This work was presented to the Graduate School of Yale University in canidacy for the degree of Doctor of Philosophy. The author is presently a member of the Courant Institute at New York University.

Carrier Arrays: An Extension to APL

Research Report #256

Paul Geoffrey Lowney

May 1983

This research was supported by the Burroughs Corporation under grant 16994. Publication costs were paid by the National Science Foundation under grant MCS-8106181.

ABSTRACT

Carrier Arrays: An Extension to APL

Paul Geoffrey Lowney

Yale University, 1983

The idiomatic APL programming style is limited by the constraints of the rectangular array as a data structure. Non-scalar data is difficult to represent and manipulate, and the non-scalar APL functions have no uniform extension to higher rank arrays. The carrier array is an extension to APL that addresses these limitations while preserving the economical APL style. A carrier array is a ragged array with an associated partition that allows functions to be applied to subarrays in parallel. The primitive functions are given base definitions on scalars, vectors, and matrices, and they are uniformly extended to higher rank arrays by the function application mechanism. The associated partition also allows the last dimensions of an array to be treated as a single item; the primitive functions are given extended definitions on arrays of these non-scalar items.

This thesis defines the carrier array and gives the accompanying changes to the definitions of the APL primitive functions. Examples of programming with carrier arrays are presented, and the carrier array is compared with other proposed extensions to APL. Methods for implementing the carrier array extension are discussed, and a prototype interpreter presented.

Acknowledgements

Working with Alan Perlis, my advisor, has been an extraordinary experience. He has been a continual source of new ideas and fresh outlooks on programming languages and computer science. He originally suggested the notion of a "carrier" array to me, and then gave me the freedom to develop it at my own pace, and in my own manner. I thank him for his guidance, and my freedom.

I thank my readers, Drew McDermott and Jim Brown, for a prompt evaluation of this work.

The research for this thesis was done on a DECsystem20 using the *Tools* programming environment. The 20 is part of an excellent computing facility managed by John O'Donnell. *Tools* is a very productive working environment developed at Yale for the 20; it was extended to support APL program development by Steve Wood. The thesis was written with the Z editor, formatted with Scribe, and produced on an Imagen laser printer. John Ellis developed the APL font used in the text.

Many thanks to my long-suffering roommate, Mark Burstein, and office mate, Steve Wood, who have been my companions throughout graduate school; I will miss them both. To Drs. Cahow, Cane, and Spiro, along with the wonderful staff of the Yale health plan and hospital, who have kept me going the past five years. To Tom Sahagian, who gave me an escape from graduate school when I needed one. To my grandmother, Mrs. Edward C. Keating, who supplemented my meager fellowship with one of her own. To my parents, who have supported my education for thirty years. And most of all, to Maggie Wolf, who stood by my side through some very trying times. I dedicate this work to her.

This research was supported by the Burroughs Corporation under grant 16994. Publication costs of the thesis were paid by the National Science Foundation under grant MCS-8106181.

Table of Contents

List of Figures

Chapter 1

INTRODUCTION

Carrier arrays are an extension to APL designed to increase the power of the APL programming style. Carrier arrays extend APL in two directions. They enable more functions to be applied in parallel to the elements of an array. And they introduce non-scalar data into the language, permitting a function to be applied to an array of non-scalar data items.

A *carrier array* is a ragged array with an associated partition. When a function is applied to a ragged array, the array is partitioned into a three level carrier array: a collection of subarrays of non-scalar items. The function is then applied independently to each subarray. The results are assembled in a new carrier array, and the partitioning is removed.

The carrier array extension requires two major changes in current APL. The basic APL data structure is changed from a rectangular, homogeneous array to a ragged, heterogeneous array. And the function application mechanism is standardized for all primitive and defined functions, and modified to include carrier partitioning.

The carrier array and its effect on the APL language are presented in detail in the thesis. To begin, we examine APL programming and the APL programming style.

1.1 APL programming style

A strength of APL is that functions can be applied in parallel to the elements of an array. This is most true of the scalar functions. The scalar functions in APL are defined for pairs of scalars and extended to higher rank arrays componentwise. Plus is a scalar function, and for two equally shaped arrays *A* and *B*, (*A*+*B*)[*I*] is equivalent to *A*[*I*]+*B*[*I*], for all valid indices *I*.

To perform *A*+*B*, we pair elements in corresponding locations in *A* and *B*, and apply plus to each pair. All the additions are performed independently. Thus we could pair the elements from *A* and *B* in a variety of ways. APL provides a means of doing this by supplying higher order functions which are called operators. The APL operators are reduction, scan, outer product, inner product, and axis.

In a strict sense, operators are functions which take a primitive function as an argument and produce a derived function as a result [21, 28, 29]. However, it is also fruitful to look at operators as a means of applying a function to the elements of an array. To evaluate the outer product *A*∘.+*B*, we form a full cartesian product of the elements in the two arrays, and apply plus to each pair. The operator, outer product, interacts with the array structure of *A* and *B* to determine how plus is applied to the elements in the two arrays.

The interaction of operator and array structure produces a data driven flow of function application. In the plus reduction +/*A*, there is no need for explicit control structure; the reduction operator and the array structure of *A* completely determine the pattern of function application. Functions, operators, and data structures combine to give APL a concise, powerful programming style. We illustrate this style in two examples below.

Consider the problem of finding the depth of parenthesis nesting of each element in an arithmetic expression. More concretely, if *V* is a character vector representing an arithmetic expression, we want to determine how many pairs of parenthesis enclose each element of *V*. The answer is a vector of length equal to *V*; each element gives the depth of nesting of the corresponding element in *V*.

An APL solution is `+\-/V∘.='()'`. Figure 1-1 presents a sample evaluation. We first apply equal with outer product (`V∘.='()'`) to identify the parenthesis in *V*; the result is a two column boolean matrix. We then apply minus with reduction (−/) to take the difference between the two columns, and apply plus with scan (+\) to the resulting vector, obtaining a running count of the level of nesting in *V*.

```
V                  ←→         ((÷B)×C)

V∘.='()'           ←→         1 0
                              1 0
                              0 0
                              0 0
                              0 1
                              0 0
                              0 0
                              0 1

-/V∘.='()'         ←→         1 1 0 0 ¯1 0 0 ¯1

+\-/V∘.='()'       ←→         1 2 2 2 1 1 1 0
```

Figure 1-1: Parenthesis nesting.

Note the pattern of function application in the program. We build a large array with outer product, and repeatedly apply scalar functions to it with the reduction and scan.

There is no explicit control structure; operators guide the application of functions.

As a second example, consider an expression to calculate the third-order determinant of a matrix. An APL solution is −/+/×/[2](0 1 2,[.5]0 2 1)⌽*M*,[.5]*M*. Figure 1-2 presents a sample evaluation.

To calculate the determinant, we first combine two copies of the matrix to form a three dimensional array (*M*,[.5]*M*). We then construct a matrix of rotators (0 1 2,[.5]0 2 1) and rotate (⌽) the rows of the two matrices. The rotation aligns the columns of the upper and lower matrices to contain the multiplicands that form the positive and negative terms in the summation of the determinant. We then reduce each column with multiplication (×/[2]), producing a vector of positive terms and a vector of negative terms. We sum both vectors (+/) and take the difference of the two sums (−/) to obtain the determinant.

Again note the pattern of function application. We build a big array, and then repeatedly apply functions to subarrays. We rotate planes, reduce columns, sum rows, and reduce a vector. The array structure is a frame, a "carrier", for the application of functions to the data in the array. There is no explicit control structure; the control of function application is driven by operators and the organization of data.

The interaction of array structure and operators gives APL its distinctive programming style. The first lesson to be learned from APL programming is that an appropriate pairing of data structures and operators permit simple, concise solutions to a variety of problems.

In examining APL programs, we notice that certain combinations of primitives occur frequently. For example, a lamination and rotation followed by a series of reductions

```
M                          ←→      2  0  3
                                  10  1 17
                                   7 12 ¯4

T0 ← M,[.5]M               ←→      2  0  3    2  0  3
                                  10  1 17   10  1 17
                                   7 12 ¯4    7 12 ¯4

T1 ← 0 1 2,[.5]0 2 1       ←→     0 1 2
                                  0 2 1

T1⌽T0                      ←→      2  0  3    2  0  3
                                   1 17 10   17 10  1
                                  ¯4  7 12   12 ¯4  7

×/[2]T1⌽T0                 ←→     ¯8  0 360
                                 408  0  21

+/×/[2]T1⌽T0               ←→    352 429

-/+/×/[2]T1⌽T0             ←→    ¯77
```

Figure 1-2: Determinant of a matrix.

calculates the determinant of a matrix, but Perlis and Rugaber have noted that the same combination of functions and operators also occurs in programs to determine whether a pair of line segments intersect, to determine whether a polygon is convex, and to determine if a point is inside a convex polygon [63, 64]. This combination of primitives is an example of an idiom. An *idiom* is a combination of primitives that occurs repeatedly in varying contexts. Idioms are the building blocks of APL programs and an important component of the APL style.

1.2 The Constraints of Rectangularity

As the examples above illustrate, simple APL solutions can be written for problems that fit the APL data structure and primitive functions. However, many problems can not be solved simply in the APL style. Their one-line solutions are complicated, unintuitive combinations of primitives. These "one-liners" have given APL its reputation for obscure, hard to read programs.

These problems can also be solved by explicitly iterating over over the elements of an array. However, compared with FORTRAN or PASCAL, APL does not have a rich set of control structure primitives, and there is a consensus in the APL community that they should not be added to the language [74]. Rather the operators, functions, and data structures of the language should be enhanced so that more problems can be solved simply in the APL data-driven style.

It is often the constraints of the rectangular array as a data structure which limit the applicability of the APL programming style. In particular, it is difficult to apply non-scalar functions and to represent non-scalar data with rectangular arrays.

1.2.1 Applying non-scalar functions.

The non-scalar, or *mixed*, functions are array manipulating functions; they test, rearrange, and select the elements of an array. They include membership (∈), rotate (⌽), take (↑), and drop (↓). The mixed functions cannot be defined independently for pairs of scalars and extended to higher rank arrays componentwise, for their definitions depend on the array structure of their arguments.

A serious restriction to APL programming is that the non-scalar functions do not extend uniformly to higher rank arrays. We demonstrate this with an example. Let V be a vector and M be a matrix with two rows. 3↑V takes the first three elements from the vector V. By analogy to the scalar functions, we might assume 2 3↑M takes two elements from the first row of M, and three elements from the second. Of course, this results in a ragged matrix, and thus cannot be a valid extension of the function. Rather 2 3↑M takes a two by three matrix from the upper left-hand corner of M.

We have encountered a constraint of rectangularity: The application of any APL function to an array must result in a rectangular array. An array cannot be decomposed into a collection of subarrays, a non-scalar function applied independently to each subarray, and the results assembled in a new array, for the resulting array may not be rectangular. There is no uniform algorithm for extending non-scalar functions to higher rank arrays, as there is for the scalar APL functions. Rather, each non-scalar function is extended to higher rank arrays in an idiosyncratic manner.

The idiosyncratic extension of the non-scalar functions has two unfortunate consequences. The first is that the non-scalar functions cannot be applied with the operators, for there is no uniform way of applying non-scalar functions to selected elements of an array. The second, and more important, is that the non-scalar functions often halt the parallel flow of function application in an expression. If an expression contains two non-scalar functions, each of them extends to higher rank arrays in a different, idiosyncratic manner and their composition probably does not extend to higher rank arrays at all, or at least not in the manner expected by the programmer.

1.2.2 Representing Ragged Structures

A more obvious constraint of rectangularity is that ragged structures and collections of non-scalar data cannot be represented easily. Non-scalar objects can be simulated by embedding data in a rectangular array. For example, we can represent a word list as a rectangular matrix, one word per row, padding out with blanks to a common length. But the primitive functions lose their expressive power when applied to these embedded objects. The functions cannot manipulate the matrix as a vector of words; rather the functions are applied to a matrix of characters. An APL programmer must always be conscious of how his data is encoded in the array structure.[1]

1.3 Nested Arrays

The APL community has been aware of the constraints of rectangular arrays for a long time. As a solution, they have proposed the nested array as a new APL data structure [14, 24, 47, 26, 10, 68, 69, 17, 15, 65].[2] A nested array is a recursive data structure where the elements of an array may be arrays themselves. Non-scalar data can be represented trivially with nested arrays, and new operators are defined to allow the parallel application of all primitive and defined functions.

Nested arrays in APL encourage their own programming style. Parallel function application is no longer as natural; with a nested data structure the programmer moves toward the LISP paradigm of recursive movement down and up the tree. In sample

[1]Robert Dewar notes that in this regard programming in APL is very similar to programming in assembly language [20].

[2]Gull and Jenkins [26] present an extensive bibliography on nested arrays.

programs [42, 26, 31], the level of nesting of data replaces the dimensions of the array as a means of structuring data. Nested arrays introduce new idioms into APL, and current idioms do not generalize naturally to a recursive data structure.

1.4 Carrier Arrays

The carrier array is an extension to APL developed with the APL programming style in mind. A carrier array is a ragged array with an associated partition; the partition permits an array to be viewed as a collection of non-scalar items. This addresses the two weaknesses of APL we outlined above. Non-scalar data can be naturally represented; the last dimensions of an array can be implicitly enclosed and treated as a single datum. And non-scalar functions can be applied in parallel; an array can be partitioned into a collection of subarrays, and the function applied independently to each subarray.

In extending an idiom-rich language such as APL, an important goal is conserving its idioms. We call this the principle of ***idiomatic similitude.*** The development of the carrier array has been driven by an attempt to generalize existing APL idioms. For example, $((V\iota V)=\iota\rho V)/V$ is an idiom to remove duplicates from a vector V. How can APL be extended to allow $((V\iota V)=\iota\rho V)/V$ to remove duplicates from many vectors in parallel? And what extensions would enable the idiom to remove duplicate words from a word list? The carrier array is an abstraction which permits both generalizations. An examination of many APL programs has shown carrier arrays preserve APL idioms and the economical APL style while greatly increasing the power of the language.

1.5 Outline

In the next chapter, we illustrate the limitations of current APL by attempting to generalize an idiom. We show how nested arrays address these limitations, and then use the idiom to motivate the carrier array and the accompanying changes to APL. Chapter 3 describes carrier APL in detail, giving definitions of the data structures, the functions, and the application forms. In chapter 4 we present several annotated carrier APL programs, and in chapter 5 we discuss the issues in implementing carrier APL. We close with a brief conclusion in chapter 6.

Chapter 2

REMOVE DUPLICATES: AN EXPOSITORY EXAMPLE

As an informal introduction to carrier arrays and the problems they address, we consider the remove duplicates idiom $((V\iota V)=\iota\rho V)/V$ in more detail. As mentioned in Chapter 1, we would like to extend the idiom in two directions: to operate on non-scalar data and to remove duplicates from many vectors in parallel. In this chapter, we use the problem of extending the idiom to evaluate extensions to APL. We first examine the limitations of current APL which make these extensions difficult. We then see how the problem is addressed with nested arrays, and analyze the effect of the nested extension on the APL programming style. We conclude by introducing the carrier array, and presenting the carrier APL solution.

2.1 Current APL

((V⍳V)=⍳⍴V)/V removes duplicates from a vector. Figure 2-1 presents a sample evaluation. *V⍳V* maps each element of *V* to the index of its first occurrence in *V*. *⍳⍴V* gives the index of each element of *V*. Comparing these two values, we can identify the first occurrence of each element of *V* and select them all with compression (/).

V	←→	*ABACBF*
(V⍳V)	←→	1 2 1 4 2 6
⍳⍴V	←→	1 2 3 4 5 6
(V⍳V)=⍳⍴V	←→	1 1 0 1 0 1
((V⍳V)=⍳⍴V)/V	←→	*ABCF*

Figure 2-1: Sample evaluation of remove duplicates idiom.

2.1.1 Non-scalar Data

The remove duplicates idiom does not extend to a vector of non-scalar data. Consider removing duplicate entries from a list of words. We first represent the word list as a rectangular APL array; this is most easily done by padding the shorter words with blanks, and storing each word as a row of a matrix. (Note this is not an adequate solution for storing a list of arbitrary text strings, for the padding character must not occur in the string.) We now adapt the functions in our idiom to this new data structure. We can use the inner product operator to derive the ∧.= function, which tests for equality between vectors. However we cannot derive a variant of dyadic iota (⍳) that uses ∧.= as its test for equality. Thus we must discard our current idiom and develop a new combination of primitives to identify the first occurrence of each row of the matrix.

A new solution is given in figure 2-2. Note it has the same design as the original idiom. 1 1⍉<*M*∧.=⍉*M* creates a mask which identifies the first occurrence of each row in *M*, and we use compression (/) to select the first occurrence of each row. But there is no simple transformation from the first idiom to the new expression. To accommodate a simple change to the data structure, the idiom must be completely rewritten.

```
R ← (1 1 ⍉<\M∧.=⍉M)/[1]M

M    ←→   APL                    R    ←→   APL
          BASIC                            BASIC
          APL                              COBOL
          COBOL                            FORTRAN
          BASIC
          FORTRAN

ρM   ←→   6 7                    ρR   ←→   4 7
```

Figure 2-2: Remove duplicates from a word list.

2.1.2 Parallel Application

It is more difficult to rewrite the idiom to remove duplicates from many vectors in parallel. We represent the vectors as rows of a matrix, padding the shorter vectors appropriately. To apply the idiom to the matrix, the primitives functions in the idiom must extend "row-wise", operating on each row of the matrix independently. However, of the primitives functions used in the idiom, only the scalar function equal (=) extends to a matrix row by row, testing equality between the elements of corresponding rows. The other functions extend to a matrix in various ways. Compression (/) and dyadic iota (⍳) apply a vector left argument to each row of a matrix right argument. A matrix left argument is not permitted. Shape (ρ) is not a decomposable function; it gives the shape of the matrix, not of the rows which compose it. Monadic iota (⍳) is only defined for scalars; it does not extend to higher rank arrays.

To extend our idiom to apply to the rows of a matrix, we must rewrite it with a new collection of primitives. The new solution is given in figure 2-3. Again, the algorithm is the same. The first line creates a mask which identifies for every row the first occurrence of each element in the row. The next three lines use this mask to select the corresponding elements from *M*. But even to an experienced APL programmer, it is not obvious we are removing duplicates from the rows of a matrix.

Note this program can be extended to non-scalar data by using the ∧.= inner product in line one, and restructuring the ravel of *M* on line four.

```
M0 ← 1 2 2⍉<\1 2 1 3⍉M∘.=M
L  ← +/M0
M1 ← L∘.>⍳⌈/L
R  ← (⍴M1)⍴(,M1)\(,M0)/,M

M    ←→   ABACBF           R    ←→   ABCF
          FFAC                       FAC
          ABBAC                      ABC

⍴M   ←→   3 6              ⍴R   ←→   3 4
```

Figure 2-3: Remove duplicates from multiple vectors.

2.1.3 Analysis

The two extensions of the remove-duplicates idiom demonstrate the limitations of the language mentioned in Chapter 1. It is difficult to represent non-scalar data. Ragged structures, such as a word list, can be embedded in rectangular arrays, but the primitive functions then lose their expressive power. And non-scalar functions cannot be applied in parallel. Each non-scalar function extends to higher rank arrays in a idiosyncratic manner, and expressions containing non-scalar functions cannot be applied independently to subarrays collected in a higher rank array.

These problems are largely a consequence of the limitations of a rectangular array as a data structure. Non-scalar data cannot be directly represented in a rectangular array, and thus the primitive functions cannot manipulate it easily. Similarly, many non-scalar functions applied independently to the rows of a matrix would produce non-rectangular results.

A historical note is of interest here. In his first published definition of APL [30], Iverson allowed a boolean matrix to compress another conformably shaped matrix; each row of the boolean matrix compressed a corresponding row in the right argument. The compressed rows could have varying length, for there was no restriction on the boolean matrix; they could not be stored as rows of a rectangular array. Therefore the result rows were catenated into one vector. This catenation, however, stops subsequent parallel function application, for the row structure which guides the application is lost. This extended compression was eliminated in later definitions of the language; it is essentially incompatible with rectangular arrays.

2.2 Nested Arrays

Nested arrays are part of a larger set of extensions to APL which have been proposed over the past decade. Many variations of the nested arrays system have been proposed, and three different systems have been implemented [10, 17, 65].[3] The APL community has not yet reached a consensus as to the best proposal [4]. We present a modest nested array extension below, with features common to all three systems,[4] and briefly summarize the differences between the nested systems. We then return to the examination of the remove duplicates idiom. We present extensions to the idiom using nested arrays and evaluate how the weaknesses of APL we have observed are addressed, and how the APL

[3]Gull and Jenkins [26] summarize the various proposals. Orth [59] compares the STSC [17] and I.P.Sharp [10] implementations.

[4]The notation used is from the STSC NARS system [17].

programming style is affected.

2.2.1 Overview

A nested array is a recursive data structure in which any element in an array can be an array itself. The nesting can continue to an arbitrary depth. An element in an array is called an *item*. A *simple* array contains no nested items; simple homogeneous arrays are the arrays of current APL.

Nested arrays require new functions to create the nested structure and a mapping operator to apply functions to the items of an array. *Enclose* (⊂) and *disclose* (⊃) are functions to create and remove layers of nesting. Enclose converts an array into a scalar, and disclose is the inverse operation which recovers the original array, i.e. $⊃⊂A \longleftrightarrow A$ for all arrays A. The new mapping operator is called called *each* (¨). If a function G is applied with each to an array A, each element of A is in turn selected and disclosed, the function G applied to the disclosed element, and the result enclosed and stored in a new array. No order for the selection of elements of A is specified. More formally, if $R{\leftarrow}G¨A$, then $R[I] \longleftrightarrow ⊂G⊃A[I]$ for all valid indices I.

Some existing APL functions are extended to nested arrays. Functions that manipulate array structure (e.g., catenate (,), reshape (ρ), transpose (⍉)) are extended in a straightforward manner. An equivalence function is introduced to test for equality between nested arrays, and it is used to extend dyadic iota (⍳) and membership (∊) to arrays of nested items. No consensus has been reached on the appropriate extension of the remaining primitive functions, and we do not discuss them here.

The major difference between the proposals for nested arrays is in their definition of a nested array. The differences are manifested in the application of enclose to simple scalars, that is, the scalars of current APL. The proposals fall into two groups. In "Axiom System 0"[5] enclose and disclose are *permissive*, that is, the enclose of a scalar is equivalent to the scalar, and the disclose of a scalar is equivalent to the scalar. In "Axiom System 1" enclose is *strict* but disclose is permissive, that is, the enclose of a scalar is a nested item not equivalent to the original scalar, but the disclose of a scalar is equivalent to the original scalar. The application of enclose to arrays other than simple scalars is identical in both systems.

The apparently minor distinction between the two definitions has ramifications throughout the nested array extension. For example, a permissive enclose introduces heterogeneous arrays into APL if we allow the catenation of enclosed arrays [26]. Because of the "telescoping scalar" effect, (⊂1),⊂'*A*' is equivalent to 1,'*A*' with permissive enclose; with strict enclose, the values of the two expressions are distinct. Moreover, the mapping operator each is defined in terms of enclose and disclose, and the distinction between the definitions affects derived functions produced with the operator. If *ID* is an identity function such as (*ρA*)*ρ*,*A*, then the derived function *ID*¨ is also an identity function with permissive enclose, but it is not an identity for a system using strict enclose, for with strict enclose *ID*¨ maps simple items to nested items.

From the point of view of carrier arrays, the distinctions between the various proposals are not important. Both introduce a recursive data structure into APL, along with functions to manipulate the structure. Both have an equally profound effect on the APL programming style.

A common feature of the nested array proposals is that they are supersets of current

[5]The classification is due to Gull and Jenkins [26].

APL. No current APL programs are effected by the changes. Thus the remove duplicate idiom is unchanged; *((V⍳V)=⍳ρV)/V* removes duplicates from a simple vector in the nested array system.

2.2.2 Non-scalar Data

To remove duplicates from a word list with nested arrays, we first create a vector *V* of enclosed character vectors. Each element of *V* represents a word in the word list. The definition of dyadic iota is extended to enclosed arrays with a recursive equivalence function: Two nested items are equivalent if their discloses have equal shape and equivalent items in corresponding locations; simple items are compared with scalar equality. The other primitives in the idiom are indifferent to the type of the elements of *V*. Thus using nested arrays, the idiom extends to non-scalar data with no changes. Figure 2-4 presents a sample evaluation.

V	←→	*(APL) (BASIC) (APL) (COBOL) (BASIC) (FORTRAN)*
(V⍳V)	←→	1 2 1 4 2 6
⍳ρV	←→	1 2 3 4 5 6
(V⍳V)=⍳ρV	←→	1 1 0 1 0 1
((V⍳V)=⍳ρV)/V	←→	*(APL) (BASIC) (COBOL) (FORTRAN)*

Figure 2-4: Removing duplicates from a word list with nested arrays.

2.2.3 Parallel Application

To remove duplicates from several vectors in parallel, we enclose each vector and assemble them in a new vector *V*. We then apply every function in the idiom to the items of *V* with the mapping operator each (¨). Figure 2-5 presents the solution and a sample evaluation.

Note this solution will also remove duplicates from multiple word lists in parallel. *V* must be constructed with two levels of nesting: each item is a vector of enclosed

V	←→	(*ABACBF*) (*FFAC*) (*ABBAC*)
(*V*⍳¨*V*)	←→	(1 2 1 4 2 6) (1 1 3 4) (1 2 2 1 5)
⍴¨*V*	←→	(6) (4) (5)
⍳¨⍴¨*V*	←→	(1 2 3 4 5 6) (1 2 3 4) (1 2 3 4 5)
(*V*⍳¨*V*)=¨⍳¨⍴¨*V*	←→	(1 1 0 1 0 1) (1 0 1 1) (1 1 0 0 1)
((*V*⍳¨*V*)=¨⍳¨⍴¨*V*)/¨*V*	←→	(*ABCF*) (*FAC*) (*ABC*)

Figure 2-5: Remove duplicates from multiple vectors with nested arrays.

character vectors. The mapping operator each applies each function to the items of *V* independently, removing duplicates as before.

2.2.4 Analysis

Nested arrays address the limitations of APL outlined in Chapter 1. Ragged structures are represented as arrays of nested items, and primitive functions are defined to manipulate nested arrays. The mapping operator each provides a mechanism for parallel function application.

In the extensions above, the nested array serves two purposes. In the extension to non-scalar data, it is a data structure for representing a collection of character strings. In the extension to parallel application, it is a frame for the application of functions with a mapping operator. Note it is not used to create a recursive structure; the deepest nesting is the three level structure used to represent multiple word lists in the combination of the two extensions.

A three level structure is a natural one for parallel function application. The lowest level holds the data, the words. The middle level contains the structures being manipulated, the word lists. The top level provides the structure to permit parallel application; it is an array of word lists. A more deeply nested structure should rarely be needed to apply functions in parallel.

Moreover, a recursive nested structure may draw unnecessary attention to levels of nesting which are not logically required by the problem. Consider removing duplicates from several collections of vectors in parallel. We represent the collections with an array of doubly nested items; each item is a vector of enclosed vectors. To apply the remove duplicates idiom to this representation, each function in the idiom must be applied with the mapping operator twice. The first each operator selects each item of the array in turn; the second maps the function along the vectors in the item as before. Figure 2-6 presents the solution.

$$R \leftarrow ((V\,\iota^{\text{¨¨}}\,V) =^{\text{¨¨}} \iota^{\text{¨¨}}\,\rho^{\text{¨¨}}\,V)/^{\text{¨¨}}\,V$$

$$V \longleftrightarrow ((ABACBF)\,(FFAC)\,(ABBAC))\ \ ((DDACFF)\,(APLA))$$

$$R \longleftrightarrow ((ABCF)\,(FAC)\,(ABC))\ \ ((DACF)\,(APL))$$

Figure 2-6: Remove duplicates from multiple collections of vectors with nested arrays.

In this example, there are only two layers of nesting, but more complicated examples require more. Whenever we have a ragged structure, we introduce a layer of nesting to represent it. Whenever we need to apply a non-scalar function to a set of arguments, we introduce a layer of nesting as a frame for the each operator. These two usages can interact to produce arbitrarily deep nesting structures. To apply a function to the leaves of the structure requires a corresponding number of mapping operators.

Notice in our example that when the each operator is used it is applied to every function in the idiom. A question arises as to why the operator cannot be incorporated into the function application mechanism, as it is for the scalar functions in current APL. With a nested structure it clearly cannot, for the each operator serves two purposes. It applies functions to frames of arguments, but it also picks through nested layers to determine the level of function application. With a nested structure, a mapping operator is required. But with a more conservative extension to the APL data structure, perhaps the each operator could be implicitly incorporated into the application of the primitives.

2.3 Carrier Arrays

The carrier array is an extension to APL which permits non-scalar functions to be applied in parallel to selected elements of an array without the introduction of a mapping operator. A carrier array is a ragged array with an associated partition; the partition permits the parallel application of functions and the representation of non-scalar data. A preliminary definition is given in [39]. In this section we present the carrier concept, which allows functions to be applied in parallel, and the notion of datum rank, which allows functions to be applied to arrays of non-scalar data. To illustrate these ideas, we examine how the remove duplicates idiom can be extended with carrier arrays. In chapter 3 we present the carrier array and the accompanying changes to the APL language in more detail.

2.3.1 The Carrier Concept

Carrier arrays are modeled on the relationship between scalar functions and arrays of scalars in current APL. The APL scalar functions are defined on scalars and extended to higher rank arrays componentwise. This extension is natural, for the elements of an APL array coincide with the domain and range of the scalar functions. In carrier APL we extend this identity to non-scalar functions by viewing an array as a "carrier" of arguments to a function, an array of subarrays.

In current APL, a number of non-scalar functions are defined for arrays of fixed rank, and are extended uniformly to higher rank arrays. Iverson groups these functions with the scalar functions and labels them *uniform* functions [27, 28, 29, 10]. For example, reversal (⌽) is defined to reverse vectors, and is extended to reverse the rows of a higher rank array. Consider the application of reversal to a rank 3 array A, i.e., ⌽A. In Iverson's terminology, the first two dimensions of A are a two-dimensional *frame* holding one-dimensional *cells*. When the function is applied, the cells are independently reversed and assembled in the corresponding locations of a new frame of the same shape. The rank and shape of the argument(s) to a uniform function determine the rank and shape of the result. Thus we know the results of the application of a uniform function to a collection of cells can be assembled in a rectangular array.

However, we cannot apply non-uniform functions such as compression in a similar manner. Compression has a natural definition for pairs of vectors; a boolean left vector identifies elements to select from an equal length right vector. Consider extending compression to two equally shaped matrices. Each row of the boolean left argument is paired with the corresponding row of the right argument, and compression is applied to each pair. The results of each compression may be of varying length; they cannot be stored as the rows of a rectangular matrix. Iverson originally proposed that they be catenated into a single vector [30], but this destroys the rank structure which guides the function application. Rather, the left argument of compression is restricted to a vector. It is paired with each row in the right argument when compression is applied and the results can be assembled in a rectangular array.

In carrier APL, we permit the application of compression to two arrays of the same shape. The basic APL data structure is changed to be a ragged array. To evaluate the compression of two matrices, corresponding rows are paired and compression applied; the results are assembled as the rows of a ragged matrix. Figure 2-7 presents a sample evaluation.

$R \leftarrow B/M$

$$B \longleftrightarrow \begin{array}{cccc} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{array} \qquad M \longleftrightarrow \begin{array}{l} ABCD \\ EFGH \\ IJKL \end{array} \qquad R \longleftrightarrow \begin{array}{l} ACD \\ EF \\ I \end{array}$$

Figure 2-7: Compression applied to two matrices.

By changing the data structure to a ragged array, we can apply in parallel all functions that are ***rank uniform***. A function is rank uniform if the rank of the result is determined by the rank of the argument(s); note there are no constraints on the shapes of the argument(s) and result. By uniformly mapping rank, we ensure that if the arguments of a function can be represented as subarrays of a ragged array, then the results can also be assembled as subarrays in a new ragged array. If both rank and shape were permitted to vary, the results of the application could only be assembled in a tree-like data structure such as a nested array.

We redefine the APL primitives as a mapping from a set of arrays with a characteristic rank (two sets for dyadic functions) to another set of fixed rank arrays. This mapping is called the ***base definition*** of the function. For example scalar plus (+) is defined to map two scalars to their sum, another scalar. Grade up (⍋), the sorting function, maps a vector of data to a vector of indices. Catenate (,) joins two vectors into a longer vector. Laminate (;̈) joins two scalars into a vector.[6] With a few exceptions, all the primitive functions can be defined as mappings of scalars, vectors, or matrices to scalars, vectors, or matrices.

The definition of each function can be characterized by a rank vector. For a dyadic function, the vector has three components: the rank of the left argument, the rank of the

[6]Laminate has a distinct token (;̈) in carrier APL.

right argument, and the rank of the result. For monadic functions only two components are necessary. We call this vector the *base rank* of the function. Figure 2-8 presents the base ranks of several primitive functions.[7]

	arg		result
compress (/)	1	1	1
plus (+)	0	0	0
grade up (⍋)		1	1
catenate (,)	1	1	1
laminate (,̈)	0	0	1

Figure 2-8: Base ranks of some primitive functions.

When a monadic function is applied to an array, the array is partitioned into a frame, or *carrier*, of *base arguments*, and the function is applied independently to each base argument. For a dyadic function, both arrays are partitioned, base arguments in corresponding locations are paired, and the function applied to each pair. In both cases the *base results* are assembled in a new carrier array, and the partitioning is removed from all carriers. The programmer need not indicate a partition; it is determined by the base rank of the primitive functions. The each operator of the nested array system is effectively incorporated into the function application mechanism.

The carrier array extension does change the meaning of some APL programs which use higher rank arrays. However, on the domain of the base definitions, the primitives of carrier APL and current APL are essentially equivalent. Thus, most programs which operate on vectors are unchanged, and in particular $((V\iota V)=\iota\rho V)/V$ removes duplicates from a vector.

[7]Iverson has similarly characterized the functions of current APL with rank vectors [27]. Note he assigns a different order to the components.

2.3.2 Parallel Application

With carrier arrays, the remove duplicates idiom extends to a collection of vectors with no changes. We represent the vectors as rows of a ragged matrix; no padding is necessary. Figure 2-9 presents the base ranks of the functions in the idiom.

	arg		result
dyadic iota (ι)	1	1	1
shape (ρ)		1	0
monadic iota (ι)		0	1
equal (=)	0	0	0
compress (/)	1	1	1

Figure 2-9: Base ranks of functions in remove duplicates idiom.

The functions are equivalent to their counterparts in current APL on the domain of the base definitions. However, there are differences when the functions are applied to higher rank arrays. Dyadic iota (ι) is defined for vectors. When applied to a pair of rank N arrays, both arrays are partitioned into collections of vectors, corresponding vectors are paired, and dyadic iota applied to each pair. It produces vectors, which are assembled in a rank N result. Shape (ρ) is defined to give the length of a vector. Applied to a rank N array, it will return a rank $N-1$ array of lengths. Monadic iota (ι) maps scalars to vectors. Applied to a rank N array, it produces an array of rank $N+1$. Equal (=) is a scalar function. It compares corresponding elements in its array arguments. And compression (/) is defined for pairs of vectors, giving a vector result. Figure 2-10 presents a sample evaluation of the idiom.

The idiom also extends to higher rank collections of vectors. A hierarchy of classifications is represented by adding dimensions to the ragged array. The extension to higher rank arrays is completely data driven; no changes in the program text are required.

V	$\longleftrightarrow$	*ABACBF* *FFAC* *ABBAC*

$(V\iota V)$	$\longleftrightarrow$	1 2 1 4 2 6
		1 1 3 4
		1 2 2 1 5

ρV	$\longleftrightarrow$	6 4 5

$\iota\rho V$	$\longleftrightarrow$	1 2 3 4 5 6
		1 2 3 4
		1 2 3 4 5

$(V\iota V)=\iota\rho V$	$\longleftrightarrow$	1 1 0 1 0 1
		1 0 1 1
		1 1 0 0 1

$((V\iota V)=\iota\rho V)/V$	$\longleftrightarrow$	*ABCF* *FAC* *ABC*

Figure 2-10: Removing duplicates from multiple vectors with carrier arrays.

2.3.3 Datum Rank

How can we represent non-scalar data without introducing a new data type such as a string, a tree, or a nested array? Ragged arrays allow non-rectangular structures to be represented, but the primitive functions manipulate the structures piecemeal. What we need is a mechanism for indicating which of the last axes of an array represent one structure, and should be treated as a unit. Enclose serves this purpose in the nested array system. In carrier APL, when a function is applied, a ***datum rank*** is specified. The datum rank indicates the rank of the data structures represented in the array. If we apply function F with datum rank K to a rank N array X (i.e., $F\{K\}X$), the last K dimensions of X are implicitly enclosed, and F is applied as defined above to a rank $N-K$ array of rank K data items.

The base definitions of most APL functions have natural generalizations to arrays of non-scalar items. The structure functions (e.g., shape (ρ), catenate (,), reverse (Φ)) are indifferent to the type of the data they manipulate and can be extended in a straightforward manner. Scalar equality and its related functions have obvious extensions. Two non-scalar items are equal if the implicitly enclosed arrays have the same shape and equal elements in corresponding locations. The corresponding elements are simple, for the implicit enclosures of carrier APL cannot be nested. Detailed definitions of the primitive functions are given in chapter 3 (see section 3.3).

Each primitive function can be applied with a variety of datum ranks. For example, we can apply equal (=) to two matrices *M*1 and *M*2 in three different ways. Figure 2-11 presents the alternatives. We can compare the two arrays as matrices of simple items, applying equal with datum rank zero and comparing corresponding scalars. Or we can compare them as vectors of vectors, using datum rank one. When the function is applied, the matrices are partitioned into collections of vectors, corresponding vectors are paired, and the function applied to each pair. Or we can compare them as scalars containing matrices, using datum rank two. When the function is applied, each matrix is enclosed and the two scalars compared.

$$M1 \longleftrightarrow \begin{array}{l} COBOL \\ COBOL \end{array} \qquad M2 \longleftrightarrow \begin{array}{l} ALGOL \\ COBOL \end{array}$$

$$M1{=}\{0\}M2 \longleftrightarrow \begin{array}{l} 0\ 0\ 0\ 1\ 1 \\ 1\ 1\ 1\ 1\ 1 \end{array}$$

$$M1{=}\{1\}M2 \longleftrightarrow 0\ 1$$

$$M1{=}\{2\}M2 \longleftrightarrow 0$$

Figure 2-11: Equal applied with three different datum ranks.

Not all datum ranks are permissible for every function. For example, for many selection functions such as take (↑) or compression (/), only the right argument to the function can contain non-scalar items; the left argument is a numeric or boolean guide to the selection of the data. When these functions are applied, the datum rank specified in the application form refers only to the right argument. Other functions such as monadic iota or encode are defined only for arrays of simple items, and an application with a non-zero datum rank is an an error.

We characterize the extended definition of a primitive function by noting the permissible ranks of the items in the arguments and the result. These ranks are assembled in a *datum rank vector* for the function. For a dyadic function, the vector has three components: the permissible ranks of items in the left and right argument, and the corresponding rank of the items in the result. For monadic functions only two components are necessary. For example, equal (=) compares two rank N items and returns a rank 0 result; its datum rank vector is N N 0, where N can be any non-negative integer. Figure 2-12 presents the datum rank vectors for several primitive functions. Note all primitive functions map datum rank uniformly, that is, the rank of the items in the argument(s) determine the rank of the items in the result.

	arg		result
gradeup (⍋)		N	0
catenate (,)	N	N	N
take (↑)	0	N	N
encode (⊤)	0	0	0

Figure 2-12: Datum rank vectors of some primitive functions.

2.3.4 Non-scalar Data

We can extend the remove-duplicates idiom to word lists by adding the appropriate datum rank to the code. Figure 2-13 presents the datum rank vectors for the functions in the idiom, and figure 2-14 presents the solution and a sample evaluation.

	arg	result
dyadic iota (⍳)	$N\ N$	0
shape (ρ)	N	0
monadic iota (⍳)	0	0
equal (=)	$N\ N$	0
compress (/)	0 N	N

Figure 2-13: Datum rank vectors of functions in remove duplicates idiom.

We represent the word list as a ragged character matrix V, one word per row; there is no need for padding. In the idiom, we interpret V as a vector of words. Thus we apply functions dyadic iota (⍳) and shape (ρ) to V with datum rank one. Dyadic iota maps a vector of non-scalar items to a vector of simple indices. Similarly, shape maps a vector to its length. Thus monadic iota (⍳) and equal (=) are applied to arrays of scalars, and we use datum rank zero. We do not need to specify it, for zero is the default datum rank. The left argument to compression (/) is a boolean vector; the right argument is V. We apply compression with datum rank one. Only the right argument is partitioned, compression is applied, and we get our result.

Datum rank gives us a mechanism in carrier APL for manipulating a subarray as a single data object. With this mechanism, APL idioms can be transformed in a routine way to apply to arrays of non-scalar data.

```
V                          ←→      APL
                                   BASIC
                                   APL
                                   COBOL
                                   BASIC
                                   FORTRAN

(V⍳{1}V)                   ←→      1 2 1 4 2 6
ρ{1}V                      ←→      6
⍳ρ{1}V                     ←→      1 2 3 4 5 6
(V⍳{1}V)=⍳ρ{1}V            ←→      1 1 0 1 0 1

((V⍳{1}V)=⍳ρ{1}V)/{1}V     ←→      APL
                                   BASIC
                                   COBOL
                                   FORTRAN
```

Figure 2-14: Removing duplicates from a word list with carrier arrays.

2.3.5 Analysis

Carrier arrays directly addresses the weaknesses of APL we outlined in chapter 1. Ragged structures can be naturally represented in ragged arrays, and the datum rank provides a mechanism for manipulating collections of non-scalar data objects. Both scalar and non-scalar functions can be applied in parallel. APL expressions can be developed for scalars, vectors, matrices or other fixed rank arrays and they extend uniformly to collections of these arrays. The power of the APL programming style is increased.

A ***carrier array*** is a ragged array with an associated three level partitioning. When we apply a function F to an array X with datum rank K (i.e., $F\{K\}X$), the array is partitioned into three levels. The lowest is the ***datum level***; the last K dimensions of the array are enclosed as determined by the datum rank. The middle level is the ***base level***; the arguments to the base definition of F. And top level is the ***carrier level,*** the collection of base arguments.

Note ((V⍳{1}V)=⍳ρ{1}V)/{1}V will remove duplicates from a multiple word lists if each list is represented as a plane in a three-dimensional array. Each primitive function is applied using its base definition, and the array *V* is partitioned into a three-level structure: a collection of lists of words.

We consider the evaluation of *V*ι{1}*V* in more detail, for it presents clearly all three levels of carrier partitioning. In the evaluation, we first form the datum level. Dyadic iota is applied with datum rank one, so we enclose the last dimension of the left and right arguments; each instance of *V* becomes a matrix of words. We next create the base and carrier levels. Dyadic iota is defined for a pair of vectors. We again partition the left and right arguments; each matrix of words becomes a collection of vectors of words. We can now see the three levels at once. The datum level contains the atomic data objects being manipulated, the words in the word list. The base level contains the arguments to the base definition of dyadic iota, the lists of words. And the carrier level is the collection of base arguments, the collection of word lists. To complete the evaluation, we pair corresponding vectors in the partitioned left and right arguments, and apply dyadic iota to each pair. The results are assembled in a new carrier, and partitioning is removed from all arrays. Figure 2-15 presents a sample evaluation.

```
V          ←          APL
                      BASIC
                      APL
                      COBOL
                      BASIC
                      FORTRAN

                      FORTRAN
                      FORTRAN
                      APL
                      COBOL

Vι{1}V     ←          1 2 1 4 2 6
                      1 1 3 4
```

Figure 2-15: A three level carrier evaluation with dyadic iota.

A partitioned carrier array is a dynamic structure. When a function is applied, the array is partitioned. After the evaluation, the partition is removed. A partitioned, three-level carrier array exists only during function application; the base data structure is a heterogeneous ragged array. This is analogous to the execution of machine code. No type is associated with a machine word; it is imposed on the word when it is fetched and an operation performed, and the type is forgotten when the word is stored. Dynamic partitioning gives carrier APL the ability to vary the application axis, as the axis operator does in current APL, for the same array in carrier APL can have many different partitionings.

Chapter 3

CARRIER APL

In this chapter we examine the components of the carrier array system in more detail. We define carrier arrays and their associated three level partition. We define how functions can be applied to a carrier of base arguments. We give the base definitions of the primitive functions. And we describe how defined functions are defined and evaluated.

3.1 Carrier Arrays

A carrier array is a ragged array with an associated partition. The partition is dynamic; it exists only during function application. The basic data structure is a heterogeneous ragged array. To define carrier arrays, we first characterize ragged arrays, and then specify how they are partitioned during function application.

3.1.1 Ragged Arrays

How do we characterize a ragged array? Knuth has noted that an array can be viewed as a special kind of ordered tree [36]; we expand this notion in defining a ragged array. We define a *uniform-depth tree* to be a ordered tree where all paths from root to leaf have the same length; we call this length the depth, or *rank*, of the tree. A *level* in the tree is the set of nodes at a fixed path length from the root; nodes at the Ith level all have a path of length I from the root. A *complete tree* is a tree with uniform branching at each level, that is, on each level, every node on the level has the same number of branches. A rectangular array is a complete, uniform-depth tree. We define a *ragged array* to be a uniform-depth tree.

Each level in the tree corresponds to a *dimension* in the array. The root is the "zeroth" dimension, which we usually recognize only in scalars, and the correspondence proceeds top-down. The first dimension in the array corresponds to the first level in the tree, the second dimension to the second level, and so forth. Data is stored in the leaves of the tree. The data items stored are either characters or numbers, the primitive data elements of current APL. An array has an associated *type* determined by the type of data in the array. A type is either character or numeric. The type of a heterogeneous array is determined by the data in the first leaf of the tree.

For example, a ragged matrix is a uniform-depth tree of rank two. All the paths from root to leaf are of length two; the leaves contain the data items in the matrix. Figure 3-1 graphically illustrates our definition of a ragged matrix. Note this definition of a ragged array is independent of how it is represented in machine memory; we describe possible representations in chapter 5 (see section 5.1).

We permit empty sub-arrays by associating a rank with a leaf node in the tree, and

```
ABC                          --root--
D                           /   |   \
EF                      node   node   node
                        / | \    |     / \
                        A B C    D     E F
```

Figure 3-1: A ragged array and the corresponding tree.

including this rank in determining the length of the path from root to leaf. Leaves which contain a data item have an associated zero rank, but other leaves may have a non-zero rank. To preserve uniform-depth, leaves with a non-zero rank have a correspondingly shorter path from root to leaf. Empty subarrays also have a character or numeric type associated with them; the type is determined when the array is created.

Empty subarrays are important in carrier APL. Carrier partitioning is not preserved between the evaluation of the various function applications in an expression. If some of the base results of a function application are empty, this information must be preserved in the unpartitioned ragged array, so that the next function in the expression may process the empty result. For example, consider 2 0 3↑M, where M is a matrix of three rows. Take (↑) pairs a scalar with a vector to produce a vector. When we evaluate 2 0 3↑M, the base results are three vectors, of length two, zero, and three respectively. These results are assembled in a new carrier. When partitioning is removed, we have a ragged matrix with an empty row.

Rosenburg and Standish both define a ragged array as a mapping from a set of indices to a set of data [67, 70]. This definition is not satisfactory for carrier APL, for it cannot distinguish between empty subarrays and the ragged edges of the array. Rather we define indexing in terms of the definition of a ragged array as a tree. An *index* of a rank K ragged array is a K-tuple of integers which pick a path from root to leaf. The first element in the K-tuple selects a branch from the root to the first level in the tree;

the second element selects a branch from the first level, and so forth. The indexed element is the selected leaf. If a non-existent branch is selected, the index is invalid.

3.1.2 Partitioning

A carrier array is produced when a ragged array is partitioned during function application. The partition consists of three levels. Each level consists of a number of contiguous dimensions in an array; the number is the *rank* of the level. The lowest level is the *datum level*; it contains the atomic data objects manipulated by the function. The middle level is the *base level*; it contains the base arguments of the function being applied. The top level is the *carrier level*; it is a frame carrying the base arguments of the function.

Partitioning can only be described within the context of a function application. Consider a monadic function F applied with datum rank K to a rank N array A, i.e. $F\{K\}A$. Let F have base rank $I\ J$, and let K be a permissible datum rank for F. When F is applied, A is partitioned into a three-level carrier array. The datum level has rank K; when F is applied, the last K dimensions of A are implicitly enclosed. The base level has rank I, which is the rank of the base arguments to F. The carrier level has rank $N-(I+K)$, the difference between the rank of the array A and the rank of the base and datum levels. This difference cannot be negative. If the rank of A is less than $I+K$, A is temporarily coerced to have rank $I+K$, by adding the necessary number of singleton dimensions; A is then partitioned again. The partitioned carrier array produced from A is is called an *M-carrier* for F with datum rank K, where $M \longleftarrow N-(I+K)$. If the rank of carrier level is zero we have a *scalar carrier*.

For example, reversal is applied with datum rank 2 to a rank 6 array A (i.e., $\Phi\{2\}A$), where A has rectangular shape 2 3 4 5 6 7. A is partitioned into a 3-carrier of base arguments; the carrier level has rank 3 and rectangular shape 2 3 4. Each base argument is a vector of matrices; the base level vectors have length 5, and the datum level matrices have rectangular shape 6 7. To complete the evaluation, each base level vector is reversed, and the results are assembled in a new 3-carrier. The partitioning is then removed from both A and the new array; the net result is equivalent to the reversal of A on the fourth dimension in current APL (i.e., ⌽[4]A).

3.2 Application Forms

The concept of an array as a carrier of function arguments permits a great freedom in the pairing of functions with arguments. In discussing carrier APL in previous chapters, we have assumed functions were applied "componentwise": a monadic function is applied independently to each base argument in a carrier array; a dyadic function is applied independently to base arguments in corresponding locations of two carriers. We call this componentwise application *scalar product*. For monadic functions, scalar product is perhaps the only reasonable application form. For dyadic functions, however, other possibilities exist. Current APL offers a rich set of operators to apply scalar functions to arrays. Along with the default application form of scalar product, scalar functions can be applied using inner and outer product, reduction, and scan. In this section we examine how these application forms can be extended to carrier APL.

3.2.1 General Considerations

An *application form* is a syntactic construct which specifies how a function is to be applied to an array (or a pair of arrays). It contains a function, an optional operator, and an optional datum rank. The datum rank is an expression that must evaluate to a non-negative integer; it is usually a constant. If no operator is mentioned, a scalar product is applied; if no datum rank is specified, zero is assumed.

Application forms specify how a function is applied to a collection of base arguments; they partition an array and then select and pair arguments from the carrier level. They do not affect the evaluation of a function. Function evaluation, including all conformability checks and exception handling, is determined independently for each pair of base arguments by the base definition of the function. Application forms apply functions to arguments, and assemble the results.

The base results of each function application are assembled in a new carrier. We can calculate the rank and the partitions of the result carrier before applying the function, for every primitive function maps base rank and datum rank uniformly. However, we cannot know the shape of the result carrier before evaluation, for the primitive functions do not map shape uniformly.

Every application form partitions its array arguments into carriers in the same manner (see section 3.1.2). They differ in how they pair the base arguments of the carriers. In the remainder of the section, we describe the pairings of the various application forms.

3.2.2 Scalar Product: $A\ F\{K\}\ B$

Scalar product is the componentwise application of a function to the base arguments in a carrier array(s). There is no defined ordering of the applications; it is as if they all are evaluated in parallel. The scalar product of a monadic function is straightforward. The array is partitioned; the function applied to each base argument; the results collected in a new carrier; and the partitioning removed. The definition for a dyadic function is similar. Both arrays are partitioned; the carriers checked for conformability; corresponding base arguments paired; the function applied to each pair; the results collected in a new carrier; and the partitioning removed.

Two carriers are conformable for the scalar product of a dyadic function if their carrier levels have equal rank and equivalent shapes. We compare shapes of the carrier level of two M-carriers by comparing the branching at corresponding nodes in their tree definition, working from the root down to the Mth level. If no differences are found, the shapes are equivalent. If one of the carriers is a scalar carrier, we have a scalar extension, which we define below. Otherwise, if the carriers are not conformable, we have an error.

An example of a dyadic scalar product is presented in figure 2-15 of chapter 2. The form is $V\iota\{1\}V$; dyadic iota is applied with datum rank 1 to two copies of a rank 3 array.

3.2.2.1 Scalar Extension: $A\ F\{K\}\ B$

A useful addition to the concept of componentwise application is scalar extension. Consider a dyadic function F applied to two arrays A and B ; the form is $A\ F\{K\}\ B$. Assume A is partitioned into a J-carrier, and B into a 0-carrier. The one base argument in B is paired with each of the base arguments in A, and the function is applied independently to each pair; it is as if the 0-carrier is extended to conform with the

J-carrier. A similar extension is performed if the left argument is the scalar carrier.

For example, suppose we want to cross reference a list of variable names with the text of a defined function, reporting the variables referenced by each function line. Assume the variable names are all one character (as they too frequently are in APL programs). We can represent the variables in a vector *V*, and the function in a matrix *F*, one line per row. (We do not include the function header in *F*.) We now want to apply the membership function between *V* and each row of *F*. Using scalar extension, we specify *V*∈*F*. Membership is defined for pairs of vectors. *F* is partitioned into a 1-carrier; *V* becomes a 0-carrier that can be applied with scalar extension. The vector base argument in *V* is paired with every base argument in *F*, and membership is applied to each pair. Each row of the result is a boolean mask indicating which variables are present in the corresponding row of the function, i.e., (*V*∈*F*)[*I*;] ←→ *V*∈*F*[*I*;]. Figure 3-2 presents a sample evaluation.

```
V       ←→       KASNIR

F       ←→       S ← ⍋,A
                 I ← 1++/S∘.>+\N ← ρA
                 R ← ,¯1+⍳N
                 K ← R[S]⌽A[I],'|'

V∈F     ←→       0 1 1 0 0 0
                 0 1 1 1 1 0
                 0 0 0 1 0 1
                 1 1 1 0 1 1
```

Figure 3-2: An example of scalar extension.

What if *V* is a scalar in the above example, e.g. *V* ←→ '*A*'? Membership requires a vector left argument. A first partitioning of *V* results in a carrier level of negative rank; *V* is then implicitly promoted to a vector. It can then be partitioned into a 0-carrier, and the evaluation proceeds as described above. After the evaluation of the scalar product, the coercion is removed, and *V* remains a scalar.

3.2.3 Outer Product: *A*∘.*F*{*K*}*B*

When a function is applied as an outer product to two ragged arrays, a full cartesian product of the base arguments in the two carriers is formed, and the function is applied independently to each pair. There is no defined ordering of the applications; it is as if they are all evaluated in parallel.

Consider the outer product form *A*∘.*F*{*K*}*B*, where *F* is being applied as an outer product with datum rank *K* to the two arrays *A* and *B*. Assume *A* is partitioned into an *I*-carrier, and *B* a *J*-carrier. We form a cartesian product of the base arguments from the two carriers in two steps. First we pair each base argument *a* from the *I*-carrier *A* with the entire *J*-carrier *B*. Then for each (*a*,*B*) pair, *a* is paired with every base argument *b* in the *J*-carrier *B*. The result is an *I*+*J*-carrier of (*a*,*b*) base argument pairs. *F* is then applied to each pair, the results assembled in a new *I*+*J*-carrier, and the partitioning removed from all arrays. Figure 3-3 presents the partitioning and evaluation of a sample outer product.

For a more interesting example, we extend the cross-reference problem discussed above. Recall we represented a function as a character matrix *F*, and the variables names used in the function as a character vector *V*. We reported the variables referenced by each line of the function. Suppose we now want to cross reference the formal parameters separately from the other variables used in the function. *V* becomes a matrix; the formals in the first row, the other variables in the second. *F* is unchanged. We want to compare each row of *V* with every row of *F*. We apply membership with outer product; the form is *V*∘.∊*F*. Membership is defined for pairs of vectors. Both *V* and *F* are partitioned into

```
A    ←→    10 20 30        B    ←→    1 2 3
           40 50
           60 70 80
```

First pairing: (a,B)

```
(10,1 2 3) (20,1 2 3) (30,1 2 3)
(40,1 2 3) (50,1 2 3)
(60,1 2 3) (70,1 2 3) (80,1 2 3)
```

Second pairing: (a,b)

```
(10,1) (10,2) (10,3)
(20,1) (20,2) (20,3)
(30,1) (30,2) (30,3)

(40,1) (40,2) (40,3)
(50,1) (50,2) (50,3)

(60,1) (60,2) (60,3)
(70,1) (70,2) (70,3)
(80,1) (80,2) (80,3)
```

Evaluation: A∘.+B

```
11 12 13
21 22 23
31 32 33

41 42 43
51 52 53

61 62 63
71 72 73
81 82 83
```

Figure 3-3: Base argument pairing and evaluation of A∘.+B

1-carriers of vector base arguments. Each base argument in the 1-carrier for V is paired with every base argument in the 1-carrier for F, and membership applied to each pair. The result is a 2-carrier containing vector base results; when partitioning is removed, it is a rank 3 ragged array. Figure 3-4 presents the example.

```
V         ←→      KA
                  SNIR

F         ←→      S ← ⍋,A
                  I ← 1++/S∘.>+\N ← ⍴A
                  R ← ,¯1+⍳N
                  K ← R[S]⌽A[I],'|'

V∘.∊F     ←→      0 1
                  0 1
                  0 0
                  1 1

                  1 0 0 0
                  1 1 1 0
                  0 1 0 1
                  1 0 1 1
```

Figure 3-4: An example of outer product.

3.2.3.1 Outer Product and Transpose: *A*∘.*D F*{*K*}*B*

The combination of outer product and transpose is very powerful, and it occurs frequently in APL programming. For example, to add a vector *V* to each column of a matrix *M*, we write 1 1 2⍉*V*∘.+*M*. Unfortunately, in most APL implementations this is a wasteful computation, for the entire outer product is calculated before the transpose is performed. Abrams has noted that the transpose can be regarded as a further specification of the pairing of arguments requested by the outer product, and the superfluous pairs can be discarded before the function is applied [1]. However, in the presence of side-effects (such as a divide-by-zero error), this merging of transpose and outer product is not equivalent to a sequential interpretation of the two functions. Because of its effect on the semantics of the language, the desirability of this optimization has been debated in the APL community for years [1, 2, 73, 75].

In carrier APL we allow the programmer to determine how a transpose, outer product combination should be evaluated. If the transpose is to be a guide in the pairing of arguments to the outer product, the transpose vector can be included in the outer product form; for example, we write *V*∘.1 1 2+*M*. If a full evaluation of the outer product followed by the transposition is desired, we write 1 1 2⍉*V*∘.+*M*.

We extend the definition of outer product to include a transpose vector. A full outer product form is *A*∘.*D F*{*K*}*B*, where *F* is being applied as an outer product with datum rank *K* and transpose vector *D* to the two arrays *A* and *B*. *D* must be a constant. We form a full cartesian product of base arguments as before, but the transpose vector *D* is used to restrict the pairing. It is as if the array of pairs is transposed by *D*. The function *F* is then applied to this restricted set of pairs.

We now define the permissible transpose vectors in an outer product form, and explain in more detail how it restricts the pairing of base arguments. Assume *A* is partitioned into an *I*-carrier, and *B* a *J*-carrier. First, *D* must be a valid transpose vector for the *I*+*J*-carrier of base argument pairs. Thus it must satisfy the following three conditions:

1. (⍴*D*)=*I*+*J*. The length of *D* must equal the rank of the carrier level of the array of pairs.
2. ∧/*D*∊⍳*I*+*J*. *D* must contain only valid dimensions of the carrier level.
3. ∧/(⍳*D*)∊*D*. *D* must be dense.

However *D* is only a means of restricting the full cartesian product of base arguments; it is not a substitute for the general dyadic transpose primitive. Thus we further restrict *D* so that the first *I* elements and the last *J* elements both form ascending sequences, and each sequence contains no duplicates. These are restrictions Abrams imposes on a transpose vector in his general dyadic form [1]. They restrict *D* to indicating how the dimensions of the two carriers are merged in the result. *I*↑*D* maps the dimensions of the *I*-carrier *A* to dimensions in the array of pairs; (−*J*)↑*D* maps the dimensions of the

J-carrier *B* to dimensions in the array of pairs; and the rank of the array of pairs is the maximum value of *D*, i.e., ⌈/*D*. If an arbitrary permutation of the dimensions is required, an explicit transpose should be used.

Because *I*↑*D* and (−*J*)↑*D* are both ascending sequences, we can construct the array of pairs by traversing the two carriers top-down, level by level, mapping levels in the carriers to levels in the array of pairs. Essentially we interleave the two carriers into one result. We give a recursive definition of the pairing below. Let *D*1 be *I*↑*D*; *D*2 be (−*J*)↑*D*; *C*1 be the *I*-carrier *A*; and *C*2 be the *J*-carrier *B*. At each level we compare the first values in *D*1 and *D*2 and based on the comparison, perform one of three actions:

1. If 1↑*D*1 is less than 1↑*D*2, then the first dimension of *C*1 is mapped to this level in the result. We split *C*1 into subarrays along its first axis, drop the first element from *D*1, and recursively pair each subarray with *C*2.
2. If 1↑*D*2 is less than 1↑*D*1, then the first dimension of *C*2 is mapped to this level in the result. We split *C*2 into subarrays along its first axis, drop the first element from *D*2, and recursively pair each subarray with *C*1.
3. If 1↑*D*1 and 1↑*D*2 are equal, then the first dimension of both *C*1 and *C*2 is mapped to this level in the result. We split both *C*1 and *C*2 along the first dimension, trim to a common length, drop the first element from *D*1 and *D*2, and recursively pair corresponding subarrays.

The pairing stops when both *C*1 and *C*2 are scalar carriers, and *D*1 and *D*2 are empty. (Note this algorithm is to define how base arguments are paired in forming the extended outer product. There are many efficient implementations of this type of array traversal [1, 62, 25].)

For example, consider adding a vector *B* to the columns of a matrix *A*. We write *A*∘.1 2 1+*B*.[8] Plus is a scalar function, and the datum rank of the application is zero. Thus *A* is partitioned into a 2-carrier and *B* a 1-carrier. *D*1←2↑*D* is 1 2, and *D*2←¯1↑*D* is

[8]Note *B*∘.1 1 2+*A* is an equivalent expression.

1. The rank of the result is two; B is mapped to the first dimensions of the result, and A is mapped to both dimensions. Using the above algorithm, we first split both A and B along their first dimension, drop the first value from $D1$ and $D2$, and pair corresponding subarrays (a,b). Then, for each (a,b) pair, we split the subarray a of A, pair with b, and drop the remaining value from $D2$. The values of $D1$ and $D2$ are exhausted, and the pairings are complete. The base argument evaluations can now proceed. Figure 3-5 presents sample pairings and evaluation. (Compare with figure 3-3.)

```
A    ←→    10 20 30          B    ←→    1 2 3
           40 50
           60 70 80
```

First pairing:

```
           (10 20 30,1)  (40 50,2)  (60 70 80,3)
```

Second pairing:

```
           (10,1) (20,1) (30,1)
           (40,2) (50,2)
           (60,3) (70,3) (80,3)
```

Evaluation: A∘.1 2 1+B

```
           11 21 31
           42 52
           63 73 83
```

Figure 3-5: Base argument pairing and evaluation of A∘.1 2 1+B

For a more interesting example, we return to the cross-reference problem. We want to cross reference two functions in parallel. Ignoring the distinction between formal parameters and other variables, V is now a matrix, each row corresponding to the name-list for a function. F is rank 3 array, each plane representing a function. Note scalar extension is not applicable here; V is no longer a 0-carrier for membership. Rather, V is a 1-carrier, and F a 2-carrier. We want to compare each base argument in the 1-carrier for

V with every base argument in the corresponding row in the 2-carrier for *F*. We specify *V*∘.1 1 2∊*F*. The outer product is evaluated as defined above. *V* and *F* are partitioned; the 1-carrier for *V* and the first dimension of the 2-carrier for *F* are merged along the first dimension of the matrix of pairs; the second dimension of the 2-carrier is mapped to the second dimension of the matrix of pairs. Membership is applied to each pair of base arguments; the results are assembled in a 2-carrier. When partitioning is removed we have a rank-3 array. Figure 3-6 presents a sample evaluation.

```
V            ←→        KASNIR
                       RDSIL

F            ←→        S ← ⍋,A
                       I ← 1++/S∘.>+\N ← ρA
                       R ← ,¯1+⍳N
                       K ← R[S]⌽A[I],'|'

                       I ← (D=S,D)/⍳1+ρS
                       L ← ¯1+I−0,¯1↓I
                       R ← Lρ(S≠D)/S


V∘.1 1 2∊F   ←→        0 1 1 0 0 0
                       0 1 1 1 1 0
                       0 0 0 1 0 1
                       1 1 1 0 1 1

                       0 1 1 1 0
                       0 0 0 1 1
                       1 1 1 0 1
```

Figure 3-6: An example of outer product with transpose vector.

3.2.4 Reduction: *F*/{*K*}*A*

Reduction is a mechanism for sequentially applying a function to selected elements of an array. Unlike scalar product and outer product, there is an explicit ordering to the function applications. Reduction is defined for vectors of base arguments of a function, and extended to higher rank carriers "row-wise"; each vector of base arguments is independently reduced.

To be applied with reduction, a function F must have base arguments and results of equal rank, i.e., the base rank of F is of the form $I\ I\ I$. Also F must preserve datum rank, i.e. its datum rank vector must be of the form $N\ N\ N$ or 0 0 0. We call a function which meets these restrictions *reducible*. For a reducible function, a carrier of the base results of the function has the same three-level structure as a carrier of either the left or right base arguments.

The reduction of a vector of base arguments can be given a simple recursive definition, as suggested by Jenkins and Michel [31]. Consider the reduction $F/\{K\}A$, where F is applied as a reduction with datum rank K to A. F must be a reducible function with base rank $I\ I\ I$. Assume A is a 1-carrier for F applied with datum rank K; since F is reducible, A has a suitable carrier structure to be a 1-carrier for either the left or right base arguments or the base results of F. Let J be $I{+}K$. Note A has rank $1{+}J$. We define the reduction as follows.

$$F/\{K\}A \quad \longleftrightarrow \quad \begin{array}{ll} (1\uparrow\{J\}A)F\ F/1\downarrow\{J\}A & \textit{if}\ \ 2\leq\rho\{J\}A \\ A & \textit{if}\ \ 1=\rho\{J\}A \\ \text{Identity or Error} & \textit{if}\ \ 0=\rho\{J\}A \end{array}$$

Since APL is right associative, this definition is equivalent to placing F between every base argument in the partitioned 1-carrier A (if $2\leq\rho\{J\}A$) and evaluating the resulting expression.

For example, we can apply catenate with reduction to ravel a word list. Let M be a character matrix representing a word list, one word per row. The reduction form is $,/M$. Catenate has base rank 1 1 1. M is partitioned into a 1-carrier for catenate, a vector of base arguments. This 1-carrier is reduced and the words are successively catenated into one vector. Figure 3-7 presents a sample evaluation. Note we first apply catenate with

```
M          ←→     THE
                  APL
                  IDIOM
                  LIST

,/M,' '    ←→     THE APL IDIOM LIST
```

Figure 3-7: A catenate reduction.

scalar extension to tag a blank onto the end of each row in the matrix.

If we assemble several word lists into a rank-3 array *A*, we can use the catenate reduction to ravel each list. *A* is a 2-carrier for catenate, a matrix of base arguments. Each row of base arguments is independently reduced. Figure 3-8 presents a sample evaluation. Note the planes of the three-dimensional array *A* are displayed adjacently.

```
A          ←→     STRUCTURED  THE       A
                  PROGRAMMING APL       PROGRAMMING
                              IDIOM     LANGUAGE
                              LIST

,/A,' '    ←→     STRUCTURED PROGRAMMING
                  THE APL IDIOM LIST
                  A PROGRAMMING LANGUAGE
```

Figure 3-8: A catenate reduction of a rank 3 array.

3.2.5 Scan: *F*\\{*K*}*A*

Scan is best defined in terms of reduction; it is the successive reduction of all prefixes of a vector. Like reduction, scan is defined for vectors of base arguments of a function, and extended to higher rank carriers row-wise.

Consider the form *F*\\{*K*}*A*, where *F* is applied as a scan with datum rank *K* to *A*. *F* must be a reducible function with base rank *I I I*. Assume *A* is a 1-carrier for *F* applied with datum rank *K*. Let *J* be *I*+*K*. The reduction of *F*/{*K*}*A* is well-defined, and the scan of *A* is defined as follows.

```
F\{K}A  ←→  F/{K}(ιρ{J}A)↑{J}A
```

The expression on the right means the following. Take has base rank 0 1 1 and datum rank of the form 0 *N N*. *A* has rank 1+*J*. Thus *A* is a scalar carrier for right base arguments to take applied with datum rank *J*. Using scalar extension, *A* is paired with each element of the index vector ι*ρ*{*J*}*A*, and take applied to each pair. The result is a new 1-carrier containing the successive prefixes from the first dimension of *A*. The partitioning is removed, resulting in a ragged array of rank 2+*J*. Reduction repartitions the result into a 2-carrier, and independently reduces each row of base arguments, giving us the scan of *A*.

We can use scan to create triangular matrices. For example, consider ,\;̈ι*N*. Flatten (;̈) is a new function; it maps a scalar to a vector (see section 3.3.5.2). ;̈ι*N* creates a one column matrix of indices, and catenate scan combines the rows. Similarly ;̈*Nρ*1 creates a one column matrix of ones, and ,\;̈*Nρ*1 combines them to form a a triangular mask. Figure 3-9 presents sample evaluations.

```
;̈ι3        ←→     1
                  2
                  3

,\;̈ι3      ←→     1
                  1 2
                  1 2 3


;̈3ρ1       ←→     1
                  1
                  1

,\;̈3ρ1     ←→     1
                  1 1
                  1 1 1
```

Figure 3-9: Two scan reductions.

3.2.6 Inner Product: $A\ F\{I\}.G\{K\}\ B$

As noted by Abrams, inner product is essentially a combination of outer product, reduction, and transpose [1]. In carrier APL, by supplying the appropriate transpose vector, it can be defined directly in terms of reduction and outer product.

Consider the inner product form $A\ F\{I\}.G\{K\}\ B$. Assume A is a I-carrier and B a J-carrier for G applied with datum rank K. F must be reducible. Then the inner product is defined as follows.

$A\ F\{I\}.G\{K\}\ B \longleftrightarrow F\{I\}/\ A\circ.D\ G\{K\}\ B$

D is $(\iota I-1),((I-1)+J),(I-1)+\iota J$ which is equivalent to

$1\ 2\ \ldots\ I-1\ I+J-1\ I\ I+1\ \ldots\ I+J-1$

If either I or J is zero, D is omitted from the definition. The transposed outer product pairs the last dimension of the I-carrier A with the last dimension of the J-carrier B. The lengths of corresponding rows along the paired dimensions must be equal. We then apply G to the base argument pairs, remove the partitioning of the result, and apply the reduction.

Note the pairing of the last dimensions of the left and right carriers is a change from current APL, which pairs the first dimension of the right argument to an inner product with the last dimension of the left argument. We make this change for two reasons. First, columns along the first dimension are not always well defined in a ragged array; they may have gaps. And second, in most non-mathematical uses of the inner product, such as the $\wedge.=$ comparison function, the data to be processed lies along the last dimension of both arrays, and the right argument must be transposed to to place the data along the first ,dimension. This transpose is not required in carrier APL.

For an example, we return to the cross-reference problem presented above. Recall we cross referenced a list of variable names with the text of an APL function, reporting the variables referenced by each line of the function. We now want, for each function line, a count of the number of distinct variables it refers to. As before, we assume all variable names are one character, and they are represented in a vector *V*. The function is represented as a matrix *F*, one line of the function per row.

To count the distinct variable references per line, we write *V*+.∈*F*. Membership has base rank 1 1 1. *F* is partitioned into a 1-carrier for membership, *V* a 0-carrier, and an outer product is performed. Since V is a 0-carrier, a transpose vector is not required in the outer product form. The result is a boolean matrix; each row identifies the variables referenced in the corresponding line in the function. Plus reduction is then applied to the rows of the boolean matrix, giving a count of distinct variable references per line. Figure 3-10 presents a sample evaluation.

```
V         ←→        KASNIR

F         ←→        S ← ⍋,A
                    I ← 1++/S∘.>+\N ← ρA
                    R ← ,¯1+⍳N
                    K ← R[S]⌽A[I],'|'

V+.∈F     ←→        2 4 2 5
```

Figure 3-10: An example of inner product.

3.3 Primitive Functions

In this section we present the base definitions of the APL primitive functions. We divide the functions into seven groups: the arithmetic scalars, the equality functions, the relations, the structure functions, the selection functions, the functions without obvious extensions to non-scalar items, and the functions of unbounded rank. We examine each group in turn.

Several APL primitives (indexing, assignment, and branching) do not fall into this classification. We define them separately as *special forms*.

3.3.1 General Considerations

As discussed in chapter 2 (see section 2.3.1), each primitive function is given a *base definition* on arrays of fixed rank. This definition is characterized by a *base rank*, a vector containing the ranks of the arguments and result of the function. For a dyadic function, the vector has three components: the rank of the left argument, the rank of the right argument, and the rank of the result. For monadic functions only two components are necessary. For example, take (↑) is defined to map a scalar and a vector to a vector, and it has base rank 0 1 1. Figure 3-11 gives the base ranks of the APL primitive functions.

An exception is the functions of unbounded rank. These functions cannot be defined for fixed rank arrays, and thus cannot be applied with the partitioning algorithms and application forms defined in sections 3.1 and 3.2. A * in a base rank represents an unbounded argument. We discuss functions of unbounded rank in greater detail below.

On the domain of simple, homogeneous arrays, we define a primitive function by restricting its current APL definition to arrays of the appropriate rank. Detailed definitions of the current APL primitive functions are given by Falkoff and Orth in their proposed APL standard [21]. Note, however, that restricting the current APL definitions to fixed rank arrays changes their effect when they are applied to higher rank arrays in carrier APL. For example, shape (ρ) is defined for vectors. When applied as a scalar product to a matrix in carrier APL, it returns the length of each row. This is quite

Monadic

	arg	result
arithmetic scalars	0	0
grade up (⍋)	1	1
grade down (⍒)	1	1
shape (ρ)	1	0
flatten (,̈)	0	1
reverse (⌽)	1	1
monadic transpose (⍉)	2	2
diagonal (⌀)	2	1
monadic iota (⍳)	0	1
matrix inverse (⌹)	2	2
ravel (,)	*	1
rank (ρ̈)	*	0

Dyadic

	arg		result
arithmetic scalars	0	0	0
relational scalars	0	0	0
membership (∈)	1	1	1
dyadic iota (⍳)	1	1	1
catenate (,)	1	1	1
laminate (,̈)	0	0	1
rotate (⌽)	0	1	1
compress (/)	1	1	1
expand (\)	1	1	1
take (↑)	0	1	1
drop (↓)	0	1	1
matrix divide (⌹)	2	2	2
encode (⊤)	1	0	1
decode (⊥)	1	1	0
deal (?)	0	0	1
dyadic transpose (⍉)	1	*	*
reshape (ρ)	*	*	*

Figure 3-11: Base ranks of primitive functions.

different than the shape of a rectangular matrix in current APL.

Most of the primitive functions have natural extensions to arrays of non-scalar data. Each extended definition can be characterized by a ***datum rank vector***, which gives the permissible ranks of the items in the arguments, followed by the rank of the items in the result. The items in an argument are either restricted to rank 0, or else any rank *N* item is permissible. The rank of the items in the result are determined by the rank of the items in the arguments; arrays of rank *N* items are mapped either to another array of rank *N* items, or to a simple array.

Figure 3-12 presents the datum rank vectors of the primitive functions. An *N* in a datum rank vector represents any integer greater than or equal to zero; in a given datum rank vector, every instance of *N* represents the same value. In addition, if any element of ¯1↓*DRV* of a datum rank vector *DRV* is zero, the corresponding arguments must be simple. For example, the left argument of compression (/) must contain boolean values;

the right argument may contain non-scalar items; and the datum rank vector is 0 *N N*. Similarly, if ¯1↑*DRV* is numeric, the rank of the items in the result have that rank for every application. For example, shape (*ρ*) maps a vector of arbitrary rank items to a simple scalar, and the datum rank vector is *N* 0.

Monadic

	arg	result
arithmetic scalars	*N*	*N*
grade up (⍋)	*N*	0
grade down (⍒)	*N*	0
shape (*ρ*)	*N*	0
flatten (⍮)	*N*	0
reverse (⌽)	*N*	*N*
monadic transpose (⍉)	*N*	*N*
diagonal (⌀)	*N*	*N*
monadic iota (⍳)	0	0
matrix inverse (⌹)	0	0
ravel (,)	*N*	*N*
rank (*ρ̈*)	*N*	0

Dyadic

	arg		result
arithmetic scalars	*N*	*N*	*N*
relational scalars	*N*	*N*	0
membership (∈)	*N*	*N*	0
dyadic iota (⍳)	*N*	*N*	0
catenate (,)	*N*	*N*	*N*
laminate (⍮)	*N*	*N*	*N*
rotate (⌽)	0	*N*	*N*
compress (/)	0	*N*	*N*
expand (\)	0	*N*	*N*
take (↑)	0	*N*	*N*
drop (↓)	0	*N*	*N*
matrix divide (⌹)	0	0	0
encode (⊤)	0	0	0
decode (⊥)	0	0	0
deal (?)	0	0	0
dyadic transpose (⍉)	0	*N*	*N*
reshape (*ρ*)	0	*N*	*N*

Figure 3-12: Datum ranks of primitive functions.

We also extend the domain of the base definitions of the primitive functions to heterogeneous arrays. The extension is straightforward; the definitions below permit heterogeneous arguments, unless explicitly prohibited.

In the remainder of this section, we give the base definitions of the primitive functions and special forms.

3.3.2 Arithmetic Scalars

The arithmetic scalars are the APL arithmetic and boolean functions. Figure 3-13 lists the arithmetic functions; figure 3-14 gives their base ranks and datum ranks. The functions are defined only for numeric data. For numeric scalars, we use the definitions of current APL.

The base definitions are extended to non-scalar data "by leaf". A non-scalar item can be viewed as a ragged array. An arithmetic function is applied to a non-scalar item as if the function is applied to its ragged array representation with a scalar product. For monadic functions there are no conformability constraints; the function is applied to each element of the ragged array comprising the item. For dyadic functions the items must have the same rank and shape, else an error occurs. The carrier partitioning ensures that the ranks are the same; thus we need only check the shapes. If conformable, the corresponding elements of the two ragged arrays are paired, and the function applied.

Monadic	Dyadic
conjugate (+)	plus (+)
negative (−)	minus (−)
signum (×)	times (×)
reciprocal (÷)	divide (÷)
floor (⌊)	minimum (⌊)
ceiling (⌈)	maximum (⌈)
exponential (*)	power (*)
natural logarithm (⍟)	logarithm (⍟)
magnitude (|)	residue (|)
factorial (!)	binomial (!)
roll (?)	circular (○)
pi times (○)	and (∧)
not (~)	or (∨)
	nand (⍲)
	nor (⍱)

Figure 3-13: Arithmetic scalar functions.

	Base		Datum	
	arg	result	arg	result
Monadic scalars	0	0	N	N
Dyadic scalars	0 0	0	$N\,N$	N

Figure 3-14: Ranks of the arithmetic scalar functions.

3.3.3 Equality

The functions concerning equality are the scalar functions equal (=) and not equal (≠) and the the two membership-testing functions dyadic iota (⍳) and membership (∈). Figure 3-15 gives their base and datum ranks.

	Base		Datum	
	arg	result	arg	result
equal (=)	0 0	0	$N\,N$	0
not equal (≠)	0 0	0	$N\,N$	0
membership (∈)	1 1	1	$N\,N$	0
dyadic iota (⍳)	1 1	1	$N\,N$	0

Figure 3-15: Equality functions.

3.3.3.1 Equal (=)

Equal maps a pair of scalars to a boolean result. We compare simple scalars with the definition of current APL. We extend equal to non-scalar data recursively, recurring on the rank of the datum, using the current APL definition as a base case.

Let $V1$ and $V2$ be vectors. Then we define equality of rank 1 data items as follows:

$V1$={1}$V2$ ⟷ ((ρ$V1$)=ρ$V2$) ∧ ∧/$V1$∘.1 1=$V2$

Note we use a transposed outer product to compare corresponding elements in $V1$ and $V2$. It will return a boolean result if $V1$ and $V2$ have different lengths, whereas the scalar product $V1$=$V2$ or the inner product $V1$∧.=$V2$ would return a conformability error.

We can generalize this definition by noting a rank K item can always be partitioned into a vector of rank K−1 items. Let $A1$ and $A2$ be rank K arrays. Then we have:

$A1$ ={K}$A2$ ⟷ ((ρ{K−1}$A1$)=ρ$A2${K−1}) ∧ ∧/$A1$∘.1 1={K−1}$A2$

We assume $A1$ and $A2$ have the same rank; because of carrier partitioning, items of different rank can never be compared by the equal function.

3.3.3.2 Not Equal (≠)

Not equal is defined to be the complement of equal. Let $A1$ and $A2$ be two rank K arrays. Then:

$A1{\neq}\{K\}A2 \longleftrightarrow {\sim}A1{=}\{K\}A2$

3.3.3.3 Membership (∈)

Membership maps a pair of vectors to a vector. Each item in the left argument is mapped to a boolean in the result which identifies if the item is present in the right argument. The argument vectors may be vectors of rank N items of any type; the two vectors may have different lengths. The result is a boolean vector of simple items; it has the length of the left argument.

Membership is defined in terms of the equal function. Let $A1$ and $A2$ be two arrays of rank K+1. Then:

$A1{\in}\{K\}A2 \longleftrightarrow \vee/A1\circ.{=}\{K\}A2$

3.3.3.4 Dyadic Iota (ι)

Dyadic iota maps a pair of vectors to a vector. Each item in the right argument is mapped to the index of its first occurrence in the left argument; if the item does not occur in the left argument, it is mapped to the one plus the length of the left argument. The argument vectors may be vectors of rank N items of any type; the two vectors may have different lengths. The result is an integer vector of simple items; it has the length of the right argument.

Dyadic iota is defined with the not equal function. Let $A1$ and $A2$ be two arrays of rank K+1. Then:

$A1 \iota \{K\} A2 \longleftrightarrow 1{+}{+}/{\wedge}\backslash A2{\circ}.{\neq}\{K\}A1$

The idiom ∧\ preserves the longest prefix string of ones in a boolean vector and sets the remainder of the vector to zero. Each rank K item along the first dimension of $A2$ is compared with all the rank K items along the first dimension of $A1$; the prefix of unequal comparisons is identified; and one is added to its length, giving the index of the first match.

3.3.4 Relations

The relational functions are the scalar functions less than (<), less than or equal to (≤), greater than (>), and greater than or equal to (≥), and the two sorting functions grade up (⍋) and grade down (⍒). Figure 3-16 gives their base and datum ranks.

	Base		Datum	
	arg	result	arg	result
less than (<)	0 0	0	N N	0
less than or equal to (≤)	0 0	0	N N	0
greater than (>)	0 0	0	N N	0
greater than or equal to (≥)	0 0	0	N N	0
grade up (⍋)	1	1	N	0
grade down (⍒)	1	1	N	0

Figure 3-16: Relational functions.

3.3.4.1 Less Than (<)

Less than maps a pair of scalars to a boolean result. In current APL, less than is defined only for numeric scalars. In carrier APL, we compare all data types. One character is less than another if it precedes the other in the atomic vector; if $C1$ and $C2$ are character scalars, then

$C1{<}C2 \longleftrightarrow (\square AV \iota C1){<}\square AV \iota C2$

Also any character is less than every number.

We extend less than to determine a lexicographic ordering of vectors. Let *V1* and *V2* be two vectors. *V1* is less than *V2* either if *V1*[*I*] is less than *V2*[*I*] for the smallest *I* for which the two vectors differ, or if there is no such *I*, *V1* is shorter than *V2*. In carrier notation, we write:

```
V1<{1}V2  ←→  T1∨T2
                    where
                       T1 ← 1↑(+/∧\V1∘.1 1=V2)↓V1∘.1 1<V2
                       T2 ← (∧/V1∘.1 1=V2)∧(ρV1)<ρV2
```

T1 finds and compares the first values for which the two vectors differ, and *T2* compares the lengths of two vectors in the case that they have no first difference.

We can generalize this definition to to arbitrary rank items by noting that a rank *K* item can be partitioned into a vector of rank *K*−1 items. Let *A1* and *A2* be rank *K* arrays. Then:

```
A1<{K}A2  ←→  T1∨T2
                    where
                       T1 ← 1↑(+/∧\A1∘.1 1={K−1}A2)↓A1∘.1 1<{K−1}A2
                       T2 ← (∧/A1∘.1 1={K−1}A2)∧(ρ{K−1}A1)<ρ{K−1}A2
```

We assume *A1* and *A2* have equal rank; because of carrier partitioning, unequal rank items cannot be compared with the less than function.

3.3.4.2 The Other Scalar Comparison Functions (≤ > ≥)

Given the definitions of less than (<) and equal (=), we can easily define the remaining scalar comparison functions:

- Less than or equal to:

  ```
  A1≤{K}A2  ←→  (A1<{K}A2) ∨ A1={K}A2
  ```

- Greater than:

  ```
  A1>{K}A2  ←→  ~A1≤{K}A2
  ```

- Greater than or equal to:

  ```
  A1≥{K}A2  ←→  ~A1<{K}A2
  ```

3.3.4.3 Grade Up (⍋)

Grade up is a sorting function; it maps a vector to a vector of indices that could be used to rearrange the original vector into ascending order. In current APL, grade up is restricted to numeric vectors; in carrier APL, it is extended to vectors of non-scalar data of arbitrary type. The result is always a vector of simple indices.

Let A be a rank K+1 array. Then ⍋{K}A is the the permutation of ⍳⍴{K}A which satisfies the following:

1. For all I and J in ⍳⍴{K}A, if I<J then

   (A[⍋{K}A])[I] ≤{K} (A[⍋{K}A])[J]

2. If A[I]={K}A[J] and I<J, then

   ((⍋{K}A)⍳I) < (⍋{K}A)⍳J

The second condition ensures that the sort is stable [37]. Indexing is a special form defined in section 3.3.9.1. To evaluate A[I], we partition A into a vector of rank K items, and index the vector with I, returning an item.

3.3.4.4 Grade Down (⍒)

Grade down is the complementary sort to grade up; it sorts into descending order. Informally, grade down is the reverse of the permutation generated by grade up. However, the reverse of a grade up does not give a stable sort, so we explicitly define grade down.

Let A be a rank K+1 array. Then ⍒{K}A is the the permutation of ⍳⍴{K}A which satisfies the following:

1. For all I and J in ⍳⍴{K}A, if I<J then

   (A[⍒{K}A])[I] ≥{K} (A[⍒{K}A])[J]

2. If A[I]={K}A[J] and I<J, then

   ((⍒{K}A)⍳I) < (⍒{K}A)⍳J

3.3.5 Structure Functions

The structure functions manipulate the structure of an array; they are indifferent to the type of the items. Thus they extend naturally to heterogeneous arrays and arrays of non-scalar data. The structure functions are shape (ρ), flatten (,̈), reverse (⌽), monadic transpose (⍉), diagonal (⌀), catenate (,), laminate (,̈), and rotate (⌽). Flatten and diagonal are new primitives, and laminate has a distinct token. Figure 3-17 gives their base and datum ranks.

	Base			Datum		
	arg		result	arg		result
shape (ρ)		1	0		N	0
flatten (,̈)		0	1		N	0
reverse (⌽)		1	1		N	N
transpose (⍉)		2	2		N	N
diagonal (⌀)		2	1		N	N
catenate (,)	1	1	1	N	N	N
laminate (,̈)	0	0	1	N	N	N
rotate (⌽)	0	1	1	0	N	N

Figure 3-17: Structure functions.

3.3.5.1 Shape (ρ)

Shape maps a vector or rank K items to its length. The result is a simple scalar. We discuss shape further in the context of reshape (see section 3.3.8.4).

3.3.5.2 Flatten (,̈)

Flatten is defined for a scalar containing a rank K item. It flattens the item, constructing a ravel order vector of its leaves. The result is a vector of simple items.

3.3.5.3 Reverse (⌽)

Reverse reverses a vector of rank K items.

3.3.5.4 Monadic Transpose (⍉)

Monadic transpose transposes a matrix of rank K items. To ensure the result is a well-defined ragged array, the rows of the matrix are truncated to a common length before the transpose is performed, i.e., for a rank K+2 matrix A

⍉{*K*}*A* ←→ ⍉{*K*}(⌊/ρ{*K*}*A*)↑{*K*}*A*

3.3.5.5 Diagonal (⌀)

Diagonal selects the diagonal of a matrix of rank K items. It is the equivalent of the 1 1⍉*M* of current APL. As for transpose, the rows of the matrix are truncated to a common length before the diagonal is selected.

3.3.5.6 Catenate (,)

Catenate joins two vectors of rank K items into a longer vector of rank K items.

3.3.5.7 Laminate (⍮)

Laminate joins two scalars of rank K items into a vector of rank K items. Note laminate has a distinct function token in carrier APL.

3.3.5.8 Rotate (⌽)

Rotate rotates a vector of rank K items as determined by an integer scalar left argument. The left argument must be simple.

3.3.6 Selection Functions

The selection functions select items from an array. They extend naturally to heterogeneous arrays and arrays of non-scalar data. The selection functions have a boolean or numeric left argument which guides the rearrangement of the data; non-scalar items are not allowed in these guides. The selection functions are compress (/), expand (\), take (↑), and drop (↓). Figure 3-18 gives their base and datum ranks.

	Base		Datum	
	arg	result	arg	result
compress (/)	1 1	1	0 N	N
expand (\)	1 1	1	0 N	N
take (↑)	0 1	1	0 N	N
drop (↓)	0 1	1	0 N	N

Figure 3-18: Selection functions.

Expand and take require fill items. The *fill item* is a singleton of rank equal to the datum rank of the application. A singleton is an array in which each dimension is a vector of length one. The data at the leaf of the singleton fill item is either zero or blank, and it is determined by the type of the array. (See section 3.1.1 for a discussion of types.)

3.3.6.1 Compress (/)

Compress pairs a boolean vector with a vector of rank K items, and selects elements from the right vector which correspond to ones in the left vector. The two vectors must have the same length. The left argument must be simple.

3.3.6.2 Expand (\)

Expand expands a vector of rank K items as determined by boolean vector left argument. The length of the right vector must equal the sum of the boolean. The left argument must be simple.

3.3.6.3 Take (↑)

Take pairs an integer scalar with a vector of rank K items, and takes the designated number of items from the vector. The scalar must be simple.

3.3.6.4 Drop (↓)

Drop pairs an integer scalar with a vector of rank K items, and drops the designated number of items from the vector. The scalar must be simple.

3.3.7 Functions without obvious extensions to non-scalar items

Several APL functions have no obvious extension to arrays of non-scalar items. To define these functions, we restrict their current APL definitions to fixed rank arrays of simple scalars. They are not defined for heterogeneous arrays or arrays of non-scalar items, and they cannot be applied with a non-zero datum rank. The functions are monadic iota (⍳), matrix inverse (⌹), matrix divide (⌹), encode (⊤), decode (⊥), and deal (?). Figure 3-19 gives their base ranks and datum ranks.

	Base			Datum		
	arg		result	arg		result
monadic iota (⍳)		0	1		0	0
matrix inverse (⌹)		2	2		0	0
matrix divide (⌹)	2	2	2	0	0	0
encode (⊤)	1	0	1	0	0	0
decode (⊥)	1	1	0	0	0	0
deal (?)	0	0	1	0	0	0

Figure 3-19: Functions with no extension to non-scalar data.

3.3.8 Functions of unbounded rank

Several primitives cannot be defined for fixed rank arrays; an argument to each of these functions has *unbounded rank*. An array argument to a function of unbounded rank cannot be partitioned into a three-level carrier of base arguments, for we cannot determine the rank of the base level. Thus, functions of unbounded rank cannot be applied with the application forms defined in section 3.2; they must be applied to an entire array. However, the functions can be applied with datum rank. Array arguments are partitioned into a a two level structure. The functions extend to higher rank arrays in an idiosyncratic manner, similar to the mixed functions of current APL.

The functions of unbounded rank are ravel (,), rank ($\ddot{\rho}$), dyadic transpose (⍉), and reshape (ρ). Rank ($\ddot{\rho}$) is a new function. Figure 3-20 presents the base and datum rank.

	Base			Datum		
		arg	result		arg	result
ravel (,)		*	1		N	N
rank ($\ddot{\rho}$)		*	0		N	0
dyadic transpose (⍉)	1	*	*	0	N	N
reshape (ρ)	*	*	*	0	N	N

Figure 3-20: Functions of unbounded rank.

3.3.8.1 Ravel (,)

Ravel is defined as in current APL; it flattens an arbitrary array into a vector. If a datum rank of K is given in the form $,\{K\}A$, then A is partitioned into an array of rank K items, and the partitioned result is a vector of rank K data. If A has rank less then K, it is coerced to have rank K, and the result is a singleton vector of rank K data. In every case, when partitioning is removed, the result is a rank K+1 array.

3.3.8.2 Rank ($\ddot{\rho}$)

Because shape (ρ) is defined to give the length of a vector, the current APL idiom $\rho\rho A$ does not return the rank of an array in carrier APL; rather it returns the shape of the shape. Rank is a new function which gives the rank of an array. $\ddot{\rho}\{K\}A$ partitions a rank N array into a rank $N-K$ array of rank K items, and returns $N-K$, the rank of the upper partition. If N is less than K, zero is returned.

3.3.8.3 Dyadic Transpose (⍉)

Dyadic transpose is defined for a vector left argument, and an arbitrary rank right argument. Consider *V*⍉{*K*}*A*, where the length of *V* is *I* and the rank of *A* is *N*. *A* is partitioned into an *M*-carrier of rank *I* subarrays, each containing rank *K* data, where *M* ←→ *N*−(*I*+*K*). *V* transposes each rank *I* subarray independently; the results are assembled in a new *M*-carrier. If *I* is greater than *N*−*K*, *A* is coerced to have rank *I*+*K*. We define how to transpose a ragged array below.

Note the rank of the subarray to be transposed depends on the length of *V*, and the rank of the result depends on the value of *V*. This is the reason for restricting the left argument of a transpose to a vector. If we allow a carrier of vectors to transpose a carrier of subarrays, we cannot partition the carrier of subarrays without restricting the vectors to a common length, and the results of the various transposes cannot be assembled in a new carrier without restricting the number of duplicates in the values of the vectors. These restrictions are in conflict with the carrier concept of the independent application of base argument pairs.

V must meet all the requirements for a transpose vector in current APL.

1. (⍴*V*)=*I*. The length of *V* equals the rank of the subarray being transposed. This is assured by the partitioning.
2. ∧/*V*∊⍳*I*. *V* must contain only valid dimensions of the subarray.
3. ∧/(⍳*V*)∊*V*. *V* must be dense.

We define a dyadic transpose as a series of applications of the monadic transpose function and diagonal function (see sections 3.3.5.4 and 3.3.5.5). Applied with an appropriate datum rank, the monadic transpose function can interchange any two adjacent dimensions in an array. If *A* is a rank *N* array, ⍉{*K*}*A* interchanges the *N*−*K* and the *N*−(*K*+1) dimensions of *A*. Similarly, the diagonal function can merge any two adjacent dimensions. Thus a dyadic transpose can be effected by a series of monadic transposes followed by a series of diagonals. The number of monadic transposes is determined by the number of inversions which keep the transpose vector from sorted order.[9] The number of diagonals is determined by the number of duplicate elements in the transpose vector. Figure 3-21 presents the definition of two transposes.

4 3 1 2⍉A ←→ ⍉{1}⍉{2}⍉⍉{1}⍉{2}A

3 2 1 1⍉A ←→ ⌀{2}⍉{1}⍉{2}⍉⍉{1}⍉{2}A

Figure 3-21: Two dyadic transposes defined.

Because the monadic transpose and diagonal functions truncate the rows of their matrix argument to a common length, the ordering of the transposes and diagonals can affect the result if the array being manipulated is not rectangular. We define a canonical ordering, which is the series of monadic transposes corresponding to the interchanges in the "bubble sort" [37] of the transpose vector, followed by the appropriate number of diagonal functions. We make repeated passes from right to left over the transpose vector, interchanging numbers in the vector that are out of order; for each interchange of numbers, we interchange the corresponding dimensions in the array with a monadic transpose. For example, to effect 3 1 2⍉A, we first interchange the first and second dimensions of A, and then the second and third. Once the dimensions have been appropriately arranged, we merge the adjacent dimensions where required with the diagonal function, again moving from left to right along the sorted transpose vector. Both the examples in figure 3-21 are in canonical order. (Note this algorithm is to define a dyadic transpose; it is not necessarily an implementation strategy.)

[9]See Knuth [37] for a discussion of inversions.

3.3.8.4 Reshape (ρ)

Reshape is defined for arbitrary rank left and right arguments. The elements of the left argument must be non-negative integers; the elements of the right argument may have any type. The definition is recursive, recurring on the rank of the left argument. For a scalar *S* and an array *A*, we use the reshape of current APL. *SρA* creates a vector of *S* elements selected in ravel order from *A*, recycling to the beginning of *A* if necessary. This extends immediately to non-scalar items. *Sρ{K}A* partitions *A* into an array of rank *K* items, and reshapes the array.

For a vector *S*, we say:

```
Sρ{K}  ←→  S↑{K}(+\0,¯1↓S)↓{K}(+/S)ρ{K}A
```

(+/*S*)ρ{*K*}*A* forms a vector of rank *K* items. Each element of (+\0,¯1↓*S*) is paired with the vector using scalar extension, and drop applied; the result before partitioning is removed is a matrix of rank *K* items. When take is applied with datum rank *K*, each element of *S* selects a prefix from the corresponding row in the matrix. The sequence of drop followed by take assures that successive elements are selected from *A*.

For *S* with rank *N*, where *N* is greater than one, we say:

```
Sρ{K}  ←→  (ρS)ρ{K+1}(,S)ρ{K}A
```

Recall ρ*S* has rank *N*−1 if *S* has rank *N*, and ,*S* is a vector. Thus the equivalence gives a recursive definition with a well-defined base case.

A ragged array introduces a new concept of shape into APL. The shape (ρ) of a rank *N* ragged array is an array of rank *N*−1, each element containing the length of the corresponding vector in the original array. We call this an ***additive*** shape, for the number of elements in an array equal the sum of the elements in its shape. Rectangular arrays have traditionally had a ***multiplicative*** shape. The multiplicative shape of a rank *N* rectangular array is a length N vector; each element of the vector giving the length of the corresponding dimension in the array. Thus the number of elements in a rectangular array is the product of the elements in its multiplicative shape. For vectors, the additive and multiplicative shapes are the same, but they differ for higher rank arrays.

Note that the multiplicative shape of any non-empty rectangular array can be calculated from its additive shape. Figure 3-22 presents the algorithm. For any ragged array, the algorithm gives the shape of the smallest rectangular array which contains it.

```
$RECTSHAPE \leftarrow \rho\{K\}A$

for $I \leftarrow K-1$ to 1 do
    $RECTSHAPE \leftarrow RECTSHAPE,\ \lceil/,\rho\{I\}A$
```

Figure 3-22: Calculating rectangular shape of a rank $K+1$ array A

Reshape constructs an array from an additive shape. We can examine the relationship of reshape with the shape function by examining the form $(\rho A)\rho A$. Let A be a rank 3 array. Then, by expanding the recursive definition of reshape, we get:

$$(\rho A)\rho A \longleftrightarrow (\rho\rho\rho A)\rho\{2\}(,\rho\rho A)\rho\{1\}(,\rho A)\rho A$$

It is clear that $\rho\rho\rho A \longleftrightarrow \rho\{2\}A$ and $\rho\rho A \longleftrightarrow \rho\{1\}A$. So we have:

$$(\rho A)\rho A \longleftrightarrow (\rho\{2\}A)\rho\{2\}(,\rho\{1\}A)\rho\{1\}(,\rho A)\rho A$$

This corresponds to an intuitive model of building a tree level by level: first constructing a collection of vectors, assembling the vectors into matrices, and the matrices into a three-dimensional array. Thus we say:

$$(\rho A)\rho A \longleftrightarrow A$$

And notice that:

$$(\rho A)\rho A \longleftrightarrow (\rho A)\rho,A$$

So we get the familiar identity of current APL:

$$(\rho A)\rho,A \longleftrightarrow A$$

By a similar analysis, we see $S\rho A$ has shape S and rank $1+\tilde{\rho}S$.

3.3.9 Special Forms

Special forms are constructs in APL which cannot be defined as monadic or dyadic functions. The special forms are indexing ([]), because of its unconventional syntax; and assignment (←), indexed assignment ([]←), and branching (→), because they are primarily evaluated for side effect. The special forms cannot be applied with the carrier partitioning and application forms defined in sections 3.1 and 3.2. Each special form determines how its arguments are partitioned and results formed.

3.3.9.1 Index ($A[I;\ldots;K]$)

The index form $A[I;\ldots;K]$ is the same as in current APL; the array A is paired with an an index list delimited by brackets. The elements of the index list can be arbitrary rank arrays of non-negative integers. A is partitioned into an array of rank one plus the number of semicolons in the form; the remaining dimensions, if any, are enclosed to form non-scalar items. If the rank of A is less than the number of semicolons in the form, A is coerced to the appropriate rank by adding singleton dimensions.

To define indexing we introduce an auxiliary function denoted by ⎕.[10] ⎕ indexes a vector with a scalar, using the definition of current APL, i.e., V⎕I ⟷ $V[I]$ where $I\in\iota\rho V$. ⎕ returns an error if $\sim I\in\iota\rho V$. ⎕ extends directly to a vector of non-scalar items. If A has rank K+1, then A⎕$\{K\}I$ partitions A into a vector of rank K items, and indexes the vector with I. The result is a scalar containing a rank K item; with partitioning removed, it is a rank K array. Thus ⎕ has base rank 1 0 0 and datum rank N 0 N.

[10]Note ⎕ is not a carrier APL primitive, and is used only in the definition of indexing.

With $\square$ we can define the indexing form. Let A be a rank 3 array. Let I, J, and K be ragged arrays of non-negative integers with ranks N, M, and O, respectively. Then:

$$A[I;J;K] \longleftrightarrow ((A\circ.\square\{2\}I)\circ.\square\{1\}J)\circ.\square K$$

The result is a rank $N+M+O$ array of items selected from A.

Note the definition fits nicely with the definition of a ragged array as a tree; we index by selecting items in a breadth first search of the tree. $A\circ.\square\{2\}I$ indexes the vector which is the first level in the tree, and for each item selected, $\circ.\square\{1\}J$ indexes the vectors which form the second level of the tree, and again, for the items selected $\circ.\square K$ indexes the vectors on the third level of the tree.

This definition for a rank 3 array extends to the general case. Let A be a rank N array, and let $I_1,I_2,\ldots,I_j$ be a sequence of arbitrary rank ragged arrays of non-negative integers. Then:

$$A[I_1;I_2;\ldots;I_j] \longleftrightarrow (\ldots((A\circ.\square\{N-1\}I_1)\circ.\square\{N-2\}I_2)\ldots)\circ.\square\{N-J\}I_j$$

The result is a rank $+/(\ddot{\rho}I_1),(\ddot{\rho}I_2),\ldots,\ddot{\rho}I_j$ array of rank $N-J$ items; with partitioning removed it is a rank $(N-J)++/(\ddot{\rho}I_1),(\ddot{\rho}I_2),\ldots,\ddot{\rho}I_j$ array.

Empty subscript positions are permissible, and if given, select every element in the corresponding dimension. This is accomplished by dropping the corresponding application of $\square$. For example, if A is a rank 3 array, then:

$$A[I;;K] \longleftrightarrow (A\circ.\square\{2\}I)\circ.\square K$$

If the array is ragged in the dimension corresponding to the empty subscript, no array value could be substituted for the empty subscript to give an equivalent result.

As in current APL, an element of an index list may contain repeated values. Figure 3-23 presents a sample application of indexing.

```
A                   ⟷    ABCD
                         EFG
                         HIJ

                         KL
                         MNO

A[1 2;2 1ρ2;2 3]    ⟷    EF
                         EF

                         EF


                         MN
                         MN

                         MN
```

Figure 3-23: An sample application of indexing.

3.3.9.2 Assignment (←)

Assignment is the same as in current APL; the value of the right expression is associated with the name on the left.

Assignment introduces a side effect which affects the order of execution of functions in an expression. To avoid ambiguity,[11] carrier APL follows the "Minnowbrook rule" as described by Lathwell [38]: Names are bound in right to left order as they appear in an expression. Thus an assignment in an expression may affect the value of identifiers to its left in the expression, but not the identifiers to its right.

[11]See Wiedmann [75] for a discussion of potential ambiguities.

3.3.9.3 Indexed Assignment ($A[I;\ldots;K]\leftarrow B$)

Indexed assignment is similar to the indexed assignment of current APL. Consider the form $A[I_1;I_2;\ldots;I_J]\leftarrow B$, where A is a rank N array. $A[I_1;I_2;\ldots;I_J]$ must be a valid index form. The result of $A[I_1;I_2;\ldots;I_J]$, before partitioning is removed, is a rank M array of rank K items, where $M \longleftarrow +/(\ddot{\rho}I_1),(\ddot{\rho}I_2),\ldots,\ddot{\rho}I_J$ and $K \longleftarrow N-J$. B is also partitioned into an array of rank K items. As in current APL, the singleton dimensions are removed from the partitioned values of $A[I_1;I_2;\ldots;I_J]$ and B, and the ranks and shapes of the resulting arrays compared. (A dimension is a *singleton* if the length of all the vectors along the dimension is one, i.e., if C is a rank N array, then the Ith dimension is a singleton if $1\wedge.=\rho\{N-I\}C$.) If, with singleton dimensions removed, the partitioned value B has the same shape and rank as the indexed value $A[I_1;I_2;\ldots;I_J]$, then the items of B replace the corresponding indexed items of A. If, with singleton dimensions removed, B is a scalar with one rank K item, then the single item replaces each indexed item in A. Otherwise $A[I_1;I_2;\ldots;I_J]$ and B are not conformable for indexed assignment.

The rule for disambiguating multiply specified values is the rule give by Falkoff and Orth in their APL Standard [21]. If an item in A is selected more than once by $A[I_1;I_2;\ldots;I_J]$ then the item assigned will be the item in B with the largest index in the vector $,\{K\}B$.

3.3.9.4 Branch ($\rightarrow$)

As in current APL, branching transfers control within the body of a defined function.

3.4 Defined Functions

In current APL the application of defined functions is more constrained than the application of primitives. A defined function is applied to its entire array argument(s). It does not extend meaningfully to higher rank arrays unless explicitly programmed with the extension in mind. Moreover, defined functions cannot be applied with any of the operators.

In carrier APL, we apply defined functions in the same manner as we apply the primitives. The partitioning algorithm and application forms defined in sections 3.1 and 3.2 are used, permitting defined functions to be applied in parallel to arrays of non-scalar items. In this section we specify how defined functions are defined, applied, and evaluated.

3.4.1 Applying defined functions

To apply a defined function with an application form, we need the same information about the function that we have for the primitives. The function must have a base definition, a base rank, and a datum rank. However, not all defined functions can be defined as a mapping from fixed rank arrays to fixed rank arrays. Thus we divide the defined functions into several categories: functions of bounded rank, functions of unbounded rank, functions which return no explicit result, and niladic functions. We consider each category in turn.

3.4.1.1 Defined functions of bounded rank

A defined function of bounded rank is a monadic or dyadic function that returns an explicit result, and that can be defined for fixed rank arrays. It can be applied with all of the application forms.

The base rank and the permissible datum ranks must be declared. Any non-negative integer is a valid base rank for an argument or result of a function of bounded rank. The permissible datum rank of an argument may be 0 or N, where an N means any datum rank is permissible. The permissible datum rank of the result may also be 0 or N, but it may be N only if one of the arguments has datum rank N. Thus defined functions of bounded rank map a rank an array of N items to a simple array, or to another array of rank N items.

The base definition is given by the function body. It must accept a base argument(s) with a base and datum level corresponding to the declared ranks for the argument, and return a base result with a base and datum level corresponding to the declared ranks of the result. We cannot determine by static examination if a function body returns results with the correct base and datum rank. However, if in the context of the evaluation of an application form, a function produces base results of incorrect rank, an error is signalled and application halts.

We declare the base rank and the permissible datum ranks in the function header by attaching the rank attributes to the corresponding formal parameter. For example, figure 3-24 presents the definition of a dyadic function *MERGE* with base rank 1 1 1 and datum rank N N N. Local variables are specified as in current APL, separated from the formal parameters by a semicolon. The function body may be a multi-line program, including labels and branching, as in current APL.

```
    ∇ R:1:N ← X:1:N MERGE Y:1:N ;L
[1] L ← (ρX)⌊ρY
[2] R ← ,(L↑X),̈L↑Y
    ∇
```

Figure 3-24: A defined function *MERGE* with base rank 1 1 1 and datum rank *N N N*.

When *MERGE* is applied with an application form, the base and datum ranks are used to partition the argument arrays into carriers, and the application form determines the pairing of base arguments. For each pair of base arguments, the left argument is bound to *X*, the right argument to *Y*, and the function body is evaluated. The base result is the value of *R*. The base results are assembled in a new carrier, as determined by the application form. Partitioning is removed, and the application is complete.

The application of a defined function with order dependent side effects to a carrier of base arguments is usually undefined. Scalar product and outer product have no defined ordering of function applications to base argument pairs. Reduction and scan apply functions sequentially to the items of a vector, but there is no defined ordering of their row-wise extension to higher rank arrays. Naturally, any implementation will order the applications of a function to the arguments in a carrier, but the ordering is implementation dependent, and may vary.

3.4.1.2 Defined functions of unbounded rank

A defined function of unbounded rank is a monadic or dyadic function which returns an explicit result, and which is not a mapping from a set of fixed rank arrays to a set of fixed rank arrays. The function header and body are identical to the defined functions of current APL. No base or datum rank vectors are specified.

A function of unbounded rank is applied to its entire array argument(s); no three-level carrier partitioning is performed. However, a monadic defined function *BAR* of unbounded rank can be applied to a rank N array A with a datum rank K by specifying $BAR\{K\}A$. A is partitioned into a rank $N-K$ array of rank K items, and the formal parameter of BAR is bound to the partitioned value. If N is less than K, the value of A is coerced to rank K by adding singleton dimensions. In a dyadic application, e.g., $A\ DBAR\{K\}B$, both array arguments are partitioned into arrays of rank K items. The permissible datum ranks are not specified; the evaluation of the function body determines if K is an acceptable datum rank, and also determines the datum rank of the result.

3.4.1.3 Defined functions with no explicit result

Functions with no explicit result may be defined as functions of bounded rank, or of unbounded rank. If defined with bounded rank, the base and datum ranks of the arguments are declared and attached to the formal parameters as described above. The function can be applied with the application forms. Arrays are partitioned into carriers, base arguments paired, and the function applied in the standard manner. Obviously no base results are produced. If defined with unbounded rank, the header of the definition is identical to the header in current APL, and the function may be applied with datum rank.

3.4.1.4 Niladic functions

Niladic functions are defined and applied as in current APL. They cannot be applied with the application forms or with datum rank.

3.4.2 Evaluating defined functions

The evaluation of a defined function is similar to current APL. The formal parameters are bound to arguments; a new environment is created; and the body of the function is evaluated in the new environment. The scoping of names is dynamic as in current APL.

Applying a defined function with a non-zero datum rank introduces an array (or arrays) of non-scalar items into the new environment. Let *FUN* be a monadic defined function with formal parameter X. If we apply *FUN* with datum rank K to an array A, i.e., *FUN*$\{K\}A$, the formal X is bound to an array of rank K items. This is true if *FUN* has bounded or unbounded rank. We call K the ***implicit datum rank*** of X. Note the implicit datum rank is associated with the array value and not the application form. The binding exists only during the evaluation of *FUN*, but the value is global to the evaluation of any form within the function body. Through side effects, the partitioned value may be returned to the global environment.

Implicit datum rank adds a partition to a ragged array. If an array A has implicit datum rank K, then A is an array of rank K items. The ragged arrays defined in section 3.1.1 are in effect arrays of rank 0 items, and they have implicit datum rank 0. Thus every array has an implicit datum rank. When we say we remove the partition from an array with implicit datum rank K, we actually map the array to an array with implicit datum rank 0.

We define how to apply a function to an array with implicit datum rank. Recall that the primitive functions map the rank of the items in an array in a very uniform manner; arrays of rank N items are mapped either to another array of rank N items, or to a simple array. Let A be an array with implicit datum rank J, and let B be the value of A

with the partition removed. Consider the application of a primitive function F to A with datum rank K; the form is $F\{K\}A$. To evaluate $F\{K\}A$ we first evaluate $F\{K+J\}B$. The evaluation of this second form is well-defined, for B is a simple array. Let C be the result of the evaluation. Then, if F preserves datum rank, we associate the implicit datum rank J with C; otherwise, if F maps a rank K item to as simple scalar, the implicit datum rank of C is zero. Note the application results in an error if $K+J$ is not a permissible datum rank for F. In the application of a dyadic function, the implicit datum rank of both argument arrays must be the same, unless one argument to the function is restricted to an array of simple items.

The definition of the application of primitives with implicit datum rank extends immediately to defined functions. Defined functions are constructed from primitive functions; thus they also map an array of rank N items to another array of rank N items, or to a simple array. Note we cannot construct an array of arbitrarily nested items by "nesting" implicit datum ranks. Each time a function is applied, the implicit datum rank is removed from the argument array and added to the datum rank of the application form. Thus, if we call one defined function from the body of another, the size of the implicit datum ranks of the values in the local environments may grow, but they never form a nested, multi-level partition.

3.4.3 Examples

3.4.3.1 Remove duplicates

Figure 3-25 presents the remove duplicates idiom as a defined function of bounded rank; it is declared to have base rank 1 1 and datum rank N N.

Let M be a character matrix representing a word list. Consider the form

```
    ∇ R:1:N ← REMDUP V:1:N
[1] R ← ((V⍳V)=⍳⍴V)/V
    ∇
```

Figure 3-25: Remove duplicates as a defined function of bounded rank.

REMDUP{1}*M*, where *REMDUP* is applied with datum rank 1. To evaluate the expression, we first partition *M* into a vector of rank 1 items and bind the formal parameter *V* to this partitioned value. *V* has implicit datum rank 1. In the body of *REMDUP*, every application of a function to *V* is equivalent to the application of the function to *M* with datum rank 1. Note that dyadic iota (⍳) and shape (⍴) both map rank *K* items to scalars. Thus the results of these functions have implicit datum rank 0, and functions applied to the results are effectively applied with datum rank 0. Thus we see ((*V*⍳*V*)=⍳⍴*V*)/*V* in the local environment is equivalent to ((*M*⍳{1}*M*)=⍳⍴{1}*M*)/{1}*M* in the global environment. This second expression is the extension of the remove duplicates idiom to non-scalar data presented in chapter 2 (see section 2.3.4).

3.4.3.2 Row sort

Defined functions of bounded rank are useful in extending idioms which otherwise cannot extend to higher rank arrays. For example, we sort a vector *V* with the form *V*[⍋*V*]. This idiom does not extend to collections of vectors because of the syntactic restrictions of the indexing form. However, we can write a defined function which can independently sort the rows of an arbitrary rank array. The definition of *ROWSORT* is given in figure 3-26; the function has base rank 1 1 and datum rank *N N*. The scalar product *ROWSORT A* independently sorts the rows of an array *A*.

```
    ∇ R:1:N ← ROWSORT V:1:N
[1] R ← V[⍋V]
    ∇
```

Figure 3-26: Row sort as a defined function of bounded rank.

Chapter 4

ANNOTATED EXAMPLES

This chapter presents some examples of programming with carrier arrays. In the first section, we examine the key-word-in-context problem and compare a carrier APL solution with solutions in current APL and nested arrays. In the remainder of the chapter, we consider a variety of programming problems and present their solutions in carrier APL.

4.1 Key word in context.

The key-word-in-context problem has often been used to portray programming techniques and style [61, 35, 63, 57, 58]. The problem is: Given a list of book titles, produce a sorted list of titles keyed by each word in the title; a title appears once for each word in it. KWIC is a non-trivial problem; Parnas estimates a KWIC system can be coded by a good programmer in a week or two [61]. A simple solution requires several pages of PASCAL.

KWIC is a natural problem for APL. An algorithm for a solution is:

1. Determine the sort order of the words in the titles.

2. Copy and rearrange the titles into sort order.
3. Highlight the key word for each entry.

The steps in the algorithm correspond directly to primitives in APL: gradeup (⍋) for sorting, indexing for rearranging, and rotate (⌽) for highlighting. A four line solution can be written in current APL if we assume the titles can be represented as a simple matrix: each word in a title is a single character, and all titles have a common length. However, the program becomes significantly more complicated when we extend it to realistic data. In carrier APL, a four line solution generalizes trivially from a simple matrix to collections of titles embedded in a rank 3 array. In this section we present a KWIC system written in carrier APL and contrast it with solutions written in current APL and nested arrays.

4.1.1 Carrier APL

4.1.1.1 Simplified KWIC

First consider a simplified version of the KWIC problem. We represent a list of titles in a ragged matrix A: each title is a row of the matrix; each word in the title is a single character. Figure 4-1 presents a solution and a sample evaluation. In the first line we calculate S the sort order of the "words"; recall that gradeup can be applied directly to a character vector. Line two contains a standard APL idiom for classifying the numbers in S into intervals of length N; it in effect reports the title containing the word corresponding to each element in S. Line three calculates the offsets R needed to rotate each word to the front of its title. We now have all the information needed for a KWIC index; in line 4, we apply this information with index and rotation to form the result K.

```
S ← ⍋,A
I ← 1++/S∘.>+\N ← ρA
R ← ,¯1+ιN
K ← R[S]⌽A[I],'|'

A          ←→          SP
                       TAIL
                       APL

S          ←→          4 7 5 6 9 2 8 1 3

N          ←→          2 4 3

I          ←→          2 3 2 2 3 1 3 1 2

R          ←→          0 1 0 1 2 3 0 1 2

K          ←→          AIL|T
                       APL|
                       IL|TA
                       L|TAI
                       L|AP
                       P|S
                       PL|A
                       SP|
                       TAIL|
```

Figure 4-1: Simplified KWIC in carrier APL.

4.1.1.2 Realistic KWIC

We now extend the solution to a more realistic data structure. Titles are represented in a three dimensional ragged array: each plane is a title; each row is a word. Figure 4-2 contains a solution and the resulting index. Note the solution is essentially identical to the simplified KWIC solution. We insert datum ranks to indicate we are now representing words as character vectors, not single characters. We also add a flatten ($\ddot{,}$) in line four to format the index for output.

```
S ← ⍋{1},{1}A
I ← 1++/S∘.>+\N ← ρ{1}A
R ← ,¯1+⍳N
K ← ;̈{2}(R[S]⌽{1}A[I],{1}'|'),' '
```

```
A     ←→     Structured      The      A
             Programming     APL      Programming
                             Idiom    Language
                             List

K     ←→     A Programming Language |
             APL Idiom List | The
             Idiom List | The APL
             Language | A Programming
             List | The APL Idiom
             Programming | Structured
             Programming Language | A
             Structured Programming |
             The APL Idiom List |
```

Figure 4-2: Realistic KWIC in carrier APL.

4.1.2 Current APL

4.1.2.1 Simplified KWIC

A similar KWIC program can be written in current APL if we represent a list of titles as a rectangular matrix: each title is a row of the matrix; each word in the title is a single character; all titles have a common length. Figure 4-3 presents the program. There are a few differences from the carrier solution: we must index the characters from *A* into an alphabet *ALPHA* before sorting, and the expressions to calculate *N* and *R* are slightly more complicated. However, the carrier and current APL solutions are nearly identical.

```
S ← ⍋ALPHA⍳,A
I ← 1++/S∘.>+\N ← (ρA)[1]ρ(ρA)[2]
R ← ,¯1+(ρA)ρ⍳(ρA)[2]
K ← R[S]⌽A[I;],'|'
```

Figure 4-3: Simplified KWIC in current APL.

4.1.2.2 Realistic KWIC

The current APL solution does not extend simply to a more realistic data structure. We now represent titles in a three dimensional array: each plane is a title; each row is a word. The planes and rows are blank padded as necessary, for the words and titles may have varying length. Figure 4-4 contains a solution.

```
M ← A∨.≠' '
S ← ⍋(,M)/,⍉(ρALPHA)⊥⍉¯1+ALPHA⍳A
I ← 1++/S∘.>+\N ← +/M
R ← (,M)/[1]¯1+((×/(ρA)[1 2]),(ρA)[3])ρ1 3 2⍉(ρA)[1 3 2]ρ⍳(ρA)[2]
X ← R[S;]⌽[2](A,[2](ρA)[1 3]ρ(ρA)[3]↑'|')[I;;]
Y ← ((ρX)[1],×/(ρX)[2 3]+ 0 1)ρX,' '
C ← T∨1⌽T ← Y≠' '
H ← ⌈/+/C
K ← ((ρY)[1],H)ρ(,(+/C)∘.≥⍳H)\(,C)/,Y
```

Figure 4-4: Realistic KWIC in current APL.

The solution is more than twice as long as the previous program, and is an order of magnitude more complicated. We first calculate *M*, a mask to identify the blank rows of *A*, and we use it as a filter throughout the program. *S* is again the sort order of the words in *A*, but we now represent each word as a number in base *ρALPHA* before applying grade up. This sorting idiom loses precision for long words; a longer, iterative method should be used. The expression to calculate the rotators *R* is significantly more complicated, for we can not rotate each word as a unit in current APL; rather, we must rotate each column of characters. The KWIC index *X* is calculated as before, but we add a four line idiom to format it for output.

4.1.3 Nested Arrays

The nested array system is a superset of current APL, and thus the current APL solutions are also nested array solutions. However, a collection of book titles can be represented as nested array without any blank padding: each title is a vector of enclosed character vectors, and the titles are in turn enclosed and assembled in a vector. For example:

```
((Structured) (Programming)) ((The) (APL) (Idiom) (List)) ((A) (Programming) (Language))
```

The KWIC solution for this data structure is not a simple generalization of the four line APL solution. Figure 4-5 presents the program. The first two lines convert the nested representation to a rectangular matrix of words for sorting. We sort the matrix of words as in current APL, converting each word to a base *ρALPHA* number. The calculation of *I*, *R*, and the KWIC index *X* are similar to the simplified APL solution. Two lines are then required to format the nested KWIC index for output.

```
A1 ← ⊃,/A
A2 ← ↑(⌈/ρ¨A1)↑¨A1
S ← ⍋,⍉(ρALPHA)⊥⍉¯1+ALPHA⍳A2
I ← 1++/S∘.>+\N ← ⊃,/ρ¨A
R ← ¯1+⊃,/⍳¨N
X ← R[S]⌽¨A[I],¨'|'
T ← ⊃¨,/¨X,¨¨' '
K ← ↑(⌈/ρ¨T)↑¨T
```

Figure 4-5: Realistic KWIC with nested arrays.

4.1.4 Analysis

Only the carrier array solution generalizes from simple to realistic data. The concept of datum rank permits a function to extend directly from array of scalars to arrays of non-scalar data. In current APL, the primitives apply naturally to a simple matrix. But when this matrix becomes a three-dimensional blank padded array, an otherwise simple algorithm must be contorted to apply the primitives to the new data structure. And in the nested system, when a nested data structure is introduced, the program must be rewritten with the new representation in mind.

Carrier arrays contain the match of data structure and primitives required for the key word problem; the current and nested APL do not. Current APL does not have an adequate representation for an array of non-scalar data, and thus the primitives cannot be simply applied. Book titles can be represented naturally in a nested array, but we must switch between nested and flat representations of data, depending on the operation we perform. A nested array is converted to a character matrix to be sorted; the nested representation is used for rotating and rearranging the titles; and the nested KWIC index is mapped to a flat character matrix for output.

4.2 Other Examples.

In this section we present more examples of carrier APL programs.

4.2.1 Tokenizing Strings.

A common APL idiom (called *MAKEARRAY* in the APL Idiom List [63]) breaks a character vector of strings separated by delimiters into a matrix, one string per row. The idiom is coded in carrier APL in figure 4-6. We assume there are no adjacent delimiters in the input vector S (unless there are null strings in S). I is the index of each delimiter in S, and L is the length of each string. L is also the additive shape of the desired result, and can be used to reshape S with the delimiters removed, producing the result R.

This program generalizes without changes to tokenize independently the rows of a higher rank array. With the insertion of appropriate datum rank, it will also tokenize

```
I ← (' '=S,' ')/ι1+ρS
L ← ¯1+I−0,¯1↓I
R ← Lρ(S≠' ')/S
```

Figure 4-6: MAKEARRAY idiom

arrays of non-scalar data.

4.2.2 Symbol Table Update.

The symbol table update problem is presented in the APL Idiom List [63]. Two structures, *A* and *B*, comprise the symbol table. *A* is a list of symbols; *B* is an associated usage count list. *X* is a new symbol to be added to the table. If *X* is in *A*, the appropriate element of *B* should be incremented. If *X* is not in *A*, it should be appended to *A*, and a count of one appended to *B*.

An elegant solution can be written in current APL if we assume each symbol can be represented as a single character. *A* and *B* are then vectors, and *X* is a scalar. Figure 4-7 presents the program. This solution extends immediately to add several symbols to the table in parallel. *X* is now a vector of symbols; we assume it contains no duplicates.

```
A ← A,(M←~X∈A)/X
B ← (B,M/0)+A∈X
```

Figure 4-7: Symbol Table Update for scalar symbols

In this example, the data structures *A*, *B*, *X*, and *M* are vectors. Membership, catenate, and compression are evaluated in the restricted domain of their base definition in carrier APL. Thus the evaluation of this program using the semantics of carrier APL is identical to its evaluation in current APL. However, by appropriately inserting datum rank, we can extend the carrier APL program to non-scalar symbols. Figure 4-8 contains the extended solution.

All the functions in this program are applied using scalar product; thus this program

```
A ← A,{1}(M←~X∈{1}A)/{1}X
B ← (B,M/0)+A∈{1}X
```

Figure 4-8: Symbol Table Update for non-scalar symbols

will also update multiple symbol tables in parallel.

4.2.3 Triangular Forms.

Ragged arrays and the carrier APL primitives make it easy to create and manipulate triangular arrays. For example:

- (⍳*N*)⍴⍳+/⍳*N* creates an *N*-row triangle of successive integers.:

```
    (⍳5)⍴⍳+/⍳5          ←→          1
                                    2  3
                                    4  5  6
                                    7  8  9 10
                                    11 12 13 14 15
```

- ⍳⍳*N* creates the first *N* sequences of positive integers.

```
    ⍳⍳5                 ←→          1
                                    1  2
                                    1  2  3
                                    1  2  3  4
                                    1  2  3  4  5
```

- ⌈\(⍳*N*)↑¨⍳*N* creates a triangle with *I* occurrences of *I* in the ith row.

```
    ⌈\(⍳5)↑¨⍳5          ←→          1
                                    2  2
                                    3  3  3
                                    4  4  4  4
                                    5  5  5  5  5
```

We can combine the last two expressions with the binomial coefficient function (!) to calculate Pascal's triangle.

```
(¯1+⍳⍳5) ! ¯1+⌈\(⍳5)↑¨⍳5    ←→          1
                                        1  1
                                        1  2  1
                                        1  3  3  1
                                        1  4  6  4  1
```

The APL Idiom List [63] presents a program which finds integers whose square is a palindrome (i.e., the decimal expansion reads the same backward and forward.) Figure 4-9 contains the solution in current APL. *Y* contains the decimal expansion of the first *N*∗2 integers, one per row. The expansions of the smaller numbers are padded with leading zeros, and the second line of the program is largely devoted to rearranging the padding.

```
Y ← ⍉((1+⌊10⍟N∗2)⍴10)⊤(⍳N)∗2
T ← Y=(−+/∧\Y=0)⌽⌽Y
Z ← (∧/T)/⍳N
```

Figure 4-9: Palindrome in current APL

In carrier APL, we can store the decimal expansions in a triangular form without padding. The test for palindromes in line 2 is now straightforward: does an expansion equal its reversal? Figure 4-10 presents the simplified carrier solution. Note we no longer need to transpose the result of the encode function.

```
Y ← ((1+⌊10⍟M)⍴10)⊤M ← (⍳N)∗2
Z ← (Y={1}⌽Y)/⍳N
```

Figure 4-10: Palindrome in carrier APL

4.2.4 Replicate.

Replicate is an extension of compression which has been implemented in several APL systems [8, 17, 65]. The left argument to replicate (/) is a vector of non-negative integers which indicate how many copies of the corresponding elements in the right vector appear in the result. For example:

```
(⍳3)/'ABC'          ←→          'ABBCCC'
```

Replicate is a one-line idiom in carrier APL; figure 4-11 contains the code. *V* is a vector of data; *M* is the numeric guide to replicating the elements of *V*. We first use reshape and transpose to create a matrix of the elements of *V*; row *I* contains

⌈/*M* copies of *V*[*I*]. We then apply take to select the desired number of elements from each row, and ravel the result.

```
,M↑⍉(⌈/M)ρ{1}V
```

Figure 4-11: REPLICATE idiom

In carrier APL, defined functions can be applied with all the application forms, in a manner similar to the primitives. Thus we can define a *REPLICATE* function with base rank 1 1 1 and datum rank 0 N N and use it in carrier APL programs as an extension to compression. Figure 4-12 contains the definition.

```
   ∇ R:1:N ← M:1:0 REPLICATE V:1:N
 [ 1 ]  R ← ,M↑⍉(⌈/M)ρ{1}V
   ∇
```

Figure 4-12: *REPLICATE* function

David Rabenhorst of IBM [66] suggested that the replicate function would simplify the KWIC index programs discussed in section 4.1. Figure 4-13 presents a carrier solution using the *REPLICATE* function. The program is essentially the same as before (see figure 4-2), but the second line is simpler. Moreover, the expressions for *S*, *D*, and *R* can be substituted into the fourth line of the program, producing a (relatively) readable one-liner. Similar simplifications are found when using replicate in the current and nested APL solutions.

```
S ← ⍋{1},{1}A
D ← (ρ{1}A) REPLICATE{2} A
R ← ,¯1+ιρ{1}A
K ← ⍪̈{2}(R⌽{1}D,{1}'|')[S],' '
```

Figure 4-13: Realistic KWIC in carrier APL with *REPLICATE*

4.2.5 Rational Numbers

In his evaluation of nested arrays, Orth [60, 59] uses rational numbers to examine the capabilities of the nested array system to support a limited form of data abstraction. He represents rational numbers in a nested array, and defines rational plus and times functions. He then examines if an APL arithmetic expression can be transformed into an equivalent rational arithmetic expression by the appropriate substitution of function names. In the nested array system, Orth shows this is not possible; a defined rational plus function cannot be equivalent to the primitive plus (+) in both a scalar product application and a reduction. With carrier arrays, we can define such a rational plus function.

In carrier APL, a rational number can be represented as a vector of two integers, a numerator and a denominator. Figure 4-14 presents the defined functions *PLUS* and *TIMES* for adding and multiplying rational numbers. Each function maps a pair of rational numbers to a new rational, which in our representation is a mapping from a pair of vectors to a vector. Thus both functions have base rank 1 1 1. The datum rank vector is 0 0 0, for the components of the representation of a rational number are scalars.

```
    ∇R:1:0 ← A:1:0 PLUS B:1:0
[1]  R←(A+.×⌽B),(¯1↑A)×¯1↑B
    ∇

    ∇R:1:0 ← A:1:0 TIMES B:1:0
[1]  R←A×B
    ∇
```

Figure 4-14: Functions for Rational Arithmetic

Rational numbers can be collected in an array by representing them as length-two vectors along the last dimension. When *PLUS* or *TIMES* is applied to arrays *X* and *Y* of rational numbers (e.g., *X PLUS Y*), *X* and *Y* are partitioned into collections of vectors, and the function is applied independently to each pair of corresponding vectors. This parallels the application of the primitive plus (+) function to two arrays of scalars: Plus is applied independently to pairs of corresponding scalars. If *PLUS* is applied with an operator such as reduction or outer product, the partitioning semantics again parallels the application of the primitive plus function to an array of scalars with the same operator.

Thus *PLUS* and *TIMES* can be directly substituted for the primitives + and × in every application context when transforming an arithmetic expression to rational arithmetic (assuming there are no name conflicts.) In the figure 4-15, *A* and *B* are arrays of integers; *AR* and *BR* are corresponding arrays representing rational numbers.

Current APL	Carrier Arrays
A+*B*	*AR PLUS BR*
A++/*B*	*AR PLUS PLUS*/*BR*
A+.×*B*	*AR PLUS*.*TIMES BR*

Figure 4-15: Integer and Rational Arithmetic

However, *PLUS* is not equivalent to the primitive plus when applied to an empty array with any of the application forms. For example, the reduction of an empty array is defined to be either the identity element of the applied function or an error if no identity exists (see section 3.2.4). There is no means of specifying the identity element of a defined function in carrier APL.[12] Thus the *PLUS* reduction of an empty array is an error, whereas the plus (+) reduction is zero.

More importantly, with or without the nested or carrier extension, APL is not a good language for data abstraction, for the programmer wants to use the primitives of the

[12]APL2 [65] addresses this problem; in APL2, the system label ⎕*ID* provides a means of specifying an identity element for a defined function.

language to manipulate the representation of the data object directly. But the primitives have no knowledge of the representation, and their application can have unfortunate consequences. For example, the fill element in the overtake of a vector of integers (¯1↑(1+ρ*IV*)↑*IV*) is 0, the identity of the primitive plus function. However, the overtake of a vector of rationals (¯1↑{1}(1+ρ{1}*RV*)↑{1}*RV*) is 1ρ0, which is not a well-defined rational.

4.2.6 A Phone List

Carrier APL is well suited for small database applications. A phone list is a small database. Our sample phone list is a collection of records, each containing a person's name, his phone number, and his office number. In this section we present a set of carrier APL functions which create and manipulate the sample list.

4.2.6.1 Parsing the data

Assume the data for the phone list is stored as a stream of text, a long character string with embedded new lines. Each line is a record; the fields within each record are separated by colons. We can use the *MAKEARRAY* idiom defined in section 4.2.1 to parse the data into a three dimensional array: one record per plane, one field per row.

Figure 4-16 defines *MAKEARRAY* as a function, and in figure 4-17 we parse the data for a sample phone list. *STREAM* contains the raw data. We first use *MAKEARRAY* to break *STREAM* at each newline character (<nl>), resulting in the matrix *RECORDS* which contains one record per row. We then apply *MAKEARRAY* in parallel to each row of *RECORDS*, resulting in the phone list *PL*. (The planes of *PL* are displayed adjacently for convenience.)

```
  ∇ R:2:N ← D:0:N MAKEARRAY S:1:N;I;L
[1]  I ← (D=S,D)/⍳1+⍴S
[2]  L ← ¯1+I−0,¯1↓I
[3]  R ← L⍴(S≠D)/S
  ∇
```

Figure 4-16: *MAKEARRAY* function

```
STREAM
                  Steve Wood:436-8160:501F<nl>Bob Nix:436-1834:420<nl>John
                  Ellis:436-8160:421<nl>Nat Mishkin:787-2660:501F<nl>

RECORDS ← '<nl>' MAKEARRAY ¯1↓STREAM
                  Steve Wood:436-8160:501F
                  Bob Nix:436-1834:420
                  John Ellis:436-8160:421
                  Nat Mishkin:787-2660:501F

PL ← ':' MAKEARRAY RECORDS
                  Steve Wood     Bob Nix       John Ellis     Nat Mishkin
                  436-8160       436-1834      436-8160       787-2660
                  501F           420           421            501F
```

Figure 4-17: Parsing Phone List

4.2.6.2 Sorting by Last Name

Figure 4-18 presents the carrier APL program to sort the phone list *PL* by last name. We first collect the last names in *LN*, noting that the last name in the name field is every character after the first blank. We then grade *LN* to get the sort order and rearrange *PL*.

```
LN ← 1↓(∨\' '=PL[;1])/PL[;1]
SPL ← PL[⍋{1}LN]
```

Figure 4-18: Sorting by Last Name

4.2.6.3 Searching by Last Name

Figure 4-19 contains programs to search the phone list for records which match the last name represented by the character vector *NAME*. We first collect the last names in *LN* as before, and then use dyadic iota (⍳) to get the first match or membership (∈) to get them all. In both cases, an empty array is returned if there are no matches. Note both programs will search for multiple names in parallel; *NAME* must be a matrix representing a collection of names, one per row.

```
I     ← LN⍳{1}NAME                M     ← LN∊{1}NAME
FIRST ← PL[(I≤ρ{2}PL)/I]          ALL   ← M/{2}PL
```

Figure 4-19: Searching by Last Name

4.2.6.4 Cross Referencing by Office

Figure 4-20 contains two programs to cross reference the phone list by office. In the first program, we first obtain the list *OL* of offices in the phone list with duplicates removed. We then use outer product to compare each distinct office with the office fields in the phone list, and select the records corresponding to each office from *PL*. The result *CR*1 is a four dimensional array; each hyperplane *CR*1[*I*] contains all the records of individuals in office *OL*[*I*].

The second program is more efficient. We first sort the phone list *PL* by office, and determine the indices *I* where the offices change. We use *I* to determine the length *L* of each sequence of records with the same office, and use *L* to reshape the phone list. The cross reference *CR*2 is equivalent to *CR*1, except that *CR*2 is sorted by office along its first axis.

```
OL  ← ((T⍳{1}T)=⍳ρ{1}T)/{1}T ← PL[;3]
CR1 ← (OL∘.={1}PL[;3])/{2}PL


SPL ← PL[⍋{1}PL[;3]]
I   ← (((1↓{1}SPL[;3])≠{1}¯1↓{1}SPL[;3]),1)/⍳ρ{2}SPL
L   ← I−0,¯1↓I
CR2 ← Lρ{2}SPL
```

Figure 4-20: Cross Referencing by Office.

4.2.6.5 Formatting for Output

The phone list *PL* is a three-dimensional array, each plane representing a record. For output, we pad each field to the maximum width of all the values of the field, and ravel the planes. Figure 4-21 presents the program and the result of formatting *PL*.

```
ML  ← ⌈/⍉ρPL
O   ← ,̈{2}(ML+1)∘.2 1 2↑PL
```

```
Steve Wood  436-8160 501F
Bob Nix     436-1834 420
John Ellis  436-8160 421
Nat Mishkin 787-2660 501F
```

Figure 4-21: Formatting for Output.

Chapter 5

IMPLEMENTATION ISSUES

Much of an implementation for carrier APL is similar to an implementation of current APL. The structure of the systems is the same. The tokenizer and syntax analyzer are essentially identical. The base definitions of the carrier APL functions are closely related to the primitive functions of current APL.

A carrier APL implementation differs from a current APL system in the application and evaluation of functions and in the representation of arrays in memory. The application routines perform carrier partitioning, a concept found only in the scalar functions of current APL, and they apply functions to arrays of possibly non-scalar items. Arrays are stored as *edge vectors* [3], vectors of pointers pointing to vectors of data. The contiguous, ravel order representation of arrays used in current APL systems is not appropriate for ragged arrays.

In this chapter, we examine the issues in implementing carrier APL. We first present the edge vector representation and examine why it is appropriate for carrier arrays. We then contrast the designs of interpreters for current and carrier APL, and sketch the routines which apply and evaluate functions in a carrier APL interpreter. We end the chapter by considering some issues in compiling carrier APL.

5.1 Edge Vector Representation

We defined a ragged array as a uniform-depth tree (see section 3.1.1). The edge vector representation is a natural realization of this definition. An edge vector is a vector of pointers, or *edges* of a graph. In our definition of an array as a tree, each dimension in the array corresponds to a level in the tree. In our representation of an array, each dimension corresponds to a collection of vectors in storage. Each row of a ragged array is stored as a vector of data; each plane is a vector of pointers to vectors of data; each hyperplane is a vector of pointers to the vectors representing planes; and so forth. Figure 5-1 presents the edge vector representation of a three-dimensional array.

```
          /----->|||----->1
         /       |
|||--/           |||----->2 3
|                |
|||--\           |||----->4 5 6
        \
         \----->|||----->7 8 9 10
                 |
                 |||----->11 12 13 14 15
```

Figure 5-1: Edge Vector Representation of 3 2ρ{1}(ι5)ρι+/ι5

Edge vectors are stored in a heap; the basic unit of storage is a homogeneous vector. Each vector has a type and length associated with it. The types are pointer, character, or one of the three numeric types: boolean, integer, and floating point. The vectors are not extensible; two vectors are catenated by creating a new vector of appropriate type and length. However, an edge vector can be a component of more than one array; this sharing gives a substantial saving in storage.

We evaluate the suitability of the edge vector representation for carrier arrays using the following criteria:

- Capabilities for representing a heterogeneous ragged array.
- Capabilities for partitioning a ragged array into a three-level carrier.
- Efficiency of access to an individual element or base argument.
- Efficiency of traversal.
- Storage utilization.
- Complexity of storage management.

These categories capture the special requirements of carrier APL as well as general desiderata for an array representation.

5.1.1 Representing a heterogeneous ragged array.

Each vector in the representation of an array is stored independently with an associated type and length. Thus the representation is well suited to represent ragged and heterogeneous collections of vectors. Note that a row of an array can be stored as a vector of data only if the row is homogeneous. If the row contains both character and numeric data, another edge vector must be introduced; the row is represented as a vector of pointers to single data elements. Throughout the remainder of this section, we assume each row of an array is homogeneous, though different rows may have different types.

5.1.2 Partitioning a ragged array into a three-level carrier.

A ragged array can be implicitly partitioned into a three level carrier by storing the ranks of the three levels; the actual partitioning is done while accessing the elements of the array. Let A be a rank N array, and let the ranks of the three levels be I J K, where $N \longleftarrow I+J+K$. Recall the three levels are carrier, base, and datum. To access the base arguments, we traverse the top I levels of the tree structure of edge vectors. If the base arguments are not scalars (i.e., $(J+K)>0$), the elements of the vectors on the ith level are pointers to base arguments; if the base arguments are scalars (i.e., $(J+K)=0$), the elements of the vector are the base arguments. To access the elements on the datum level, we similarly traverse the top $I+J$ levels of the tree structure, and to access the components of each datum, we traverse all N levels of the tree.

5.1.3 Efficiency of access to an individual element or base argument.

An arbitrary element or base argument can be accessed very efficiently with the edge vector scheme; we simply follow a chain of pointers. For simplicity, we use zero origin indexing in the following discussion, and assume that the size of each vector element is the same as the smallest addressable unit of storage. If AC is an accumulator containing the address of an edge vector in the Kth dimension of an array, then $CONTENTS(AC+I)$ is the address of the Ith component of $K+1$ dimension. Thus to access an element $A[I1;I2;I3]$ in a rank 3 array, we must execute $AC \leftarrow AC+I$ three times, each time setting I to the appropriate index. If we unroll the loop, this requires six instructions on a PDP-10: three moves and three adds. By comparison, the equivalent index calculation on a rectangular array stored in ravel order requires three multiplications and three additions.

5.1.4 Efficiency of traversal.

Ravel order traversal of the elements in an edge vector representation of a ragged array is easy and efficient. However, traversal along any other axis is expensive. Traversal along the first dimension requires following a chain of pointers from root to leaf for every element access. Similarly traversal along the Kth dimension of a rank N array requires following a chain of at least $N-K$ pointers for each access.

Traversal along an arbitrary dimension is not as important in carrier APL as it is in current APL. The axis operator is replaced with datum rank in the application of the primitive functions. Rather than access elements along the Kth dimension of a rank N array, in carrier APL we partition the array into a collection of rank $N-K$ data items, and manipulate the Kth dimension as a collection of vectors of these non-scalar items. Accessing successive vectors of non-scalar data in ravel order is as simple as traversing the elements of the array.

5.1.5 Storage utilization.

The edge vector representation requires some overhead for pointers. However, it is easy to show that if every vector on every level of an array contains at least two elements, the number of pointers in the representation is less than the number of data items, and that as the size of the edge vectors increase, the ratio of pointers to data items decreases. Thus if we assume a pointer takes the same amount of space as a data item, and ignore the overhead of type and length fields for the moment, we see storage utilization in the representation of an array is always greater than 50 percent, and often much greater. Moreover, the edge vector permits the implicit sharing of subarrays between arrays. The pointer overhead in representing a single array may be recouped by the sharing of portions of the representation with other arrays.

First consider the relationship of the edge vector representation to the definition of a ragged array as a tree. Pointers correspond to edges in the tree. Note, however, that in the representation we have collapsed the lowest level of the tree by collecting the leaves into vectors of data.[13] Thus pointers in the representation are in one-to-one

[13]This is only true if each row of the array is homogeneous. See section 5.1.1.

correspondence with edges into internal nodes in the tree.

Now consider a ragged array containing d data elements; the corresponding tree has i internal nodes and d leaves. Assume each node in the tree has at least k branches, where $k \geq 2$. Following a proof by Knuth [36], we note there are i+d-1 edges, for there are no cycles. Also we know there are at least ki edges, for each internal node has at least k branches. Thus we have

$$ki \leq i+d-1$$

which gives

$$(k-1)i+1 \leq d$$

In the edge vector representation of this array, we know there are i-1 pointers and d data elements (assuming the rows of the array are homogeneous). Thus we have

$$\frac{\text{pointers}}{\text{data}} = \frac{i-1}{d} \leq \frac{i-1}{(k-1)i+1} \leq \frac{1}{k-1}$$

If all nodes on each level in the array have two or more branches, the ratio of pointers to data is no greater than 1, and as the branching factor increases, the ratio decreases. If all nodes have 10 or more branches, the ratio of pointers to data is 1 out of 10. In the representation of the triangular array shown in figure 5-1, we have 7 pointers and 15 data elements.

To estimate more accurately the storage utilization we must estimate the storage requirements of a pointer, the type and length fields associated with a vector, and the data in the vector itself. A pointer on a PDP-10 is an 18 bit halfword. The type and length can be encoded in another 18 bit halfword, if no vector is longer than 2^{15} (roughly 32,000) elements. (Alternatively the type could be implicit in the address of the data, as in some PDP-10 LISP implementations [71], and the length could be a full halfword.)

Thus the address, length, and type of a vector can be represented in a full word, which is the "pointer" in the above analysis. (Alternatively, the length and type can be stored with the vector of data, and the pointer in a separate halfword; this arrangement may cause alignment problems for the data in the vector.) Integers and floating point numbers take a full word in a PDP-10. Thus for numeric arrays, storage utilization is 50 percent or more. Character data can be packed five to a word, and boolean 36 to a word. For character and boolean data, the edge vector overhead is considerably higher.

The edge vector representation permits the sharing of data between different arrays; this sharing often offsets the extra storage required for pointers. For example, to laminate two matrices into a rank 3 array does not require the copying of the two matrices, as it does in current APL. Only one new edge vector, with two pointers, is created; one pointer points to the first matrix, the other to the second.

5.1.6 Complexity of storage management.

The internal pointers in the heap introduced by the edge vector representation significantly complicate storage management when compared to the techniques used in current APL systems. However, efficient garbage compactification algorithms have been developed whose running times are linear in the size of the heap.

To reclaim storage, active edge vectors must be identified, arbitrary length vectors of pointers must be relocated, and their pointer values updated. This problem is addressed by Wegbreit in the design of a compactifying garbage collector for ECL [72, 19]. In his algorithm, contiguous blocks of active storage are marked, and a forwarding address of each block is calculated. Pointers can then be updated, and the blocks moved. Pointer updating in Wegbreit's algorithm is worse than linear in the sum of the number of pointers and the size of the heap.

Morris has developed an improved algorithm which is linear in the size of the heap [55, 56, 76, 23]. Morris makes two sweeps of the marked heap. Moving from low addresses to high, he chains together all "upward-pointing" references to each cell. When a cell is reached, its new location is known, and all upward references are updated to the new location. Moving from high addresses to low, all "downward-pointing" references are chained. When a cell is reached, it can be moved to its new location (we are compacting to the high end of memory), and all downward references updated.

Martin has recently published a new algorithm which is more efficient than Morris's, though it remains linear in the size of the heap [40].

In the discussion above, we do not consider reference counting schemes or the "copying" garbage collectors which are suitable for virtual memory machines. Cohen provides a general survey and a comprehensive bibliography of garbage collection techniques [18].

5.1.7 Alternative representations.

Several alternative representations of ragged arrays are possible, but none of them fit the requirements of carrier APL. Two representations are suggested by current APL programming techniques for manipulating ragged structures. A ragged array can be padded with fill characters and stored in ravel order as a rectangular array. In this representation, the overhead of blank padding is significant, and it offers none of the flexibility or capabilities for sharing that accompany the edge vector representation. Moreover, heterogeneous arrays can not be represented. Alternatively, the data from the array can be stored contiguously in ravel order; the structure is stored separately as a collection of vector lengths or boolean masks. With this type of representation, accessing an arbitrary element is difficult. And again, heterogeneous arrays cannot be represented. Rosenberg considers storing ragged arrays by hashing [67]. This method is inappropriate for carrier APL, for ravel order traversal of the elements is expensive and storage utilization is low.

5.2 Interpretation

A prototype carrier APL interpreter has been implemented in APLSF [5] on the DECsystem20; it is a large, 7000 line APL program. All carrier array programs in this thesis have been run on the system. The interpreter supports the full language with a few exceptions.[14] The system contains a memory model and a storage management system including a compactifying garbage collector. The edge vector representation is implemented in this memory model.

In this section, we describe an interpreter for carrier APL, using the prototype carrier APL interpreter as a guide. We first sketch an interpreter for current APL, and show that an interpreter for carrier APL has the same structure. We then describe routines to apply and evaluate functions in carrier APL. The implementations of these routines are straightforward, and often closely parallel the definition of application forms and primitive functions given in chapter 3 (see sections 3.2 and 3.3). Thus we do not go into great detail.

[14]Indexed assignment and branching are not implemented, and the definitions of monadic transpose, dyadic transpose, and indexing have been revised since the interpreter was completed. Defined functions of unbounded rank, defined functions with no explicit result, and niladic functions are not implemented.

5.2.1 Current APL

The design of an interpreter for APL is simple and well-understood. The first complete interpreter was the APL\360 system designed by Breed, Lathwell, and Moore [13]; most of the subsequent APL interpreters have followed the same design [38]. APL program text is tokenized as it is read by the interpreter, and no more analysis is done until a statement is evaluated. The interpreter interleaves parsing and evaluation. When a statement is to be evaluated, it is scanned from the right, and each well-formed APL expression is evaluated as soon as it is recognized. The parser is a simple state machine with a stack. The state machine requires only three states [77, 16], though it is often convenient to use more. The evaluator is a dispatch to the appropriate function or operator routine.

Array data is stored in ravel order in a heap. Ravel order storage is well suited to array operations. An access polynomial permits fast translation between a multi-dimensional array index and a ravel order storage index. Traversal along an arbitrary axis is very efficient; elements along a common axis are a fixed distance apart and can be accessed by the successive additions of a constant. Storage is used efficiently, for the data is contiguous. Storage management is straightforward. All heap entries are referenced indirectly through pointers in the symbol table or execution stack. Each heap entry contains two fields in addition to the data stored. One is the length of the entry and the other is a backpointer to the stack or symbol table. The doubly-linked pointers makes garbage collection and relocation easy.

The complexity of the APL interpreter is in the routines which implement the primitive functions. Here there is almost unlimited opportunity for optimization. For example, dyadic iota and membership can be implemented with a linear-time hashing algorithm [7, 33]; transpose can be effected without moving any data by manipulating an array descriptor [1]; and the plus reduction of a boolean vector can be implemented very efficiently using the "translate" instruction of the IBM/360 series [41].

5.2.2 Carrier APL

An interpreter for carrier APL has the same structure as an interpreter for current APL. Carrier APL program text is tokenized as it is read into the system. Parsing and evaluation are interleaved as before. There are minor changes in the APL syntax, but the language can still be recognized by a simple state machine with a stack. The evaluator is a dispatch to the routine which handles the application form in the expression.

Arrays are stored as edge vectors in carrier APL, as explained in section 5.1. Each array has a header block which contains the rank of the array, the implicit datum rank of the array,[15] and a pointer to the tree of edge vectors. If the array is a scalar, the value is stored directly in the header. Storage management requires a compactifying garbage collector. All arrays are referenced indirectly through stack entries or the symbol table. However, since the garbage collector traces all active chains of pointers, backpointers to the stack or symbol table are not stored in array headers.

The evaluator is organized around the application forms. Each type of application form is implemented in a separate routine. Each routine partitions argument arrays and passes successive pairs of base arguments to another routine which evaluates the function being applied. This second routine either implements the base definition of a primitive function or evaluates defined functions.

[15]See section 3.4.2 for a definition of implicit datum rank.

The edge vector representation permits the implicit sharing of data between arrays, and the primitive functions routines exploit this whenever possible. Unlike LISP, APL has "by-value" assignment and parameter passing [26]. Assignment by value means that for any assignment $A \leftarrow <EXP>$, subsequent changes to the identifiers referenced in $<EXP>$ will not affect A. Carrier APL maintains the value semantics of APL; thus there can be no explicit sharing of data between values. However, data can be implicitly shared if the sharing does not affect the semantics of an expression. Indexed assignment is the only carrier APL operation which appears to restrict the sharing of data. If implemented in a manner similar to the RPLACA operation in LISP, it destroys the value semantics defined above. Rather, we implement indexed assignment by copying the edge vector that is assigned new values. Thus, for a vector V we have:

$$V[I] \leftarrow <EXP> \qquad \leftarrow\rightarrow \qquad V \leftarrow ((I-1)\uparrow V), <EXP>, I\downarrow V$$

This by-value implementation permits all the other primitive functions to share data wherever possible, as in LISP.

5.2.3 Application Forms

Each of the five application forms (scalar product, outer product, reduction, scan, and inner product) is implemented by a routine associated with the form. Each routine determines the carrier partition of the argument arrays and of the result of the application, performs conformability checks on the carrier level of the argument arrays, and creates the edge vector structure for the carrier level of the result array. The routine then cycles through the carrier level of the argument arrays, appropriately pairing base arguments arguments, and passing them to a routine which implements the function being applied. The results of each call to the function routine are assembled in the new carrier. The routines are distinguished in how they traverse the carrier level to access base arguments, and in the conformability checks they perform.

The outer product form contains a transpose vector which guides the pairing of the base arguments. The transpose vector is restricted to a catenation of two ascending vectors. One maps the dimensions of the left argument to the result; the other maps the dimensions of the right argument. Since both vectors are ascending, the base arguments can be accessed and paired by traversing the the carrier levels in the argument arrays in ravel order without backing up. (See section 3.2.3.1.)

5.2.4 Primitive functions

The base definition of each primitive function is implemented in a separate routine. Each routine accepts base arguments and a datum rank, allocates storage for the result, and performs the function. The routines differ from current APL implementations in that they accept arrays of non-scalar items. We briefly sketch the implementation of each class of primitive functions below.

5.2.4.1 Arithmetic Scalars

For scalar data, the functions are the same as in current APL. For non-scalar data, corresponding elements in the data items are first paired, and the current APL routine applied.

5.2.4.2 Equality

For scalars, the equal function is the same as in current APL. For non-scalar items, the pointers to the two arguments are compared for a inexpensive test of equality. (Recall that edge vectors can be shared between two arrays.) If the pointers are unequal, the respective lengths, types, and data of the edge vectors on the first level of the two arguments are compared. If the types are both pointer, the equal function is applied recursively in a breadth first search of the two trees of edge vectors.

To implement membership, we sequentially search the right argument for an occurrence of each element in the left argument, using the equal function to test for a match. For dyadic iota, the arguments are reversed. This algorithm takes time proportional to the product of the lengths of the two arguments. Faster algorithms can be used. With the definition of an appropriate hashing function, Bernecky's linear time implementation [7, 33] can be extended to vectors of non-scalar items. Alternatively, the non-scalar items can be mapped to a tree structure, as Bernecky has done for nested arrays in SHARP APL [9], and the searches performed on this representation, giving an expected-time O(n log n) implementation.

5.2.4.3 Relations

The relational functions define a lexicographic ordering of arrays. For numeric scalars, the current APL function is used. Character scalars are compared using their ⎕*AV* values, and characters are less than numbers. Vectors of scalars are compared by searching sequentially from the left for the first difference, and then comparing the differing elements; if no difference is found, the shorter vector is less than the longer. Vectors of non-scalar data are compared in a similar manner. The equal function defined above is used to find the first difference, and less than is applied recursively to the differing pair of elements.

Using this comparison function, gradeup and gradedown are implemented with quicksort [37].

5.2.4.4 Structure and Selection

The structure and selection functions treat vectors of scalar and non-scalar data alike. With the exception of flatten and shape, they copy and rearrange the edge vector on the first level of the argument array. If the edge vector is a vector of pointers to non-scalar data items, these items will be shared between the argument and result arrays. The edge vector that is being rearranged is always copied, so the value of the argument is unchanged.

Flatten ($\ddot{,}$) is defined for a scalar of arbitrary rank datum, and it flattens the datum. The result is a scalar containing a vector. Flatten is implemented by copying the data in the leaves of the edge vector tree into a new vector.

Shape (ρ) simply returns the length of the edge vector on the first level of the argument.

5.2.4.5 Functions without obvious extensions to non-scalar items

The functions with no obvious extensions to non-scalar items (monadic iota, matrix inverse, matrix divide, encode, decode, and deal) are implemented as they are in current APL.

5.2.4.6 Functions of unbounded rank

The functions of unbounded rank are implemented in a straightforward manner. Ravel flattens an arbitrary array into a vector; the array is traversed and the items assembled in a new edge vector. Rank returns the rank field in the array header. Dyadic transpose walks the right argument array to determine the shape of the result, creates a new array of appropriate shape, and traverses the argument array again, copying elements to their new locations in the result array. Reshape copies the items in its right argument into new edge vectors; the lengths of the edge vectors are determined by the left argument.

5.2.4.7 Special Forms

The special forms are also implemented in a straightforward manner. Indexing follows chains of pointers as described in section 5.1.3, and stores the indexed values in a new array. Assignment updates the value pointer for a symbol in the symbol table. Indexed assignment is implemented by copying the edge vectors which are receiving new values. Branching selects the next line of the function body to be processed by the evaluator.

5.2.5 Defined Functions

There are two major categories of defined functions: defined functions of bounded rank, and defined functions of unbounded rank. One routine evaluates both categories. Functions of bounded rank can be applied with the application forms. The routine that evaluates the application form partitions the array arguments and passes the defined function with successive pairs of base arguments to the routine which evaluates defined functions. Functions of unbounded rank cannot be applied with the application forms and the function is passed with its array arguments to the evaluating routine directly, without invoking an application routine. Functions of unbounded rank can be applied with a datum rank; if so, the array arguments are partitioned into arrays of non-scalar items before being passed to the evaluation routine.

In the evaluation of the function, the binding of formals and treatment of local variables is conventional; current APL's dynamic scoping is supported. However, if the function is applied with a non-zero datum rank K, the formal is bound to an array of rank K data items. K is an implicit datum rank, and its value is stored in a special field in the array header for the formal. As discussed in chapter 3 (see section 3.4.2), an implicit datum rank is inherited by the result of the application of any function which preserves datum rank. If the defined function preserves datum rank, its result will also be an array of implicit datum rank K; this datum rank must be removed from the array header when the function environment is exited.

5.3 Compilation

Compiling APL is attractive because of the optimizations to be found in analyzing an APL expression as a unit. The difficulty in compiling APL is in determining the attributes of identifiers in an expression. In the absence of declarations, these attributes must be inferred.

In his thesis, Miller analyzes the binding of the attributes of APL identifiers into compiled code [43, 44, 45]. He determines that the most important attributes for compiling APL are the valence of functions and the rank and type of arrays. Function valence determines the syntax of the expression, array rank determines the control structure of the generated code, and array type determines the selection of machine instructions. The valence of defined functions rarely changes and can almost always be determined. Array attributes can be determined from initial values and propagated through expressions.

Miller gives an algorithm for propagating array attributes through an APL expression. He determines which features of APL halt attribute propagation, and uses the occurrence of these features to break programs into blocks which are compiled separately. Once the input values to a compilation block are known, attributes can be propagated and the block compiled as a unit.

The "carrier" concept of applying functions in parallel along the dimensions of an array suggests a straightforward streaming of base argument evaluations, and makes carrier APL an attractive candidate for compilation. In this section, we examine how Miller's analysis of expressions can be applied to carrier APL. We first give a brief overview of his compiler. We then examine array attributes in current and carrier APL, and discuss rules for breaking programs into compilation blocks. We conclude by observing that in some ways carrier APL is more amenable to compilation than current APL.

5.3.1 Compiler Overview

Miller's system interleaves compilation and execution; it is similar to the HP-3000 compiler [32]. When a program is compiled, it is first broken into compilation blocks. When a block is to be executed for the first time, all existing array attributes are propagated through the block. The block is analyzed as a unit, and highly optimized code is generated using transformations similar to Abrams' drag along and beating [1]. The code is then evaluated. A preamble of "signature code" is attached to the head of the compiled code; this preamble checks for attributes which are bound into the code. On a subsequent execution, if any of these attributes are changed, the block is recompiled; otherwise the compiled code is executed directly.

Miller's compiler generates code for a special-purpose architecture, the ladder machine [46]. However, his analysis of the problems of compiling APL is applicable to a compiler for any machine.

5.3.2 Array Attributes

The array attributes Miller considers important to a compiler are rank, type, and shape. Of these three, rank and type can usually be determined and bound into the compiled code for an expression. In carrier APL, the three-level partition of a carrier array is the most important attribute in an expression, and it is can almost always be determined and bound into the compiled code.

5.3.2.1 Current APL

In Miller's compiler, rank is important, for it determines the control structure of the generated code. Miller argues that the ranks of array values do not often change between evaluations of an expression, except for functions that are designed to be rank independent. Rank can easily be propagated through an APL expression; of the APL primitives, only transpose and reshape are not rank transparent. (A function is transparent with respect to an attribute if the values of the attribute in the function arguments determine the value of the attribute in the result.) In a study of APL programs, Bauer and Saal found that 80 percent of array ranks in the programs can be determined by static analysis of the program text [6].

The type of array elements is important in generating machine instructions. Type is a stable attribute, and it can be easily propagated through an APL expression. All APL primitives are type transparent, and Bauer and Saal have found that almost 90 percent of APL domain checking can be done statically. However, the APL primitives are not transparent with respect to machine representation; a numeric value may be boolean, integer, or floating point. Miller does not address this issue, for the arithmetic instructions on the ladder machine are type sensitive.

The shape of an array is a more volatile attribute; the shape of array will often change between evaluations of an expression. Moreover, shape cannot be as easily propagated through an expression. Five primitive functions (compression, expansion, take, drop, and reshape) are not shape transparent; in two surveys of APL programs, these functions accounted for approximately a fifth of the primitive functions in the program text [11, 12, 34]. Bauer and Saal estimate only 60 percent of APL length checking can be done statically. Miller addresses this problem by propagating three types of shape: minimum, maximum, and actual. Minimum and maximum shape define a range of possible shapes for an array, and they are useful for storage allocation and a limited amount of conformability checking. All primitive functions except reshape allow the propagation of minimum and maximum shape.

5.3.2.2 Carrier APL

In carrier APL, the three-level carrier structure can be easily propagated through an expression if we restrict the datum rank in the application form to a constant. This restriction should not seriously limit carrier APL programming; all examples in this thesis use constant datum ranks.

The rank of every value in an expression can usually be determined. Every application of a primitive function or special form is rank transparent except for dyadic transpose. (The rank of the result of a dyadic transpose depends on the value of the left argument.) Every application of a defined function of bounded rank is rank transparent. An application of defined function of unbounded rank is not, and a call to such a function halts rank propagation.

Given the rank of an array, we can determine the carrier structure of an application of a function to the array. The ranks of the three levels of a carrier array sum to the rank of the array. Both the base and datum rank can be determined statically. The datum rank is a constant in the application form, and the base rank is determined by the function being applied. Thus, knowing the rank of an argument, we can determined the rank of the carrier level.

Type and shape can not be propagated easily in carrier APL. Heterogeneous arrays make type propagation very difficult. We cannot know the type of an arbitrary element selected from a heterogeneous array unless we know the type of each element in the array and the value of each selector. Thus the selection functions (take, drop, compression, and indexing) are not type transparent. These functions are used frequently in current APL; in two surveys of APL programs, these functions accounted for more than a quarter of all primitive functions in the program text [11, 12, 34]. We expect they would also be used frequently in carrier APL.

Carrier APL inherits the problems of shape propagation Miller found in current APL. The same primitive functions halt the propagation of shape in both current and carrier APL. Moreover, the shape of a ragged array contains many elements, one for every vector on each level of the array, and it is expensive to propagate.

5.3.3 Determining Compilation Blocks

5.3.3.1 Current APL

A compilation block in a current APL program is a set of expressions which can share a common control structure in the generated code. The ranks of the arrays in an expression determine the control structure of the generated code. When the rank of an array cannot be determined, the block ends, for the following expressions cannot share control structure with the previous expression. Similarly, an occurrence of either of the APL primitives gradeup (⍋), matrix inverse (⌹), or matrix divide (⌹) end a block. These primitives are implemented in a complicated, self-contained control structure which cannot easily be merged with other code.

Miller gives several occurrences used to break APL programs into blocks. We list them below along with a brief description of why they determine a block boundary.

- The source or target of any branch. To be able to identify target locations, Miller restricts APL branches to labelled statements. A compilation block contains no loops or branches.
- A dyadic transpose. Dyadic transpose is not rank transparent.
- A reshape. Reshape is not rank transparent.
- A call of a defined function. The rank of the result of a defined function is unknown. A defined function may also change through side effects attributes of other identifiers in the expression.
- Any primitive function whose evaluation cannot be streamed, e.g., gradeup (⍋) matrix inverse (⌹), and matrix divide (⌹).

The first category restricts compilation blocks to straight line code. The remaining categories potentially divide a single APL expression into several blocks.

5.3.3.2 Carrier APL

In carrier APL, the control structure of the generated code is determined by the three-level carrier structure. As noted above, this three-level structure can be determined if we know the ranks of the argument and result arrays. Thus, as in current APL, we break programs into compilation blocks when we cannot propagate rank. Note, however, that the complicated functions such as gradeup do share a carrier structure with the surrounding expression, and thus we do not need to break programs when they occur.

There are three occurrences in carrier APL which stop the propagation of rank; we break programs into blocks at these occurrences.

1. A dyadic transpose. Dyadic transpose is not rank transparent.
2. A call of a defined function of unbounded rank.
3. The target of any branch.

There are fewer break points in carrier APL, and they occur less frequently than the corresponding break points in current APL. Thus, compilation blocks are larger in carrier APL. Note a call to a defined function of bounded rank does not halt the propagation of rank, and thus the call can be incorporated in a larger block. We expect that defined functions of unbounded rank will be rarely used in carrier APL, except in the case when the rank of the function arguments and results are purposefully indeterminate. Branching should occur less frequently than in current APL, for all functions can be applied with application forms. Dyadic transpose should not occur frequently in carrier APL. The outer product form contains a transpose vector. More importantly, data is typically stored and manipulated along the last dimensions of an array, and the frequent use of transpose with inner product and the encode and decode functions has been replaced by modifying the application axis of these forms.

Of course, a defined function of bounded rank can alter attributes of identifiers in an expression through side effects. To be conservative, we should break when they occur. Alternatively, we could insist the global side effects be declared, or warn the the programmer to compile defined functions with caution.

5.3.4 Compiler Evaluation

From our analysis, we see that more control structure can be shared in compiled carrier APL, but that the type and shapes of the identifiers cannot be as tightly bound in the compiled code.

The control structures implicit in the application forms of carrier APL are well suited to streamed evaluation. As Guibas and Wyatt note, we can organize the evaluation of A++/B∘.+C in a natural way such that the some of the result has been produced before any of the subexpressions have been completely evaluated [25]. In carrier APL, all functions, not just the scalars, can be applied with application forms. Consider the remove duplicates example discussed in chapter 2. When removing duplicates from the rows of a matrix, the evaluator can process one row entirely before starting another, with significant savings in temporary storage and loop overhead. Moreover, base argument streaming allows the more ambitious APL optimizations proposed by Abrams, Perlis, and others [1, 62, 25, 32, 43, 44] to be applied largely within base argument computations, e.g. removing duplicates from a vector. On this restricted domain, these optimizations may prove more successful.

Chapter 6

CONCLUSION

In our eyes, the major strength of APL is the data-driven semantics of the primitive functions and operators, the replacement of explicit looping with the parallel flow of function application along the planes of an array. Carrier APL is a natural generalization of this data-driven control flow. The rank structure of the array, which permits the implicit iteration, is maintained, but the constraints of rectangularity and homogeneity are removed. Primitive functions are redefined to make their parallel application more uniform.

The major discovery of carrier arrays is that the APL primitives can be limited to the domain of scalars, vectors, and matrices, and that the higher dimensions of an array viewed as a *carrier* of the scalar, vector, and matrix computations. The introduction of the carrier concept into the language requires a change in the data structure from a rectangular to ragged array. With the new data structure, ragged data objects can be represented, and with the three-level carrier partition, a non-scalar object can be manipulated as a unit. Almost all the primitive functions have natural extensions to arrays of non-scalar items.

Carrier arrays are a bottom-up extension to APL. Their development has been driven by examples and by the APL programming style. This is in marked contrast to the development of nested arrays, which has been shaped by a mathematical theory of arrays [24, 47, 48, 49, 50, 51, 52, 53, 54].[16] But, as Orth has pointed out, it is not clear how relevant array theory is to the extension of a programming language [60]. Identities are important in "smoothing" a language, removing, wherever possible, special cases and exceptions that must be explicitly tested for. And theoretical considerations are important in assigning meanings to programs, and proving them correct. But it is pragmatic considerations, how people program and how problems are solved, that should guide an extension to a language.

[16]Adin Falkoff contested this, noting that nested extensions developed by him and Iverson have a pragmatic base [22].

References

[1] Abrams, Philip S.
An APL Machine.
Technical Report SLAC-114 UC-32 (MISC), Stanford Linear Accelerator Center, February, 1970.

[2] Abrams, Philip S.
What's Wrong with APL.
In *APL 75*, pages 1-8. ACM, June, 1975.

[3] Aho, Alfred V. and Jeffrey D. Ullman.
Principles of Compiler Design.
Addison-Wesley, Reading, Mass., 1977.

[4] Anderson, William L.
APL 81: A View from the Sidelines.
APL Quote Quad 12(2):4-5, December, 1981.

[5] *APLSF Programmer's Reference Manual*
Digital Equipment Corporation, Maynard, Massachusetts, 1977.

[6] Bauer, Alan M. and Harry J. Saal.
Does APL Really Need Run-Time Checking.
Software -- Practice and Experience 4:129-138, 1974.
Cited in Miller, Terrence. *Tentative Compilation: A Design for an APL Compiler.* Technical Report 133, Department of Computer Science, Yale University, May, 1978.

[7] Bernecky, Bob.
Speeding Up Dyadic Iota and Dyadic Epsilon.
In P. Gjerlov, H.J. Helms and Johs Nielsen (editors), *APL Congress 73*, pages 479-482. North Holland/American Elsevier, New York, August, 1973.

[8] Bernecky, Bob.
Replication.
Technical Report SATN 34, I.P.Sharp Associates, September, 1980.

[9] Bernecky, Robert.
Representations for Enclosed Arrays.
September, 1981.

[10] Bernecky, Bob and Kenneth E. Iverson.
Operators and Enclosed Arrays.
Presented at A Standard APL Workshop, Minnowbrook Conference Center, September 30 - October 3, 1980.

[11] Bingham, Harvey W.
Content Analysis of APL Defined Functions.
In *APL 75*, pages 60-66. ACM, June, 1975.

[12] Bingham, Harvey W., and Kenneth T. Carvin.
Dynamic Usage of APL Primitive Functions.
In *APL 76*, pages 83-86. ACM, September, 1976.

[13] Breed, L.M., and R.H. Lathwell.
The Implementation of APL\360.
In Melvin Klerer, Juris Reinfelds (editors), *Interactive Systems for Experimental and Applied Mathematics*, pages 390-399. Academic Press, New York and London, 1968.

[14] Brown, J.A.
A Generalization of APL.
PhD thesis, Syracuse University, 1971.

[15] Brown, J.A.
Evaluating Extensions to APL.
APL Quote Quad 9(4-Part 1):148-155, June, 1979.

[16] Budd, Tim.
The Design of an APL Compiler: Rules for Boxes.
unpublished memorandum, U.C.Berkeley APL Project, Spring 1979.

[17] Cheney, Carl M.
APL-PLUS Nested Arrays System
STSC, Inc., Bethesda, Maryland, 1981.

[18] Cohen, Jacques.
Garbage Collection of Linked Data Structures.
Computing Surveys 13(3):341-367, September, 1981.

[19] Conrad, William R.
A Compactifying Garbage Collector for ECL's Non-Homogeneous Heap.
Technical Report, Harvard University Center for Research in Computing Technology, February, 1974.

[20] Dewar, Robert.
Private Communication, March 1982.

[21] Falkoff, A.D., and D.L. Orth.
Development of an APL Standard.
APL Quote Quad 9(4-Part 2):409-453, June, 1979.

[22] Falkoff, Adin.
Private Communication, April 1982.

[23] Fisher, D.A.
Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment.
Information Processing Letters 3(1):29-32, July, 1974.

[24] Ghandour, Ziad, and Jorge Mezei.
General Arrays, Operators and Functions.
IBM Journal of Research and Development 17(4):335-352, July, 1973.

[25] Guibas, Leo J., and Douglas K. Wyatt.
Compilation and Delayed Evaluation in APL.
In *Conference Record of the Fifth Annual ACM Symposium on the Principles of Programming Languages*, pages 1-8. ACM, January, 1978.

[26] Gull, W.E., and M.A. Jenkins.
Recursive Data Structures in APL.
Communications of the ACM 22(1):79-96, January, 1979.

[27] Iverson, Kenneth E.
Operators and Functions.
Technical Report RC 7091, IBM Thomas J. Watson Research Center, 1978.

[28] Iverson, Kenneth E.
The Role of Operators in APL.
APL Quote Quad 9(4-Part 1):128-133, June, 1979.

[29] Iverson, Kenneth E.
Operators.
ACM Transactions on Programming Languages and Systems 1(2):162-176, October, 1979.

[30] Iverson, Kenneth E.
A Programming Language.
John Wiley and Sons, Inc., New York, 1962.

[31] Jenkins, M.A., and Jean Michel.
Operators in an APL Containing Nested Arrays.
APL Quote Quad 9(2):8-20, December, 1978.

[32] Johnston, Ronald L.
The Dynamic Incremental Compiler of APL\3000.
APL Quote Quad 9(4-Part 1):82-87, June, 1979.

[33] Jordan, Kevin E.
Speeding up the Floating Point Cases of Dyadic Iota and Epsilon.
Presented at A Standard APL Workshop, Minnowbrook Conference Center, September 30 - October 3, 1980.

[34] Kanner, Raymond.
The Use and Disuse of APL: An Empirical Study.
APL Quote Quad 13(1):154-159, September, 1982.

[35] Kernighan, Brian W. and P.J. Plauger.
Software Tools.
Addison-Wesley, Reading,Massachusetts, 1976.

[36] Knuth, Donald E.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms.*
Addison-Wesley, Reading,Massachusetts, 1968.

[37] Knuth, Donald E.
The Art of Computer Programming. Volume 3: *Sorting and Searching.*
Addison-Wesley, Reading,Massachusetts, 1973.

[38] Lathwell, Richard H.
Some Implications of APL Order-of-Execution Rules.
APL Quote Quad 9(4-Part 1):329-332, June, 1979.

[39] Lowney, P. Geoffrey.
Carrier Arrays: An Idiom-Preserving Extension to APL.
In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 1-13. ACM, January, 1981.

[40] Martin, Johannes J.
An Efficient Garbage Compaction Algorithm.
Communications of the ACM 25(8):571-581, August, 1982.

[41] McDonnell, Eugene E.
Recreational APL: Sum of the Bits.
APL Quote Quad 11(1):30-32, September, 1980.

[42] Mezei, J.E.
Uses of General Arrays and Operators.
In *6th International APL User's Conference*, pages 334-348. APL User's, May, 1974.

[43] Miller, Terrence.
Tentative Compilation: A Design for an APL Compiler.
Technical Report 133, Department of Computer Science, Yale University, May, 1978.

[44] Miller, Terrence.
Type Checking in an Imperfect World.
In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, pages 237-243. ACM, January, 1979.

[45] Miller, Terrence.
Tentative Compilation: A Design for an APL Compiler.
APL Quote Quad 9(4-Part 1):88-95, June, 1979.

[46] Minter, Charles.
A Machine Design for the Efficient Implementation of APL.
Technical Report 81, Department of Computer Science, Yale University, May, 1976.

[47] More, Trenchard, Jr.
Axioms and Theorems for a Theory of Arrays.
IBM Journal of Research and Development 17(2):135-175, March, 1973.

[48] More, Trenchard, Jr.
Notes on the Development of a Theory of Arrays.
Technical Report Report 320-3016, IBM Scientific Center, Philiadelphia, May, 1973.

[49] More, Trenchard, Jr.
Notes on the Axioms for a Theory of Arrays.
Technical Report Report 320-3017, IBM Scientific Center, Philiadelphia, May, 1973.

[50] More, Trenchard, Jr.
A Theory of Arrays with Applications to Databases.
Technical Report Report 320-2106, IBM Scientific Center, Cambridge, September, 1975.

[51] More, Trenchard, Jr.
Types and Prototypes in a Theory of Arrays.
Technical Report Report 320-2112, IBM Scientific Center, Cambridge, May, 1976.

[52] More, Trenchard, Jr.
On the Composition of Array-Theoretic Operations.
Technical Report Report 320-2113, IBM Scientific Center, Cambridge, May, 1976.

[53] More, Trenchard, Jr.
The Nested Rectangular Array as a Model of Data.
APL Quote Quad 9(4-Part 1):55-73, June, 1979.

[54] More, Trenchard, Jr.
Nested Rectangular Arrays For Measures, Addresses and Paths.
APL Quote Quad 9(4-Part 1):156-163, June, 1979.

[55] Morris, F. Lockwood.
A Time- and Space-Efficient Garbage Compaction Algorithm.
Communications of the ACM 21(8):662-665, August, 1978.

[56] Morris, F. Lockwood.
On a Comparison of Garbage Collection Techniques.
Communications of the ACM 22(10):571, October, 1979.

[57] Morris, James H., Eric Schmidt, and Philip Walder.
Experience with an Applicative String Processing Language.
In *Conference Record of the Seventh Annual ACM Symposium on the Principles of Programming Languages*, pages 32-46. ACM, January, 1980.

[58] Morris, James H.
Real Programming in Functional Languages.
Technical Report CSL-81-11, Xerox Palo Alto Research Center, July, 1981.

[59] Orth, D.L.
A Comparison of the IPSA and STSC Implementations of Operators and General Arrays.
APL Quote Quad 12(2):11-18, December, 1981.

[60] Orth, D.L.
A User's View of General Arrays.
Technical Report RC 8782, IBM Thomas J. Watson Research Center, April, 1981.

[61] Parnas, D.L.
On the Criteria To Be Used in Decomposing Systems into Modules.
Communications of the ACM 15(12):1053-1058, December, 1972.

[62] Perlis, Alan J.
Steps Towards An APL Compiler - Updated.
Technical Report 24, Department of Computer Science, Yale University, March, 1975.

[63] Perlis, Alan J., and Spencer Rugaber.
The APL Idiom List.
Technical Report 87, Department of Computer Science, Yale University, April, 1977.

[64] Perlis, Alan J., and Spencer Rugaber.
Programming with Idioms in APL.
APL Quote Quad 9(4-part 1):232-235, June, 1979.

[65] Rabenhorst, D.A.
APL2 Language Manual
First edition, IBM, 1982.
Installed User Program, Program Number 5798-DJP.

[66] Rabenhorst, D.A.
Private Communication, April 1982.