Technical Report

Santa Teresa Laboratory
San Jose, CA

IBM

TR

THE PRINCIPLES OF APL2 *BY DR. JAMES A. BROWN*

MARCH 1984 *TR* 03.247

March 1984
TR 03.247

The Principles of APL2

by

Dr. James A. Brown

International Business Machines Corporation
General Products Division
Santa Teresa Laboratory
San Jose, California

ABSTRACT

This paper presents the rules governing the APL2 language and the principles that motivated the design decisions.

TABLE OF CONTENTS

The Principles of APL2

by

Dr. James A. Brown

1.0 INTRODUCTION

IBM has many products which follow the IBM internal standard for *APL* (*VSAPL*, *APLSV*, PC *APL*, 8100 *APL*). This level of the *APL* language is referred to as *APL*1 in this paper.

*APL*2 is based on this writer's PhD Thesis [Br1], the array theory of Trenchard More [Mo1], and most of all on *APL*1. See Appendix 3 for a brief chronology of the development of *APL*2. *APL*2 removes many of the restrictions of *APL*1. *APL*2 generalizes many of the fundamental concepts of *APL*1 and extends or completes many functions and operators.

Although this paper attempts to be formal in its approach, some statements are made without proof when the details of the proof would not add to understanding. While a presentation of principles would be brief, this paper derives and discusses the rules and principles. No attempt is made to present the new language in its entirety. Familiarity with the concepts of *APL*2 is assumed. Those wishing a more complete description of *APL*2 may refer to the APL2 publications library [1-11]. This is rather a discussion of how *APL*2 was designed and what motivated the choices that were made.

2.0 THE OBJECTIVES OF *APL2*

2.1 *APL1*

APL1 has proven itself to be an extremely powerful, usable language. This is because it has arrays as data; has functions which apply directly to arrays and produce arrays as results; has operators which apply to functions in a uniform manner producing families of related functions; and has syntax governed by a few simple, easily understood rules.

Arrays are collections of numbers or collections of characters. Arrays are rectangular in the sense that in a matrix, for example, each row has the same number of columns.

Numbers are one-sorted in that all numbers are real numbers and a user of the notation does not have to manage number representations.

Array operations allow computations to be performed where the data itself controls the limits of the operation. A summation operation is controlled only by the data being summed. No loop is explicitly written. You can do arithmetic on whole collections of numbers in a single operation.

Operators provide a means for controlling the application of functions to data. When you understand how an operator works when applied to one function, you know how it applies to any function in its domain. If you understand +/, then you can figure out -/ and ∘/ even if you've never seen them.

The simple syntax of *APL1* does not assign precedence to functions. You do not need to remember which functions are evaluated before which other functions. Only their position in an expression determines when they are evaluated.

2.2 *APL2*

If we understand what is good about *APL1*, we may proceed to consider extensions which will enhance the good features. The extensions range from the removal of somewhat arbitrary restrictions to incorporation of pervasive changes.

In *APL2* the *APL1* restriction which forbids numbers and characters in the same array is relaxed. An array may be a collection of numbers and characters. Numbers are one-sorted in that all numbers are taken from the complex number field. The term nonreal number is reserved for numbers with a nonzero imaginary part. Real numbers are the proper subset which have an imaginary part of zero.

APL2 extends the array concept by permitting any item of an array to be any other array. Actual data is not always rectangular. *APL2* retains the useful properties of rectangular data yet lets non-rectangular data be easily represented in nested arrays. Nested arrays can be used with the same ease as *APL1* arrays. Users are not burdened with the managing of the data structures. More operations are controlled by the data and the need for explicit controls on these computations is removed. The result is a reduction in the complexity of programs.

Operators from *APL1* are generalized so that they apply to all dyadic functions - even those defined by the user. One new primitive operator is defined and users may define their own operators.

The syntax of *APL2* is essentially unchanged from *APL1*. Restrictions on the use of parentheses are removed so that one simple rule governs the use of all parentheses. This provides the ability to write expressions of operators and a convenient way to write lists of arrays (especially nested vectors) as an extension of constant vector notation.

In summary: the objective of *APL2* is to produce a more powerful and productive language; the challenge is to make it formally correct and complete while providing a rich set of tools for problem solving.

3.0 CRITERIA

There is no doubt that the set of possible extensions is inter-related. It is not possible to study each extension in isolation from the others. Yet there has to be some measure by which we judge any change.

Four criteria against which any proposal for an extension may be measured are:

- compatibility
- formality
- simplicity
- usability

3.1.1 COMPATIBILITY

Compatibility is the measure of the extent to which a proposal imposes a change on something in *APL1*. There are four kinds of changes to functions which might affect compatibility:

- errors become answers,
- answers become different answers,
- answers become errors, and
- errors become different errors.

Many changes cause *APL2* to give answers where *APL1* produced an error message. In general, these are not considered serious compatibility issues and changes of this kind have been taking place throughout the history of *APL*. However a system where every expression gives an answer no matter what would be a difficult one in which to develop an application.

Changes which cause answers different from those in *APL1* are serious compatibility issues and should be adopted only for very strong reasons.

Changes which cause errors where *APL1* gave answers appear serious at first glance but are not because the error message and the carets point precisely to the offending statement. These are therefore easier to fix than the second case above where a failure may not occur.

In some situations *APL2* gives a different error message than *APL1*. This does not affect compatibility because there was no

way under program control to determine which error occurred. It does affect documentation and user practice. Since *APL2* has error handling facilities, future changes in error messages could affect compatibility.

Another factor that can influence the decision is the extent to which a change can be automatically detected or even corrected.

A second form of compatibility has to do with the rules users apply when using the notation. *APL1* has many identities which help people understand and remember how functions work. For example the shape of the result of the indexing function may be difficult to figure out without the identity:

$$\rho M[I;J] \leftrightarrow (\rho I),(\rho J)$$

It is important to preserve useful identities whenever possible.

3.1.2 FORMALITY

Formality is the measure of the extent to which a proposal follows rules. This is where consistency must be determined. A formally incorrect proposal can never be considered. However there may be many correct formalisms and the choice among them must be guided by other principles.

Formal arguments are normally phrased in terms of identities. As described in the previous section, *APL1* has many identities and judgement is required in choosing those which are to remain true and those which will be violated. Very often the identities of *APL1* dictate which of several definitions for a primitive should be chosen. Of course it is a matter of judgement which identities are the important ones. In *APL2* the shape identities from *APL1* (as in indexing, outer product, etc.) are considered important.

Also formal correctness of the extended notation may be assured by choosing easily understood -- universally true identities describing the new function. For example, in *APL2* the new function disclose (⊃) is defined as a left inverse of enclose (⊂):

$$A \leftrightarrow \supset \subset A$$

The function disclose is also given a meaning when applied to nonscalar arguments but even so the above identity remains strictly and universally true.

3.1.3 SIMPLICITY

Simplicity is the measure of several things. Having few rules is better than having many; having classes of objects with similar properties is better than treating each object separately; results that are conformable with arguments are better than results that are not; a rule without an exception is better than one with an exception.

Again these are not independent considerations. It is clearly good that the class of functions called the scalar functions all operate the same way. Saying all functions are ambi-valent lets us then talk about all functions as a class. Yet operators are not ambi-valent because the additional parentheses that would be needed are not desirable.

The simplicity principle is sometimes called Occam's Razor -- if two explanations describe some behavior then the simpler one is probably correct. That is how renaissance astronomers concluded that the Sun was the center of the solar system - the motion of the planets was simpler to describe.

3.1.4 USABILITY

Usability is the measure of the ease with which the notation can be understood and applied. Although important, it is the most difficult criterion to measure or subject to an objective test. This is probably the reason why companies have whole departments devoted to usability studies and human factors. In the end, usability is strongly influenced by formality and simplicity.

Given that the other principles are followed, usability follows from conscious decisions to make things easy to do. It can arise from making the most common things people want to do easy to express even when another formulation may be desirable for other reasons. One proof of usability is the ability to correctly predict how an operation works in an unfamiliar situation -- the law of least surprise. Usability in *APL2* most often follows from the knowledge of a few simple rules -- again the application of identities.

3.2 DESIGN DECISIONS

These principles are not independent properties. At first it is tempting to try to order them by priority. Perhaps formality is more important than compatibility; or usability is more desirable than simplicity. If a given change falls only under one of these headings, that may decide the question. For example a prototype of *APL2* returned 0 as the answer to 0÷0. There are

good arguments why that answer is better, but 0 is not more formally correct than 1. (Formally any number is correct - 0÷0 is not undefined, it's over-defined.) Only compatibility with *APL1* dictated that we must return 1. More often the decision is a trade off between several principles and in these cases human judgement must be applied to determine the decision. This is why language design is an art rather than a science and one reason why reasonable people can come to different conclusions given the same problem.

Common to all four criteria is the notion of an identity. *APL2* must be compatible with useful identities in *APL1*; identities are the key to formality; identities make ideas simple to understand and that makes them usable.

In the following sections an attempt will be made to identify how these principles are applied in making the decisions embodied in *APL2*.

4.0 METHOD

There are two very different approaches to extending *APL* which were used in the design of *APL2*. The approaches were used for different purposes and at different times in the design process. The constructive approach assures us that the desired function is available in the new language. The deductive approach assures us that the new language is formally correct.

4.1 THE CONSTRUCTIVE APPROACH

In the constructive approach we start with what we have (*APL1*) and make extensions -- at each point being sure we have not introduced any formal inconsistency, anything difficult to use, etc. We build new things from the old things.

4.2 THE DEDUCTIVE APPROACH

In the deductive approach we start with where we want to be and make sure that the resulting notation is compatible, formally correct, simple, and usable. We begin by adding things one at a time, then look back and formalize what was produced. The rules may be proposed when you start but are not validated until you finish. The real rules are discovered after we know where we want to be. The process is iterative -- extend, discover rules, adjust the extensions to meet the rules, and repeat the process refining the ideas and the notation at every step. When using this approach, since it is a matter of judgement as much as with the constructive approach, it is advisable to stay away from the limiting cases where intuition may not serve well. Therefore we stay away from arrays of rank 0 and 1, from axes of length 0 and 1, and therefore from arrays having 0 or 1 items. Rather we examine extensions away from the limits, attempt to understand what is true, write the identity that describes what is true, and then approach the limit formally. If the identity fails at the limit, either the identity is not universally true, we wrote the wrong identity, or we discover some new property of the data or the operations which make it true.

The section on empty arrays contains two examples of the deductive approach. In the first we choose to believe the formal equation and postulate data that makes the equation universally true. In the second, the example is universally true but does not do what is expected in the empty case. This time we reject the equation in favor of another which is also universally true but which does what is expected as well. The choice you make in any given instance is, alas, a matter of judgement.

4.3 COMPARISON OF THE APPROACHES

Neither approach is more correct than the other and in cases where the same conclusion is reached by either approach very little judgement need be employed. Which approach is chosen in a particular instance depends on the impact of the proposed extension.

When adding a single primitive operation it is clearly most important that it fit with what exists. If the primitive is isolated from other considerations such as syntax and new data structures it may be considered in isolation. The only questions are "Is it a good definition?", "Is it easy to use?", etc.

When adding a large extension that pervades a significant part of the notation, you still begin with the constructive approach -- adding things carefully one at a time. After everything is in place, however, the resulting notation should be compatible, formal, simple, and usable. This involves writing down what you believe to be true about the extended notation (keeping away from the limits), then studying the limits making adjustments to rules or extensions until you are satisfied that all the criteria are satisfied.

In particular, extending the universe of data to nested arrays must, by its very nature, affect every operation. Given nested arrays, some new primitives must be viewed as part of the whole and not as isolated extensions.

The following sections introduce *APL2* objects. Starting with general abstract notions, we apply controls to limit the set of objects. Since we know what we want to achieve, we know what controls to impose. Thus, when talking about functions in the abstract, we make choices knowing that we want a linear infix notation that can allow at most two arrays as arguments. It is therefore no surprise when talking about syntax that we can easily devise ways to write the functions we have included.

5.0 THE OBJECTS OF *APL2*

APL2 recognizes three classes of objects: arrays, functions, and operators. In this section each object class is examined first as an abstract notion and then as the subset actually permitted in *APL2*.

5.1 ARRAYS

The arrays of *APL2* are based on this writer's PhD dissertation [Br1]and on the array theory of Trenchard More [Mo1 etc.] and influenced by numerous other papers (see references). Array theory is an attempt to give a scientific basis to the theory of data and has been proven to be correct (i.e. consistent) if set theory is correct.

Arrays are the objects least constrained by *APL1*. Evidence of this is the existence of two very different nested array implementations: one based on array theory and one based on other principles. Other possibilities also exist. One can conceive of arrays with different numbers of items in each row (ragged arrays) and these would not conflict with rectangular arrays which would be a proper subset of them. Geoff Lowney's PhD dissertation [Lo1], for example, is an interesting (although not entirely compatible) possible direction for extension to arrays.

5.1.1 THE CONCEPT OF AN ARRAY

Arrays may be viewed in the abstract. An array is an ordered collection of items. It is the analog of a set which is an unordered collection of items. At this level of abstraction the items can be anything at all and they can be ordered in any conceivable manner.

The concept of ordered collections stands on its own and may be considered and understood without a syntax to write collections or operations to manipulate them. *APL2* adds to this concept controls which organize and define a subset of the set of all possible arrays on which we define operations. It then defines a syntax for expressing some arrays, operations to transform them into other arrays, and rules for applying the operations.

5.1.2 CONTROLS ON ARRAYS

The first control on arrays is rectangularity. We include in our universe of data only arrays where the number of items along any axis is independent of the position along the other axes (that is the axes are orthogonal).

The next control is to specify the nature of the items. We will include in our universe of data only arrays whose items are arrays.

While it is conceivable to have arrays in which the recursion continues without end, we know we want to express computations on real data. We therefore stipulate as the next control the conditions for the termination of nesting [Mo8]. We declare that a single number or character (that is a simple scalar) contains itself as its only item. Thus arrays contain arrays as items and recursion is effectively terminated by a simple scalar.

The next control on arrays is finiteness. The length of any of the orthogonal axes of the rectangular arrays is finite. The number of axes is finite and the depth of nesting is finite.

5.1.3 ARRAYS IN *APL2*

The arrays of *APL2* are finite rectangular arrays which contain arrays as items. When the term array is used, it means this subset of all possible arrays.

The arrays of *APL2* are the same as the arrays of Array Theory and in particular empty arrays have structure as defined by Array Theory [Mo1 etc.].

An array one of whose items is other than a single number or character (a simple scalar) is called a nested array. An array containing only numbers or containing only characters is called a homogeneous array. An array all of whose items are either single numbers or single characters is called a simple array. The arrays of *APL1* are simple and homogeneous.

In some sense every array in *APL2* is nested because it contains other arrays. The term is reserved for those which contain at least one item which is not a single number or character. Thus the universe of arrays is partitioned into two subsets: simple arrays and nested arrays.

5.2 FUNCTIONS

The functions of *APL2* are defined as in *APL1*. Although *APL2* has several new functions and old functions have been extended, the definition of and controls on functions are not different and are included here for completeness.

5.2.1 THE CONCEPT OF A FUNCTION

A function is a mapping from the members of its domain to the members of its codomain. (The codomain is sometimes called the range.) Both the domain and the codomain are sets (as opposed to arrays) because there is no implied ordering. The mapping represented by a function may be specified by a formula (especially if the domain has an infinite number of members) or by a table listing the codomain member corresponding to each member of the domain.

APL2 adds to this concept controls which select the subset of functions we wish to make representable.

5.2.2 CONTROLS ON FUNCTIONS

Since the data of *APL2* is arrays, we want functions to operate on arrays and produce arrays. An *APL* function may take either one or two arrays as arguments and produce one array as a result. Therefore the first control we impose on functions is that the largest domain is the set of all arrays together with the set of all pairs of arrays. It is this choice of domain that makes all functions ambivalent. Which is written in any instance is a syntactic decision not a semantic one (see section on array expressions).

The second control is that the largest codomain of functions is the set of all arrays. When a function is evaluated, the array from the codomain that is produced is called the explicit result of the function.

Thus the functions of *APL2* map arrays (monadic) or pairs of arrays (dyadic) onto arrays.

5.2.3 FUNCTIONS IN *APL2*

While the arrays of *APL2* match those of array theory, the functions (and the symbols used to represent them) are taken from

*APL*1. Not all possible functions are made primitive but a way to generate functions is provided (operators) as well as a way to define functions.

Given the new data structures, the restrictions on the number of arguments to functions are not severe. Although *APL* allows functions of at most two arguments and at most one result, nested arrays give an easy way to package arrays into a vector. The vector may be thought of as multiple arguments even though syntactically it is one argument.

The function universe of *APL*2 also includes functions which do not produce a result. These are not functions in the strict sense because they do not have a codomain. In a strict function this could only be true if the domain were also empty. The only primitive function which returns no result in some circumstances is execute (⍎). User defined functions without results are also permitted.

Notice that the description of functions does not include *APL*'s concept of niladic function (a function with no arguments) and in general when we talk about "all functions" we do not mean the niladic ones. Except for compatibility with *APL*1 terminology, we would give these a different name. Syntactically they are treated like arrays.

5.3 OPERATORS

Operators are one of the most powerful concepts in *APL* and will probably provide the most interesting direction for future extensions. *APL*1 has a very limited set of operators which can be used only with specific functions. *APL*2 has an unlimited set of operators.

5.3.1 THE CONCEPT OF AN OPERATOR

An operator is a mapping from a member of its domain to a member of its codomain. This is exactly what was said about functions.

Operators differ from functions because the largest domain for operators is larger than that for functions; the codomain is different from that for functions; and operator binding to operands is stronger than function binding to arguments. Operators and functions as a group are referred to as operations.

5.3.2 CONTROLS ON OPERATORS

Operators in *APL1* are defined as applying to functions giving new functions as a result. In *APL2*, operators with array operands are allowed so the first control on operators is that the largest domain of operators is the union of arrays, functions, and pairs of arrays and functions in any combination. Thus an operator may take one array or one function as an operand (monadic); or two arrays, two functions, or a function and an array as operands (dyadic). It is the concept of an operator taking array operands that permits using / as both reduction and replicate (compress). In *APL2* / is an operator. It is not possible in *APL1* to tell if / is a function or an operator. No expression of the form array/array could give a different answer depending on the interpretation of array/ as a monadic operator with an array operand or array/array as a dyadic function.

The second control on operators is that the domain of any particular operator contains either arrays and/or functions or pairs of arrays and/or functions. It is this choice of domain that prohibits operators from being ambi-valent. When an operator is defined a semantic choice is made that determines the number of operands allowed. Context does not influence the decision. No operator is both monadic and dyadic.

The third control on operators is that the largest codomain is the set of all functions. Therefore operators return functions as their results and the functions so produced are called derived functions. Operators that produce arrays may be correct but are not included in *APL2*.

5.3.3 OPERATORS IN *APL2*

Not all possible primitive operators are defined and no general way to generate operators (derived operators) is provided. A mechanism to define the derived functions of user operators is provided.

The operators of *APL2* match those of *APL1* in concept but there are several important generalizations and one new operator.

The operators of *APL1* are extended in *APL2* so they apply to all dyadic functions -- primitive, derived, and defined. The new operator each (¨) transforms the concept of iteration into one of an array operation and applies to all functions -- monadic and dyadic.

The controls put on operators are strict and examining relaxations of the controls could lead to exciting new concepts. For example, currently there is no capability to deal with collections of functions as there is with collections of data.

Note that nothing in the language precludes the inclusion of other objects perhaps of higher binding power than operators but none have been included.

6.0 NAMES

In the previous section we defined *APL2* objects in the abstract. In this section we discuss how the objects are identified in a written notation.

A name is a string of one or more characters which is, or may be, associated with an *APL2* object. Some names are always associated with the same object, others may not be associated with objects at all or may be associated with different objects at different times.

Names are considered atomic, indivisible units of writing even when they take more than one character to represent.

6.1 PRIMITIVE NAMES

Primitive names are those that are defined as part of the definition of the language. They have fixed associations in that a given primitive name is always associated with the same object.

6.1.1 PRIMITIVE ARRAY NAMES

APL2 arrays are collections of numbers and characters. The primitive arrays (the ones given names) are single numbers and single characters (that is simple scalars).

Numeric scalars are written using their decimal representations. Complete rules for writing numbers may be found in [1]. Here are examples of various styles of numbers.

```
      245.5
```

is the name of a single numeric scalar. It is treated as an indivisible unit despite the fact that it occupies 5 print positions.

Negative numbers are written as positive numbers prefixed with a high bar.

```
      ¯245.5
```

Notice that, unlike conventional mathematics, the negative attribute of a number (¯) is distinguished from the subtraction operation (-).

Numbers may be represented in scaled form by specifying an integer power of ten scaling factor.

2.35*E*13

Complex numbers may be written as a real and imaginary part connected with a *J*

2*J*3
4.3*J*1*E*¯13

and in polar form with magnitude and angle expressed in degrees or radians.

2*D*45
1*R*1.716

While any rational number may be written, in an implementation not all are associated with a scalar object. For example, 2*E*987654 is a legal name for a number but is not associated with an object in most implementations because the number is not representable.

A given numeric object may be associated with many names. For example, the number "fifteen" can be written:

15
15.
15.0
1.5*E*1
15*J*0
etc.

Character scalars are written by enclosing the graphic associated with the character in single quotation marks.

'*A*'

This is a single character and is treated as an indivisible unit despite the fact that on input it occupies three print positions. The use of the quotes means it is always possible to distinguish between a number which is represented by a single digit and the character whose graphic is that digit.

2 cannot be confused with '2' in an expression

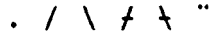
While the *APL*2 implementation allows for 2,147,483,647 different characters, not all may be written as constants. See [1] for the list of characters allowed in quotes.

6.1.2 PRIMITIVE OPERATION NAMES

Primitive operations are named by single symbols each of which occupies one print position.

There is a large set of primitive functions using the symbols:

There are only a few primitive operators using the symbols:



Note that dot (.) is an ambiguous symbol used as a decimal point in addition to its use as an operator. Which is intended in any instance is clear from context.

6.2 CONSTRUCTED NAMES

Constructed names are strings of one or more characters with the following constraints:

Initial or only character is from the set

*ABC...XYZ*Δ⎕
*ABC...XYZ*Δ

and remaining characters (if any) are from the set

*ABC...XYZ*Δ¯_
*ABC...XYZ*Δ
0123456789

6.2.1 USER NAMES

User names follow the above rules except that the initial character may not be ⎕. Any name constructed according to these rules is valid (no length limitation) and none has any value (i.e. none is associated with an object) until some action is taken to specify the association. User names may be associated with any class of *APL2* object.

Arrays and user names are associated through use of the specification arrow (←), through parameter substitution caused by invoking a defined operation, and as an implicit result of the ⎕*TF* function. A name which is associated with an array is called a variable. Thus a variable is said to be array valued. It is different from a constant array in that at different times it may have a different array as value.

Functions and operators are associated with user names as an implicit result of the ⎕*FX* and ⎕*TF* functions. Functions may also be associated with user names through parameter substitution in a

defined operator. Thus a user name may indicate the same function as a primitive function or even a derived function.

Facilities outside the language can also provide the associations (editors, system commands, etc.).

For historical reasons, the implementation gives special propeties to names beginning with $S\Delta$ and $T\Delta$. These names shall not be discussed in this paper.

6.2.2 DISTINGUISHED NAMES

Names which begin with the character ⎕ are reserved for fixed uses in the language. Any distinguished name is valid but only a few are associated with objects.

Distinguished names associated with arrays are called system variables. System variables are associated with new values through use of the specification arrow. They are shared variables and provide communication with the system or environment in which *APL2* is executed. Shared variables are not further discussed in this paper.

Distinguished names associated with functions are called system functions. They provide an alternative to the use of symbols for system related operations.

No distinguished names associated with operators are provided in *APL2*.

6.3 WRITING NAMES

Primitive operations are named by single symbols. When writing a linear sequence of names, a primitive operation name need never be separated from adjacent names. Thus the two names + and - written next to each other

```
+-
```

can never be confused with a single different operation. All other names may require more than one symbol. When writing a linear sequence of names, these names, if adjacent, must be separated to avoid confusing the combination with a single different name. Thus the two names 12 and 34, when written next to each other, must be separated to avoid confusion with the name 1234. The separation character is a blank if no other nonblank character falls between them. For example

```
12 34  blank needed for separation
12(34) blank not needed for separation
```

In later sections when substitution is discussed, removal of a redundant parenthesis may imply the insertion of a blank to prevent two names from becoming one new name. For example:

```
12(34)  becomes 12 34
        not     1234
```

6.4 DISPLAYING NAMES

When the character representation of expressions is produced (because of an error or because of execution of ⎕*CR* or ⎕*TF*), names are always displayed as they were entered. This is different from *APL*1 only in the case of constant array names. For example, entering 2.5000 causes a display of the representation of the value 2.5:

```
      2.5000
2.5
```

However, if the same name is used in an expression containing an error, the original name, not a representation of the value, is displayed:

```
      2.5000×
SYNTAX ERROR
      2.5000×
            ∧
```

The assumption is that a user chooses how he enters a number for some reason and the representation should not be altered. In the above example he may know the value to five digits precision. As a second example consider a defined function containing the following expression:

```
DFN[3] PI←3.14159265358979323846
```

Here, even if the particular implementation could not handle this much precision, the function could be transferred to some other implementation that could represent *PI* without loss of precision.

6.5 GLOBAL AND LOCAL NAMES

The set of names that are defined outside the context of the evaluation of defined operations are called global names. When a defined operation is evaluated, names are defined that are associated only with this evaluation. These names are called local names. When the defined operation completes evaluation, the values associated with local names are discarded and values associated with the names prior to evaluating the defined

operation are restored. When a local and a global share the same name, the global value is said to be shadowed by the local value.

The concept of local and global does not affect the definition of any primitive operation. Evaluation of expressions is affected only in that any reference to a name yields its most local value. Therefore local names will not be considered further.

7.0 SYNTAX

This section and following sections show the derivation of the definition of syntax for *APL2*. Appendix 1 includes a summary of the rules.

The syntax of *APL* is simple, straightforward, and easy to learn. This is so because of the great care exercised by the creators of *APL1*. Similar care is required in making any extensions or changes to syntax. With the exception of the removal of mixed output, the syntax has been unchanged since the early days of the language. Therefore extensions to syntax are probably the most constrained by *APL1*. The resulting syntax must retain at least the following properties:

- It is linear - we do not want to introduce anything that cannot be written on a line (like superscripts, subscripts, radical signs, and so forth).
- Primitive operations are represented by single symbols.
- It uses a function symbol for two (usually related) functions - one monadic and one dyadic - that is primitive functions are ambi-valent.
- No functional precedence - all functions have equal precedence and execute according to their position in an expression.
- Operators have higher binding power than functions.

The syntax of *APL2* must be able to express:

- arrays,
- functions and their application to arguments, and
- operators and their application to operands.

The linear collection of special symbols and names (primitive and constructed) used to write arrays, functions and their application to arguments, and operators and their application to operands, is called an expression.

The names and symbols used to write an expression are divided into six syntax classes:

- array
- function
- monadic operator

- dyadic operator
- assignment arrow
- brackets

(Note that the object class operator is divided into two syntax classes; brackets and their contents are treated as one class; ∘ and → are treated like functions.) To these classes are added parentheses -- the only punctuation symbols in an expression.

Evaluation of an expression may produce any of the three objects or may produce no object at all and be correct (although an attempt to display or assign the result of an expression that produces a function or operator generates an error).

Expressions are classified by the object they produce:

- array expression: one that evaluates to an array
- function expression: one that evaluates to a function
- operator expression: one that evaluates to an operator.
- valueless expression: one that evaluates to no object.

With these thoughts in mind we examine how each kind of expression may be written. First only the simplest forms are explored and in particular parentheses are not introduced until later. Formation and evaluation of expressions are approached by examining the binding strengths of the objects. The concept of binding strength brings together in one measure all the concepts of syntax -- order of execution, precedence of operators over functions, building lists of arrays, etc.

Evaluation involves scanning the names (in a strictly right to left order), determining binding strengths of objects next to each other, and evaluating operations whenever they are completely determined.

Thus the fundamental concept of syntax is that of adjacency or juxtaposition and its use for the most important actions: forming of vectors, applying functions to arguments, and applying operators to operands.

In the following we examine binding strengths of various combinations of objects. The objective is to arrive at a simple linear hierarchy that is easy to use in practice to parse expressions. Bindings are chosen so that useful expressions can be written without parentheses. Parentheses are then introduced as a way to delay certain bindings.

7.1 EXPRESSIONS WITHOUT PARENTHESES

First, we investigate how to write arrays, functions, and operators and discover the bindings implied when symbols and names of objects are placed next to each other.

7.1.1 VECTOR EXPRESSIONS

There is one rule for writing a simple vector: write the simple scalars which are the items of the simple vector next to each other with separating blanks as needed. Notice that, because the rule involves a separation of items, the resulting vector must have at least two items.

Here are three examples of simple constant vectors. The first is all numeric, the second is a mixture of numbers and characters, and the third is all character:

```
      2 3 4
      2 'B' 4
'A' 'B' 'C'
```

The last example is a different way of writing a simple character vector than provided in *APL*1.

This is the first extension to syntax. It is a simplification. There is now one rule for writing a vector: write the scalar items separated by spaces. This may be generalized by saying that when two arrays are written next to each other, a binding exists between them. Thus if *I* and *J* are arrays, writing them next to each other implies construction of a vector containing them as items. In the following this is called vector binding. Later a rewriting rule is presented that gives a compatible way to write a character vector.

7.1.2 ARRAY EXPRESSIONS

Given that we can write some arrays we may now consider how we write functions and apply them to arrays. The rule is the same as in *APL*1: a function symbol may represent two functions - one monadic (one argument or valence 1) and one dyadic (two arguments or valence 2). A monadic function is written with its single argument on the right and a dyadic function is written with arguments on the left and the right (infix notation).

```
monadic function      ÷2
 dyadic function    5÷2
                      24
```

There is only one reasonable way to interpret these expressions. It could be argued that if ÷2 is a monadic function that 5÷2 is the number 5 sitting to the left of a monadic function. This is even easier to argue if instead of ÷ we use a symbol which does not have a dyadic definition. For example the symbol used for enclose (⊂) has not been given a dyadic meaning. One could argue that 2⊂3 is really a 2 next to a monadic function. *APL* solves this possible ambiguity with the rule:

> All functions are ambi-valent (both valences)
> and the one evaluated in any instance is determined
> only by context.

Thus functions in the abstract are ambi-valent but at evaluation time (call time) the syntax uniquely determines which function is intended. If you write a function symbol with an argument on each side, you have written a dyadic function. If it has a meaning, it is evaluated and otherwise it is an error. In the case of ⊂ if this should ever be given a dyadic meaning, it will not be considered a change to the syntax of *APL* it is a change to the semantics. This is why in *APL2* attempting to execute such an expression does not give *SYNTAX ERROR*. It gives *VALENCE ERROR*, meaning that the function is not defined for the given number of arguments.

In the same sense that arrays written next to each other have vector binding, writing arrays next to functions have argument binding. In the following this is called left argument binding and right argument binding.

When an expression is written containing more than one function, rules for determining which is to be evaluated first must be given. In the expression:

```
2×3+4
```

which is done first -- the multiplication or the addition? Another way of phrasing this question is: "Which gets bound to the 3? × or +?" This can be answered many ways all of which lead to the same result (because we know what we want to reach). We want all functions to be the same syntactically and we want precedence to be positional with the rightmost function whose arguments are available to be evaluated first. Here it is most convenient to phrase the rule in terms of binding. We want 3 to be the left argument of + not the right argument of × so that the rightmost function is evaluated first. Therefore we declare that left argument binding is stronger than right argument binding.

Binding strength (strongest on top)

> left argument
> right argument

Thus, the above expression means 3+4, then 2× the result. This is equivalent to the right-to-left rule:

In an unparenthesized expression without operators, functions are evaluated right to left.

By this rule, the rightmost function is evaluated first with its explicit result becoming the right argument of the next function or the value of the expression if there is no next function.

The next question to answer is: Where does vector binding fit in with argument binding. There are three choices: below right argument binding, above left argument binding or between them. Beginners in *APL*, not being told otherwise, often assume that vector binding is lower than right argument binding so that in the expression:

```
2×3  4+5   (extra spaces for emphasis)
```

times binds it right argument 3 and plus its left argument 4 getting two results 6 and 9 and that then these are bound giving the two item vector 6 9. There is absolutely nothing wrong with this analysis except that *APL* chooses to put vector binding higher than argument binding. Thus *APL2* has the following hierarchy

Binding strength

vector
left argument
right argument

In the above example 3 is bound to 4 first and then the pair is bound to + as its left argument. It is this choice that gives *APL2* its array processing capabilities. The fundamental data in *APL2* is arrays. We therefore make it easy to construct arrays and apply functions to them.

7.1.3 FUNCTION EXPRESSIONS

Without operators the only function expression that can be written is one which contains only the name of a function. Thus:

```
×
```

is a syntactically correct function expression. It means we are talking about the function itself as opposed to its application to arguments. Therefore the above expression results in the function "times". Although it is an error to attempt to display or assign this result, in the future even this could be allowed and would not be an extension of syntax. Without these extensions, function expressions are useful only in expressions containing operators. The reason for allowing function expressions becomes clear after parentheses are discussed.

Operators can be used to write other function expressions, in which case the function result is called a derived function.

The syntax of operators is in many aspects the mirror image of the syntax of functions. A monadic operator is written with its single operand on the left

```
+/     for / a monadic operator
```

A dyadic operator is written with its operands on the left and the right.

```
+.×    for . a dyadic operator
```

Each of these evaluates to a derived function and so is a valid function expression. As before the attempt to display the derived function generates an error.

Operators differ from functions (even in mirror image) in that they are not ambi-valent. A particular operator is either monadic or dyadic but never both. This is why operators are represented by two syntax classes.

*APL*1 only allows function expressions consisting of a single operator applied to scalar functions. *APL*2 permits the operand of an operator to be any function -- even the function which results from the application of another operator (that is its derived function). We therefore have to answer the question: "In the following function expression, which operator is evaluated first?"

```
+.×/
```

This could be an inner product between + and ×/ or it could be a reduction by an inner product. The question is further complicated by the possibility of array operands.

As with functions the answer can be approached by specifying the binding strengths of operators to their operands. Unlike binding of arguments to functions, *APL*1 gives no help with determining what the binding of operands to operators should be. Either ranking of left operand binding versus right operand binding is correct. Since the operands are presented in the mirror image of functions, we choose binding strengths in the mirror image. Thus we stipulate the following:

Binding strength

```
right operand
left operand
```

with the understanding that monadic operators have no binding strength on the right at all. Therefore the conclusion is that in the expression

```
+.×/
```

the right binding strength of . is stronger than the left binding strength of / and the expression is a reduction by an inner product.

The next question to be addressed is: "Where does operand binding fit with argument binding?" The answer is entirely arbitrary but is guided by *APL1* and designed to make using operators as simple as possible. In the expression

```
A+.×B
```

we do not want ×*B* to be computed and so we say that right operand binding is higher than left argument binding. (We also did not allow functions to take operators as left arguments when controls were imposed.) This gives:

Binding strength

right operand
left argument
right argument

Left operand binding could go in any of three places (since it is below right operand binding) but since we are not trying to express the sum of A with anything we make left operand binding higher than right argument binding. Because no object is both a function and an operator, the ordering of left argument and left operand does not matter. Therefore the binding hierarchy for functions and operators is defined as:

Binding strength

right operand
left operand
left argument
right argument

It is in this sense that operators have higher precedence than functions; they have stronger bindings.

With the binding of operands placed in the hierarchy, we can say why it is not desirable for operators to be ambi-valent.

Suppose that operators were ambi-valent in a system with the same binding hierarchy and consider the expression:

```
+/A÷B
```

If operators were ambi-valent, then because right operand binding is higher than left argument binding, / would get *A* as its right operand and the derived function would apply to ÷*B*. The monadic use of / (or any other monadic operator) would require parentheses to limit the binding. This is not wrong but is not compatible with *APL1* and would make the use of operators more difficult. The remedy is to stipulate that operators are not ambi-valent and that / is strictly monadic. Of course, the

derived function from any operator is still a function and all functions are ambi-valent. Therefore a strictly monadic operator may produce both a monadic and a dyadic function. For example:

```
+/B
A+/B
```

are both syntactically correct and are respectively the monadic and dyadic use of the derived function +/.

The next question to be answered is: "Where in the binding hierarchy does vector binding belong? At the moment we only know that it is above left argument binding. Therefore it could go in any of three positions:

Binding strength

```
                ←----
right operand
                ←----
left operand
                ←----
left argument
right argument
```

Because / is an operator, compatibility with *APL*1 requires that vector binding be higher than left operand binding. In the expression

```
1 0 1/A
```

we want the vector formed before the left operand of / is bound. Therefore vector binding must be stronger than left operand binding. This reduces the possibilities to:

Binding strength

```
                ←----
right operand
                ←----
left operand
left argument
right argument
```

Either of these positions is correct and both were tried experimentally in the *APL*2 IUP (which did not allow array left operands). The question is exemplified by the following expression using a dyadic defined operator DOP because there is no primitive dyadic operator that takes an array right operand:

```
+ DOP A B
```

If vector binding is above right operand binding, this is a function expression with *A B* as the right operand. If vector binding is below right operand, this is an array expression which applies the derived function + *DOP A* to argument *B*. This second

choice makes operators with array right operands easier to use because otherwise parentheses are always needed. This is the order chosen.

Therefore the binding hierarchy for functions, operators, and vectors is:

Binding strength

right operand
vector
left operand
left argument
right argument

7.1.4 OPERATOR EXPRESSIONS

The only operator expressions are a single operator name or a single operator name to the left of brackets. (Brackets are discussed separately.)

7.1.5 VALUELESS EXPRESSIONS

User-defined functions that do not return explicit results may be written. The only valueless expressions that can be written involve such a user-defined function, the primitive function execute (⍎) whose evaluation includes such a function, or an empty expression such as

```
      L1:    ⍝ EMPTY EXPRESSION
```

7.2 EXPRESSIONS WITH PARENTHESES

In *APL1* parentheses are used only to group functions with their arguments. In *APL2* there is the need to express other groupings (for example, grouping an operator with its operands). Rather than use a new pair of grouping symbols, a new simplified parentheses rule is adopted. This rule is:

Parentheses are used for grouping.

They may be used anywhere as long as they are properly paired and what is inside the pair evaluates to an array, a function, or an operator. An expression inside parentheses (or one which could be put in parentheses without changing the evaluation of anything) is called a subexpression.

Evaluating expressions with parentheses is only a matter of evaluating what is inside the parentheses and then substituting for the parenthesized expression the value it produces.

Some parentheses that are correct can be removed from an expression without affecting the result of the expression. Correct parentheses that don't delay any bindings are called redundant parentheses. In particular, parentheses surrounding a name of an object (primitive or constructed) or a parenthesized expression do not have the action of grouping and are always redundant. Here are examples of parentheses redundant by this rule:

```
2(+)3          Constant operation name
A +(.)× B      Constant operation name
(A)←3          Constructed name
(2)+1          Constant array name
((2-3))+1      Parenthesized expression
```

Here is an example of parentheses that seem redundant by this rule but are not:

```
(NDFN)         niladic function without result
```

These parentheses are not correct because what is inside does not evaluate to an array, a function, or an operator.

Redundant parentheses may be added to or removed from expressions freely without changing the value of the expressions.

Additional rules for when parentheses may be removed are given in the following sections. The effect is to say that parentheses which do not delay any bindings are redundant.

7.2.1 VECTOR EXPRESSIONS IN PARENTHESES

In expressions of arrays, parentheses that do not separate a group from another part of the expression are redundant. Here are some examples of redundant parentheses:

```
2 (3) 4        These do not group.
(2 3 4)        These group but do not separate.
```

Notice that in each case what is inside the parentheses is a correct *APL2* expression.

Nonredundant uses of parentheses in vector expressions give a facility for writing nested vectors. For example consider:

```
2 (3 4)
```

What is inside the parentheses is a valid *APL2* expression and so the parentheses are correct. Evaluating what is inside the

parentheses gives us an array (a two-item vector). Vector binding tells us that writing 2 next to an array gives us a vector. Thus parentheses may be used to write nested vectors. This is called vector notation in *APL2* and strand notation by others[Mo1]. It is seen as a consequence of the simplified parentheses rule. Vector notation is an extension of the concept of a numeric vector constant in the sense that numeric constants are covered by the rules for vector notation.

Now that we have array expressions with parentheses we can state the rewriting rule that permits the *APL1* style of simple character vectors. The rule is:

> If a vector in parentheses is made up entirely of single characters, it may be rewritten with a single pair of enclosing quotes.

The parentheses must be part of the rule even though they appear redundant. Thus in the following example even though *'B' 'C'* is made up entirely of single characters, the rewriting rule may not be applied.

'A' 'B' 'C' is not *'A' 'BC'*

The following is a correct application of the rule:

('A' 'B' 'C')	is rewritten	*('ABC')*	Rewriting rule
('ABC')	is rewritten	*'ABC'*	Remove redundant parentheses

This gives a compact way of writing character vectors that is compatible with *APL1*.

7.2.2 ARRAY EXPRESSIONS IN PARENTHESES

Parentheses in any expression may be used to delay the binding of arguments to functions. Therefore if it is desired to add 2 to 3 then divide the result by 4 we may delay the binding of divide to its left argument by using parentheses:

```
(2+3)÷4
```

and this causes 3 and + to be bound even though the left argument binding of ÷ is stronger.

Parentheses in array expressions are redundant if they group the right argument of a function or a vector left argument of a function.

```
2×(3÷4)   Group right argument.
(2 3)×4   Group vector left argument.
```

7.2.3 FUNCTION EXPRESSIONS IN PARENTHESES

Parentheses in any expression may be used to delay the binding of operands to operators. To express an outer product where the function applied is an inner product, we write:

```
      ∘.(+.×)
```

This causes + to be bound to the dot on its right even though the right operand binding of the other dot is stronger.

Parentheses in function expressions are redundant if they group the left operand of an operator.

```
      (+.×)/   Group left operand.
```

Parentheses around a function expression are redundant if the left parenthesis does not separate two arrays.

```
      A (+.×) B  Group function expression.
```

However, the following parentheses are not redundant because the left parenthesis separates arrays.

```
      A (B/) C  Required parentheses
```

Note that (*B/*) evaluates to an ambi-valent function which, in this case, is dyadic because it has two arguments *A* and *C*. This results in a *VALENCE ERROR* because *B/* has no dyadic meaning in *APL2*.

7.2.4 OPERATOR EXPRESSIONS IN PARENTHESES

It is not possible to write an operator expression that uses non-required parentheses. Even in an operator expression involving brackets, parentheses are redundant. (Brackets are discussed separately.) Thus in any syntactically valid operator expression, parentheses are redundant.

7.2.5 VALUELESS EXPRESSIONS IN PARENTHESES

A valueless expression may not be a subexpression (that is it may not be within parentheses). Writing a valueless expression in parentheses results in a *VALUE ERROR*.

7.3 STATEMENTS

An *APL2* statement is made up of three parts each of which is optional: a label followed by a colon; an expression; and a comment. A defined operation may validly contain a statement having none of the optional parts. The canonical representation of such an operation will have a blank row. In immediate execution mode (see below), a blank or empty line may be thought of as an execution of an empty expression.

A label is a user name which, if present, must be followed by a colon. The expression is any *APL2* expression. A comment is a string of characters whose first nonblank character is the lamp symbol (⍝).

The statement is the executable unit of work. Expressions presented for evaluation as independent lines are said to be executed in immediate execution mode. Statements selected from a defined operation for evaluation are said to be executed in function execution mode. In addition implementations normally have a definition mode where statements are collected into defined operations for later execution. This will not be considered further here.

7.3.1 IMMEDIATE EXECUTION MODE

In immediate execution mode labels and comments are ignored and only the expression is evaluated.

7.3.2 FUNCTION EXECUTION MODE

In function execution mode, labels and comments are also ignored at the time a statement is selected for evaluation. However, at the time that a defined operation begins execution, each label becomes a local name associated with a scalar whose value matches the line number of the statement in which the label appears.

8.0 THE SUBSTITUTION PRINCIPLES

The notion of substitution is one of the most primitive notions that people learn. In the abstract, the substitution rule says that equals may be substituted for equals. In practice this is much too general to apply. If we substitute 4 for 2+2, we do have equal values but not equal statements - one is a constant the other is the application of a dyadic function. We must specify what it is that we may substitute and what remains the same after we do it.

APL2 has two substitution rules. One preserves values (expression substitution) and the other preserves syntax (syntactic substitution). The rules are intimately connected with the parentheses rules. They involve the insertion of redundant parentheses followed by the treatment of what is within the parentheses as an expression or value for which a substitute may be supplied.

In this section, applications of the substitution rules are written with equivalent expressions arranged vertically with the reason for the rewriting on the right.

8.1 EXPRESSION SUBSTITUTION

The rule of expression substitution covers replacing one expression with another which, when evaluated, produces the same object (array, function, or operator). In general, the syntax of the expression will change -- there will be different numbers of arrays, functions, and operators.

> The Rule of Expression Substitution
>
> In an expression, any subexpression may be replaced by another expression that computes the same value without changing the value of the original expression.

For example:

```
(¯1↓⍳4)-1   may be written
(⍳3)-1      expression substitution
```

The new expression has fewer functions but evaluates to the same value as the original.

Because a single array is the limiting case of an array expression the following corollary holds:

Corollary 1: Array Substitution

In an expression, any array in parentheses may be replaced by another array or array expression having the same value as the original array without changing the value of the original expression.

Conversely,

In an expression, any array expression in parentheses may be replaced by an array that has the same value.

This means, for instance, that a constant may be replaced by a variable having the same value. Here is an example of an array being substituted for an array:

If *A*←2 3 4

```
2 3 4,5      is written
(2 3 4),5    add redundant parentheses
(A),5        array substitution
A,5          remove redundant parentheses
```

Here are examples of expressions being substituted for arrays.

```
2+3          is written
(2)+3        add redundant parentheses
(4÷2)+3      array substitution
```

```
2+3          is written
2+(3)        add redundant parentheses
2+(4-1)      array substitution
2+4-1        remove redundant parentheses
```

Examples of the converse are obtained by reading the above examples from bottom to top.

Because arrays contain arrays as items, the following corollary holds:

Corollary 2: Item Substitution

In an expression, any item of an array enclosed in parentheses may be replaced by another array or array expression having the same value as the original item without changing the value of the original expression.

Conversely,

in an expression, any array expression that is an item of an array may be replaced by the array having the same value without changing the value of the original expression.

If *A*←2 3 4

```
(2 3 4) (5 6)   is written
(A) (5 6)       item substitution
A (5 6)         remove redundant parentheses
```

If *B*←'*ABC*'

```
B (5 6)          is rewritten
(B)(5 6)         redundant parentheses
('ABC') (5 6)    item substitution
'ABC' (5 6)      remove redundant parentheses
```

```
2 3 4       is written
2 (3) 4     add redundant parentheses
2 (2+1) 4   array substitution
```

Examples of the converse are obtained by reading the above examples from bottom to top.

All the examples given are for array-valued objects because those are the most useful cases. In general, there are no useful alternate ways to write function expressions. If there were a primitive *APL2* function called summation, we could do substitution between it and +/. If *APL2* ever allows functional specification as in

CARTESIAN← ∘.,/

Then we could give examples. The rule would not need to be changed to cover this case.

8.2 SYNTACTIC SUBSTITUTION

The rule of syntactic substitution covers substitutions that can be made that do not alter the syntax of an expression. There will be the same number of arrays, functions, and operators before and after the substitution. The parts of the expression not touched never become different in meaning, binding strength, valence, value, and so forth. No dyadic function will become monadic, no vector will become shorter or longer. However the value of the expression will generally be different.

> The Rule of Syntactic Substitution
>
> In an expression, any object in parentheses may be replaced by an object of the same class without changing the syntax of the statement.

(Recall that monadic and dyadic operators are different syntax classes.)

Operator example:

```
2+,/A      may be written
2+,(/)A    add redundant parentheses
2+,(¨)A    syntactic substitution
2+,¨A      remove redundant parentheses
```

and the syntax is unchanged. There is still one monadic operator producing a monadic function and one dyadic function.

Function example:

```
2+3×4      may be written
2(+)3×4    add redundant parentheses
2(*)3×4    syntactic substitution
2*3×4      remove redundant parentheses
```

and the syntax is unchanged. In order for syntactic substitution to hold, every object in a given syntactic class must have equal precedence. Therefore the rule for syntactic substitution implies the following property of functions in *APL2*:

> Functions in an expression have no precedence. The order of execution depends only on position in the expression.

Replacing one function with another can never change the fundamental syntax of an expression. (This is not true in standard arithmetic notation.) Only the evaluation of the expression changes and only at the point where the substitution was made.

Array examples:

```
2 3 4 5,B       is written
2 (3) 4 5,B     add redundant parentheses
2 (9) 4 5,B     syntactic substitution
2 9 4 5,B       remove redundant parentheses
```

```
2 3 4 5,B       is written
2 (3) 4 5,B     add redundant parentheses
2 ('A') 4 5,B   syntactic substitution
2 'A' 4 5,B     remove redundant parentheses
```

and the syntax is unchanged. The left argument to catenate is still a four-item vector.

8.3 THE SIGNIFICANCE OF SUBSTITUTION

The substitution rules are intimately tied up with the nature of *APL2*.

Expression substitution is the process used by an implementation (and by readers of *APL* code) to evaluate expressions.

Syntactic substitution expresses the fundamental precedence of operations that has always been a characteristic of *APL*. Functions have no precedence so one may substitute one function for another function without affecting the syntax of the expression. This is not true of traditional mathematics or of most programming languages.

Each of the substitution rules simplifies how one looks at and deals with *APL* expressions.

9.0 BRACKETS

Brackets are a special syntactic construction for writing lists of arrays for use in indexing and axis specifications. They are correct if correctly paired and if what is inside is one of the following:

- Nothing [ ]
- An array expression [1] [2+2]
- More than one of the above separated by semicolons [;] [1;] [1;2 3 4]

Brackets are used for two different purposes: indexing and axis specification. In each case evaluating a bracket expression is a substitution in that brackets to the right of an array (indexing) produces an array, and brackets to the right of a function or operator (axis specification) yield a function or operator respectively.

9.1 INDEXING

Brackets indicate an indexing function when written to the right of an array expression (a single name or an expression in parentheses).

```
A[2]
(matrix expression)[3;]
```

Such constructions are always syntactically correct but there are domain restrictions implied by the semantics of brackets. Namely the rank of the array indexed must equal 1 plus the number of semicolons inside the brackets. The consequences of this are that brackets cannot be used to index a scalar and cannot be used to the right of an expression that at different times produces an array of different rank.

Bracket indexing is a function in that it maps an array and some indices (which may be considered a vector of arrays) to a new array (i.e. it is a dyadic function). It fails to be like other *APL2* functions because more than one symbol is needed to write the function. Even though it is dyadic it does not use infix notation. It may not be used as the operand to an operator. There appears to be no way to introduce a related monadic function. And the same symbols when used for axis specification do not represent a function.

9.2 AXIS SPECIFICATION

Brackets indicate an axis specification when written to the right of a function or operator expression (a single name or an expression in parentheses).

```
      ⌽[1]
```

The brackets are considered to be a notation for writing an operation related to the one on its left. It cannot be considered an operator because the definition of the related function cannot be expressed, in a uniform way, in terms of the original function.

Writing the brackets next to a function or operator is always syntactically correct but evaluation of the related function or operator succeeds only under specific conditions. An *AXIS ERROR* is generated when the conditions are not met. The conditions are:

- The bracket expression must contain no semicolons.
- If the related function is used monadically, the original function must be one of ⊃ ⊂ , ⌽ ⊖.
- If the related function is used dyadically, the original function must be one of ⌽ ⊖ , ↑ ↓ and the scalar functions.
- If the related operator is monadic, the original operator must be one of /⌿\⍀.

The primitive functions mentioned above may be written as primitive symbols or as user names having the primitive operation as value (because of parameter substitution in a defined operator).

Here are examples of incorrect axis specifications:

```
      2↑[2;3]A
```

gives an *AXIS ERROR* because the bracket expression contains semicolons; and

```
      ⍳[3]A
```

gives an *AXIS ERROR* because ⍳ is not one of the monadic functions mentioned above.

```
      ⍴¨[1] A
```

gives *AXIS ERROR* because ¨ is not one of the monadic operators mentioned above. The reason why the brackets are not treated as applying to the derived function ⍴¨ is presented in the next section.

Evaluation of the related function could yield many error conditions including *AXIS ERROR* for other reasons. For example:

```
      ϕ[5] 2 3 4
```

is allowed by the conditions but gives an *AXIS ERROR* because 5 does not indicate an axis of the argument array.

When an operation can be written with an axis specification, there is always a choice of axes which gives the same result as the function without axes.

For example:

L↑R ↔ *L↑[⍳⍴⍴R] R*

9.3 BINDING STRENGTH

Brackets are not an array, a function, or an operator. They are treated as members of a special syntactic class. We must, therefore, make an individual assessment of where they fall in the binding hierarchy. The following example shows that there is a choice. Let *DOP* be a dyadic operator:

```
      + DOP ϕ[1]
```

If right operand binding is higher than bracket binding, this must mean

```
      (+ DOP ϕ)[1]
```

which gives an *AXIS ERROR* because the rules do not include any valid use of brackets with a derived function. If bracket binding is higher than right operand binding this must mean

```
      + DOP (ϕ[1])
```

which is a legal function expression. Neither choice is more formally correct. The second option lets us write a useful expression without parentheses and is the option chosen in *APL2*. As usual, parentheses may be used to delay binding but no useful expression can be so produced.

If brackets have stronger binding than right operands then ,if we are to maintain the simple linear hierarchy, their binding is stronger than any other binding giving the following binding hierarchy:

Binding strength

brackets
right operand
vector
left operand
left argument
right argument

This implies that in the expression

```
+/[1]A
```

the brackets bind to the operator / producing a new monadic operator which binds to + as its left operand.

A useful way to phrase the binding strength of brackets is to say that "Brackets are tightly bound to the object on their left." For example:

```
A +.×[2] B
```

expresses an inner product with operands + and ×[2]. If *A*, *B*, and *C* are vector arrays, then

```
A[1] B[2] C[3]
```

expresses the three item vector whose first item is *A*[1] and whose second item is *B*[2] and whose third item is *C*[3].

```
A B C[2]
```

is a three item vector whose first item is *A*, whose second item is *B*, and whose third item is *C*[2]. Application of expression substitution in the above example, shows that

```
2 3 4[2] ↔ 2 3 (4[2])
```

Which is a *RANK ERROR*. Such constant vectors are viewed as expressions containing the names of three scalars. This is different from *APL*1. Indexing of a constant numeric vector requires parentheses. (Note that)*MCOPY* (Migration *COPY*) and)*IN* make this change in defined functions migrated from *APL*1.)

The practical effect of this placement of brackets in the hierarchy is that brackets become syntactically transparent. Whenever brackets are seen in an expression (for indexing or axis specification) they bind tightly to whatever is on the left and the combination may be immediately evaluated and replaced by the computed value from the same class. This is why brackets and their contents may be treated as a single syntax class. Parentheses around brackets and the object to their left don't delay any bindings and are always redundant.

Brackets, which have always been an exceptional case in *APL*1 (sometimes described as a function and sometimes as an operator), are now regularized and explained.

9.4 EVALUATION PATTERNS FOR AXIS SPECIFICATIONS

Even though axis specifications do not follow one uniform rule, there are, none the less, guiding principles for definitions of functions with axis.

Those uses of axis specification from *APL*1 are preserved and new ones obey identities or follow a single predictable pattern.

The pattern consists of splitting an array into subarrays along some axes, applying a function to each subarray, then gathering the results together into an array.

The function *SPLIT* in Appendix 2 takes as left argument the axes along which the right argument is to be split. The function *UNSPLIT* takes as left argument the result axes into which the items of *R* will be merged. *UNSPLIT* is the left inverse of *SPLIT*.

Most functions which have an axis specification may be described in terms of *SPLIT* and *UNSPLIT*. The function enclose with axis (⊂[*X*]) is the simplest of these and is defined as follows:

$$\subset[X]\ R \leftrightarrow X\ SPLIT\ R$$

Disclose with axis is defined as the left inverse of enclose with axis.

$$\supset[X]\ R \leftrightarrow X\ UNSPLIT\ R$$

giving the identity

$$R \leftrightarrow \supset[X]\ \subset[X]\ R$$

By extension disclose with axis accepts an argument where not all the items have the same shape. This extension is not defined in this paper and is not used in the equations below.

In the following discussion, other functions with axis are defined by using enclose with axis to split arrays, applying the function under discussion without an axis to each item of the split array (using the each operator), then reassembling the result with disclose with axis. This form fails to be a universal formula because the way results are reassembled depends on the shape of arrays produced from application of the function to the items of the split arrays. Furthermore, for the dyadic scalar functions, the argument split depends on the relative ranks of the arguments.

The monadic functions with axis are defined as follows:

```
φ[X]R ↔ ⊃[X] φ¨ ⊂[X]R
⊖[X]R ↔ ⊃[X] ⊖¨ ⊂[X]R
,[X]R ↔ ⊃[⌊/X] ,¨⊂[X]R for non-fractional X
,[X]R ↔ ⊃,¨⊂[X]R       for empty X
```

A single expression can be written for ravel with axis but the two above are easier to apply.

The monadic operators with axis are defined as follows:

```
F⌿[X]R ↔ ⊃[X] F⌿¨ ⊂[X]R for F array or function
F/[X]R ↔ ⊃[X] F/¨ ⊂[X]R for F array
F/[X]R ↔      F/¨ ⊂[X]R for F function
```

For dyadic functions with axis, one argument is generally treated as a control to be applied to each of the subarrays of the other argument. Therefore, in the formula, the control argument is enclosed:

```
  L ↑[X]R ↔ ⊃[X] (⊂L) ↑¨ ⊂[X]R
  L ↓[X]R ↔ ⊃[X] (⊂L) ↓¨ ⊂[X]R
L F/[X]R ↔ ⊃[X] (⊂L) F/¨ ⊂[X]R
```

The scalar functions with axis follow the above pattern if the left argument is of lower rank. Otherwise the following formula is used:

```
L DF[X]R ↔ (⊂[X]L) DF¨ ⊂R
```

Catenate has a slightly different pattern where the array of lower rank is not enclosed:

```
L,[X]R ↔ ⊃[X](⊂[X] L),¨R for R lower rank
L,[X]R ↔ ⊃[X] L ,¨⊂[X]R  for L lower rank
```

Rotate has a pattern similar to catenate:

```
L⌽[X]R ↔ ⊃[X] L ⌽¨ ⊂[X] R
L ⊖[X]R ↔ ⊃[X] (⊂L) ⊖¨ ⊂[X]R
```

These patterns are retained for compatibility with *APL*1.

Catenate and ravel with fractional axis specification are related by the following identity:

```
L,[X] R ↔ (,[X]L),[⌈X] ,[X]R
  for scalar fractional X
```

It is possible to define these patterns or any others in a defined operator. Appendix 2 gives an example of a *RANK* operator similar to the one defined by Iverson [Iv3]. Because these operators would not make use of bracket notation, they could be defined uniformly on all functions. Any such defined operator which proved to be especially useful could be considered the definition of some new primitive axis operator.

10.0 OTHER SPECIAL SYMBOLS

APL2 includes the use of several special symbols that do not represent arrays, functions, or operators. These are parentheses, brackets, semicolons, right and left arrows, and jot. Parentheses, brackets, and semicolons have been treated previously.

10.1 ASSIGNMENT

The assignment arrow (←) is the only syntactic construction for associating names with arrays. There are two kinds of assignment: one which associates a name (perhaps with no value) with an arbitrary array (direct assignment) and one which merges an array into indicated positions in another array already associated with a name (selective assignment). In each case one parameter is an array and the other is either a name or positions in a named array. Therefore the assignment arrow can be neither a function nor an operator (since these operate on values not names). The assignment arrow is in a separate syntactic class.

The name whose value is replaced or modified must be a constructed name having no value or having an array value. This, in particular, excludes names of niladic defined functions which are otherwise treated syntactically as arrays.

10.1.1 ASSIGNMENT SYNTAX

To fit assignment into the binding hierarchy, we must consider the relative strengths with which a left arrow binds with what is on its left and what is on its right. *APL1* answer both these questions.

Consider the expression:

```
A←2+3
```

Clearly left argument binding must be stronger than assignment right binding so that the addition is done before the assignment. Assignment right binding must therefore be placed either just above or just below right argument binding. Because the left arrow cannot be a function, the order is immaterial. We therefore select to place assignment right binding as lowest, giving the following binding hierarchy:

Binding strength

brackets
right operand
vector
left operand
left argument
right argument
assignment right

*APL*1 only helps a little in determining assignment left binding. The expression:

```
2+A←3
```

shows that assignment left binding is stronger than right argument binding. Because *APL*1 did not have operators with array operands, we may choose how much stronger than right argument binding it is.

Consider the following expression where *DOP* is a dyadic operator with array right operand:

```
+ DOP A←3
```

If right operand binding is stronger than assignment left binding then this means:

```
(+ DOP A)←3
```

which is an error. If assignment left binding is stronger than right operand binding, this means:

```
+ DOP (A←3)
```

which is a legal function expression. This is the choice made in *APL*2 giving the hierarchy:

Binding strength

brackets
assignment left
right operand
vector
left operand
left argument
right argument
assignment right

(Because brackets do not bind on the right at all, assignment left could have been put at the top.)

This choice of assignment left binding has the practical effect of tight binding a left arrow to the thing on its left. Thus an assignment can always be immediately evaluated and replaced by

its value (which is always the array on its right) making assignments syntactically transparent.

10.1.2 ASSIGNMENT RESULT

While assignment is not treated like a function, it may be thought of as a function whose explicit result is the value of its right argument. Alternatively it may be considered syntactically transparent in the sense that after the assignment is complete, the arrow and whatever is bound to it on the left are removed from the expression leaving the right argument array as value. In either case, after the assignment, a value is left and is considered the explicit result of the assignment. This may then be used in further computation.

Here are some examples of assignments in value expressions and the value that is computed:

```
Expression      Value after execution
A←3             3
(A←3)           3
(A←2),(B←3)     2 3
2+A←1           3
(A←2) (B←4)     2 4
```

The following rule determines when the value of an expression should be printed:

> If the last syntactical action (binding or evaluation) in an array expression is an assignment, the final array value of the expression is not printed. If any binding occurs after the last assignment, or if there is no assignment, the final array value is printed.

Here are executions of the above examples using this rule:

```
      A←3
            no display -- last action is assignment

      (A←3)
            no display -- last action is assignment
                          parentheses are redundant

    (A←2),(B←3)
  2 3       display -- last action is binding of 2 and 3 to
                       catenate (followed by execution)

    2+A←1
  3         display -- last action is binding of 2 and 1 to
                       plus (followed by execution)

    (A←2) (B←4)
  2 4       display -- last action is binding 2 to 4
                       (no function executed)
```

10.1.3 ASSIGNMENT SEMANTICS

In the case of direct assignment, the semantics are perfectly clear -- the name is associated with the array on the right.

In the case of selective assignment, the expression in parentheses on the left of the arrow follows the same rules of syntax as any other expression. The semantics, however, are quite different. One of the names in the expression will have its array value modified by the assignment. We need to define which name is the one into which values are merged. The rule adopted in APL2 is that the first name from the right end of the expression not in brackets is the one into which values will be merged. The value associated with this name will not change in either rank or shape because of the assignment. Scanning the name produces an array indicating positions in that array. (These were called position scalars in [Br1] and the arrays were called name arrays.) The remaining functions in the expression operate on this array of positions. Any function which selects, subsets, or rearranges this array is allowed in the expression. No arithmetic is allowed on these arrays. When the expression completes, the assignment completes by inserting the values from the right argument into the indicated positions of the selected array.

These name arrays could be considered a new data type but are not because they can never be associated with a name and they can only be manipulated in the context of selective assignment.

In the following example the array into which values are merged is *A*.

```
A[2]←3
(1↑A)←3
(2↓2 3,A)←3
(I⊃A)←3
```

Note that in the implementation, not all functions formally permitted in an expression left of an arrow are actually supported.

10.2 BRANCH AND ESCAPE

The right arrow, when used to control sequencing in a defined operation or when used to resume execution, is called Branch. It is syntactically like a function and so does not influence the binding hierarchy. It fails to be a function in the strict sense because it does not have an explicit result. It can therefore only be used in a valueless expression. The execute function (⍎) and user defined operations may also fail to return an explicit result but are none the less still considered functions. Branch is not considered a function semantically and in particular cannot be the operand of an operator. Its only purpose is the determination of the next line to be executed.

When the right arrow is used without a right argument it is called escape and it must be the only symbol in the expression. Syntax is not a question because nothing is next to it. It is simply treated as a special case request to clear out any executions that are pendant on completion of the current operation (or in the case of escape in immediate execution to clear out executions that are pendant and suspended).

10.3 JOT

The jot symbol '∘' is used as a special symbol to distinguish between the two derived functions of the array product operator dot (.). If the left operand of matrix product is a function (*F.G*), the derived function is inner product. If the left operand of matrix product is jot (∘.*G*), the derived function is outer product. Inner product (*F.G*) takes two functions as operands. Outer product (∘.*G*) takes one function as operand and the jot is a place holder for the other operand. Its use is not exploited or extended beyond its use in *APL*1.

Strictly speaking, jot is in its own syntactic class. Syntactically, however, it is treated as a function when it is used in the context of outer product and so does not influence the binding hierarchy. It cannot be used as an operand to other operators but expanding its use would introduce no formal problems.

11.0 FUNCTION PROPERTIES

The fundamental concept of syntax is adjacency. Expressions are written by writing names of objects next to each other along with special symbols. Evaluation of an expression is an iterative process composed of identifying operations, evaluating them, and then using the result of the evaluation in identifying the next operation. The next sections deal with the second of these -- the application of functions (primitive, defined, or derived) to arguments.

When more than one function exhibits some behavior, that behavior may be isolated and defined on its own in which case it is called a function property.

We first discuss some ordinary mathematical properties, then some new ones of particular interest to *APL2*.

11.1 MATHEMATICAL PROPERTIES

The mathematical properties of functions may be defined in a general way and are defined here because they are useful in definitions of other properties.

- The Commutative Property

  Dyadic function *F* is a commutative function if:

  A F B ↔ *B F A* for constant values *A* and *B*

- The Associative Property

  Dyadic function *F* is an associative function if:

  A F(B F C) ↔ *(A F B)F C*
  for constant values *A*, *B*, and *C*

- The Distributive Property

  Dyadic function *F* distributes over dyadic function *G* if:

  A F(B G C) ↔ *(A F B) G (A F C)*
  for constant values *A*, *B*, and *C*

  Dyadic function *F* distributes over monadic function *G* if:

  A F(G C) ↔ *G (A F C)*
  for constant values *A* and *C*

The properties might be used to write concise expressions. For example, in:

```
(A×B)+C
```

because + is commutative, an equivalent expression can be written without parentheses as

```
C+A×B
```

In general, though, the mathematical properties apply to actual values but not necessarily to expressions. For example:

```
2+3 ↔ 3+2
```

but

A+B is not *B+A*

when *A* and *B* are shared variables or niladic functions. Plus is commutative on values not on names.

The implementation can take advantage of mathematical properties to compute results more efficiently. For example since + is associative the expression:

```
+\ 2 3 4
```

which is defined as

```
(+/2) (+/2 3) (+/2 3 4)
```

may be computed by a single left to right pass through the vector using the I th result in computing the I+1st result.

```
(+/2) ((+/2)+3) ((+/2 3)+4)
```

Formally this gives the same result and is much more efficient than the exact definition of scan. Numerically (because of the precision of the machine) round off may cause the two expressions to give very different results.

11.2 *APL2* FUNCTION PROPERTIES

In *APL2* a few new properties are defined which relate to large classes of functions and which, therefore, simplify understanding of how the functions operate.

11.2.1 THE SCALARWISE PROPERTY

The scalarwise property is defined using one of the mathematical properties. A function is scalarwise if indexing distributes over the function. For monadic *F* where *Z←F R*:

$Z[I] \leftrightarrow (F\ R)[I] \leftrightarrow F\ R[I]$
for all I that can index R

for dyadic F where $Z \leftarrow L\ F\ R$:

$Z[I] \leftrightarrow (L\ F\ R)[I] \leftrightarrow L[I]\ F\ R[I]$
for all I that can index R and
where scalar extensions (if any) have
already been done.

Note that because the indexing function of *APL* has special syntax, the expression above does not exactly match the form for the distributive property given previously. Also the arguments may be of any rank even though the bracket notation for indexing cannot be used on variable rank arguments. If the symbol ⌷ were defined as an indexing function (it is not in *APL2*), the formulas would read:

$I \quad ⌷ \quad Z \leftrightarrow I \quad ⌷ \quad (L\ F\ R) \leftrightarrow (I \quad ⌷ \quad L)\ F\ (I \quad ⌷ \quad R)$
$I \quad ⌷ \quad Z \leftrightarrow I \quad ⌷ \quad (F\ R) \leftrightarrow F\ (I \quad ⌷ \quad R)$

which does follow the form and would work on any rank array.

In *APL2*, two sets of functions have the scalarwise property: the scalar functions and the derived functions of the each operator. The implication is that these functions may be defined by saying how they operate on scalars, then saying that they apply independently to each scalar of any other array.

Thus, for monadic functions, a length n argument implies a length n result made up of the application of the function to each scalar of the argument. For example:

```
÷2 10
(÷2) (÷10)
.5 .1
```

and

```
⍴¨V←(2 3⍴⍳6) (4 3 2⍴0)
(⍴¨V[1]),(⍴¨V[2])
(⍴2 3⍴⍳6) (⍴4 3 2⍴0)
(2 3) (4 3 2)
```

For dyadic functions with the same shape argument on each side, the scalarwise property implies application of the function to corresponding scalars one from each side. For example (leaving out the indexing step):

```
2 3⍴¨ 4 5
(2⍴4) (3⍴5)
(4 4) (5 5 5)
```

It is impossible to define a function which has the scalarwise property but which is not defined in terms of each. Thus it is

both correct and necessary to phrase the above discussion in terms of scalars and not items.

By extension one argument is allowed to be a scalar or one item vector (perhaps nested) in which case the item is paired with each item of the other argument. This is called scalar extension.

11.2.2 THE PERVASIVE PROPERTY

The pervasive property is also defined in terms of a mathematical property. A function is pervasive if pick distributes over it. For monadic function *F* where *Z* ↔ *F R*:

I⊃*Z* ↔ *I*⊃(*F R*) ↔ *F I*⊃*R*
for all *I* that can pick from *R*

For dyadic function F where *Z* ↔ *L F R*:

I⊃*Z* ↔ *I*⊃(*L F R*) ↔ (*I*⊃*L*) F (*I*⊃*R*)
for all *I* that can pick from *R*
where scalar extensions (if any) have
already been done.

Since the pick function may select an item at an arbitrary depth in a nested array, it may select deep enough to access a simple scalar (because nested arrays have finite depth). Thus a pervasive function may be thought of as applying independently to each simple scalar in its argument(s).

Suppose an argument *J* is found which picks a simple scalar from array *R*. Then for monadic *F*:

J⊃*Z* ↔ *J*⊃*F R* ↔ *F J*⊃*R* by definition or Pervasive

Pick can select from a scalar with an argument of (⊂⍳0) and a simple scalar is defined, by the control on termination of nesting, to be an array which contains itself as its only item:

5 ↔ (⊂⍳0)⊃5

Therefore it is also true, by definition of the pervasive property, for the same monadic F and array R that:

(*J*,⊂⍳0)⊃*Z* ↔ (*J*,⊂⍳0)⊃*F R* ↔ *F* (*J*,⊂⍳0)⊃*R*

where pick gets the only item of the simple scalar. But since a simple scalar contains itself, both picks produce the same scalar:

(*J*,⊂⍳0)⊃*R* ↔ *J*⊃*R* (same simple scalar)

and so

$(J,\subset\iota 0)\supset Z \leftrightarrow J\supset Z$

and is a simple scalar by definition. Therefore a pervasive function eventually applies to simple scalars and returns simple scalars and the result of a pervasive function has the same structure as its argument(s) (after scalar extension). This means that it is impossible to write a defined operator that applies a nonpervasive function in a pervasive manner. There is no general way to force a simple scalar result from simple scalar arguments. Another way to say this is that for a function to be pervasive it is necessary (but not sufficient) that when it is applied to simple scalars, it returns a simple scalar result.

For computational purposes, the pervasive property may be stated in terms of the each operator. (In fact this may be taken as an alternate definition of pervasive.) For monadic pervasive function *F* where $Z \leftrightarrow F\ R$:

$$Z \leftrightarrow F\ R \leftrightarrow F\ddot{}R$$

For dyadic pervasive function *F* where $Z \leftrightarrow L\ F\ R$:

$$Z \leftrightarrow L\ F\ R \leftrightarrow L\ F\ddot{}\ R$$

Then, in computing the result of a pervasive function, if the arguments are not simple scalars, apply the each operator recursively.

In *APL2* the scalar functions and only the scalar functions are pervasive.

Example:

```
÷(1 2) ((5 10) 20)
(÷1 2) (÷(5 10) 20)
((÷1)(÷2)) ((÷5 10) (÷20))
(1 .5) (((÷5)(÷10)) .05)
(1 .5) ((.2 .1) .05)
```

11.2.3 THE LEAFWISE PROPERTY

The leafwise property is related to the pervasive property except it is more restricted in its application. A function is leafwise if pick distributes over it with left arguments that do not select from simple scalars (i.e. do not end in $\subset\iota 0$). For monadic function *F* where $Z \leftrightarrow F\ R$:

$$I\supset Z \leftrightarrow I\supset(F\ R) \leftrightarrow F\ I\supset R$$
where: $\sim(\iota 0) \equiv \uparrow\phi I$

For dyadic function *F* where $Z \leftrightarrow L\ F\ R$:

```
I⊃Z ↔ I⊃(L F R) ↔ (I⊃L) F (I⊃R)
ignoring scalar extension
and where :~(⍳0) ≡ ↑⍎I
```

The restriction on the index means that while *I* may be a path to a simple scalar, no paths are allowed that pick into a simple scalar. The practical difference between a leafwise function and a pervasive function is that when a leafwise function applies to simple scalars the result need not be a simple scalar and the result of a leafwise function might not have the same structure as the arguments(s).

For computational purposes, the leafwise property may be stated in terms of the each operator. (In fact this could be taken as an alternate definition.) For monadic leafwise function *F* where *Z* ↔ *F R*:

```
Z ↔ F R ↔ F¨R
      for R not a simple scalar
```

For dyadic leafwise function *F* where *Z* ↔ *L F R*:

```
Z ↔ L F R ↔ L F¨ R
      for L and R not both simple scalars
```

Then, in computing the result of a leafwise function, if the arguments are not simple scalars apply each recursively.

In *APL2* the scalar functions have the leafwise property. The operator *LEAF* in Appendix 2 is a defined operator whose derived function has the leafwise property. For example:

```
  2 ⍴LEAF (5 10) 20
 (2 ⍴LEAF 5 10) (2 ⍴LEAF 20)
((2 ⍴LEAF 5)(2 ⍴LEAF 10)) (20 20)
((5 5)(10 10)) (20 20)
```

Thus, unlike the pervasive property, an operator may be applied to any function giving a derived function having the leafwise property.

12.0 APPLICATION OF FUNCTIONS

Some functions may be simplified by breaking down the application of the function into simpler pieces. Other functions cannot usefully be broken down. These are called intrinsic functions.

There are three fundamental concepts relating to the application of functions to arguments: application of functions to items of array arguments; splitting of array arguments into contiguous subarrays followed by the application of functions to the subarrays; and application of intrinsic functions to data.

Many functions involve more than one of these concepts. For example, the derived function reduction on a multi-dimensional array splits the array into vectors. Vector reduction is then applied to each vector and is defined in terms of items. Eventually the vector reduction reaches an intrinsic function.

12.1 FUNCTIONS ON ITEMS

Many functions are defined by selecting items from their arguments, then applying a simpler function to these arrays. This is the most fundamental concept of function application. Many functions use the concept directly; functions which split arguments into subarrays are best described by making the subarrays items of a lower-rank array; and the intrinsic definition of many functions is best described in terms of items.

As shown previously, any function which has the pervasive or leafwise property is clearly defined in terms of items. Vector reduction is defined in terms of items in that if:

```
V←A B C
```

then

```
F/V ↔ ⊂A F B F C
```

i.e. the function of reduction is applied between the items of the vector producing a result having rank one less than the argument.

Outer product is defined in terms of items in that its result contains the application of its operand function between all pairs of items -- one from the left argument and one from the right argument. For `Z←L ∘.F R`:

```
(I,¨J)⊃Z ↔ (I⊃L) F (J⊃R)
```

12.2 FUNCTIONS ON SUBARRAYS

Many functions are defined on higher-rank arrays by splitting the array along some axes into subarrays of lesser rank then applying a simpler function to the subarrays. This is the case with most of the primitive functions and operators which take an axis specification.

Functions that apply to subarrays in *APL1* are normally defined in terms of indexing which selects the subarrays by indexing along one or more axes with one or more scalars giving arrays of smaller rank. For example reduction of a matrix can be defined:

```
(+/R)[I] ↔ +/R[I;]
```

the implication being that the above expression is evaluated for each scalar integer I possible.

In *APL2* such functions are normally defined in terms of enclose along an axis which builds an array whose items are the subarrays of interest. Then the application of the simpler function to each of the subarrays may be expressed in closed form as an application of the each operator. For example reduction of a matrix can be defined:

```
+/R ↔ ⊃+/¨⊂[2]R
```

and this is a complete executable definition. The final disclose in this expression undoes the split implied by the enclose by assembling the scalars resulting from each reduction into the desired result. Thus functions defined on subarrays are best described in terms of an array containing the subarrays as items.

12.3 INTRINSIC FUNCTIONS

The intrinsic functions are defined directly on their array arguments. For example, the function matrix divide (⌹), applied to a matrix, cannot be easily broken down to a simpler function that applies to rows or columns or items of its argument. Even when a primitive function can be applied to subarrays or items, eventually a point is reached where further simplification is not reasonable. Even the scalar functions eventually are reduced to the case where the arguments are simple scalars at which point the intrinsic function is applied. The *APL2* Language Manual [1] only defines the intrinsic scalar functions with the discussion of their application to other arrays factored out and described in a single place.

Many of the structural functions in *APL2* (reshape, ravel, transpose, etc.) cannot be simplified, yet their intrinsic behavior is still defined in terms of items. For example, vector catenate can be described as producing the vector containing the

items from the left argument followed by the items from the right argument.

13.0 RELATED FUNCTIONS

A function in mathematics is a mapping from a domain to a codomain. A function in *APL2* is a collection of five related mathematical functions. Whenever an *APL2* function is evaluated, one function from this collection is selected to satisfy the evaluation. The collection of functions is divided into two groups: principal functions and implied functions. The principal functions are used when functions are directly executed and either a principal or an implied function is used when functions are executed by an operator.

13.1 DIRECT EXECUTION OF FUNCTIONS

When an *APL2* function is evaluated directly (not under control of an operator), one of two principal functions is selected. If a left argument exists, the dyadic principal function is selected; otherwise the monadic principal function is selected. Thus when we say that a function is ambi-valent, we mean there are two principal functions with the one chosen depending only on context. If one of the principal functions is not defined, a *VALENCE ERROR* is generated on an attempt to use it. The syntax is correct -- a principal function is missing.

13.2 INDIRECT EXECUTION OF FUNCTIONS

When an *APL2* function is the operand of an operator, one of its principal functions may eventually be evaluated. However, under some circumstances, one of the implied functions is evaluated instead. *APL2* defines three implied functions: two fill functions and an identity function. For a particular *APL2* function, if an implied function is not defined, a *DOMAIN ERROR* is generated on an attempt to use it.

13.2.1 THE FILL FUNCTIONS

When the derived function of each or outer product is applied to empty arrays, the operator definition specifies that the principal function of its operand should not be applied and that instead a fill function should be applied. Each function has a monadic fill function and a dyadic fill function. The one applied depends on context and on the operator. The scalar functions may be defined in terms of each so the above discussion also holds for them.

The fill functions are selected so that empty arguments do not cause discontinuities in identities or other unexpected behavior. Definitions and specific cases are discussed in the section on empty arrays.

13.2.2 THE IDENTITY FUNCTION

When the derived function reduce is applied to empty arrays, the operator definition specifies that the principal function of its operand should not be applied and that instead an identity function should be applied. Identity functions are an extension of the notion of identity elements from *APL*1 [Br4]. The identity functions are selected so that they produce arrays which are the nested array equivalents of identity items from *APL*1. Definitions and specific cases are discussed in the section on empty arrays.

14.0 FILL ITEMS

The *APL2* functions take (↑), expand (*A*\), and replicate (*A*/) produce results consisting of some items from their right argument along with some fill items. In *APL1*, where a given array is either all numeric scalars or all character scalars, the fill item chosen is zero (0) or blank (' ') respectively. Thus the result has the same properties as the argument. A numeric array remains numeric and a character array remains character. There is no choice here because *APL1* only has homogeneous array. We could not fill a character array with zeros and certainly not with nested fill items.

Given the wider variety of arrays in *APL2*, a scheme for determining what fill item to use with these functions is required. For example, suppose we apply take to the following nested array:

```
A←(1 2) (3 4) (5 6)
Z←N↑A
```

The array A is called a uniform array because each item has the same structure. (In particular any array containing one or fewer items is uniform.) Clearly if the value of *N* is 1, 2, or 3, *Z* is also uniform array. If take is to retain properties where possible (this is an objective) then for *N*=5, *Z* should still be uniformly nested. Thus a reasonable answer for 5↑*A* is:

```
5↑A ↔ (1 2) (3 4) (5 6) (0 0) (0 0)
```

When take produces a result bigger than its argument (along any axis) the operation is called an overtake.

If the original array is not uniformly structured, there does not seem to be a reasonable choice for fill that fits every conceivable situation. Any choice of fill would give a result that was not uniformly nested. The choice for fill in *APL2* is chosen to satisfy the following guidelines:

- Do what *APL1* does, where defined.
  - Keep simple arrays simple.
  - Keep numeric arrays numeric.
  - Keep character arrays character.
- Keep uniform arrays uniform.
- Fill item must be predictable.

In order to do what *APL1* does where defined, we need an expression that will produce a zero if an array is numeric and a blank if an array is character. Let the function *TYPEOF* be a monadic function

which given a simple scalar returns zero if the scalar is a number and blank if the scalar is a character (see Appendix 2). Then a definition that works for *APL1* is "Fill with the type of the first item (*TYPEOF* ↑ *R*)".

In order to satisfy the second guideline we need to be able to compute an array whose structure is the same as a given array *A* but with zeros replacing numbers and blanks replacing characters. Such an array is called the type of *A*. Thus the type of an array is itself an array. Appendix 2 contains the definition of such a *TYPE* function. A definition for fill that works for all arrays is: "Fill with the type of the first item (*TYPE* ↑ *R*)". For every nonempty array *A*, type of first (*TYPE* ↑) is well-defined and the resulting array is called the prototype of the array (proto type ↔ first type). For uniform arrays, filling with the prototype preserves structure and type but not values.

All of the functions which produce fill items in the manner described here have been described in the section on brackets as being defined on multi-dimensional arrays by splitting the array into arrays of smaller rank, then applying a simpler case of the function. This affects the production of fill items. For example, the function expand is intrinsically defined on vectors. (Iverson would call it a rank 1 or a vector function [Iv3].) When applied to multi-dimensional arrays, the array is split into vectors as follows:

```
    If A← 2 2⍴ 2 'A' 3 'B'
       A
  2 A
  3 B

    Then      1 0 1⍀A
    implies   (1 0 1⍀2 3) and (1 0 1⍀'A' 'B')
    giving
2 A
0
3 B
```

Each vector is filled with a fill item appropriate to that vector.

The function take is not a vector function yet it still can be defined in terms of splitting its right argument into arrays of lower rank by using the identity:

$$L\uparrow[X]R \leftrightarrow (\uparrow L)\uparrow[\uparrow X]\ (1\downarrow L)\uparrow[1\downarrow X]\ R$$

Therefore if

```
      X← 2 2 2⍴ 1 2 3 4 'A' 'B' 'C' 'D'
      X
  1 2
  3 4

  A B
  C D
```

An argument similar to the one above for expand explains the following result

```
      3 3↑[2 3] X
  1 2 0
  3 4 0
  0 0 0

  A B
  C D
```

The question of fill items for empty arrays is discussed in the section on empty arrays.

15.0 EMPTY ARRAYS

The arrays of *APL2* are finite rectangular collections which contain arrays as items. The items of an array are ordered along zero or more directions called axes. The number of axes is called the array rank. The vector containing the length of each axis is called the array shape. The number of items in the array is the product of the shape. Any array whose shape contains a zero (i.e. has an axis of length zero) has no items and is called an empty array.

The designers of *APL1* were very careful to be sure that primitives operated correctly in the limiting cases. As a result when expressions are applied at the limits (axis lengths of 1 or 0 and ranks of 1 or 0), there are no surprising discontinuities. Functions in *APL1* work correctly in the limiting cases.

In extending *APL1* great care is taken to preserve this behavior at the limits. Achieving correctness is especially difficult because our intuition cannot always be trusted in these cases. We therefore must rely on more formal means of discovering the behavior of functions in limiting cases. The following sections detail the approach to discovering the nature of empty arrays and the application of functions to empty arrays.

15.1 EMPTY NESTED ARRAYS

There are two approaches to defining empty arrays as discussed previously (see Methods). The constructive approach is the most obvious approach but, because we are dealing at the limits, we know that the deductive approach must also be examined. The following two discussions will show that it is possible to get two different answers to a limiting case question depending on the approach taken.

15.1.1 THE CONSTRUCTIVE APPROACH TO EMPTY ARRAYS

In the constructive approach to extensions, we start with what we have and extrapolate.

In *APL1* we have simple rectangular collections of numbers and characters. The constructive approach to nested arrays is to allow any of those numbers or characters to itself be a rectangular collection. Then by recursively substituting arrays for items, we can construct arbitrarily complicated arrays. Indexed specification can be used to replace any scalar with another scalar.

```
A[I;...;K]← 5
```

Therefore nested arrays can be achieved merely by postulating a function which when applied to any array produces a scalar in such a way that the original array can be reconstructed. In *APL2* we use the function enclose (⊂) to do this. So any nested array can be constructed by indexed specifications of the following sort:

```
A[I;...;K]← ⊂B for arbitrary array B
```

The question of interest here is "Can an empty array be nested?" and the answer is obvious. Since we start with simple arrays and construct nested arrays by substituting enclosed arrays for the simple scalars, it is clear that an empty array cannot be nested because an empty array has no items for which we can substitute.

The conclusion is that introduction of nested arrays does not introduce any new empty arrays. All empty arrays are simple.

Perfectly well-defined and usable extensions can be defined using this conclusion. The following discussion shows that the deductive approach leads to different results.

15.1.2 THE DEDUCTIVE APPROACH TO EMPTY ARRAYS

In the deductive approach we start from where we want to be; write the equations that describe the behavior away from limits; then investigate what happens as the limits are approached and believe what we discover. As in *APL1*, we wish to have no surprising discontinuities when we reach the limit. We want functions to work in expected ways in limiting cases.

The identity that defines the relationship between enclose with axis and disclose with axis is an easy one to believe and is restated here assuming *A* is a matrix and the function is applied along axis 2:

```
A ↔ ⊃[2] ⊂[2] A
```

or

```
A ↔ 2 UNSPLIT 2 SPLIT A
```

This says that if we form a vector whose items are the rows of a matrix, then form the items back into a matrix, the same array results. This is easy to believe because it is a spatial relationship that we can picture - it's obvious.

The relationship can be explored at the limits by starting with a nonempty matrix and examining what happens as we reduce the length of an axis to zero. The assumption is that the equation

will hold when A becomes an empty array. Since we are talking about relationships we can picture we will use words instead of equations to investigate the identity. (This is not, of course, a mathematical proof. See [Mo1] for a more formal approach to this subject.) Let the array A be defined as follows:

```
      A←2 3⍴⍳6
```

Consider approaching emptiness along the column axis:

```
If the shape of A is 2 3, then
   the shape of ⊂[2] A is 2
   and the shape of each item of ⊂[2] A is 3

If the shape of A is 2 2, then
   the shape of ⊂[2] A is 2
   and the shape of each item of ⊂[2] A is 2

If the shape of A is 2 1, then
   the shape of ⊂[2] A is 2
   and the shape of each item of ⊂[2] A is 1

If the shape of A is 2 0, then
   the shape of ⊂[2] A is 2
   and the shape of each item of ⊂[2] A is 0
```

Clearly, at each step the result of ⊂[2] is well-defined and the reconstruction of *A* by ⊃[2] is no problem. Thus the equation holds at least in some empty cases.

Now consider approaching emptiness along the row axis:

```
If the shape of A is 2 3, then
   the shape of ⊂[2] A is 2
   and the shape of each item of ⊂[2] A is 3

If the shape of A is 1 3, then
   the shape of ⊂[2] A is 1
   and the shape of each item of ⊂[2] A is 3

If the shape of A is 0 3, then
   the shape of ⊂[2] A is 0
   and the shape of each item of ⊂[2] A is 3
```

Although this last step follows the progression of numbers naturally with items always being length 3, it cannot be pictured visually. This might be a demonstration that the "obvious" equation is not true! It appears that the result of the enclose on axis 2 is an empty vector so how could each item have shape 3? Yet, if the equation is to hold everywhere, the information that the original argument had three columns must be retained somewhere!

Because we believe the equation and know that our intuition cannot be trusted at the limits, we declare that the equation must hold everywhere and that the result of the enclose is a new

object, implied by the theory, that we did not know existed: an empty nested array. In this case it is an empty vector of three-item vectors. The implication is that there is more to arrays than shape and items -- there is information about the structure of arrays. In the example above, the information about the structure is that the array contains 3-item vectors. The function first is extended so that when applied to an empty array it returns an array that describes that structure. Thus

```
B←⊂[2] A←0 3⍴0
↑B ↔ 0 0 0 by definition
```

Furthermore if *A* is a character array:

```
B←⊂[2] A←0 3⍴' '
↑B ↔ '   ' by definition
```

The empty array *A* contains no data. Since the array describing the structure must have something for items, it contains zeros and blanks indicating data type. The function *TYPE* has no effect on such an array so above where we have first (↑) we may also say "type of first" (*TYPE* ↑) and by definition this is the prototype of *A*. Thus the array that describes the structure of an empty array is its prototype.

An empty array having any desired prototype can be constructed by use of the reshape function. For example an empty vector whose prototype is a 2 3 numeric matrix is constructed by reshaping the enclose of the desired prototype as follows:

```
0⍴⊂2 3⍴0
```

Keep in mind that since only type and shape are kept that nonzero values will become zeros in the prototype of the empty array.

Adopting the above definition for first answers the question posed earlier about the fill item on overtake of an empty array. If fill items are determined by the expression "type of first" (*TYPE* ↑), then fill items are defined for empty arrays because first (↑) is defined for empty arrays. For example consider the following expression involving take and a uniform nested array.

```
N↑ (1 2) (3 4) (5 6)
```

It has already been shown that, for *N*>0, the result is a uniformly nested array. With this definition of first and the existence of prototypes, the result is uniformly nested for all *N*.

Thus prototypes are used to complete the definitions of functions and make them work in expected ways even at limiting cases. This is why *APL2* adopts the answer provided by the deductive approach rather than the answer provided by the constructive approach. The section on scalar extension gives another example of the deductive approach. This simple and elegant solution to the problem of emptiness is due to many years of intensive work by Dr. Trenchard More. He applied the scientific method to data and

discovered a new "particle" -- empty nested arrays. It is similar to what Mendeleev did in working out the periodic table. He arranged the elements in order of atomic weight and in a system of rows and columns that divided them into natural families with all elements in a column showing similar properties. When an element didn't fit he did not conclude that his notion was wrong, he assumed that the atomic weights were computed incorrectly. When there was a hole in the table, he predicted (correctly) the existence of new elements. In nested arrays we are merely arranging items instead of elements.

The following sections will point out several other situations where empty nested arrays allow definitions to hold universally - even in empty cases.

15.2 EMPTY ARRAYS AND FILL FUNCTIONS

When functions are applied with operators, usually one of the principal functions (the monadic one or the dyadic one) is selected. For example, consider the following use of the each operator:

```
L←N↑(1 2)(3 4)(5 6)
R←N↑(7 8 9)(8 7 6)(5 4 3)
L ,¨ R
```

By definition of each this implies evaluation of the dyadic principal function of catenate N times -- once for each corresponding pair of items one from each argument. If N is 3 this evaluates as follows:

```
(1 2)(3 4)(5 6) ,¨ (7 8 9)(8 7 6)(5 4 3)
(1 2,7 8 9) (3 4,8 7 6) (5 6,5 4 3)
(1 2 7 8 9) (3 4 8 7 6) (5 6 5 4 3)
```

If N=0, the principal function is evaluated zero times (that is not at all). Instead the related dyadic fill function is evaluated with arguments ($\uparrow L$) and ($\uparrow R$) and the result computed defines the prototype of the result of ,¨ (which is by definition empty). Thus for N=0 the expression becomes:

```
L←0↑(1 2)(3 4)(5 6)
R←0↑(7 8 9)(8 7 6)(5 4 3)
L ,¨ R
```

The fill function for catenate is catenate (as is often but not always true of the primitive functions) so it is called:

```
(↑L) , (↑R)
0 0 , 0 0 0
0 0 0 0 0
```

This five item vector becomes the prototype of the result of ,¨

```
L ,¨ R ↔ 0⍴⊂0 0 0 0 0
```

Thus the original expression always results in a vector of five item vectors -- even when *N* is zero.

As a second example consider a monadic derived function and a fill function different from the principal function.

```
R← N⍴ (2 3⍴⍳6) (2 3⍴⍳6) (2 3⍴⍳6)
⌸¨ R
```

Ignoring the numerical result of the inverse primitive, for nonzero *N* the shape of each item (if a *DOMAIN ERROR* is not signalled) is 3 2 because ⍴⌸*A* ↔ ⌽⍴*A* (even in *APL*1). The fill function for inverse is transpose so if *N*=0:

```
R← 0⍴ (2 3⍴⍳6) (2 3⍴⍳6) (2 3⍴⍳6)
⌸¨ R
```

The fill function is called with argument ↑*R*:

```
      ⍉ ↑R
0 0
0 0
0 0
```

This becomes the prototype of the result of ⌸¨:

```
⌸¨ R ↔ 0⍴⊂3 2⍴0
```

Again the expected structure is achieved even in the empty case.

All the discussion above deals with vector arguments. The same analysis holds for higher rank arrays. For example let *R* be rank 3:

```
R← 2 0 3⍴ (2 3⍴⍳6) (2 3⍴⍳6) (2 3⍴⍳6)
⌸¨ R
```

The fill function is called:

```
      ⍉ ↑R
0 0
0 0
0 0
```

This becomes the prototype of the result of ⌸¨:

```
⌸¨ R ↔ 2 0 3⍴⊂3 2⍴0
```

The fill function for *F*¨ and ∘.*F* are the same as the fill function for *F*. *APL2* does not provide a means to specify the fill function of a defined function.

15.3 EMPTY ARRAYS AND SCALAR EXTENSION

15.3.1 SCALAR EXTENSION

The dyadic scalar functions normally require arguments with matching shapes:

$L+R$ requires $(\rho L) \equiv (\rho R)$

The same is true with derived function of the each operator.

$L\ F\ddot{}\ R$ requires $(\rho L) \equiv (\rho R)$

The scalar functions may be defined in terms of each so, in the following discussion, we will only consider functions derived from each as representative of any function having the scalarwise property.

Dyadic scalarwise functions relax the conformability requirements and admit one argument that is scalar when the other is not scalar. In addition, one item vectors are treated like scalars for compatibility with *APL1*. The item of the scalar argument is then paired with each item of the non-scalar argument. This repeated use of scalars is called scalar extension.

Here is an example of scalar extension using a nested scalar:

```
     (⊂2 3) ⍴¨ 4 5 (⍳6)
     (2 3⍴4) (2 3⍴5) (2 3⍴⍳6)
4 4 4  5 5 5  1 2 3
4 4 4  5 5 5  4 5 6
```

This is a three item vector of 2 by 3 arrays.

In general, to apply *F* with left argument *L* against each item of an array *R*, we may write:

$(\subset L)\ F\ddot{}\ R$

(In array theory this is accomplished by using two primitive operators -- each right and each left. These operators are not needed in *APL2* because of the definition of scalar extension.)

In the case where R of the above formula is an empty array, our verbal definition of scalar extension does not tell us precisely how to compute the result. Using the deductive approach (because this is an empty case), we write an equation which we believe describes scalar extension. For simplicity, we consider only a scalar (*S*) left argument. A similar discussion would hold for a

scalar right argument. We propose the following formula as the defining equation for scalar extension:

```
S F¨ R ↔ ((ρR)ρS) F¨ R
```

This is the definition from *APL*1 which says "reshape the scalar to be the same shape as the nonscalar argument".

Now we want to examine this equation in the limit. Consider the statement:

```
(⊂2 3) ρ¨ V  for vector V
```

Clearly, for nonempty vector *V*, the result will always be a vector of 2 by 3 arrays. Suppose *V* is an empty vector. Then by the proposed formula:

```
(⊂2 3) ρ¨ V ↔ ((ρV)ρ⊂2 3) ρ¨ V
            ↔ (0ρ⊂2 3) ρ¨ V
```

Now both arguments have the same shape (namely 0) and we know the fill function is called with the prototypes as arguments. The fill function for reshape is reshape so:

```
(↑0ρ⊂2 3) ρ ↑V ↔ 0 0 ρ ↑V
```

and this becomes the prototype of the empty result. Thus for nonempty *V* we get a vector of 2 by 3 arrays but for empty *V* we get a vector of 0 by 0 arrays.

This is not an incorrect result, only a surprising one. We expect prototypes to take care of discontinuities of shape. Note that there is a discontinuity of shape even though the equation we propose does hold universally. We have two choices -- believe the equation and admit that the empty case is a singularity; or find another defining equation.

Another equation is found by noting that what we really want is to bind the scalar argument to the function, and to apply the resulting monadic function to each item of the the nonscalar array. A binding of this sort may be accomplished by a composition operator (see Appendix 2).

```
L COMP ρ
```

gives a monadic function that does an L reshape. This may then be used in another defining equation for scalar extension:

```
S F¨ R ↔ ((↑S) COMP F)¨ R
```

where redundant parentheses are used for clarity. Note that this and the earlier proposed formula both define scalar extension properly for *APL*1. Since *APL*2 does not provide a way to specify the fill function of a defined operation, the above equation will not execute in the empty case.

This equation reduces scalar extension on dyadic functions to each on a monadic function. Each on a monadic function does not involve scalar extension and so is completely defined. Now in the case where *V* is empty, we find that:

```
(⊂2 3) ⍴¨ V
```

gives an array whose prototype is a 2 by 3 array. This second equation solves the singularity of shape in scalar extension and is, therefore, adopted as the defining equation for scalar extension.

Note that in practice we do not need to introduce the concept of composition to make scalar extension understandable. We may say "the scalar is paired with each item of a nonempty argument or with the prototype of the empty argument". This definition of scalar extension also applies to inner product, compress, expand, etc. but no new insight is gained by studying those functions.

15.3.2 OUTER PRODUCT

When outer product is used with a scalar argument, it produces the same result as the each operator:

```
S ∘.F R ↔ S F¨ R    for scalar S
```

While no proof of this is given, we can see that each side of the equation means "apply *S* to each item of *R*". If this equation is to be universally true, it defines how outer product works when *R* is empty. Clearly the prototype of the result must be:

```
(↑S) F ↑R
```

Thus `2 ∘.⍴ R` will return an array of 2 item vectors even when *R* is empty.

15.4 EMPTY ARRAYS AND THE IDENTITY FUNCTION

Reduction on nonscalars is defined so that the rank of the result is always one less than the rank of the argument. In particular, the reduction of a vector is a scalar.

In the expression:

```
F/ N⍴V
```

If *N*>0, the principal function of *F* is evaluated *N*-1 times. When *N*=1, the principal function is not evaluated at all. The result

is the scalar $(\iota 0)\rho 1\rho V$ -- a scalar containing the same item as $1\rho V$.

When $N=0$, again the principal function is not evaluated. Instead the identity function is applied to the prototype of the argument producing an array I. The scalar result is then defined as $\subset I$. (In the case of reduction of an empty higher rank array A, the result is $(^{-}1\uparrow\rho A)\rho\subset I$.)

Consider <u>times</u> <u>reduction</u> of a vector:

```
R ← ×/NρV
```

If V is a simple vector then R is a simple scalar.

```
6 ↔ ×/2 3       1 ↔ ×/0ρ0
```

If V is a uniformly nested vector then R is a scalar whose item has the same structure as items of V:

```
8 15 ↔ ↑×/(2 3)(4 5)
1 1  ↔ ↑×/0ρ⊂0 0
```

Thus in $F/N\rho V$, as N is reduced to zero, there is no sudden change in the structure of the result when N reaches zero.

The scalar functions, for which identity function are defined, have a specified simple scalar identity item (i.e. 1 for times, 0 for plus, etc.). The identity functions for these scalar functions are defined to return arrays whose structure is that of their argument (the prototype of the original argument) but with all values replaced by the identity item.

If II is the identity item of some scalar function, then the identity function for that scalar function is:

```
     ∇Z← IDF R
[1]  Z←II+R
```

Since R is a prototype it can contain only zeros. The addition of the scalar II gives a Z which is the same structure as R except it has II where R has 0. Any characters in R will cause a domain error.

Consider the evaluation of reduction on a nested empty vector:

```
     ↑×/0ρ⊂2 3ρ0
1 1 1
1 1 1
```

The identity function for <u>times</u> is called with a prototype as argument $\uparrow 0\rho\subset 2\ 3\rho 0$ which is $2\ 3\rho 0$. The identity function computes $1+2\ 3\rho 0$ (because 1 is the identity item for <u>times</u>) and the scalar result of reduction is computed as $\subset\ 2\ 3\rho 1$

The identity functions defined in *APL2* are listed in the Language Reference [1]. Application of the identity function produces an array which usually is a left and right identity item for the function. If a function has only a left or only a right identity, that value is used. Sometimes an identity function is defined for a subset of arguments (for example = which has identity item 1 only when applied to logical arrays).

Identity functions for the mixed functions are derived by considering reduction of nonempty uniformly nested vectors. This is done because as we reduce the length of the vector toward zero, it becomes uniformly nested when the length reaches 1.

As an example let *V* be the vector made up of *A*, *B*, and *C* which are identically shaped matrices. Then consider catenate reduction of *V*:

```
,/V←A B C
```

The identity function for catenate must return an array *I* such that

```
,/A B C  ↔  ,/ A B C I
```

Therefore the identity function chosen is ((¯1↓⍴*P*),0)⍴*P* where *P* is the prototype of *V*.

16.0 CONCLUSION

APL2 contains a large set of extensions to *APL1*. They are defined as proper extensions of *APL1* and in the same spirit as *APL1*. They give the appearance that the designers of *APL1* had these extensions in mind when they designed the original *APL*. While this was probably not the case, it is no accident that the new facilities fit so nicely. The creators of *APL1* were careful to adopt simple rules that applied uniformly. Such rules can be easily extended and applied in new situations. The recognition of operators as different from functions gave tremendous functional capability without an explosion of symbols. Extensions to expressions of operators was achieved in *APL2* with almost no change from the original rules of *APL1*.

APL2 provides further simplification of concepts and rules. More statements can be made that hold universally. Fewer exceptions need to be described. Binding, the unified field theory of APL2 syntax, gives one concept that ties together the concepts of order of execution, precedence of operators over functions, use of parentheses, etc. The final result is that the extensions look like they were always part of *APL*.

The resulting language is a powerful and productive tool for the writing of applications and the solving of problems. It has fewer exceptions; more things work as expected (the law of least surprise); fewer loops are needed; etc.

As a result, a novice can solve a wider class of problems. He can make use of the powerful features (such as nested arrays) for ordinary tasks (such as report writing) without knowledge of the complete language or of the underlying theory. The user of *APL1* can use *APL2* immediately and grow into the new features because *APL1* is a subset of *APL2*. The professional programmer can write applications in fewer lines and in a shorter time.

The summation of the simplifications, functional enhancements, and usability enhancements leads to a notation which can be used as a tool for thinking and for the solution of problems.

17.0 ACKNOWLEDGEMENTS

I would like to thank the many people who influenced the design and implementation of APL2 and the people who read many drafts of this paper and provided valuable feedback. These include: Phil Abrams, Doug Aiton, Bob Bernecky, Norm Brenner, Karen Brown, Larry Breed, Dick Doyle, Dick Dunbar, Ted Edwards, Ed Eusebi, Ron Frank, Ziad Ghandour, Jean-Jacques Girardot, Alan Graham, Bruce Hartigan, Dick Harvey, Hans Hegei, Bob Hendricks, Ken Iverson, Mike Jenkins, Dick Lathwell, Dieter Lattermann, Gary Logan, Blair Martin, Ken Martin, Gene McDonnell, Jon McGrew, Jorge Mezei, Eben Moglen, Mike Montalbano, Alex Morrow, Chuck Norcutt, Andre Orlans, Don Orth, Sandra Pakin, Ray Polivka, Dave Rabenhorst, Jim Ryan, Stan Schmidt, Christina Shen, Bob Smith, Howard Smith, Karl Soop, Ray Trimble, Mike van der Meulen, and Ron Wilks.

I wish to give special thanks to Dr. Garth Foster who guided my early work on nested arrays leading to my thesis; Dr. Kenneth Iverson who laid the foundations of *APL*; Dr. Trenchard More who provided the theoretical background with his array theory and gave me an appreciation of the deductive approach; Adin Falkoff from whom I learned a way of thinking about extensions and who was my manager on and off for a decade; Ev Allen who worked with me on the project almost from the start; Mike Wheatley who engineered many of the formal and political interfaces for the product; Phil Benkard who applied the concepts of binding power to APL2 syntax; and especially to John Gerth and John Bunda whose insistence on discovering and applying principles led to the formalization of the rules for *APL2* and the writing of this paper.

18.0 REFERENCES

These references include papers directly relating to the topic of this paper as well as general references for *APL* extensions.

[1] *APL2* Programming: Language Reference, IBM Corp., 1984, SH20-9227

[2] *APL2* General Information, IBM Corp., 1984, GH20-9214

[3] *APL2* Programming: System Services Reference, IBM Corp., 1984, SH20-9218

[4] *APL2* Programming: Using Structured Query Language (SQL), IBM Corp., 1984, SH20-9217

[5] *APL2* Language Reference Card, IBM Corp., 1984, SX26-3738

[6] *APL2* Programming Guide, IBM Corp., 1984, SH20-9216

[7] *APL2* Installation and Customization under CMS, IBM Corp., 1984, SH20-9221

[8] *APL2* Installation and Customization under TSO, IBM Corp., 1984, SH20-9222

[9] *APL2* Diagnosis Guide, IBM Corp., 1984, SY26-3931

[10] *APL2* Diagnosis Reference, IBM Corp., 1984, SY26-3932

[11] *APL2* Messages and codes, IBM Corp., 1984, SH20-9220

[12] *APL2* Language Manual, IBM Corp., 1982, Installed User Program 5798-DJP, SB21-3015.

[13] *APL2* Terminal Users Guide, IBM Corp., 1982, Installed User Program 5798-DJP, SB21-3014.

[14] *APL2* Introduction Manual, IBM Corp., 1982, Installed User Program 5798-DJP, SB21-3014.

[Be1] R. Bernecky and K.E. Iverson, "Operators and Enclosed Arrays", 1980 *APL* users meeting, 319-331.

[Be2] J.P. Benkard, "Valence and Precedence in *APL* Extensions", *APL* Quote Quad, Vol. 13, No 3, pp. 233-242.

[Br1] J.A. Brown, "A Generalization of *APL*", Doctoral Thesis, 1971, Dept. of Computing and Information Science, Syracuse University. New York, Clearing House 74h004942 AD-770488.

[Br2] J.A. Brown, "*APL* Language Extensions", Proceedings of SEAS 1978 anniversary meeting, Stresa, Italy, Vol. 1, pp. 335-353.

[Br3] J.A.Brown, "Evaluating Extensions to *APL*", *APL* Quote Quad, Vol. 9, No. 4 - part 1, June, 1979, pp. 148-155.

[Br4] J.A.Brown, "The *APL* identity Crisis", Proceedings of APL81, San Francisco, California, Oct. 1981.

[Br5] J.A.Brown, "APL Syntax -- Is it Really Right to Left", *APL* Quote Quad, Vol. 13, No 3.

[Bu1] J.D.Bunda and J.A.Gerth, "*APL* two by two - Syntax Analysis by Pairwise Reduction", Proceedings of *APL*84, Helsinki, Finland.

[Ch1] Cheney, "Nested arrays reference manual", STSC 1981

[Fa1] A.D. Falkoff and K.E. Iverson, "*APL*\360 User's Manual", IBM Corp., FH20-0683-1, 1970.

[Fa2] A.D. Falkoff, K.E. Iverson, and E.H. Sussenguth, "A Formal Description of System 360", IBM Systems Journal,3. pp 198 ff, 1964.

[Fa2] A.D. Falkoff, "Criteria for a System Design Language", Report on NATO Science Committee Conference on Software Engineering Techniques, April 1970.

[Gh1] Z. Ghandour, and J. Mezei, "General Arrays, Operators and Functions", IBM Journal or Research and Development, Vol. 17, no. 4, July, 1973.

[Gu1] W.E. Gull, and M.A. Jenkins, "Recursive Data Structures in *APL*", CACM vol. 22, no. 4, Feb. 1979, pp.79-96.

[Ha1] H.R. Haegi, "The Extension of *APL* to Treelike Data Structures", *APL* Quote Quad, Vol. 7,no.2.

[Iv1] K.E. Iverson, "Operators and Functions", IBM Research report #7091, Yorktown Heights New York.

[Iv2] K.E. Iverson, "A Programming Language", Wiley, New York, 1962.

[Iv3] K.E. Iverson, "Rationalized *APL*", I.P. Sharp Research Report #1, April, 1983/

[La1] R.H. Lathwell and J.E. Mezei, "A Formal Description of *APL*", Colloque *APL*, Institut de Recherche d'-Informatique et d'Automatique, Rocquencourt, France, 1971.

[Lo1] P.G. Lowney, "Carrier Arrays: An Extension to *APL*", PhD Thesis, Yale University, May, 1983.

[Mr1] R. Mercer, "Extensions of *APL* to include Arrays of Arrays", University Computing Center Report, University of Mass., Amherst.

[Mo1] T. More, "Notes on the Development of a Theory of Arrays", Philadelphia Scientific Center report 320-3016, May 1973.

[Mo2] T. More, "Notes on the Axioms for a Theory of Arrays", Philadelphia Scientific Center report 320-3017, May 1973.

[Mo3] T. More, "Axioms and Theorems for a Theory of Arrays", IBM Journal of Research and Development, Vol. 17, no. 2, 1973.

[Mo4] T. More, "A Theory of Arrays with Applications to Databases", IBM Cambridge Scientific Center report G320-2106, Sept 1975.

[Mo5] T. More, "Types and Prototypes in a Theory of Arrays", IBM Cambridge Scientific Center report g320-2112, May 1976.

[Mo6] T. More, "On the composition of Array-theoretic Operations", IBM Cambridge Scientific Center report G320-2113, May 1976.

[Mo7] T. More, "Nested Rectangular Arrays for Measures, Addresses, and Paths", ACM-STAPL/SIGPLAN APL79 Conference Proceedings, APL Quote Quad #9, No. 4 - Part 1, June 1979, PP. 156-163.

[Mo8] T. More, "The Nested Rectangular Array as a Model of Data", Invited address, ACM-STAPL/SIGPLAN APL79 Conference Proceedings, APL Quote Quad #9, No. 4 - Part 1, June 1979, PP. 55-73.

[Mo9] T. More, "Rectangularly Arranged Collections of Collections", Invited address, APL82 Heidelberg

[Mo10] T. More, "Notes on the Diagrams, Logic, and Operations of Array Theory", IBM Cambridge Scientific Center, TR G320-2137 Sept., 1981.

[Mo11] T. More, "Structures and Operations in Enginerring Second Lerchendal Book and Management Systems", Edited by Oyvind Bjorke and Ole I. Franksen, Tapir Publishers, Trondheim, Norway 1981. pp. 497-666.

[Pa1] S. Pakin, "*APL*\360 Reference Manual", Science Research Associates, Inc., Chicago, 1968.

[Sm1] R. Smith, "Nested arrays, Operators and Functions", *APL* 81 Proceedings, 286-290.

19.0 APPENDIX 1: SUMMARY OF RULES AND DEFINITIONS

Object Classes: There are three classes of objects:
- arrays
- functions
- operators

Array: An ordered rectangular collection of items each of which is itself an array

Simple Array: An array each of whose items is a single number or a single character.

Nested Array: An array at least one of whose items is not a single number of a single character.

Homogeneous array: An array containing only one type of data -- numbers or characters.

Function Valence: All functions are ambi-valent (both valences) and the one evaluated in any instance is determined only by context.

Operator Valence: Operators are not ambi-valent. A given operator is either monadic or dyadic determined by definition not context.

Syntax classes: There are six syntax classes:
- arrays
- functions
- monadic operators
- dyadic operators
- assignement arrow
- brackets

Parentheses rule: Parentheses are used for grouping. They are correct if properly paired and if what is inside evaluates to an array, a function, or an operator.

Redundant Parentheses: Correct parentheses that don't alter any bindings are redundant

- general
  - group a single name (primitive or constructed)
  - group an expression in parentheses
- array expressions
  - do not both group and separate
  - group right argument of a function
  - group vector left argument of a function
- function expressions
  - group left operand of an operator
  - group function expression and left parenthesis does not separate two arrays
- bracket expressions
  - group brackets and object to the left

Expression: A linear string of names and symbols, taken from the six syntax classes, punctuated with parentheses.

Right to left rule: In an unparenthesized expression without operators, functions are evaluated from right to left.

Function Precedence: Functions in an expression have no precedence. The order of execution depends only on position in the expression.

Rewriting rule for character vectors: If a vector in parentheses is made up entirely of single characters, it may be rewritten with a single pair of enclosing quotes.

Expression substitution: In an expression, any sub-expression may be replaced by another expression that computes the same value without changing the value of the original expression.

Syntactic Substitution: In an expression, any object in parentheses may be replaced by an object of the same syntax class without changing the syntax of the statement.

Printing results: If the last syntactical action in an array expression is an assignment, the final array value of the expression is not printed. If any binding occurs after the last assignment, or if there is no assignment, the final array value is printed.

Scalarwise function: A function over which indexing distributes.

Pervasive function: A function over which pick distributes

Leafwise function: A function over which pick distributes where no pick operation selects the item from a simple scalar.

Scalar extension: When a scalarwise function is applied to two arrays and one array is a scalar and the other is not, the item of the scalar is paired with each item of the nonempty nonscalar array, or the item of the scalar is paired with the prototype of the empty array -- defining the prototype of the result.

Binding Hierarchy

```
brackets
assignment left
right operand
vector
left operand
left argument
right argument
assignment right
```

```
brackets --------- binding of brackets to what
                   is on the left
assignment left -- binding of a left arrow
                   to what is on its left
right operand ---- binding of a dyadic operator
                   to its operand on the right
vector ----------- binding of an array to
                   an array
left operand ----- binding of an operator to
                   what is on its left
left argument ---- binding a function to its
                   left argument
right argument --- binding of a function to its
                   right argument
assignment right - binding of a left arrow
                   to what is on its right
```

Brackets and monadic operators have no binding strength on the right.
Right arrow is syntactically a function that produces no value.
Niladic functions are syntactically arrays.

APL2 function set: An *APL2* function is a collection of the following related functions:

- Monadic principal function
- Dyadic principal function
- Monadic fill function
- Dyadic fill function
- Monadic identity function

20.0 APPENDIX 2: DEFINED FUNCTIONS

The defined operations presented here are definitions or examples and in general do no checking for bad arguments or exceptional conditions.

The *SPLIT* function defines enclose with axis.

```
     ∇Z←I SPLIT R;IRHO;ZRHO          ⍝SPLIT R ALONG AXES I
[1]   IRHO←(⍴R)[I]                   ⍝SHAPE OF RESULT ITEMS
[2]   ZRHO←(⍴R)[(⍳⍴⍴R)~I]            ⍝SHAPE OF RESULT
[3]   R←(⍉((⍳⍴⍴R)~I),I)⍉R           ⍝MOVE I AXES TO RIGHT
[4]   Z←0⍴⊂R                        ⍝SET PROTOTYPE OF RESULT
[5]   →(0∈⍴ZRHO)/END                ⍝NO LOOP IF RESULT EMPTY
[6]   R←,R                           ⍝GET LIST OF ITEMS
[7]  LP:Z←Z,⊂IRHO⍴R                  ⍝BUILD NEXT RESULT ITEM
[8]   →(0≠⍴R←(×/IRHO)↓R)/LP          ⍝DROP ITEMS USED
[9]  END:Z←ZRHO⍴Z                    ⍝RESHAPE TO RESULT SHAPE
      ∇
```

The *UNSPLIT* function defines disclose with axis. It is the left inverse of *SPLIT* and in addition pads if items do not have the same shape.

```
     ∇Z←I UNSPLIT R      ⍝UNSPLIT R FORMING AXES I
[1]   R←⊃R               ⍝PUT AXES ON RIGHT
[2]   I←((⍳⍴⍴R)~I),I     ⍝COMPUTE TRANSPOSE
[3]   Z←I⍉R              ⍝MOVE AXES TO I POSITIONS
      ∇
```

The *LEAF* operator produces a derived function that has the leafwise property.

```
      ∇Z←L(F LEAF)R          ⍝APPLY F TO SIMPLE SCALARS
[1]    →(2=⎕NC 'L')/DYAD     ⍝SEPARATE DYADIC CALL
[2]   ⍝
[3]    →(0= ≡ R)/MF          ⍝BRANCH SIMPLE SCALAR
[4]    Z←F LEAF¨R            ⍝APPLY F TO EACH ITEM
[5]    →0
[6]   MF:Z←F R               ⍝APPLY F TO SIMPLE SCALAR
[7]    →0
[8]   ⍝
[9]   DYAD:→(1= ≡ L R)/DF    ⍝BRANCH SIMPLE SCALARS
[10]   Z←L F LEAF¨R          ⍝APPLY F TO EACH ITEM
[11]   →0
[12]  DF:Z←L F R             ⍝APPLY F TO SIMPLE SCALARS
[13]   →0
      ∇
```

The *RANK* operator applies a function left operand *F* to contiguous subarrays from its arguments as defined by its right operand *N* [Iv3]. *N* is a three item integer vector containing the ranks of the arguments to which *F* is ultimately applied. The first item of

N is the rank for monadic uses of the derived function. The second and third items of *N* are the ranks of the left and right arguments ultimately used with application of *F* for dyadic uses of the derived function. The ranks specified are maximums. If an argument has smaller or equal rank than the item of *N*, the argument is used unchanged. If an argument has greater rank than the item on *N*, then the argument is split into rank *N* arrays and *F* applied to each piece. Axes of the results produced by each application of *F* are arranged as right axes of the result of the derived function. A negative *N* means use complementary rank (¯2 means split except for first two axes).

```
      ∇Z←L(F RANK N)R;RL;RR ⍝APPLY F WITH RANK N
[1]    N←0⌈N+(N<0)×((⍴⍴R)-|0⌊N)-N ⍝MAKE NEGATIVE A POSITIVE
[2]    →(0≠⎕NC 'L')/V2             ⍝SEPARATE DYADIC CALL
[3]   ⍝
[4]    RR←-(⍴⍴R)⌊1⍴N               ⍝USE SMALLER OF N AND RANK
[5]    Z←⊃F¨⊂[RR↑⍳⍴⍴R]R            ⍝APPLY F TO EACH PIECE
[6]    →0
[7]   ⍝
[8]   V2:RR←-(⍴⍴R)⌊¯1↑3⍴N          ⍝USE SMALLER OF N AND RT RANK
[9]    RL←-(⍴⍴L)⌊1↑1↓3⍴N           ⍝USE SMALLER OF N AND LFT RANK
[10]   Z←⊃(⊂[RL↑⍳⍴⍴L]L)F¨⊂[RR↑⍳⍴⍴R]R ⍝APPLY F TO PIECES
[11]   →0
      ∇
```

The *TYPEOF* function takes a simple scalar argument and returns a zero if it is a number and a blank if it is a character.

```
      ∇Z←TYPEOF R ⍝GET DATA TYPE OF SIMPLE SCALAR
[1]    Z←(0 ' ')[R ≡ ⍕R]
      ∇
```

In *APL2* the type of an array R is an array of the same structure but with all numbers set to zero and all characters set to blank. The following are two different definitions of a function which computes the type of an array. The first recursively searches the argument for simple scalars, then applies *TYPEOF* from above. This function is a constructive definition because it actually finds the scalars then changes them. It fails on empty arrays because there is no way to specify the fill function related to the derived function *TYPEOF LEAF*. The second uses the intimate relationship between array types and empty arrays to extract the type array directly. It always produces the correct answer.

```
      ∇Z←TYPE R ⍝ GET TYPE OF ARRAY R
[1]   Z←TYPEOF LEAF R
      ∇

      ∇Z←TYPE R ⍝ GET TYPE OF ARRAY R
[1]   Z←↑0⍴⊂R
      ∇
```

The *COMP* operator forms a monadic function by associating with a dyadic function a fixed left argument. Thus while + is the addition function and 1+*A* is its dyadic use, 1 *COMP* + is the

increment function and (1 *COMP* +) A is its monadic use (with redundant parentheses used for clarity).

```
     ∇Z←(A COMP F) R ⍝COMPOSE ARRAY WITH DYADIC FUNCTION
[1] Z←A F R
```

21.0 APPENDIX 3: A BRIEF CHRONOLOGY OF *APL* DEVELOPMENT

APL, as a notation for writing about data processing, grew from work begun in 1957 by Dr. Kenneth E. Iverson. His book, "A Programming Language" (1962) presented a version of the language referred to as "Iverson Notation". Shortly after Iverson joined IBM in 1960, he teamed up with Adin Falkoff and for many years they led various groups in designing and developing APL products. In early 1963 Falkoff began work on the "Formal Description of System 360", which was completed with the collaboration of Iverson and Sussenguth. In 1964, Iverson and Falkoff designed the first APL type ball for the IBM selectric typewriter. This required the linearization and regularization of the language. Larry Breed and Phil Abrams produced an early implementation on the IBM 7090 in 1965. Dick Lathwell joined Falkoff and Iverson in 1966 and with Breed and Rodger Moore produced an implementation for the IBM 360. This evolved into IBM's first APL program product APL\360.

In 1971 Falkoff's group began serious investigation of shared variables under the direction of Dick Lathwell. This work evolved into APLSV. Alex Morrow led the development group that produced further releases of APLSV. Bob Creasy, Tony Hassit and Len Lyon led the group that developed APL/CMS which became the VSAPL program product.

Dr. Trenchard More joined Falkoff in 1967. He was one of the first to recognize operators as different from functions. His work on array theory provided the theoretical background for the arrays of APL2 -- especially empty arrays. Vector notation in APL2 derives from More's strand notation.

Dr. James A. Brown joined Falkoff in 1969 and participated in the development and release of APL\360, APLSV, and VSAPL. His thesis "A Generalization of APL" (1971) included significant contributions from Dr. Garth Foster, Syracuse University, and became one of the bases for APL2.

APL2 entered the IBM product plan in 1981 due to the efforts of Karen Riley and an experimental version was released as an Installed User Program (IUP) in June 1982.

Benkard provided the background theory that unified APL2 syntax in 1982.

Dr. Brown is currently manager of APL Language Development at IBM's Santa Teresa Laboratory in San Jose. Mr. Falkoff is manager of the APL Design Group at IBM Research in Yorktown Heights.