in a list (e.g., (A B C)) and put a dash after each entry in it (e.g., to produce (A - B - C -)). It attempts to do this by side-effect as follows. It enumerates each of the sublists in the original list (e.g., [(A B C) (B C) (C)]) and splices in a dash after the first element of each sublist (e.g., producing [(A - B C) (B - C) (C -)]).

```
(defun dash-list-buggy (list)
  (letS* ((sublist (Esublists list)))
    (rplacd sublist (cons '- (cdr sublist))))
  list)
```

Particularly from the point of view of the expressional metaphor, the above algorithm sounds very plausible; however, it doesn't work. What actually happens is that the program goes into an infinite loop splicing in dashes after the first item in the input list. If the loop starts with the list (A B C) then the first sublist is (A B C). The RPLACD alters this sublist to (A - B C) and therefore the list itself to (A - B C). So far this is all as intended. Unfortunately, an internal variable in ESUBLISTS has a pointer into the list in order to keep track of what sublist to enumerate. The list is altered *before* the second sublist is actually enumerated and as a result (- B C) gets enumerated as the second sublist instead of (B C).

It is possible to construct a loop expression for this algorithm which will work more or less as intended. For example, the program DASH-LIST1 combines everything into one enumerator which enumerates the next sublist before the RPLACD operation. Alternatively, DASH-LIST2 uses a modified enumerator which makes allowances for the actions of the RPLACD.

```
(defun dash-list1 (list)
  (letS* ()
    (enumerateS #'null
                #'(lambda (l) (prog1 (cdr l) (rplacd l (cons '- (cdr l)))))
                list))
  list)

(defun dash-list2 (list)
  (letS* ((sublist (enumerateS #'null #'cddr list)))
    (rplacd sublist (cons '- (cdr sublist))))
  list)
```

However, due to the antagonistic interaction between the RPLACD and the enumerator, there is no aesthetic way to express the stated algorithm using the expressional notation.

Domain of Applicability

The expressional loop notation is oriented towards the kinds of straightforward loops which are most common. In order to make it easier to express these loops, it deliberately sacrifices more general applicability. As a result, there are a number of situations where the expressional notation is not appropriate.

The basic approach of the notation is to express a loop as a composition of fragments of looping behavior represented as sequence functions. There are two main situations in which this approach is ineffective: when a loop cannot be separated into multiple fragments, and when the notation is not capable of expressing the required fragments as sequence functions.

It is quite possible that even a large loop will not be decomposable into fragments. In order to break a loop down into two fragments A and B, it must be the case that A and B are both self contained units. In particular this means that there can be no interaction between A and B other than data flow from A to B. Note that there cannot be any data flow from B to A. In some loops, all of the computation is linked together in a tight net of data flow. In this case it cannot be decomposed. For example, consider the program BINARY-MEM which tests whether a given integer is in a sorted vector of integers by doing a binary search.

```
(defun binary-mem (integer vector)
 (prog (left mid right item)
    (setq left 0)
    (setq right (1- (array-iength vector)))
   L (cond ((> left right) (return nil)))
    (setq mid (// (+ left right) 2))
    (setq item (aref vector mid))
    (cond ((> item integer) (setq right (1- mid)))
       ((< item integer) (setq left (1+ mid)))
       (T (return T)))
    (go L)))
```

This program cannot be decomposed into a composition of fragments because each part affects every other part. The values of LEFT and RIGHT are used to compute MID which is used to compute ITEM which is used in a test which determines the next values of LEFT and RIGHT. Because it cannot be decomposed, there is no way to write the program more clearly using the expressional notation. The best that could be done would be to write the program as one huge sequence function.

The expressional loop notation is also limited in the kind of loop fragments which it can represent. There are a number of sources of limitation. Perhaps the biggest limitation is the requirement for registration. Another limitation is the fact that the only facilities available for creating (as opposed to combining) sequence operations are the seven basic sequence functions. Experience has shown that these are capable of creating a wide range of useful fragments. However, there are a variety of plausible fragments which cannot be created. For example, TRUNCATES performs the truncation test at the start of each cycle of the loop. It is not possible to create a fragment where the truncation test is performed at the end of each cycle.

Another reason why the expressional loop notation may not be appropriate in a given situation is that some other paradigm may be more appropriate. For example, consider the function GCD. Writing it as a recursive program makes it very easy to understand because the structure of the program mirrors the structure of the standard proof of correctness for the algorithm. No iterative rendition would be as clear.

```
(defun gcd (x y)
 (cond ((< x y) (psetq x y y x)))
 (let ((r (remainder x y)))
  (cond ((zerop r) y)
     (T (gcd y r)))))
```

In addition, it should be noted that, unlike some looping notations, the expressional notation does not handle anything but simple loops. For example, it does not support multiple entry points nor exits to multiple points.

IV - Language Independence

The above presentation of the expressional loop notation is couched in terms of Lisp. However, none of the ideas behind the notation are inherently dependent on any specific language. As a result, there is no reason why the expressional notation could not be implemented as an extension to almost any language.

Consider what the basic ideas behind the notation are. To start with, there are three themes which underlie the notation.

The Expressional Metaphor - The idea that loops can be expressed as compositions of fragments of looping behavior is the fundamental motivation behind the notation.

The Element at a Time Metaphor - The additional metaphor that a loop can be conveniently specified as a set of operations on typical elements also underlies the notation as a whole.

Efficient Compilation - From the beginning, it was decided that it had to be possible to compile the notation into efficient looping code. This effected many of the design decisions.

There are six basic features of the notation which together support these themes.

Sequence Functions - These embody the fundamental notion of a fragment of looping behavior. The fact that they look and can be reasoned about essentially just like ordinary functions supports the expressional metaphor. Restrictions on the kinds of sequence functions allowed (e.g., the requirement for registration between elements of their inputs and outputs) support the element at a time metaphor and efficient compilation.

Sequences - These are the mode of communication between sequence functions. The fact that they look like and can be reasoned about much of the time just like ordinary aggregate data objects supports the expressional metaphor. The fact that they are defined to be one dimensional series of slots containing unitary values where each slot corresponds to one cycle of the loop which will eventually be produced is the fundamental underpinning for the element at a time metaphor and is essential for efficient compilation.

Basic Sequence Functions - The seven basic sequence functions embody the fundamental computational capabilities supported by the notation. New sequence operations can be created by composing the basic sequence functions together. The fact that this is the only way in which new operations can be created guarantees that restrictions such as registration will always be satisfied.

User Definition of Sequence Functions - The fact that the user can define his own sequence functions in analogy with the definition of ordinary functions greatly extends the utility of the notation.

Loop Expression Blocks - Calls on LETS* serve two basic purposes: delineating groups of loop expressions which are to be combined into a single loop, and supporting the notion of variables which have sequences as their values. The body of such a block is the place where the element at a time metaphor is most prominent.

Coercions - The existence of coercions such as the automatic introduction of MAPS is an important underpinning for the element at a time metaphor. Other coercions such as automatic conversions between sequences and unitary values exist merely as a convenience for the user. Note that in order to make coercions practical, variables containing sequences have to be readily identifiable as such.

It is easy to extend a language in order to support the expressional loop notation syntactically as long as the language has aggregate data structures, functions, user function definition facilities and block structure. The semantic support for the notation can be implemented as either an extension to the compiler or a separate preprocessor as in the Lisp implementation.

For example, you could add the expressional loop notation to the language Ada [1] by supporting the six basic features of the notation as follows:

Sequence Functions - As in the Lisp implementation, calls on sequence functions would look syntactically exactly like calls on other functions; however, they would be handled like macros by a preprocessor in order to create loops as described above. A set of built-in sequence functions would be provided as part of the standard environment.

Sequences - A new data type **sequence of** would be added. This could be used to specify the data types of variables and of the arguments to sequence functions.

Basic Sequence Functions - The same seven basic sequence functions would be provided. Since these sequence functions take functional arguments they are analogous to generic functions in Ada. It is suggested, however, that a syntactic sugaring be introduced so that these seven functions can appear syntactically to be ordinary functions which take functional arguments.

User Definition of Sequence Functions - A new kind of function declaration **function on sequences** would be added. Using this, sequence functions would be defined exactly like ordinary functions. These would be the only functions allowed to have parameters and/or return values of type sequence. Similarly, **procedure on sequences** would be used to define procedures operating on sequences.

Loop Expression Blocks - A new keyword **begin computation on sequences** would be introduced. This could be used in place of **begin** in begin blocks, subprogram bodies etc. Only these blocks would be allowed to have variables of type sequence. Each such block would be compiled into a single loop.

Coercions - Given that the sequence data type would be used to identify all of the variables which carry sequences, various coercions could be supported by the preprocessor in exactly the same way as in the Lisp implementation.

The following examples show what loop expressions would look like in Ada. The first is a version of the program DIGITS-TO-NUMBER which takes in a vector of digits (least significant digit first) and computes the corresponding integer. The second program illustrates the definition of a sequence function.

```
type VECTOR is array (INTEGER range <>) of INTEGER;
type INTEGER_SEQUENCE is sequence of INTEGER;

function DIGITS_TO_NUMBER_ADA(DIGITS: VECTOR) return INTEGER is
  function TIMES_TEN(X: INTEGER) return INTEGER is
    begin return X*10; end;
  DIGIT, SCALE: INTEGER_SEQUENCE;
begin computation on sequences
  DIGIT := EVECTOR(DIGITS);
  SCALE := GENERATES(TIMES_TEN, 1);
  return RSUM(DIGIT*SCALE);
end;

function on sequences RSUM(INTEGERS: INTEGER_SEQUENCE) return INTEGER is
begin computation on sequences
  return REDUCES(+, 0, INTEGERS);
end;
```

Due to the type information which has to be specified and the fact that there is no compact representation for literal functions, the above programs are somewhat more lengthy than their Lisp counterparts; however, they are identical in basic structure.

V - Comparison With Other Loop Notations

Consider the program SUM-POSITIVE-EXPRESSIONAL (reproduced below) which was used as an example in the beginning of this paper. There are many different computationally equivalent ways to represent any given loop. All of these representations are capable of expressing the same basic looping algorithm. In order to evaluate the usefulness of these representations, we must look at other characteristics beyond expressiveness.

```
(defun sum-positive-expressional (vector)
  (Rsum (Fpositive (Evector vector))))
```

The paramount property required of a looping representation is *understandability* i.e., how easy is it to look at a loop and determine what the loop is computing. Two closely related properties are also of great importance. The first is *constructibility* i.e., given a specification, how easy is it to build up a loop which satisfies the specification. The second is *modifiability* i.e., given a loop, how easy is it to change it in accordance with a change in its specification.

The key idea behind the expressional loop notation is that most looping algorithms are built up out of stereotyped fragments of looping behavior and therefore loop programs are easier to understand, construct, and modify if these fragments are expressed as easily identifiable syntactic units. In the expressional notation, loop fragments are represented by sequence functions. Many other looping notations have methods for representing at least some loop fragments. Discussion of these methods is the major theme of the comparisons below.

Two things act as the focus for the following sections. The first is the loop in the program SUM-POSITIVE-EXPRESSIONAL. Each section shows how the loop notation being discussed could be used to express this algorithm. The second focus is the six basic features of the expressional notation. The sections are ordered from simple constructs which have very few of these features to languages like APL and Hibol which embody most of them.

PROG and GO

The program SUM-POSITIVE-GO shows how our example loop could be implemented using a PROG and GO. The program is not very easy to understand because PROG and GO suggests a particularly unfortunate way to think about the loop, namely that it is basically a straight line piece of code which is converted into a loop by the addition of a GO. This notation embodies none of the basic features of the expressional notation. The key idea which is being missed by this way of thinking is that straightforward loops like this one are built up out of standard fragments of loops and not out of standard straight line fragments.

```
(defun sum-positive-go (vector)
  (prog (sum i end)
        (setq sum 0)
        (setq i 0)
        (setq end (1- (array-length vector)))
      L (cond ((> i end) (return sum)))
        (cond ((plusp (aref vector i))
               (setq sum (+ sum (aref vector i)))))
        (setq i (1+ i))
        (go L)))
```

Instead of highlighting the loop fragments, the program breaks them up into pieces and then mixes the pieces together. For example, the enumerator is broken up into three pieces: an initialization which sets the starting value for I, a termination test that terminates the loop after the last index is produced, and a repetitive step which increments I each time around the loop.

It is just as difficult to see how the fragments interact as it is to identify the fragments themselves. The enumerator and the filter interact by sharing the variable I. In contrast, the interaction between the filter and the reducer is represented by embedding part of the reducer inside of the filter COND. This is particularly confusing because the COND looks like it is implementing an ordinary straight line conditional fragment. One has to look carefully at the surrounding context in order to see that this is not the case.

Although the above points have been presented primarily as problems of understandability, they cause just as much trouble with regard to constructibility and modifiability. In particular, the fact that the fragments are not localized means that neither the construction nor modification processes can be localized. This greatly complicates both tasks. Another problem is that since the various fragments are just mixed together, there is no support for keeping them semantically separate. One must be particularly careful that introducing a new fragment will not disturb one of the other fragments.

Another kind of problem with PROG and GO as a notation for straightforward loops is that it supports a number of features which are needed only in complex situations and which obscure simple loops by cluttering them up. Two examples of these are: the fact that PROG supports multiple tags, GOs and RETURNs; and the fact that it allows multiple assignments to the key variables involved. These features are particularly problematical because even when they are not being used, you have to look very closely in order to determine that they are in fact not being used. In the example, you have to verify that there is only one tag, one GO, and one RETURN and that there is only one assignment to each of the critical variables in the loop before you can have any confidence in what is going on.

There are algorithms for which a PROG and GOs are particularly appropriate. For example, if a program implements a finite state automaton, GOs can be used to directly model the transitions. GOs can also be used to implement various exotic multiple entry and multiple exit loops. However, it is generally recognized that GOs are never the best way to implement simple loops.

Tail Recursive Style

The program SUM-POSITIVE-RECURSIVE is written in tail recursive style. Though it looks very different from SUM-POSITIVE-GO it specifies essentially exactly the same algorithm. A compiler which knew about tail recursion could produce the same object code for the two programs. SUM-POSITIVE-RECURSIVE is somewhat easier to understand because much of the verbiage is removed. There is no longer any possibility of multiple tags, GOs, or RETURNs. As a result, the reader does not have to worry about them. In addition, the fact that each value changes only once on each cycle of the loop is easy to see.

```
(defun sum-positive-recursive (vector)
  (sum-positive-recursive1 vector 0 0 (1- (array-length vector))))

(defun sum-positive-recursive1 (vector sum i end)
  (cond ((> i end) sum)
        (T (sum-positive-recursive1 vector
                                    (cond ((plusp (aref vector i))
                                           (+ sum (aref vector i)))
                                          (T sum))
                                    (1+ i)
                                    end))))
```

Like PROG, the tail recursive style suggests a particular way of looking at a loop. Namely, that we should generalize the task at hand into a problem that can be recursively reduced a step at a time to a problem that is trivial to solve. In this case the trivial problem is adding up the positive elements of a sub-vector of length zero. The generalized problem is adding up the positive elements of a sub-vector and adding this to an initial partial sum. The recursive step involves adding one element into the partial sum, and reducing the size of the sub-vector.

There are loops which can be best understood by looking at them from the recursive viewpoint. However, this program is not one of them. The problem is that the tail recursive style is no better than a PROG at highlighting the fragments that the loop is composed of. As above, the fragments are broken up and mixed together. In addition, the way the fragments interact is still unclear. For example, part of the reducer is still nested in the filter. The "(T SUM)" clause which has to be added into the filter COND makes that interaction even less clear than in the PROG above. Like PROG and GO, the tail recursive style does not support any of the features of the expressional notation.

FOR

The next few sections describe notations which begin to support the idea of a sequence function (i.e., fragments of looping behavior as identifiable units). They do not however support any of the other features of the expressional notation.

Most algorithmic languages have looping constructs which facilitate the construction of simple loops. A typical example of these is the Ada FOR construct [1]. The Ada program SUM_POSITIVE_FOR illustrates the use of this construct. One benefit of FOR is that like the tail recursive style, it clearly delimits the extent of the loop and makes it clear that there is no exotic control flow going on in conjunction with the loop. Unfortunately, it is less helpful with regard to the data flow. There is no easy indication that each of the critical variables is only modified once.

```
type int_vector is array (integer range <>) of integer;

function sum_positive_for(vector: int_vector) returns integer is
  sum: integer;
begin
  sum := 0;
  for i in vector'range loop
      if vector(i)>0 then
         sum := sum+vector(i);
      end if;
  end loop;
  return sum;
end;
```

A much more interesting aspect of FOR is that it explicitly represents one of the fragments -- the enumerator of integers over the range of the array. This explicit representation of the fragment is particularly useful because (unlike the FOR constructs in most other languages) the Ada FOR construct protects the semantic integrity of the fragment by prohibiting the loop counter from being modified inside the loop. As a result, this particular fragment is easy to understand, construct, and modify.

Unfortunately, FOR is only capable of supporting this one kind of enumerator. There is no support at all for any of the other fragments in the loop. They are represented using straight line code in exactly the same way as in SUM-POSITIVE-GO. As a result, FOR is not an improvement over GO with regard to these other fragments.

Iterators in CLU

The language CLU [8] has extended the concept behind the FOR construct so that it can represent other enumerators besides integer enumeration. In CLU you can define a program called an *iterator* which takes in some unitary arguments and creates a sequence of objects. The iterator can then be used in a FOR in order to enumerate a sequence of elements to be processed in the body of the FOR. CLU provides a number of standard iterators including one corresponding to EVECTOR. As an illustration, the first program below shows how EVECTOR could be defined if it did not already exist. The program SUM_POSITIVE_CLU then shows how the iterator could be used.

```
Evector = iter(a: array[int]) yields(int)
  i: int := array[int]$low(a)
  end: int := array[int]$high(a)
  while i <= end do
    yield(a[i])
    i := i + 1
  end
end Evector

sum_positive_clu = proc(a: array[int]) returns(int)
    sum: int := 0
    for e: int in Evector(a) do
        if e > 0 then sum := sum + e end
    end
    return(sum)
end sum_positive_clu
```

Because there are no restrictions on the form that the body of an iterator can take (for example there is no requirement that it even be a loop), iterators are more general than the enumerators presented here. However, this power has drawbacks. For example, it would be difficult to treat iterators like macros and compile them inline in the loops which use them. The current CLU compiler implements iterators as separate procedures which return one element of the sequence every time they are called.

From the standpoint of understandability, an important aspect of iterators is that their semantic integrity is protected by the fact that they encapsulate their own state. In fact, iterators embody the logical concept of enumeration fully as well as the enumerators presented here. (It should be noted that the language Alphard [14] has a similar construct called a *generator*.) Unfortunately, neither of these languages provided any support for any fragments other than enumerators. As a result, each of these constructs is only a limited (though significant) improvement over simple FOR in the direction of supporting fragments.

Lisp DO

Another variant on FOR is the Lisp DO construct. This construct is interesting because it recognizes the existence of loop fragments other than enumerators and attempts to group their parts more closely together. Each of the forms in the first part of a DO is capable of representing a loop fragment. For example, the initialization and repetitive step of the range enumerator are combined together in the first line of the DO in the program SUM-POSITIVE-DO.

```
(defun sum-positive-do (vector)
  (do ((i 0 (1+ i))
       (end (1- (array-length vector)))
       (sum 0))
      ((> i end) sum)
    (cond ((plusp (aref vector i))
           (setq sum (+ sum (aref vector i)))))))
```

Unfortunately, DO is very restrictive in the way it can represent fragments. For example, the termination test of the enumerator has to be specified separately, causing the enumerator to be less conveniently represented than in a FOR. In addition, there is no good way to represent a filter at all. Going beyond this, the interactions between the fragments have to be represented in the same clumsy ways as in the programs above. For example, a COND still has to be used to express the interaction between the filter and the reducer.

At a more fundamental level, although DO makes it easier to write loop fragments as identifiable units, it does not enforce their semantic integrity. For example, you could easily put an assignment to I in the body of the DO. If you did this the computation involving I would no longer be an range enumeration. This would be particularly confusing because the first line of the DO would still look like an ordinary range enumeration.

All in all, it is clear that the various FOR and DO constructs are quite beneficial because they make it easier to locate a simple loop, and to verify that it is indeed simple. However, although these constructs point in the direction of explicitly supporting loop fragments they do not do this in either a very thorough way or a very semantically strong way. As a result, they are only a modest help in the understanding, construction, and modification of loops.

The Lisp Map Functions

The Lisp MAP functions are very restricted in what they can do. For example, they cannot be used to express the algorithm used in the examples above. However, when they can be used they are very compact and easy to understand. Each of the six MAP functions is an abbreviation for a particular combination of loop fragments. The example below illustrates the fragments corresponding to MAPCAR.

```
(defun mapcar (fn list)
  (Rlist (mapS fn (Elist list))))
```

Each MAP function embodies a certain set of fragments and protects their semantic integrity. If these fragments are appropriate to the algorithm at hand, then the use of the MAP function leads to a program which is easy to understand, construct, and modify. The expressional loop notation is designed to extend the basic idea embodied in the MAP functions to a much wider domain of programming.

The Lisp Macro LOOP

The Lisp macro LOOP [3] is a significant improvement over the constructs presented above because it recognizes loop fragments of all kinds as full fledged constituents. Consider the program SUM-POSITIVE-LOOP. In this program, the enumerator, filter, and reducer are each represented on a separate line in the loop. This gives a program which is much easier to understand, construct, and modify than the ones above. A number of standard loop fragments are supplied as part of the macro.

```
(defun sum-positive-loop (vector)
  (loop for item being each vector-element of vector
        when (plusp item)
        sum item))
```

In addition to supporting relatively general fragments and their combination, LOOP supports the creation of user defined fragments of all kinds. The example below shows how one could define VECTOR-ELEMENT OF which is the equivalent of the sequence function EVECTOR.

```
(define-loop-path vector-element Evector (of))

(defun Evector (ignore variable ignore phrases ignore ignore ignore)
  (sublis (list (cons 'expr (cadar phrases))
                (cons 'variable variable)
                (cons 'vector (gensym))
                (cons 'i (gensym))
                (cons 'end (gensym)))
          '(((vector) (i 0) (end))
            ((setq vector expr)
             (setq end (1- (array-length vector))))
            (> i end)
            (variable (aref vector i))
            nil
            (i (1+ i)))))
```

Unfortunately, LOOP neither develops the concept of a sequence, nor the analogy of treating loop fragments as functions. This prevents it from expressing loops as sequence expressions in analogy with ordinary unitary expressions. Instead, LOOP supports a keyword-based syntax which specifies both the fragments to be used, and how they are combined. The way fragments can be combined is rather restricted because it is tied up with the keyword parsing algorithm.

In addition, the LOOP macro has a body part (not used in the example above) just like the body of a DO. This body can contain arbitrary computation -- there is no attempt to protect the semantic integrity of the individual fragments in the initial part of the LOOP.

Another problem with LOOP is that the facilities it provides for defining the equivalent of new sequence functions are rather cumbersome. Unlike the expressional notation, there is nothing corresponding to the basic sequence functions. The user has to define a function which can deal with parsing parts of the LOOP syntax and which returns a list of six pieces which are put in different places in the loop being constructed. Acting together, these pieces have to perform the actions of the desired sequence function. At the most basic level, this is quite similar to what happens in the expressional notation. However, it seems better if the user does not have to interact with the system at this low a level.

APL

There are several programming languages which support what are essentially expressional loop notations. The oldest of these is APL [10]. It is interesting to note that there is no reason to believe that the developers of APL had anything like the expressional loop notation in mind. Rather, they were just seeking to provide a set of very useful operations on arrays. However, a style of writing APL has evolved where sequences are implemented as arrays.

The implementation of sequences as bona fide data objects automatically supports four of the six features of the expressional notation (i.e., sequences, sequence functions, user definition of sequence functions, and loop expression blocks). As illustrated below, both sequence functions and the vector summing algorithm can be very compactly represented in APL. Note that since sequences are directly represented as vectors, there is no need for the function EVECTOR.

```
         ∇ RESULT←FPOSITIVE VECTOR
    [1]    RESULT←(VECTOR>0)/VECTOR
         ∇

         ∇ RESULT←RSUM VECTOR
    [1]    RESULT←+/VECTOR
         ∇

         ∇ SUM←SUMPOSITIVEAPL VECTOR
    [1]    SUM←RSUM(FPOSITIVE(VECTOR))
         ∇
```

APL also has operators similar to the basic sequence functions. For example, "*function*/*value*" is the same as (REDUCES *function init value*). (Note that the *init* is automatically chosen to be the identity element under *function*.) Unfortunately, user defined functions cannot be used with these operators, so each one only actually corresponds to a small number of built-in sequence functions. APL supports the notion of a filter in a more general way than it supports REDUCES. "(*function*(*value*))/*value*" is the same as (FILTERS *function value*). This operator (the two argument form of /), which is called compression, takes in two vectors and creates a vector of elements from the second vector which correspond to non-zero elements of the first vector. Any arbitrary function can be used to create the first vector. A binary function rather than a unary one is used in the example. (Note that compression makes a shorter vector, rather than introducing empty elements.)

APL has no operators corresponding to the sequence functions GENERATES, ENUMERATES, or TRUNCATES. Since sequences are represented as arrays, there does not have to be any equivalent of the sequence functions EVECTOR and RVECTOR. Further, since arrays are the only composite data structure supported by APL, there do not have to be any enumerators or reducers which deal with other data structures. Since all arrays are finite, there need not be any generators or truncators. APL does have an operator (the index generator "ι*N*") corresponding to (ERANGE 1 N). Note that the fact that the operators provided by APL are somewhat limited does not prevent the user from defining any kind of sequence function he desires by simply using more primitive constructs to write the appropriate function on arrays.

APL also supports the idea of implicit MAPS to some extent. Every scalar function can be applied to vectors with the meaning that the operation is to be applied to every element of the vector. Also, scalars are coerced to vectors wherever necessary. Both of these processes are happening in the expression (*VECTOR*>0) above which takes in VECTOR and produces a vector of zeros and ones which indicate which elements of VECTOR are greater then zero. Coercion cannot be done as completely as with the expressional notation presented here because there is no mechanism for differentiating between arrays which are arrays, and ones which are intended to be sequences.

There are two ways in which APL is more powerful than the expressional notation presented here. First, it supports a number of operators which are much more powerful. For example, it has a number of operators which rearrange the order and structure of an array such as reshape, concatenation (of two vectors), expansion (the inverse of compression), reversal, rotation, and grade up (sort). It has complex operations on pairs of arrays such as outer product and inner product which produce outputs which are not the same shape as the inputs. In addition to all this, arrays are of course also just data objects, and you can operate on them as such. You can retrieve and set individual elements and perform arbitrary computations.

Another way in which APL is more powerful is that while sequences are analogous to vectors, the standard intermediate structure in APL is the array. The fact that arrays are multidimensional makes them a more flexible representation. All of the operators above can be applied to arrays, and to selected parts of arrays producing results of similar or dissimilar shape.

The powerful features provided by APL make it possible to compactly express a wide variety of complex mathematical algorithms which cannot be expressed in the expressional notation at all. For these algorithms, APL has the virtue of easy understandability, constructibility, and modifiability. Unfortunately APL has several drawbacks. First, it does not support any data structures other than numbers, characters, and arrays. Second, although APL supports the expressional metaphor almost completely, it does not support the element at a time metaphor at all. Third, due to that fact that it supports such complex array functions and the fact that it rejects the element at a time metaphor, APL cannot in general be compiled into efficient code. Fourth, APL's approach to loops is embedded into a somewhat cryptic and forbidding syntax. Together, these features have limited APL's impact.

The expressional loop notation presented here eliminates these problems. First, it can handle arbitrary data structures. For example, to deal with a new aggregate structure, the user need only define new enumerators and reducers to convert the aggregate to a sequence and vice versa. Second the element at a time metaphor is part of the basis for the notation. Third, the expressional loop notation deliberately omits all those operations on sequences which would make it hard to compile. Fourth, the expressional notation is designed to be added into preexisting languages as a natural extension of their syntax. One need not learn a new language and environment in order to use it.

The Listless Transformer

In a Lisp-like language one could decide to support the expressional metaphor by implementing sequences as lists. Wadler [15] has implemented an interesting prototype system (the Listless Transformer) which is capable of transforming programs containing sequences implemented as lists and eliminating the actual computation of many intermediate lists. The loop notation supported by his system is at heart essentially identical to APL with lists substituted for arrays. The target of his system is a Lisp-like language called Iswim [15]. The example below shows how sequence functions can be defined and used in this language.

```
def Evector(v) =
  Evector1(v, 0, length(v))
    where rec Evector1(v, i, end) =
      if i>end then nil
                else cons(aref(v, i), Evector1(v, i+1, end))

def rec Fpositive(xs) =
  case xs of
    nil => nil
    cons(x ,rest) => if x>0 then cons(x, Fpositive(rest))
                            else Fpositive(rest)

def Rsum(xs) =
  Rsum1(xs, 0)
    where rec Rsum1(xs, total) =
      case xs of
        nil => total
        cons(x, rest) => Rsum1(rest, total+x)

def sum-positive-listless(v) =
  Rsum(Fpositive(Evector(v)))
```

It is not clear whether any of the basic sequence functions are supported; however, they would be easy to implement as macros. In any case, the user can implement any sequence function he desires be defining arbitrary functions on lists. Iswim is a typed language and coercions like implicit introduction of **MAPS** can be supported.

Like APL, Wadler's notation is more powerful than the notation described here in that it supports arbitrarily complex sequence functions and sequences can be multi-dimensional sequences of sequences. It goes beyond APL in being able to deal with arbitrary data structures.

Wadler's notation also shares APL's greatest weaknesses. It does not support the element at a time metaphor. In addition, due to the fact that arbitrarily complex sequence functions are allowed, it cannot be efficiently compiled in the general case. It also shares the problem that, since it is not oriented toward the element at a time metaphor, loops involving side-effects cannot be efficiently compiled.

Coroutines

Another language which supports most of the expressional metaphor is the coroutine language of Kahn and MacQueen [7]. They have suggested using parallel processes (coroutines) in order to represent computations communicating via one way *channels* (sequences). In their approach, unbounded sequences are implemented as real data objects which are passed an element at a time through channels between processes executed in parallel. The code below shows one way the vector summing algorithm could be implemented in their system. Each sequence function is defined as a separate process. These processes can have ordinary (unitary) inputs (e.g., the VECTOR input of EVECTOR) and outputs (e.g., the return value of RSUM). They can also have channel (sequence) inputs (e.g., the CHANNEL1 argument of FPOSITIVE) and outputs (e.g., the CHANNEL output of EVECTOR). An element is retrieved from a channel by the function (GET *channel*). An element can be put into a channel by the function (PUT *item channel*). In order to use the sequence functions, they are combined together in an expression as in the function SUM-POSITIVE-COROUTINE. This expression is placed in a DOCO form which causes the three processes to be executed concurrently.

```
process Evector vector => channel;
  vars i;
  1 -> i;
  repeat
     put(i, channel);
     increment i;
  until i>upper-bound(vector);
  put(done, channel)
endprocess;

process Fpositive in channel1 => channel2;
  vars n;
  repeat
    get(channel1) -> n;
    if n=done or n>0 then put(n, channel2) close
  until n=done
endprocess;

process Rsum in channel => sum;
  vars n, sum;
  0 -> sum;
  repeat
    get(channel) -> n;
    if not(n=done) then sum+n -> sum close
  until n=done;
  return(sum)
endprocess;

process sum-positive-coroutine vector
  start doco Rsum(Fpositive(Evector vector)) closeco
endprocess;
```

The language of Kahn and MacQueen supports neither the basic sequence functions, nor automatic coercions. However, they could be added if desired.

The coroutine approach is more powerful than the expressional notation along a different dimension from APL. Each process is a truly independent parallel process. One aspect of this is that sequences can really be infinite. In addition, it is possible for one process to terminate without forcing the other ones to terminate and processes can dynamically spawn whole networks of other processes. This makes it possible to express modes of computation which cannot be conveniently expressed with any of the other notations discussed here. However, this brings with it a certain overhead. In the example above, the special token DONE is passed around between the processes so that the termination of the EVECTOR process will trigger the termination of the other processes.

The key drawback of the coroutine approach is that it is not clear how it can be compiled. Like APL, it supports the definition of arbitrarily complex sequence functions. Going beyond this, given that the coroutine notation is capable of expressing arbitrary parallel computations, one would expect that it will be extremely difficult to write an optimizing compiler which reliably detects groups of processes which interact merely as simple loops. However, without such a compiler, the coroutines impose an unacceptable overhead on the execution of simple loops.

The expressional loop notation presented here is based on ideas very similar to the coroutine notation; however it is restricted so that it is trivial to compile. The intention is to use the expressional notation to represent simple loops while reserving the coroutine notation for those situations where its greater power is required.

Hibol & Model

The language Hibol [12,13] is the oldest language which both supports the idea of a sequence and is completely compilable. It is a very high level business data processing language based on the concept of a flow (which is basically equivalent to a sequence). It is very strongly oriented toward the element at a time metaphor and relies heavily on the concept of the implicit introduction of MAPS. The body of each Hibol program is a nonprocedural set of expressions specifying the computations on typical sequence elements.

The program SUM_POSITIVE_HIBOL computes the sum of the positive elements in a file. (The only aggregate data type supported by Hibol is a file.) The language provides a few standard sequence functions (e.g., the operator SUM in the program below). In addition, the operator IF implements the basic sequence function FILTERS. These facilities make it possible to specify the body of SUM_POSITIVE_HIBOL as a simple expression. The DATA DIVISION part of the program describes the files accessed by the program in a format very similar to Cobol.

```
/* the program sum_positive_hibol */
data division
  key section
    key index
        field type is integer
  input section
    file vector_item
         key is index
         type is integer
  output section
    file sum_positive
         type is integer
computation division
  sum_positive is (sum of (vector_item if vector_item > 0))
```

Hibol is more powerful than the expressional notation in that flows are multidimensional objects like arrays where each level of index is an alphanumeric key rather than a number. The Hibol operators can be selectively applied to specific dimensions of a flow. A set of defaulting mechanisms make it possible to specify a simple program like the one above without having to explicitly specify which dimensions operators are being applied to.

The operations which can be applied to flows have been carefully selected so that all flow expressions can be compiled into efficient loop code and the Hibol compiler clearly shows that an expressional looping notation can be straightforwardly compiled even if it supports multidimensional sequences. Nevertheless, when designing the expressional loop notation presented here it was decided to omit this feature for two reasons. First, it was judged that the frequency of its use would not justify the extra complexity of supporting it. Second, when a loop algorithm becomes complex enough that the user is forced to specify which dimensions operators are being applied to, the syntactic mechanisms required cause the resulting expressions to begin to lose the virtue of easy understandability.

From the point of view of this discussion the primary weakness of Hibol is that it does not provide very much support for the expressional metaphor. First, it provides a few built in sequence functions, but does not allow the user to define new ones. Note that files are the only aggregate data structure supported by Hibol, and that enumeration and reduction of files occurs implicitly. Second, it supports only two of the basic sequence functions: MAPS (introduced only implicitly) and FILTERS (the form IF). Implicit nested loops can be specified but there is no notion of an explicit looping block.

As discussed above, the expressional loop notation presented here addresses these problems because it can deal with arbitrary data structures, because it supports the creation of user defined sequence functions, and because it is intended to be embedded in a language which supports standard control flow constructs. The expressional notation being presented here could be looked at as taking some of the key ideas embodied in Hibol and separating them out from the business data processing language context of Hibol in a form in which they can be conveniently added into other languages.

More recently, another language has been developed which is very much like Hibol. This language (Model [11]) is based on the same idea of a multidimensional sequence, and is also primarily intended for business data processing applications. It is somewhat more powerful, and has a somewhat wider range of features, but at the level of this discussion it is essentially identical to Hibol. It is fully compilable and has the same basic advantages and disadvantages. It serves as yet another example that the idea of a sequence appears in many different forms in many different languages.

Acknowledgments

Both the LETS macro package and this paper have benefited from many helpful comments made by people at the AI laboratory. In particular, I would like to thank C. Rich for suggestions which led to a number of very significant improvements in the clarity and power of the notation, E. Ciccarelli for comments which materially improved the presentation in this paper, and D. Chapman for being the first real user of LETS and making a number of useful suggestions on form and content.

References

[1] J.G.P. Barnes, "Programming in Ada", Addison-Wesley, London, 1982.
[2] T.A. Budd, "An APL Compiler", Univ. of Arizona, Dept. of Comp. Sci. TR 81-17, October 1981.
[3] G. Burke and D. Moon, "Loop Iteration Macro", MIT/LCS/TM-169, July 1980.
[4] D.P. Friedman and D.S. Wise, "CONS Should Not Evaluate Its Arguments", Indiana Tech. Rep. 44, Nov. 1975.
[5] L.J. Guibas and D.K. Wyatt, "Compilation and Delayed Evaluation in APL", in Proc. 5th ACM POPL Conf., Sept, 1978.
[6] P. Henderson and J.H. Morris, "A Lazy Evaluator, presented at the SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Atlanta, Jan. 1976.
[7] G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Processes", in 1977 Proc. IFIP congress, North-Holland, Amsterdam The Netherlands, 1977.
[8] B.H. Liskov, et. al., "CLU Reference Manual", Lecture Notes in Computer Science, G. Goos and J. Hartmanis editors, V114 Springer-Verlag, New York, 1981.
[9] D.A. Moon, "MacLisp Reference Manual", MIT Cambridge MA, April 1974.
[10] R.P. Polivka and S. Pakin, "APL: The Language and Its Usage", Prentice-Hall, Englewood Cliffs NJ, 1975.
[11] N.S. Prywes, A. Pnueli, and S. Shastry, "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development", ACM TOPLAS, V1 #2, October 1979, pp 196-217.
[12] G.R. Ruth, "Data Driven Loops", MIT/LCS/TR-244, 1981.
[13] G.R. Ruth, S. Alter, and W. Martin, "A Very High Level Language for Business Data Processing", MIT/LCS/TR-254, 1981.
[14] M. Shaw and W.A. Wulf, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", CACM V20 pp 553-564, Aug. 1977.
[15] P. Wadler, "Applicative Languages, Program Transformation, and List Operators", PhD Thesis, Carnegie-Mellon Univ., 1982.
[16] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, Dec. 1978.
[17] R.C. Waters, "A Method for Analyzing Loop Programs", IEEE Trans. on Soft. Eng., V5 #3, May 1979.
[18] D. Weinreb and D. Moon, "Lisp Machine Manual", MIT Cambridge MA, July 1981.

Appendix A: The Compilation Process

The first section in this appendix describes some assumptions which the macro expansion process makes about the form of the loop expressions to be processed. The user must be careful to ensure that these assumptions are satisfied. The rest of the sections discusses the actual macro expansion process in detail. This discussion is intended to function both as detailed documentation for the actual program, and as a guide to anyone who wishes to implement a similar system.

The compilation process revolves around a data structure representing the key information about a fragment of looping behavior. Each fragment data structure contains all of the information needed to create a loop corresponding to a sequence function, and information about the inputs and outputs of the sequence function. A call on a sequence function is represented as an application of a fragment data structure to a list of arguments. The process of combining several sequence functions together into a single loop proceeds by combining together the fragment data structures corresponding to them.

Given a program that contains one or more loop expressions, macro expansion will proceed normally until the outermost macro in one of these loop expressions is encountered. At that time, the LETS macro package processes the loop expression, converting it into an iterative loop. Macro expansion then continues normally until another loop expression is encountered.

The process of converting a loop expression into a loop occurs in several steps. After locating an expression, all of the calls on sequence functions in it are located and the expression is parsed performing any necessary coercions (e.g., implicit MAPS introduction). Once this is done, all of the separate loop fragments in the expression are combined into one large loop fragment. This structure is then converted into an iterative loop.

As an example, the following shows the code which is produced for the loop in the program SUM-POSITIVE-EXPRESSIONAL.

```
        (macroexpand '(Rsum (Fpositive (Evector vector))))

yields: (prog T (sum2 element10 index6 f4 last7)
                (setq last7 (1- (array-length vector)))
                (setq index6 0)
                (setq sum2 0)
             L0 (cond ((> index6 last7) (go E0)))
                (setq element10 (aref vector index6))
                (setq f4 (plusp element10))
                (cond (f4 (setq sum2 (+ sum2 element10))))
                (setq index6 (1+ index6))
                (go L0)
             E0 (return-from T sum2))
```

Interaction With Other Macros

There are two aspects to the way in which the LETS macro package interacts with other macros which must be kept in mind when using the package. The first stems from the fact that the macro package has almost no knowledge of fexprs and other macros. In order to avoid the problems that could arise, you should never have a variable which is the same name as a function. This restriction can only be considered to be a bug in the system and should be rectified in later versions. The second issue is more fundamental and stems from the fact that when producing a loop, the LETS macro package gathers together a group of sequence functions which may be intermixed with calls on other macros. The user must be aware of the fact that all of the sequence functions will be processed before any of the other macros are processed.

The fact that the LETS macro package does not have any special understanding of fexprs or other macros leads to two specific restrictions on the kinds of loop expressions you can write. The first restriction comes from the fact that the LETS macro package assumes that every instance of a name of a sequence function is a call on that sequence function. (Note that this is not just every instance of the name that is the CAR of a list. The macro must look for sequence functions in a wider range of contexts than this due to idiocyncracies in some of the standard system macros. Note that inside of a backquoted list in MacLisp ,(ELIST X) reads in as (|',/|| ELIST X).) In order to avoid problems, you should never use the name of a sequence function as a variable name. One of the reasons why each of the sequence function names is given a prefix letter is to reduce the probability of variable name conflicts happening by mistake.

The second restriction is that, for each variable name in the argument list of a DEFUNS, bound by LETS*, or in the lambda list of a literal lambda expression passed as an argument to a sequence function, every occurrence of that symbol in its scope must be an instance of a reference to that variable. The function SUBST will be used to rename this variable when necessary to avoid name conflicts. The two main ways that trouble could arise is if you use a variable name which is the same as a function name, or if you rebind the variable name in some inner scope. Note that you cannot even use the variable name in a quoted list.

Both of the above restrictions could be removed by adding more knowledge into the system. The problem has been localized to four key functions which have to be rewritten so that they properly understand fexprs and macros. All of the rest of the package is correct as it stands.

There are two key limitations which stem from the fact that sequence functions are processed before other macros. The first limitation is that only sequence functions can expand into loop fragments. In particular, an ordinary macro cannot expand into code which is supposed to be a loop fragment. This will not work because the macro will not be expanded until after the loop it is in has already been completely constructed. Note that it is all right for a macro to contain a complete loop expression which will be converted into a loop as a whole by itself. The appropriate way to make macros describing loop fragments is to use DEFUNS. For example, compare the following two definitions of a loop fragment which enumerates the CARs of the elements of a list. Only the second one will work.

```
(defmacro car-Elist-buggy (input)
  (list 'car (list 'Elist input)))

(defunS car-Elist (input)
  (car (Elist input)))
```

Another consequence of the fact that the LETS macro package does extensive processing before other macros are expanded is that you cannot nest one of the expressional macros inside a call of a macro that looks inside of its argument. For example, even assuming that you define a SETF property for ELIST, you cannot write "(SETF (ELIST L) X)". The problem is that since the loop macros are expanded first, SETF will never get to see the ELIST. Also note that instances of loop macros are usually replaced by variables in the resulting loop. However, you can say things like the following "(SETF (CAR (ELIST L)) X)" because the SETF does not need to look at the argument of the CAR.

The Representation for a Fragment

Loop fragments are represented internally by the following structure:

```
(S-frag args returns
  icode code1 code2 pcode ucode)
```

The *args* field is a list of argument descriptors. Each descriptor is a list of four parts (*kind mode var info*) The symbol *var* is the name of the argument. Every internal use of the argument is represented by that symbol. There is one argument declaration for every input, and auxiliary variable used by the fragment. The order of the declarations is used to match the inputs up with parameters when the fragment is used. The symbol *var* is created by GENSYM and is guaranteed to be unique and occur only in this single fragment so that SUBST can validly be used to rename it. If the fragment is copied, then the *vars* are renamed to new unique GENSYMs.

Note that free variable inputs and outputs are not mentioned in the argument descriptors. Rather, they are just referred to in the body of the fragment where appropriate. Due to the fact that the order of execution in a loop expression is preserved, things work out all right when fragments are combined together without the system having to take any explicit action. In fact, the system ignores the presence of free variables entirely.

The *kind* is one of four keywords indicating what kind of input is being described.

&INPUT - An obligatory input passed in by nesting in argument position.

&OPTIONAL - An optional input passed in by nesting in argument position. When the fragment is applied to arguments this variable is either converted to an &INPUT variable or an &AUX variable. The *info* field is an expression which will be used to compute a value which will be used to initialize the variable when no argument is supplied.

&REST - This is bound to a list of the remaining nested inputs (if any). There can be at most one &REST variable.

&AUX - An internal auxiliary variable.

The *mode* field specifies the kind of value which is contained in the variable. It is one of three keywords.

&UNITARY - This is a unitary value which is available before the repetitive computation of the loop begins.

&SEQUENCE - This is a sequence value. A new element of the value is computed on each cycle of the loop.

&END-UNITARY - This is a unitary value which is not available until after repetitive computation of the loop ends.

The *returns* are also a list of argument descriptors which are very much the same as those for the *args* except for a few key points. First, the *var* does not have to be unique. Rather, it can directly refer to one of the *vars* in the *args*. This indicates that this value is going to be directly exported from the fragment. If it is unique then it indicates that an internal variable must be maintained in order to store the value which will be exported. Second, the *kind* field is one of the following two keywords.

&OUTPUT - The is an ordinary output passed out as the return value. There must be exactly one of these. The LETS macro package does not support multiple outputs from sequence functions.

&FLAG - This is an auxiliary flag used in filtered computations. The filtered sequences themselves are carried in separate variables. The *info* field is a list of all of the free variables and return values which are under the control of this filter flag.

The remainder of the fragment specifies the computation to be performed. The *icode* is a list of zero or more expressions which are executed exactly once just before the repetitive part of the loop is executed. The *icode* cannot refer to any sequence arguments. It can read only unitary inputs. It can write any aux, or unitary output. Its effect is to give initial values to variables. Typically, every unitary output is given some default value.

The *code1* and *code2* are the repetitive body of the fragment. They are the only places where sequence arguments can be referred to. Both of these are lists of zero or more expressions. Both of them are executed on every cycle of the loop and can read sequence values. The *code1* (but not the *code2*) can write sequence values.

There are two different slots here because of the following property. All the *code1* parts of all the fragments being used will be executed before all of the *code2* parts. This gives you control over what is going on. In particular all terminations are placed in *code1* parts. As a result, you can depend on the fact that the *code2* will not be executed on the cycle where the loop terminates. (The *code1* may be.)

The *pcode* is a list of zero or more expressions which is executed exactly once after the loop, if it terminates normally. It cannot refer to any sequence quantities. Its purpose is to perform epilog computations involving the unitary outputs.

The *ucode* is a list of zero or more expressions which is executed in an UNWIND-PROTECT wrapped around the loop eventually produced. It cannot refer to any sequence quantities. Its purpose is to perform epilog computations involving the unitary outputs which must be performed no matter how the loop is terminated.

Each of the seven basic sequence functions makes it possible to specify a particular piece of the fragment data structure. AT-START, MAPS. AT-END, and AT-UNWIND specify computation to be performed in the *icode*, *code1*, *pcode*, and *ucode* respectively. PREVIOUS specifies computation to be performed in the *code2* and an auxiliary variable to remember prior values. TRUNCATES specifies computation to be performed in the *code1* which is then tested by a COND which contains a DONE to trigger termination.

FILTERS specifies computation to be performed in the *code1* in order to compute a flag value. It also sets up the correct *info* field for the flag in order to specify what value is controlled. If this value is subsequently read by the *code1* or *code2* of another fragment, then a COND predicated on the flag will be used so that they will be evaluated only on those cycles where all of the filtered inputs are available.

The following examples illustrate the fragment representation. The first corresponds to the sequence function RLIST. Note the use of some *pcode* in order to reverse the list CONSed up. The second corresponds to ERANGE. Note the use of an optional parameter BY, the presence of a terminator, and that the incrementation of the state is placed in *code2* so that it will not be done on the cycle on which the loop terminates. The final fragment corresponds to FILTERS. Notice the flag variable and the output which comes directly from an input.

```
(S-frag ((&input &sequence item nil))                          ;Rlist
        ((&output &end-unitary result nil))
  ((setq result nil))
  ((setq result (cons item result)))
  ()
  ((setq result (nreverse result)))
  ())
```

```
(S-frag ((&input &unitary state nil)                        ;Erange
         (&input &unitary end nil)
         (&optional &unitary by 1))
        ((&output &sequence int nil))
  ()
  ((cond ((> state end) (done))) (setq int state))
  ((setq state (+ state by)))
  ()
  ())

(S-frag ((&input &unitary fn nil)                           ;filterS
         (&input &sequence main nil)
         (&rest &sequence others nil))
        ((&output &sequence main nil)
         (&flag &sequence flag (main)))
  ()
  ((setq flag (apply fn (list* main others))))
  ()
  ()
  ())
```

All of the internal macro processing revolves around fragments represented in the above form. They are combined together into larger and larger fragments and then converted into normal loop code.

Locating Loop Expressions

Before a loop expression can be processed, it has to be located in its entirety. There are two ways in which this can happen. The first case is when the loop is delimited by a LETS* or DEFUNS. In that situation there is no difficulty in identifying it. The second case occurs whenever any of the sequence functions is encountered unexpectedly (i.e., not during the processing of a loop which has already been located). When this happens, the sequence function application is wrapped in a LETS* and processing continues as if the LETS* had always been there. Note that if a loop is nested inside another one, the inner loop must be explicitly placed in a quoted LAMBDA or a LETS*. One effect of this is that it is trivial to locate such inner loops.

LetS*

One purpose of a LETS* is to delineate a loop as discussed above. The other is to define sequence variables. All of the variable value pairs are handled as shown below by putting the initializing expressions inside the LETS*. Note that this means that these expressions cannot refer to the values which any of the bound variables have outside of the LETS*. The macro S-DESETQ is used to handle destructuring. The LETS macro package provides its own destructuring macro because DESETQ is not a standard LispMachine form.

```
         (letS* ((x (Erange 1 10))
                 ((a b) (Elist list)))
           (reverse (Rlist (list 'item x (+ a b)))))
becomes: (letS* (x a b)
           (setq x (Erange 1 10))
           (s-desetq (a b) (Elist list))
           (reverse (Rlist (list 'item x (+ a b)))))
```

Note that the only variables which carry sequences inside the body of the LETS* are the ones specified in the bound variable list. All of the free variables referred to in the body are unitary no matter what they are in the place where they are defined. This reflects the fact that if this loop is nested in another loop then it will be MAPSed and so any sequences in that loop will look like unitary values from its point of view. Note that if sequences were multidimensional objects as in APL then things would be much more complicated because each level of looping would only strip a single dimension off of a sequence.

Implicit MapS and Coercions

The processing of the body of a LETS* starts by parsing the body and performing any necessary coercions. The parsing is done by recursive decent. Each argument to a sequence function is coerced to be of the type required by the function. While this is going on the body is checked to see that each sequence function has the correct number of arguments and that there are no improperly nested loops.

As a convenience when debugging, things are arranged so that if you do a MACROEXPAND-1 of a LETS* (or DEFUNS) the form you get shows the result of parsing without other operations being performed. The result of the parsing phase is illustrated by the following example.

```
           (macroexpand-1
             '(letS* ((x (Erange 1 (1+ limit)))
                      ((a b) (Elist list)))
                (reverse (Rlist (list 'item x (+ a b))))))

yields: (s-letS (x a b)
          (mapS-no-ret #'(lambda (v1) (setq x v1))
                       (Erange 1 (at-start #'(lambda () (1+ limit)))))
          (mapS-no-ret #'(lambda (v2) (s-desetq (a b) v2)) (Elist list))
          (at-end #'(lambda (v3) (reverse v3))
                  (Rlist (mapS #'(lambda () (list 'item x (+ a b)))))))
```

S-LETS (and S-DEFUNS) are special internal forms which are used to represent the result of the parsing process. MAPS-NO-RET is one of a group of special internal sequence functions which are used for coercions in situations where no actual value is desired.

Combining Fragments

Once all of the appropriate coercions have been applied, the fragments are combined together into one large fragment. First each sequence expression in the body is processed as described below. Then the resulting fragments are combined together using the same combination process which is used in the processing of the individual expressions. The single resulting fragment is then converted into loop code as discussed in the next section.

An application of a sequence function is processed in three stages. First, the &OPTIONAL and &REST arguments in the fragment representing the sequence function itself are processed. If a parameter is supplied for an &OPTIONAL argument, then the argument is converted into an ordinary &INPUT argument before further processing. If no parameter is supplied then an &OPTIONAL argument is converted into an &AUX argument and the initializing expression is used to give the argument a value either in the *icode* (if it is unitary) or in the *codel* (if it is a sequence value). Note that this expression is evaluated inside the fragment and can refer to all of the arguments which precede it. In order to handle &REST arguments, ordinary &INPUT arguments are first created for each actual parameter supplied. After this, each occurrence of the &REST argument in the body is replaced by a LIST of the newly created &INPUT parameters.

Second, each of the parameters to the sequence function which is not a variable or a constant is recursively processed in order to convert it into a loop fragment. Third, each parameter fragment is combined into the main fragment being applied as follows. If a parameter is a constant or variable then it is substituted into the main fragment in place of the input variable.

Substitution is used instead of simply using a SETQ to transfer the value in order to reduce the number of variables which are necessary. The system checks carefully to see that substitution can actually be done. If the destination fragment modifies an input variable then a SETQ must be used if the source variable must be protected. Considerable care is taken in the exact way in which the built-in sequence functions are defined in order to maximize the readability of the loop code which will eventually be produced and in order to minimize the number of variables which will be needed.

If a parameter is a fragment the input variable is renamed to be the same as the output variable carrying the return value and the two fragments are combined as shown below. The new fragment is created merely by concatenating the corresponding parts of the two initial fragments. As a result, the order of evaluation is preserved. Note that due to the renaming of the input variable the data flow works out right without any special processing being necessary. Also since all variable names in the original fragments were GENSYMs there is no possibility of unintentional name clashes.

```
          ((S-FRAG argsa returnsa icodea code1a code2a pcodea ucodea)
           (S-FRAG argsb returnsb icodeb code1b code2b pcodeb ucodeb))
becomes:  (S-FRAG argsa-argsb returnsa-returnsb
            icodea-icodeb
            code1a-code1b
            code2a-code2b
            pcodea-pcodeb
            ucodea-ucodeb)
```

The only complexity is involved with filters. If any of the variables read by *code1b* or *code2b* are controlled by filter flags in the first fragment, then both *code1b* and *code2b* are nested in CONDs predicated on the AND of these flags. For example, suppose that *code1b* reads two sequence variables S1 and S2, which are controlled by the flags F1 and F2 respectively. In this case, *code1b* would be converted to (COND (AND F1 F2) . *code1b*) before combination. *Code2b* would be converted analogously. The *info* fields of the flag outputs in *returnsa* are updated to reflect the fact that they are now also controlling all of the outputs in *returnsa.*

The Form of the Loops Produced

Once all of the fragments have been combined into a single large fragment, this fragment is converted into a loop as indicated below. The various parts of the fragment are merely concatenated together into the body of a PROG. *Var-list* is a list of all of the aux, flag, and return variables which are specified in *args* and *returns.* The *return-value* is the &OUTPUT variable from *returns.* If the fragment contains any *ucode* then the PROG produced is wrapped in an UNWIND-PROTECT containing this *ucode.*

```
          (S-FRAG args returns icode code1 code2 pcode nil)
becomes:(PROG T var-list
                icode
            L   code1
                code2
                (go L)
            E   pcode
                (RETURN-FROM T return-value))
```

Note that the PROG produced is just basic Lisp. (On the LispMachine this PROG is named T so that it will be transparent to the user.) The PROG contains a number of variables and tags created by the macros. These are all GENSYMs and so that they cannot conflict with any user variables. All of the variables specified by the user become variables in the PROG. At a break point, you can look at these variables in order to see the current element in each of the corresponding sequences.

The form (DONE) expands into (GO E). The form (DONE *result*) expands into (RETURN-FROM T *result*). Note that no special action is taken with regard to terminations, they just end up in the right places as things are combined together.

Apply Simplification

The LETS macro package makes a special effort in order to ensure that functional arguments to sequence functions get efficiently compiled inline. Consider for example, what happens to the call on FILTERS shown below. Substituting the arguments into the FILTERS fragment (shown earlier in this appendix) produces (among other things) the APPLY shown. Using special knowledge of APPLY, LIST*, and LIST the arguments are substituted into the lambda body as shown.

```
           (filterS #'(lambda (a b) (> (car a) (car b))) sequence1 sequence2)
 produces: (setq flag (apply #'(lambda (a b) (> (car a) (car b)))
                             (list* sequence1 (list sequence2))))
 becomes:  (setq flag (> (car sequence1) (car sequence2)))
```

DefunS

The purpose of a (DEFUNS *name lambda-list . body*) is to define a sequence function. The *body* is exactly like the body of a LETS*. In addition the aux variables in the *lambda-list* are just like LETS* variables. These variables and the *body* are processed exactly as described above in order to create a fragment. The arguments in the *lambda-list* specify that some of the free variables in the fragment are actually non-free inputs. The fragment is modified to reflect this. Note that these variables must be unique in the body so that the system can use SUBST to rename them. A sequence function macro is then constructed with the appropriate *name*.

Appendix B: Functional Summary

This Appendix is intended as a short reference manual for the system. It assumes that you have already read the rest of the paper and just gives a very brief description of each of the macros available to the user. (Note that all of these macro names are global on the LispMachine.) The macros are listed in logical groupings. The summary begins with a description of the basic macros.

`letS*` `((`*var value*`) ...) &rest` *body*

This has two purposes: to define a group of variables which contain sequences of values, and to indicate that a group of sequence expressions (the *body*) should be combined together into a single loop. Each *value* will be coerced to a sequence. If it is omitted (or if the var-value pair is rendered as merely a symbol) then the initial value is undefined and the variable must be written before it can be read. A tree of *vars* instead of a symbol can be specified, in which case destructuring is performed. Note that every free variable is per force unitary. In the body, you can use `SETQ` to assign to a sequence variable.

All of the expressions in the body are combined into a single loop. Each unitary expression in the body will be automatically `MAPS`ed if possible. The only time it is not possible is if it uses the output of some reducer. In this latter case, the expression will be automatically computed `AT-END`. The value of the last expression in the *body* is coerced to unitary and returned as the value of the loop.

`defunS` *name lambda-list* `&rest` *body*

The purpose of this form is to define a new sequence function. The *lambda-list* is just like an ordinary lambda list except that in addition to the keywords `&OPTIONAL`, `&REST`, and `&AUX` it supports two additional keywords: `&UNITARY` and `&SEQUENCE`. `&UNITARY` indicates that following arguments are unitary. This is the default to start with. `&SEQUENCE` indicates that the following arguments carry sequences. Just as in an ordinary `DEFUN`, the user can include an optional documentation string and/or a declaration specifying the arg-list, after the lambda list.

`DEFUNS` defines a macro of the specified *name* defining the sequence function specified by *body*. The *body* is exactly like the body of a `LETS*` except that it is not immediately coded up into a loop, and the value of the last expression is not coerced to unitary. Rather, this value is returned whether it is unitary or a sequence.

`done` `&optional` *result*

In a loop expression the macro `DONE` can be executed in order to indicate that the loop should be immediately terminated. If no *result* is specified, then the loop will be terminated normally executing all `AT-END` code, and returning the result specified by the last expression. If a *result* argument is supplied then it will be returned as the value of the loop. Note, however, that in this case any `AT-END` code will be skipped. Any `AT-UNWIND` code is executed in either case.

The Basic Sequence Functions

There are seven basic sequence functions which support the basic capabilities of the expressional loop notation. All of the other sequence functions are defined in terms of these functions. Note that all of the basic sequence functions take functional arguments. These will be efficiently compiled inline as long as they are quoted functions.

at-start *function* &rest *args*

This computes (APPLY *function args*) in the initialization code before a loop begins. All of the *args* must be unitary values.

at-end *function* &rest *args*

This computes (APPLY *function args*) in the epilog code after a loop ends. All of the *args* must be unitary values. They can be values returned by reducers. Note that this will not be executed if the loop is terminated via a DONE with arguments or by some extraordinary exit such as a THROW.

at-unwind *function* &rest *args*

This computes (APPLY *function args*) in an UNWIND-PROTECT wrapped around the loop. All of the *args* must be unitary values. They can be values returned by reducers. The difference between this and AT-END is that it will be executed no matter how the loop is terminated.

mapS *function* &rest &sequence *args*

The nth element of the output sequence is computed by applying *function* to the nth elements of the input sequences. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. The length of the output sequence is the same as the length of the shortest input sequence.

e.g., `(mapS #'+ [1 _ 2 3 4] [1 2 _ 3]) => [2 _ _ 6]`

filterS *function* &sequence *source* &rest *args*

This embodies the idea of selecting particular items out of a sequence. The elements of the output sequence are computed as follows. If the result of applying *function* to the nth elements of the input sequences (the *source* and *args*) is non-NIL then the nth element of the *source* is used as the nth element of the output; otherwise the nth output element is empty. However, if the nth element of any of the input sequences is empty then *function* is not applied and the nth element of the output is empty. The output sequence is exactly the same length as the shortest input sequence; however, some of the output sequence slots may be empty.

e.g., `(filterS #'> [1 _ 2 3 4] [0 0 _ 0]) => [1 _ _ 4]`

truncateS *function* &sequence *source* &rest *args*

This embodies the idea of terminating a loop. The *function* argument is applied to successive groups of corresponding elements of the input sequences. The output sequence is composed of the elements of the *source* up to but not including the first element corresponding to a non-NIL evaluation of *function*. As with the other sequence functions, if any of the nth elements of the input sequences are empty then *function* is not applied and the nth output element is empty. Note that the output sequence is typically shorter than any of the input sequences, and can be of length zero.

e.g., `(truncateS #'< [1 _ 2 3] [0 0 _ 4]) => [1 _ _]`

e.g., `(truncateS #'> [1 _ 2 3] [0 0 _ 4]) => []`

previouS *init function* `&rest &sequence` *args*

This sequence function embodies the idea of feedback between cycles of a loop. It takes in a group of sequences and returns a sequence. If there are no empty slots in any of the inputs then the first element of the output is the value *init* and the nth element of the output is computed by applying *function* to the (n-1)th elements of the input sequences. If there are empty slots then this is generalized as follows. The nth slot of the output is empty if and only if the nth slot of any input is empty. The first non-empty slot of the output contains *init.* after that each non-empty slot is computed by applying *function* to the previous group of non-empty input values. The length of the output sequence is the same as the length of the shortest input sequence. (Note that *function* is applied to the last values of the input sequences even though the result is not part of the output.)

e.g., `(previouS NIL #'ncons [_ A _ B C]) => [_ nil _ (A) (B)]`

Five additional sequence functions are defined which embody stereotyped uses of `PREVIOUS`.

generateS *function init* `&rest &sequence` *args*

This uses an internal state variable in order to generate a potentially infinite sequence of values. The unitary value *init* specifies the initial (first) value of the state. On the nth cycle of the loop, *function* is called with the nth value of the state as its first argument and the nth elements of the input sequences (if any) as its remaining arguments in order to compute the next value of the state. However, if the nth element of any of the input sequences is empty then *function* is not called and the value of the state is not changed. The output sequence consists of all of the values of the state including the first one *init.* If there are no input sequences (the normal case) or if none of them are finite, then the output will be infinite. If any of the input sequences is finite, then the length of the output will be the same as the length of the shortest input. Note that in this case, the final value of the state will not be returned as part of the output.

e.g., `(generateS #'1+ 0) => [0 1 2 3 4 5 6 7 ...]`

enumerateS *truncate-function generate-function init*

This is simply `(TRUNCATES` *truncate-function* `(GENERATES` *generate-function init*`))`. It is the preferred way to define an enumerator.

e.g. `(enumerateS #'zerop #'1- 5) => [5 4 3 2 1]`

scanS *function init* `&rest &sequence` *args*

This is just like `MAPS` except that it has an internal state variable. The initial (zeroth) value of this variable is the unitary value *init.* The elements of the output are the successive values of the state not including its zeroth value. The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the sequence inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied, the state is not changed, and the nth element of the output is empty. The length of the output sequence is the same as the length of the shortest input sequence.

e.g., `(scanS #'+ 0 [1 _ 2 3 4]) => [1 _ 3 6 10]`

reduceS *function init* `&rest &sequence` *args*

This creates a sequence function with an internal state variable. The state is initialized to the (unitary) value *init.* The nth value of the state is computed by calling *function* with the prior value of the state as its first argument and the nth elements of the inputs as its remaining arguments. However, if the nth element of any of the input sequences is empty then *function* is not applied and the state is not changed. When the input sequences are exhausted, the final value of the state variable is returned as the (unitary) result. If there are no non-empty elements in the input sequences then the value *init* will be returned.

e.g., (reduceS #'+ 0 [1 2 _ 3] [1 _ 2 3 4]) => 8

e.g., (reduceS #'+ 0 [] [1 _ 2 3 4]) => 0

Pvalue *sequence* &optional (*first* NIL)

This takes in a sequence and returns a sequence which is shifted right one position. *First* is used as the first element of the output, and the last element of the input is discarded. Note that while the elements in the non-empty slots are shifted right, the pattern of empty slots remains fixed.

e.g., (Pvalue [1 _ 3 4] 0) => [0 _ 1 3]

Predefined Generators

Gsequence *arg*

This takes in a unitary argument and produces an infinite sequence of that value. Note that the successive elements of the sequence will all be EQ.

e.g., (Gsequence 1) => [1 1 1 ...]

Glist *list*

This generates the successive elements of *list*. It will get an error if it encounters a non-list CDR.

e.g., (Glist '(1 2 3)) => [1 2 3 NIL NIL NIL ...]

Gsublists *list*

This generates the successive CDRs of *list*. It will get an error if it encounters a non-list CDR.

e.g., (Gsublists '(1 2 3)) => [(1 2 3) (2 3) (3) NIL NIL NIL ...]

Grange &optional (*first* 1) (*step-size* 1)

This generates fixnums from *first* adding *step-size* at each step. Note that *step-size* can be negative.

e.g., (Grange 10 2) => [10 12 14 ...]

Predefined Enumerators

Elist *list*

This enumerates the successive elements of *list* up to and not including the first NULL sublist. It will get an error if it encounters a non-list CDR.

e.g., (Elist '(1 2 3)) => [1 2 3]

e.g., (Elist nil) => []

Esublists *list*

This enumerates the successive CDRs of *list* up to and not including the first NULL sublist. It will get an error if it encounters a non-list CDR.

e.g., (Esublists '(1 2 3)) => [(1 2 3) (2 3) (3)]

Elist* *list*

This enumerates the successive elements of *list* up to and including the first NULL or non-list sublist.

e.g., (Elist* '(1 2 . 3)) => [1 2 3]

e.g., (Elist* NIL) => []

Eplist *plist*

This creates a sequence of pairs of successive property names and property values of the naked plist *plist.* Note that the function PLIST returns the CDR of a naked plist, not a naked plist.

e.g., (Eplist '(NIL A 1 B 2)) => [(A . 1) (B . 2)]

Ealist *alist*

This creates a sequence of pairs of successive keys and values respectively of *alist.* It requires that the lists of values associated with each key be a non-NIL list.

e.g., (Ealist '((A 1) (B 2 3))) => [(A . 1) (B . 2) (B . 3)]

Erange *first last* &optional (*step-size* 1)

Creates a sequence of integers by counting from *first* to *last* by the positive increment *step-size.*

e.g., (Erange 4 8 2) => [4 6 8]

Evector *vector* &optional (*first* 0) (*last* (1- (array-length vector)))

This enumerates the successive elements of a one dimensional array. You can specify a subrange of indices by specifying *first* and *last.* (Note that this will not work on MacLisp arrays of numeric type.)

e.g., (Evector <1 2 3>) => [1 2 3]

Efile *file-name*

This creates a sequence by doing successive reads on the file until end of file is reached. *File-name* can be anything acceptable to OPEN.

e.g., (Efile "data.lisp") => [1 2 3]
    if the file contains "1 2 3 "

Predefined Filters and Terminators

Fselect &sequence *source boolean-sequence*

This creates a sequence whose values are the values of *source* corresponding to non-NIL values of *boolean-sequence.*

e.g., (Fselect [1 2 3 4] [NIL T T NIL]) => [_ 2 3 _]

Fpositive &sequence *sequence*

This takes in a sequence of fixnums and restricts it to a sequence containing only elements greater than zero.

e.g., (Fpositive [-1 0 1]) => [_ _ 1]

Fgreater &sequence *sequence* &optional &unitary *limit*

This takes in a sequence of fixnums and restricts it to a sequence containing only elements greater than *limit.*

e.g., (Fgreater [1 2 3] 2) => [_ _ 3]

Tselect &sequence *source boolean-sequence*

This creates a sequence whose values are the values of *source* up to and not including the one corresponding to the first non-NIL value of *boolean-sequence.*

e.g., (Tselect [1 2 3 4] [NIL T T NIL]) => [1]

Predefined Reducers

Rlast &sequence *sequence* &optional &unitary (*default* NIL)

This takes in a sequence and returns its last value. If the sequence has zero length then *default* is returned.

e.g., (Rlast [1 2 3]) => 3

e.g., (Rlast []) => NIL

Rignore &sequence *sequence*

This takes in a sequence and returns NIL. It is useful in many of the same situations as **MAPC**.

e.g., (Rignore [1 2 3]) => NIL

Rlist &sequence *sequence*

This creates a list of the elements in *sequence.* The order of the elements is preserved.

e.g., (Rlist [1 2 3]) => (1 2 3)

e.g., (Rlist []) => NIL

Rbag &sequence *sequence*

This creates a list of the elements in *sequence.* The order of the elements in the list is undefined. This is more efficient if you really do not care what the order is. (The order ends up reversed, but you should not depend on that, because it could change at any time.)

e.g., (Rbag [1 2 3]) => (3 2 1)

Rlist* &sequence *sequence*

This creates a list of the elements in *sequence* with the last element of the *sequence* ending up as the CDR of the last CONS cell in the list.

e.g., (Rlist* [1 2 3]) => (1 2 . 3)

e.g., (Rlist* [1]) => 1

e.g., (Rlist* []) => NIL

Rnconc &sequence *sequence*

This creates a list by NCONCing together the successive elements of *sequence.* This is what **MAPCAN** does to create its output.

e.g., (Rnconc [(1 2) NIL (3 4)]) => (1 2 3 4)

Rappend &sequence *sequence*

This creates a list by APPENDing together the successive elements of *sequence.*

e.g., (Rappend [(1 2) NIL (3 4)]) => (1 2 3 4)

Rset &sequence *sequence*

This combines the elements in *sequence* into a list omitting any duplicate elements. The order of this list is undefined. The predicate which is used to test for duplicates is EQUAL.

e.g., (Rset [A A (B) (B)]) => ((B) A)

Reqset &sequence *sequence*

This is the same as RSET except that the test for duplicates is EQ instead of EQUAL.

e.g., (Reqset [A A (B) (B)]) => ((B) (B) A)

Rplist `&sequence` *properties values*

This takes in a sequence of property names, and a sequence of values and creates a naked plist. Note that the function `SETPLIST` expects to receive the `CDR` of a naked plist as its second argument.

e.g., `(Rplist [A B] [1 2]) => (NIL A 1 B 2)`

Ralist `&sequence` *keys values*

This takes in a sequence of keys, and a sequence of values and creates an alist. All of the values which have the same key are combined into a single entry in the alist headed by the key. The predicate which is used to test for equality of keys is `EQUAL`.

e.g., `(Ralist [(A) B (A) B] [1 2 3 4]) => ((B 4 2) ((A) 3 1))`

Reqalist `&sequence` *keys values*

This is identical to `RALIST` except that the test for key equality is `EQ`.

e.g., `(Reqalist [(A) B (A) B] [1 2 3 4]) => (((A) 3) (B 4 2) ((A) 1))`

Rvector *vector* `&sequence` *sequence* `&optional &unitary` (*first* `0`) (*last* `(1- (array-length vector)))`

This takes in a one dimensional array and a sequence of elements and stores those elements in successive positions in the array. You can specify a specific subrange in the array. (This will not work with MacLisp arrays of numeric type.) Note that this reducer is unusual in that it contains a terminator and will stop the loop as soon as the vector is full.

e.g., `(Rvector <NIL NIL NIL NIL> [1 2 3]) => <1 2 3 NIL>`

e.g., `(Rvector <NIL NIL> [1 2 3]) => <1 2>`

Rfile *file-name* `&sequence` *sequence*

This takes in a sequence and writes all of its elements into a file. *File-name* can be anything acceptable to `OPEN`.

e.g., `(Rfile "data.lisp" [1 2 3]) => T`

`"<cr>1 <cr>2 <cr>3 " is printed in "data.lisp"`

Rsum `&sequence` *integers*

Computes the sum of the integers in its input.

e.g., `(Rsum [1 2 3]) => 6`

Rsum$ `&sequence` *flonums*

Computes the sum of the flonums in its input.

e.g., `(Rsum$ [1.1 2.2 3.3]) => 6.6`

Rmax `&sequence` *numbers*

Computes the maximum of the numbers in its input. Returns `NIL` if the input has length zero.

e.g., `(Rmax [1 2 3]) => 3`

e.g., `(Rmax []) => NIL`

Rmin `&sequence` *numbers*

Computes the minimum of the numbers in its input. Returns `NIL` if the input has length zero.

e.g., `(Rmin [1 2 3]) => 1`

e.g., `(Rmin []) => NIL`

Rcount &sequence *sequence*

Computes the number of elements in its input.

e.g., `(Rcount [1 2 3]) => 3`

Rand &sequence *sequence*

Computes the `AND` of all of the elements of *sequence*. As with `AND`, the return value is either `NIL` or the last element of the input.

e.g., `(Rand [1 2 3]) => 3`

e.g., `(Rand [1 NIL 2]) => NIL`

e.g., `(Rand []) => T`

Rand-fast &sequence *sequence*

This is the same as `RAND` except that the loop is terminated as soon as a `NIL` value (if any) is encountered.

e.g., `(Rand-fast [1 2 3]) => 3`

Ror &sequence *sequence*

Computes the `OR` of all of the elements of *sequence*. As with `OR`, the return value is either `NIL` or the first non-`NIL` element of the input.

e.g., `(Ror [1 2 3]) => 1`

e.g., `(Ror [NIL NIL]) => NIL`

e.g., `(Ror []) => NIL`

Ror-fast &sequence *sequence*

This is the same as `ROR` except that the loop is terminated as soon as a non-`NIL` value (if any) is encountered.

e.g., `(Ror-fast [1 2 3]) => 1`