AN ENVIRONMENT FOR MULTIPLE-VALUED RECURSIVE PROCEDURES

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 40

AN ENVIRONMENT FOR MULTIPLE-VALUED RECURSIVE PROCEDURES

DANIEL P. FRIEDMAN

DAVID S. WISE

OCTOBER, 1975

Research reported herein was supported (in part) by the National Science Foundation under grant no. DCR75-06678 and no. MCS75-08145.

To be presented at the 2nd Symposium on Programming (Paris, 1976).

An Environment for Multiple-valued Recursive Procedures

Daniel P. Friedman
David S. Wise
Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract - A problem with existing language provisions for programming procedures which return multiple results is the facility with which such results are passed along as arguments to other functions. We define the functional operation of combination which provides a method of using such multiple results conveniently, say, for writing a multi-value procedure which recurses directly on itself. "Lists" of functions, called combinators, take lists of arguments and the length of the result is determined by the shortest of the combinator or its arugments. Notation is presented for constructing lists which are arbitrary repetitions of a single element so that one parameter can be "spread" across a combinator, or a single function can be spread into a combinator across lists as arguments. Finally, the provision of a functional application notation is introduced and shown to be a very important adjunct to the other features of our enhanced recursive language. Several examples of purely recursive code illustrate its power, including one on coroutining and one on batch searching.

Keywords and Phrases - functional combination, pure recursion, functional application, form evaluation, extractor, mapping functions, projections, LISP, lexpr, expr.

CR Categories - 4.22, 4.12, 4.13.

Introduction

The importance of functions (procedures) in programming has been demonstrated by several languages which depend solely on function or macro invocation to control computations. The central theme of this paper is enhancement of the standard treatment of functions. This enhancement not only facilitates multiple results from functions (i.e. multi-dimensioned range), as in LISP or APL, but also provides for the use of such a multiple-value as an argument to other functions without requiring its explicit decomposition.

The value of a function which returns a multiple result may be pictured as a fixed length vector in algebraic languages or as a list in list processing languages. If the language under consideration depends on function invocation for program control, then those vectors or lists must be allowed as arguments, as well as results, for functional composition. When the multiple result of one function is to be an argument to the next, treatment of the formal parameter which represents the multiple result as a data structure of several results becomes necessary. Extra assignments or nested functions can be used to rebind elements of that structure to individual variables for access by the next function, but such intervening overhead is confusing. The programmer bothered with unnecessary rebindings is more likely to make mistakes; the reader of such a program is more likely to be confused; the compiler of such a program will likely generate less efficient code then would be possible if a standard access protocol were available.

The immediate need for a protocol for linkage of such arguments arose from an ongoing project aimed at the development of a practical translator of "stylized recursions" [F&W74] into stackless iterations. Early in this project the decision was made to restrict the source language to the purer forms of recursion. This frustrated attempts to provide legal function definitions which could be translated to code like that for common iterative programs which yield several results. The preimage of such an iteration under such a translation must be recursive and must return several values. Stylized preimage code is possible if the tool of "functional combination" introduced in this paper is included as a technique of stylized recursion.

Although functional combination can be simulated at the cost of the rebindings discussed above, it does not exist in any major programming language. That fact is a bit surprising after one considers a few examples of its use. It is particularly powerful when a called function has already been defined to return several results in a non-trivial structure. Such is usually the case when a multiple-value-returning function is being defined recursively and it, itself, is the called function. Highly recursive languages, like pure LISP, can use the expressive power particularly well, and we therefore choose a form of LISP as the main communication language below. We refer the reader to [Wei67] for an explanation of list processing and to [Fri74] for an introduction to recursive programming in LISP.

Functional combination is a scheme for weaving a new function from conceptually parallel invocations of extant functions, so that the structures of the parameters and of the results for the new function are determined by a fixed arrangement of those structures and the original functions. It is a version of the Caretsian product of functions in common use by algebraists [M&B67,B&L74], modified to make the parameter-structure more like the structure of the result and generalized using a star notation to allow tuples of a function (or an argument) to be of indefinite dimension. The first change makes it a comfortable tool for composing functions in a programming language. The latter generalization extends the formalism to practical cases wherein the determination of the length of the result can be deferred until data is available.

The remainder of this paper is in seven sections followed by conclusions. The first section on motivation is followed by a section presenting notation. The third section introduces functional combination in its elementary form, which is extended in the fourth section to allow for structures of arbitrary size. These sections present elementary examples of these functional construction techniques, which become a foundation for two sections on the implications of this scheme for the function interface and upon the need for functions defined for an arbitrary number of arguments. The final section presents two uses of functional combination, each presenting recursive definitions for algorithms which require special techniques in iterative languages.

Motivation

Consider the common problem in statistical analysis of computing the mean and variance of a set of data. The principal computational step is to sum the list of data and to derive the sum of their squares. The two PASCALesque [Wir71] functions below represent a standard recursive scheme to derive each of these results:

```
function sigma (var arr: vector; n: integer): real;
     sigma := if n=0 then 0.0
                       else arr[n] + sigma(arr,n-1)
```

and

```
function sumsq (var arr: vector; n: integer): real;
     sumsq := if n=0 then 0.0
                       else arr[n] * arr[n] + sumsq(arr,n-1) .
```

Because these functions are often used together, one might consider a syntax for invoking them simultaneously. That is, the expression [sigma,sumsq](a,15) might be expected to return the ordered pair [sigma(a,15),sumsq(a,15)] . An advantage of dispatching the function calls simultaneously is that the compiler would be allowed to provide for simultaneous computation as a single task. In this example simultaneous computation could result because the recursion patterns are identical. If the recursion paths part at some point in the computation then separate tasks would be invoked. Note that the second (ordered pair) expression above requires collateral elaboration in the ALGOL 68 [vWi69] sense.

A well founded criticism of recursive programming has been its limited capacity to express functions which return more than one value [Lan66]. When the program above is written with an iterative loop, two registers are required to maintain the sums; when written recursively (in PASCAL) one register might contain the value of the recursive function and the other register could transmit its value through a parameter passed _specifically_ to receive the second value. With careful study of the implications of reference parameters used with recursion, one might discover the following recursive function:

```
function sigma (var sumsq: real; var arr: vector; n: integer): real;
    if n=0 then
        begin
           sigma := 0.0;
           sumsq := 0.0
        end
    else
        begin
           sigma := arr[n] + sigma(sumsq,arr,n-1);
           sumsq := arr[n] * arr[n] + sumsq
        end .
```

The envisioned use of the above code would require compilation of the recursion into the standard style iterative code. However, the problems of compiling recursions with reference parameters is so complex that one quickly wishes them away and turns to the problem of compilation of pure recursion, which uses only _value_ parameters [D&B73,B&D75,Ris73,F&W74].

The syntax described below allows the function invocation given by [sigma(a,15),sumsq(a,15)] to be expressed, in an extension of LISP (as described in Chapter 1 of [McC62]), by ([sigma sumsq] *a *15). Moreover, it allows a trivial definition of a pure recursive function which returns the same pair from a _single_ recursion.

Defintions and notations

The term _list_ is used to refer to a linear structure of arbitrary length, the actual length to be determined at execution time by the structure of data. A _vector_ is a linear structure whose length may be determined by inspection of the code at program-definition time. For instance, if x is to be an arbitrary list (of length greater than one) which will be bound at execution time, then an occurence of (list (car x)(cadr x)) in the code may be described as a vector since it must have length 2; but an occurence of (cdr x) can only be a list.

Many functions can be generalized to take an arbitrary number of arguments. The term _lexpr_ (as opposed to _expr_, which takes a fixed number of arguments [McC62]) used in Stanford LISP 1.6 [Qua69] will describe such functions. For example, _sum_ is a lexpr which sums its arguments, _append_ is one which concatenates its arguments, and _equal_ is one which determines if its arguments are the same.

We present notations for elementary operations on sequential structures: square brackets for list building; angle brackets for functional application; integers as extractor functions; the use of the hash "#" symbol as an ignored argument. In LISP each of

these may be locally represented using existing notation, but only by allowing for some other programming structures which we wish to prohibit. These notations provide the expressive power which we need without allowing any dangerous generalities.

The notation [a b c] is used in two contexts for representing a vector or record of fixed length. In the function position it represents a functional combinator (discussed in the next section) formed from the functions a, b, and c. As an argument the notation is synonomous with (list a b c) used by Teitelman [Tei74] and Hewitt and Smith [H&S75]. Evaluation of such an expression results in a vector of the evaluated expressions within the brackets. The length of this vector is the same as the length of the bracketed expression. For instance, [(sum 4 2) 3 (product 3 3)] evaluates to (6 3 9)[1].

The function _apply_ takes a function and a list (or vector) of its arguments as parameters. A common use is the application of a fixed function to an arbitrary list of parameters which is yielded by an expression. In LISP this case often appears as (apply (function f) x) where evaluation of x yields a list of

---

[1] [] is a function _invocation_ which returns the empty list. This fact offers a very palatable alternative to the notational problems surrounding the atom NIL in LISP. NIL is a distinguished atom which at once represents the empty list and logical false. It appears frequently in programs in one role or the other, usually unQUOTEd because NIL is one of a very few atoms in LISP which is globally bound to itself. These two uses of NIL have previously been distinguished [Fri74] by systematically using F for _false_ and () for the empty list. The latter convention is distasteful because it depends on the global binding of NIL and on the behavior of the lexical scanner in interpreting as NIL the form () which appears to be a bogus function call. Therefore, when the empty list is needed, the notation [] will be used.

arguments, perhaps of fixed length. We define the notation <f x> for such an application in order to avoid the words *apply* and *function*; (f a b c) is usually[2] synonomous with <f [a b c]>. (See also [H&S75] and their discussion of message passing.) There are always exactly two elements between the angle brackets [Bac73]: a list or vector and a function. This notation for functional application is particularly useful for lexprs applied to data structures: if arr is a list of numbers, then <sum arr> sums its elements.

All integers are implicitly defined as lexprs. For example, the function (or projection) 17 returns its seventeenth argument. (3 (sum 4 2) 3 (sum 3 4) (product 5 7)) returns 7 and <3 x> returns the third element of the list x. An integer in the functional position within angle brackets is useful for extracting a specific element from a vector which results from a function invocation in the argument position. Since (car x) is the same as <1 x>, (cadr x) is the same as <2 x>, and so forth, function invocations returning multiple arguments are often the object of the application of integer functions to access a single piece. Such applications are called *extractions*.

The symbol # is used as a pseudo-argument. Within square brackets it is dropped during argument evaluation before parameter binding. The evaluation of # converges to some value which is dropped and is, therefore, immaterial. For example, <f [# a # b # # c]> is synonomous with <f [a b c]> .

---

[2] The exceptions involve *funarg* and #.

Functional Combination

In our scheme, when a bracketed form occurs in the function position of a form it is left unevaluated. A form which begins with a bracketed form is called a C-form (combinator form), and the car of the C-form is called a combinator [F&W75]. A combinator is composed of functions or combinators. Each argument for the combinator must also evaluate to a vector or be starred (discussed in the next section). The semantics of the evaluation for such a functional combination is as follows: let

$([f_1\ f_2\ \ldots\ f_{m_0}]\ \rho_1\ \rho_2\ \ldots\ \rho_n)$ be a C-form where each

ρ_i is $(a_{i1}\ a_{i2}\ \ldots\ a_{im_i})$; its value is $[<f_1\ \gamma_1>\ <f_2\ \gamma_2>\ldots\ <f_m\ \gamma_m>]$

where $m = \min\limits_{0\leq i\leq n} m_i$ and $\gamma_j = (a_{1j}\ \ldots\ a_{mj})$ for $j\leq m$ but excluding values $a_{ij} = \#$.

The argument, ρ_i, can be viewed as a row of a jagged matrix, $\{a_{ij}\}$, which is passed to the combinator in column-major fashion. The length of the vector which results from an invocation of functional combination is the minimum of the lengths of the combinator and of all its arguments. This definition allows the value of a C-form to be determined by the leftmost functions in the combinator if an argument yields a row, ρ_i, which is too short (i.e. $m_i < m_0$).

The examples will have their arguments written with each row on a separate line and vertically aligned to suggest the columnar relationship. Furthermore, the function names will be hyphenated to suggest the length of the vector result. For example,

(s-p-q-r [1 9 7 5] [0 1 1 10]) evaluates to (1 9 7 5)

where

```
     s-p-q-r (v w) ≡ ([sum product quotient remainder]
                                                  v
                                                  w ).
```

This example shows that C-forms can be interpreted out of the context of recursion, yet it is in that context where they will find the most use. The availability of commonly used alternatives (e.g. call-by-reference illustrated above) allows programmers to avoid the C-form with its unorthodox order of argument evaluation and parameter binding. It can promote terseness, however, in instances where assignment statements or auxiliary functions would be needed to rearrange argument order from that provided as the result of nested functions. That facility is important when one works with functions such as <u>divide</u> which return vectors (in [McC62] (divide x y) is defined to be [(quotient x y)(remainder x y)]). It is particularly important when calls on those functions are recursive calls, because the order of evaluation would be clouded by intervening rearrangements.

For the recursive definition of the summation function described earlier one would be very likely to treat arr as a list of numbers. Then the function, here named <u>len-sigma-sumsq</u>, would return an ordered triple: [the length, the sum, the sum of squares] and might be defined as follows:

```
(len-sigma-sumsq arr) ≡
      if (null arr) then [0 0 0]
                   else ([addl sum       sum]
                         [ #  (car arr) (product (car arr)(car arr))]
                         (len-sigma-sumsq (cdr arr))) .
```

For example, (<u>len-sigma-sumsq</u> [1 2 3]) evaluates to (3 6 14).

To complete the computation of mean and variance it is only necessary to invoke an auxiliary function of three parameters, al-pha, on the list which is the result of len-sigma-sumsq:

```
(al-pha len sigma sumsq) ≡
    [(quotient sigma len) (difference (quotient sumsq len)
                                      (square (quotient sigma len)) ) ];
(mean-variance arr) ≡ <al-pha (len-sigma-sumsq arr)> .
```

The example of mean-variance is chosen as a common example of a computation which returns two results from a single pass. The function, lt-eq-gt, has as its parameters a number, n, and a list, s, of numbers in ascending order and returns a vector of three lists as a result of its single recursion. Each member of the triple contains elements of s: [a list of numbers less than n, a list of numbers equal to n, and a list of numbers greater than n]. Thus (lt-eq-gt 5 [1 2 5 5 5 6 8 9]) is ((1 2)(5 5 5)(6 8 9)).

```
(lt-eq-gt n s) ≡
      if (null s) then [[][][]]
   elseif (greaterp (car s) n) then [[][] s]
   elseif (lessp (car s) n) then ([cons     l    l]
                                  [(car s)  #    #]
                                  (lt-eq-gt n (cdr s)))
                            else ([ l   cons     l]
                                  [ #   (car s)  #]
                                  (lt-eq-gt n (cdr s))) .
```

This example exhibits the use of the # sign as an ignored argument within a functional combination. Within a bracketed argument it sustains the length of the rows which are arguments to the combinator, and acts as a place marker which is ignored when the columns of values are passed as arguments to the elements of the C-form.

Star on arguments and functions

The notation *p will stand for a list of infinitely many instances of p and it is an instance of a new data type which is called starred. Any data type, including starred, can be starred. For instance **0 is "the zero matrix" of two dimensions and infinite size. Starred arguments to a bracketed inatance of a functional combinator act like a vector of the same length as the combinator or shortest unstarred ρ_i because of the minimization of row lengths:

```
    ([sum product]
         *2
      [ 3    4   ])
```

evaluates to (5 8) and

```
    <1 ([sum product]
             *3
             *3 )>
```

evaluates to 6.

A starred function may appear wherever a combinator may and is considered a special case of a combinator. This convention totally subsumes all uses of mapcar [Wei67]. Such a combinator acts as if it were molded to the length of the shortest unstarred argument with every element of the combinator being the function which was starred:

```
    (*sum
      [1 2]
       *5)
```

evaluates to (6 7). If all arguments are starred, then a starred

result may be computed. After a single star is "stripped" from the combinator and each argument, normal evaluation proceeds and finally, the star is "replaced:"

```
    (*sum
     *2
     *3)
```

evaluates to *5 . A starred result is useful only as an intermediate value. Its effective length may be fixed by the row minimization rule in subsequent and more rigorous invocations of functional combination.

Because the arguments of C-forms may be structures which are solely data dependent, many instances of starred functions within C-forms are expanded according to the data which is not available until execution time. For example, the <u>dotproduct</u>[3] of two linear arrays may be expressed as

```
(dotproduct arr1 arr2) ≡ <sum (*product
                                 arr1
                                 arr2 )> .
```

Their <u>outerproduct</u>, based on any function <u>prdct</u> is given by the function

```
(outerproduct arr1 arr2) ≡ (*scalarproduct
                             *arr1
                             arr2)
```

with

---

[3]For the masochistic, if a matrix is a list of rows then Gaussian matrix multiplication is

```
(matrixmpy m1 m2) ≡ (*row m1 *<*list m2>)
where
(row r mt) ≡ (*dotproduct *r mt) .
```

```
(scalarproduct arr elem) ≡ (*prdct
                              arr
                              *elem).
```

With the lexpr *equal* discussed above

```
    (*equal
       x1
       x2
       x3 )
```

returns a pattern of *true*'s and *false*'s whose length is that of the shortest of the lists x1, x2, and x3. The *true*'s indicate the position where all lists have the same element and the *false*'s indicate the positions where they differ.

Using the * notation, we may rewrite *len-sigma-sumsq* as follows:

```
(len-sigma-sumsq arr) ≡
    if (null arr) then *0
                  else (*sum
                         [1 (car arr)(product (car arr)(car arr))]
                         (len-sigma-sumsq (cdr arr))) .
```

The result of *len-sigma-sumsq* is usually separated by extraction using integer functions. For instance, if n-s-s is such a result then <1 n-s-s>, <2 n-s-s>, and <3 n-s-s> are the three pieces. If arr is initially empty then n-s-s is *0, but each of the three pieces is still zero; any extraction from a starred value yields that value.

A final example summarizes the generality of functional combination and demonstrates some of the possibilities for nesting combinators:

```
    ([2 *cons        [product quotient] quotient]
     [5 *0           [  5       20    ] ]
     [8 [[1 2][3 4]]        *5          0      ])
```

evaluates to (8 ((0 1 2)(0 3 4)) (25 4)) .

The function junction

The preceding discussion introduces functional combination with extractor functions and functional application. These programming structures vastly enrich the facility for function linkage by allowing multiple values to be handled easily. We have also introduced three pseudo-types for purposes of interfacing functions. These are starred values, heterogeneous vectors and homogeneous lists. Although some of these types are consistent with uses beyond the function interface (namely input and output for the program) this classification makes sense only at the interface. For this reason the types are called "pseudo."

The classification is useful for describing the allowable arguments and values of various kinds of functional combination. If the combinator is bracketed then arguments are usually heterogeneous vectors or starred values, and the result is a vector. If the combinator is starred, then allowable arguments are starred values and homogeneous lists; the result is a homogeneous list (unless all arguments are starred yielding a starred value). Although starred values and heterogeneous vectors may be treated as special cases of homogeneous lists which, in turn, are elementary data structures, usually these pseudo-types are used internally only by larger programs. The multiple results of one function should be passed to another combinator, to an extractor, or to an invocation of functional application with the multiple value as the parameter list.

When a multiple value is passed to extractors, it is common that each value is extracted from apparently duplicated invocations

of functional combination. (See equaltips below.) Duplicated invocations, particularly in a sequence of extractions, are intended to be collapsed into one whose result is stored locally for all to use.

The other use of multiple value is as a list of arguments for a functional application. The notation <f arglist> often has a C-form for arglist. The resultant type of the C-form should be consistent with f. If f takes a fixed number of heterogeneous arguments (an expr in LISP parlance) then the C-form should return a vector. If f takes an arbitrary number of homogeneous arguments (a lexpr) then the C-form may return a list. A starred value may be used in the former case where the parameter count is fixed, but not in the latter.

The above discussion on parameter linkage and result generation at the function junction has intentionally omitted any specification of the order in which independent events occur. For instance, the arguments to the functional combination may be evaluated in any order. The application of an element in the combinator to its column of arguments may occur whenever the column is available, perhaps before other columns become available. Since we are considering a side-effect free environment the unspecified nature of the evaluation protocol cannot affect final results. It does, however, permit implementors to take advantage of local machine characteristics. We intend that argument evaluation and application of combinators should be implemented using parallel processors wherever available. Furthermore, we anticipate that many instances of functional combination can be compiled to run on a single processor with no sacrifice in performance. Table 1 illustrates the various pseudo-types as parameters to each of the new kinds of function invocation given above: application of extractors, exprs, lexprs, and functional combination using bracketed and starred combinators.

[lexprs not obsolete]

The convenient angle bracket notation facilitates writing recursive lexprs. For instance, the function, max, which returns the largest of an indefinite number of arguments, is expressed in terms of the binary function greater which returns the larger of its two arguments. If x represents the list of evaluated arguments for max, then the body of max may be

```
    if (null (cdr x)) then (car x)
                          else (greater (car x) <max (cdr x)>) .
```

The use of functional application in defining this lexpr is a typical recursive call. An invocation of max appears as (max a b... z). Using the other bracketing notation this same invocation can be (max [a b ... z]) when max is defined as an expr:

```
(max x) ≡
    if (null (cdr x)) then (car x)
                          else (greater (car x)(max (cdr x))) .
```

This observation brings two interesting points to the surface. First, the respective codes for the lexpr and expr versions of max differ only in the bracketing used in expressing their recursive calls. Second, the convenience of the bracketing which allows trivial construction of the expr invocation suggests that all potential lexprs be written instead as exprs with bracketing changes at their invocations [H&S75].

Functional combination provides another sort of lexpr invocation which precludes such lexpr conversion. When a lexpr appears as an element of a combinator with its arguments extracted from the argument of the C-form it is impossible to use bracketing to assemble its

arguments into a list. For example, the C-form

```
    ([max cons min]
     [ 3    #    2 ]
     [ 7    5    # ]
          p-q-r
     [ #    #    9 ])
```

where p-q-r is bound to (1 (6 7 8) 1) evaluates to (7 (5 6 7 8) 1). If <u>max</u> and <u>min</u> were written as one argument exprs there would be no way to avoid errors in the parameter binding conventions for C-forms in any version of this example.

The above example shows that the bracketing conversion does not avoid the need for lexprs. The same argument shows that the availability of the notation [a b c] for (list a b c) does not obviate the need for the lexpr, <u>list</u>. In Footnote 3 we applied <u>*list</u> as a combinator in order to transpose a matrix. Without a name for the function of arbitrarily many arguments performed by the brackets it would have been impossible to use that function within a combinator.

The integers, defined as extractor functions, are also lexprs but all uses of them hitherto have been under functional application; which could be handled if integers were exprs which extracted from their single argument. That would allow <2 x> to be written as (2 x). The reason that the lexpr definition, which allows redundancies like (2 a b c d) as b, was chosen again lies in the use of extractor functions within combinators. In that role the ith extractor selects the value in the ith row of its column which is a very useful projection function. It plays the role of <u>progi</u> (i.e. prog1, prog2, or progn), a function which was long thought to have no role in pure LISP. In an environment enriched with functional combination, however, this assumption collapses.

Examples using the extended LISP

With the notational extensions and the functional combination tool in hand, we present two examples to demonstrate the power of these extensions. The first illustrates the use of direct recursive calls as elements of a combinator and the second illustrates how these extensions handle a problem from coroutining lore.

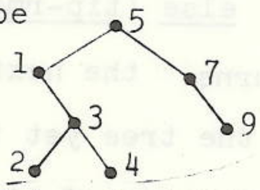
The first example is borrowed from the area of searching and sorting [Knu73]. Let tree be a binary search tree (one whose inorder [Knu68] traversal visits the nodes in order of their distinct values). Let s be a sorted list of (perhaps duplicated) elements with the same ordering relation. For simplicity, assume we are dealing with numbers and let s and the inorder of tree be in ascending order. The function, batch, returns a list whose i^{th} element is either the subtree of tree whose root has as its key the ith element of s or the empty list if the ith element does not occur in tree. The previously defined function, lt-eq-gt, whose value is a list of three sorted lists is an auxiliary function of batch.

The program below performs a binary search for each element of s on tree. It does this, however, with each node of tree visited at most once[4]. Combinators allow us to solve this problem by taking advantage of the efficiencies of batch-probing complex data structures [S&G75]. Each node has three fields: key, left, and right [Knu68, page 424].

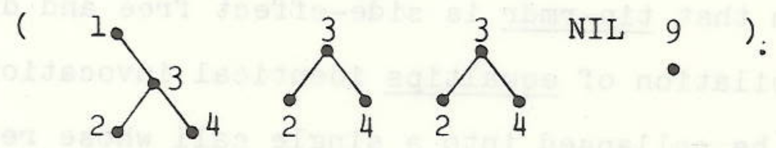
[4]The problem has an obvious solution which handles each target in the list separately but which may visit a node in tree once for each element of s:
```
(*find
   s
   *tree).
```

```
(batch s tree) ≡
      if (null s) then []
   elseif (null tree) then (*[] s)
                        else <append ([batch        *2        batch ]
                                       (lt-eq-gt (key tree) s)
                                       [(left tree) *tree (right tree)]])> .
```

For example, let tree be



then (batch [1 3 3 8 9] tree) yields the list



The final example illustrates the application of functional combination to a problem taken as typical of the use of coroutines [B&D75] : test if the atomic elements of two list structures are equal under left-to-right traversal with a single traversal of each list.

```
(equaltips x1 x2) ≡
     if (equal <1 (tip-rmdr x1)> <1 (tip-rmdr x2)>)
              then (equaltips <2 (tip-rmdr x1)> <2 (tip-rmdr x2)>)
              else false .
```

The auxiliary function, tip-rmdr is defined as follows

```
(tip-rmdr x) ≡
    if (null x) then [[][]]
 elseif (atom (car x)) then [(car x)(cdr x)]
 elseif <1 (tip-rmdr (car x))> then ([ 1    cons]
                                      (tip-rmdr (car x))
                                      [ #    (cdr x)])
                                else (tip-rmdr (cdr x)) .
```

The auxiliary function returns the next atom found and a structure of the remainder of the tree yet to be traversed. In a compiling environment the compilation of tip-rmdr should yield the information that tip-rmdr is side-effect free and during the subsequent compilation of equaltips identical invocations of tip-rmdr could be collapsed into a single call whose result is maintained locally. This function can easily be generalized (as a lexpr) to test an arbitrary number of lists for equality of atoms using the * notation:

```
(equaltips x) ≡
    if <equal <*1 (*tip-rmdr x)>> then <equaltips <*2 (*tip-rmdr x)>>
                                  else false.
```

Conclusion

The entire system described in this paper is implemented and running transparently in LISP 1.6 . A description of the interpreter appears as the appendix. In an earlier paper [F&W75] we used lists in place of bracketed expressions to represent combinators. The current notation better relates to its bracketed arguments (rows) and preserves INTERLISP's error trapping feature which is keyed to lists as functions.

Experience with this system indicates that this formulation of functional combination is both convenient and powerful. Even before it was implemented, some of this notation crept into our verbal communications with colleagues and students. The star notation is particularly convenient for expressing "MAPping" results which usually require trivial and conventionally constructed auxiliary functions. Coding quality has improved with the convenient notation for a function to build and use multiple results from a single recursion [Abr66].

Future work is directed at the construction of a translator for "stylized recursion," which will include functional combination, into stackless iterative code. The degree to which recursive calls from within a combinator will be translatable will depend on characteristics of the target machine. A rich multi-processing environment will guarantee maximum efficiency for the image code of a programmer who uses all the features of functional combination.

Appendix

Embellished LISP interpreter in the form of Chapter 1 of [McC66].

Several functions: bracketed, bracket, unbracket; starred, star, allstarred; angled, unangle, are assumed to be implementation dependent, based on the structure which the lexical scanner builds for [a b], *(a b), and <a b> respectively. It is assumed that (car *a) = a and (cdr *a) = *a.

The key words label and funarg (which is not available to the user except through FUNCTION), are handled differently when detected in eval versus apply. Argument evaluation proceeds in an environment determined by which stage of the interpreter uncovered the key word.

```
(apply fn x a) ≡
     if (atom fn) then
          if (numberp fn) then
               if (onep fn) then (car x)
               else               (apply (subl fn) (cdr x) a)
          elseif (eq fn NIL) then NIL
          elseif (eq fn #) then <1 x>
          elseif (eq fn CAR) then (car <1 x>)
          elseif (eq fn CDR) then (cdr <1 x>)
          elseif (eq fn CONS) then (cons <1 x> <2 x>)
          elseif (eq fn EQ) then (eq <1 x> <2 x>)
          elseif (eq fn ATOM) then (atom <1 x>)
          else               (apply (eval fn a) x a)
     elseif (starred fn) then
          if (allstarred x) then
               (star (apply (car fn) (omit# (*car x)) a))
          elseif (member NIL x) then NIL
          else               (cons (apply (car fn) (omit# (*car x)) a)
                                   (apply fn (*cdr x) a)   )
     elseif (bracketed fn) then
          if (eq (unbracket   fn) NIL) then NIL
          elseif (member NIL x) then NIL
          else    (cons (apply (car(unbracket fn)) (omit# (*car x)) a)
                        (apply (bracket(cdr(unbracket fn))) (*cdr x) a))
     elseif (angled fn) then (error)
     elseif (eq <1 fn> LAMBDA) then
          (eval <3 fn> (append (*cons <2 fn>  x) a))
     elseif (eq <1 fn> NLAMBDA) then
          (eval <3 fn> (append (*cons <2 fn> *x) a))
     elseif (eq <1 fn> FUNARG then (apply <2 fn> x <3 fn>)
     elseif (eq <1 fn> LABEL) then
          (apply <3 fn>  x  (cons (cons<2 fn><3 fn>) a))
     else      (error)
(eval e a) ≡
     if (atom e) then
          if (numberp e) then e
          elseif (eq e NIL) then e
          elseif (eq e #) then e
          else       (cdr (assoc e a))
     elseif (starred e) then (star (eval (car e) a))
     elseif (bracketed e) then (*eval (unbracket e) *a)
     elseif (angled e) then
          (apply <1(unangle e)> (omit# (eval <2(unangle e)> a)) a)
     elseif (atom <1 e>) then
          if (eq <1 e> QUOTE) then <2 e>
          elseif (eq <1 e> FUNCTION) then [FUNARG <2 e> a]
          elseif (eq <1 e> COND) then (evcon (cdr e)  a)
          elseif (numberp <1 e>) then (apply (car e) (*eval (cdr e) *a) a)
          elseif (member <1 e> [CAR CDR CONS EQ ATOM] then
                    (apply (car e) (*eval (cdr e) *a) a)
          else      (eval (cons (eval (car e) a) (cdr e)) a)
     elseif (or (starred<1 e>)(bracketed<1 e>)(angled<1 e>)
          (member <1<1 e>> [LAMBDA NLAMBDA]) ) THEN
                    (apply (car e) (*eval (cdr e) *a)  a)
     elseif (eq <1<1 e>> FUNARG) then (eval (cons <2<1 e>>(cdr e)) <3<1 e>>)
     elseif (eq <1<1 e>> LABEL) then
          (eval (cons <3<1 e>> (cdr e))  (cons (cons<2<1 e>><3<1 e>>) a) )
     else (error)
(omit# x) ≡
     if (eq x NIL) then x
     elseif (eq (car x) #) then (omit# (cdr x))
     else              (cons (car x) (omit# (cdr x)))
```

REFERENCES

Abr66 P. W. Abrahams. Discussion of P. J. Landin, The next 700 programming languages. Comm. ACM. 9, 3 (March, 1966), 163.

Bac73 J. Backus. Programming language semantics and closed applicative languages. IBM Research Report RJ1245, Yorktown Hts., NY (1973).

B&L74 W. Brainerd and L. Landweber. Theory of Computation, Wiley, New York (1974).

B&D75 R. M. Burstall and J. Darlington. Some transformations for developing recursive programs. ACM SIGPLAN Notices 10, 6 (June, 1975), International Conference on Reliable Software, 465-472.

D&B73 J. Darlington and R. M. Burstall. A system which automatically improves programs. Proc. 3rd International Conference on Artificial Intelligence, Stanford University (1973), 479-489.

Fri74 D. P. Friedman. The Little LISPer. Science Research Associates, Palo Alto, CA (1974).

F&W74 D. P. Friedman and D. S. Wise. Unwinding structured recursions into iterations. Technical Report 19, Computer Science Dept., Indiana University (December, 1974).

F&W75 D. P. Friedman and D. S. Wise. Multiple-valued recursive procedures. Technical Report 27, Computer Science Dept., Indiana University (April, 1975).

H&S75 C. E. Hewitt and B. Smith. Towards a programming apprentice. IEEE Trans. on Software Engineering SE-1, 1 (March, 1975), 26-45.

Knu68 D. E. Knuth. Fundamental Algorithms (2nd ed.), Addison-Wesley, Reading, MA (1973).

Knu73 D. E. Knuth. Sorting and Searching, Addison-Wesley, Reading, MA (1973).

Lan66 P. J. Landin. The next 700 programming languages. Comm. ACM. 9, 3 (March, 1966), 157-162.

McC62 J. McCarthy et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, MA (1962).

M&B67 S. MacLane and G. Birkhoff. Algebra, MacMillan, New York (1967).

Qua69 L. H. Quam. Stanford LISP 1.6 manual, Stanford Artificial Intelligence Project, Stanford University (1969).

Ris73 T. Risch. REMREC--a program for automatic recursion removal in LISP. Datalogilaboratoriet, Uppsala, Sweden (1973).

S&G75 B. Shneiderman and V. Goodman. Batched searching of sequential and tree structured files. Technical Report 32. Computer Science Dept., Indiana University (June, 1975).

Tei74 W. Teitelman. INTERLISP Reference Manual, Xerox PARC, Palo Alto, CA (1974).

Wei67 C. Weissman. LISP 1.5 Primer, Dickenson, Belmont, CA (1967).

vWi69 A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. Report on the algorithmic language ALGOL 68. Numerische Matematik 14 (1969), 79-218.

Wir71 N. Wirth. The programming language PASCAL. Acta Informatica 1 (1971), 35-63.

Acknowledgement

The insightful comments of Mitchell Wand during the development of this material, and the critical review of Stuart C. Shapiro in its writing are deeply appreciated.

Function type / Argument type	Application of extractor: 2	Application of expr: cons	Application of lexpr: append	Bracketed combinator: [2 cons]	Starred combinator: *append
Heterogeneous vector v = (A (B C D))	<2 v> (B C D)	<cons v> = (A B C D)	Usually a type conflict among arguments	([2 cons] v v) = (A ((B C D) B C D))	Often a type conflict among arguments
Homogeneous list ℓ = ((P Q)(R)(S))	<2 ℓ> = (R)	Possible type conflict or too few arguments <cons ℓ> = ((P Q) R)	<append ℓ> = (P Q R S)	Type conflict possible ([2 cons] ℓ ℓ) = ((P Q) ((R) R))	(*append ℓ ℓ) = ((P Q P Q)(R R)(S S))
Starred value s = *(A B C)	<2 s> = (A B C)	<cons s> = ((A B C) A B C)	Diverges unless s is the starred identity	([2 cons] s s) = ((A B C) ((A B C) A B C))	(*append s s s) = *(A B C A B C A B C)

Table 1. Examples of various invocation schemes on pseudo-typed functions.