APL 82, W. H. Janko, W. Stucky (ed.),
APL Quote Quad, Vol. 13 No. 1,

A SURVEY OF EXTENSIONS TO APL

Karl Fritz Ruehr
Sperry Univac - Major Systems Division
2276 Highcrest Road
Roseville, Minnesota, 55113 USA
(612) 631-6216

Abstract

A survey of proposed extensions to the APL language is made with emphasis placed on the motivations for various proposals, the differences between them and the consequences of their adoption. Some issues of a more general nature concerning the purpose, process and direction of language extension are also discussed. An extensive bibliography is provided with annotations concerning the nature, development and influence of various authors' works. Areas of extension encompassed by the survey include nested arrays, complex numbers, uniform application of functions, laminar extension, primitive functions, control structures, direct definition, operators, system functions and variables, name scope control and event trapping.

I. Introduction

APL was originally conceived by Dr. Kenneth Iverson in the late 1950's as a powerful new mathematical notation which embodied the principles of uniformity, generality and brevity of expression in a functional form. As its development proceeded APL was implemented on a digital computer as an interpreted language for interactive use. The implementation process necessitated certain changes in the language and subsequent experience with interactive use motivated still other modifications. Throughout this early process of change the original design principles of uniformity, generality and brevity were used as guidelines to help maintain and enhance the power and utility of the language (for more on the design and evolution of APL, see Falkoff and Iverson [Fa1, Fa2]).

The combined power and simplicity of APL were instrumental in its rise to widespread popularity after its public release in 1968. This popularity in turn soon brought the language to a diverse number of applications, many of them far removed from its original use in mathematical exposition. As the popularity and range of application of APL grew, certain deficiencies in the language became evident, both in the form of anomalies and irregularities in its original definition, and in the need to extend its capabilities to an even broader domain and an even greater expressiveness (see Abrams [2], Bork [Bo1] and Elliot [El1]). Consequently many researchers began to propose modifications and extensions to the language in order to meet these needs, some of which have since been incorporated into new implementations and others of which continue to be discussed by the APL community.

It is the purpose of this paper to survey these proposals for extensions to APL, to compare alternative proposals where they conflict and to draw some general conclusions about the motivations, directions and consequences of APL language extension. Before undertaking this task, it is necessary to better define the notion of an extension: for the purposes of this paper, extensions are defined to be refinements, proper extensions or additions (in the sense of Brown [20]) to APL as it is widely accepted today. Of course this definition leaves some room for judgement about what is currently accepted in APL, but an attempt will be made to at least discuss most of the borderline cases (e.g., files and shared variables).

The organization of this paper partitions the extension proposals into four major groups according to the areas with which they are concerned: Data Types and Structures, Functions and Operators,

Evaluation and the System Environment, and Miscellaneous Extensions. These groups are then further subdivided to various degrees in order to provide more structure to the paper and to allow easier location of specific topics. In addition to the paper, an extensive bibliography of published work on APL and related topics is provided, with annotations concerning the nature, development and influence of some works where such notation would seem helpful.

Unfortunately, due to a number of factors (including unintended bias on the author's part, the lack of availability of some research, the large number of extensions which have been proposed, and the necessary limits imposed on the length of this paper), many of the proposals are not covered with the detail which they may merit, and some proposals may have been omitted entirely. There are a few specific problems of accuracy and detail which should be stated more explicitly to avoid some anticipated misunderstandings: since the research surveyed spans some 20 years, it should be understood that the views of some of the researchers whose works are discussed may have changed since they were originally expressed; the references made are to the publications cited and do not necessarily reflect the authors' current views. It is also difficult in many cases to trace an idea for an extension back to its original source, especially if that source is not a widely available published work. Finally, this author's own research interests are reflected in the fact that the different areas of extension covered are not all treated at a uniform level of detail.

It is hoped that these shortcomings can be overcome at some future date with the publication of an expanded and updated version of this paper. Any comments or suggestions regarding omissions or corrections to this paper or its bibliography, or regarding the design of an expanded update will be greatly appreciated by the author.

A note on the references: all referenced papers having a specific relevance to extensions are gathered in the bibliography and are referred to by number (e.g., [12] or [12, 20]). All other referenced papers are listed in the references section and are referred to by letters and a number (e.g., [Fa2]).

II. Data Types and Structures

Much of APL's power and simplicity result from its novel approach to data types and structures. APL recognizes only two distinct types of data, numeric and character, and only one type of data structure, the arbitrary rank rectangular array of scalar data. The limitation of the number of data types which are recognized in the language eliminates the need for explicit type declarations for variables and formal parameters. The recognition of array structure allows functions and operators to be applied to whole arrays and thus implicitly to components or sections of an array, or between components of the same or different arrays. This implicit application eliminates the need for the detailed looping control of other languages. However, the simplicity of APL's data types and structures also limits the usefulness of the language for some applications; proposals for extensions which would alleviate some of these problems are discussed in this section. For more information on related topics, see these sections: on overlays, the Miscellaneous section in Names, Naming and Data Access; on arrays of functions, the Miscellaneous Extensions section; and on packages, contexts and namespaces, the section on Collections of Named Data.

II.A Files

Systems for accessing sequential files of arrays kept in off-line storage are now so common in implementations of APL that it is reasonable to consider them as "fully accepted in current APL". Most implementors also consider their file systems to be separate from their APL per se, and thus they are not truly extensions to the language. However, because they have become indispensable in many applications and because of their implications for other extensions, they will be discussed here briefly.

The addition of files to APL was largely motivated by the deficiencies of arrays for handling large, business-oriented applications: first, their homogeneous rectangular structure was too restrictive to easily represent complex structures of numeric and character data; second, their residence in main memory severely limited their size; and last, their idiosyncratic representation kept them from being easily accessed by other processors.

The incorporation of file systems largely solved these problems by providing a systems interface in APL to a file processor which can create, activate or drop files and insert, remove or update components of these files (the components are usually APL data arrays). This interface is usually provided through such means as system functions or shared variables, although some implementations provide pseudo-primitive functions for file manipulation (e.g., Burroughs APL/700, [193]). Although they solve many of the problems of arrays, files have also revoked many of their advantages: first,

most file systems can handle only linear structures, as opposed to rectangular ones; second, the limitation to reading and storing a component at a time necessitated a return to the "word-at-a-time" looping and processing style of other languages, albeit at one level removed. In these ways, files represent not so much an extension to APL data structures as they do a more conventional superstructure in which to embed APL arrays.

II.B Trees

Proposals for adding a tree structure to APL have been made by several authors including Alfonseca and Tavera [4], Murray [139] and Vasseur [183]. In most of these proposals, new functions are defined to manipulate tree structures whose nodes or leaves in turn hold APL arrays. None of these proposals seem to have gathered much support in recent years however, as more attention has been focused on an extension of arrays to nested arrays. The tree structure extensions are largely isomorphic to the nested array proposals, but they tend to promote diversity rather than uniformity, in that they introduce a whole new structural type (trees), whereas the nested array proposals simply generalize an already existing structure.

II.C Nested Arrays

A more "APL-ish" solution to the problem of the limited structure of arrays is the extension of APL arrays to hold non-scalar data, i.e., arrays which may hold other arrays as components, which in turn may hold other arrays, etc. (though one proposal would limit this recursion to two levels of nesting; see Lowney and Perlis [104]) in a manner similar to the nesting of lists in LISP (see Jenkins [85]). Such structures would retain the advantages of arbitrary rank rectangularity and the application of functions and operators to arrays as wholes, but would allow the representation of even richer and more complex structures than can be represented with sequential files.

Proposals for extending APL to handle nested arrays began appearing as early as 1966 (by Abrams [1]), and have since become predominant in the extension literature. Much of the work on nested arrays was consolidated and rationalized with the advent of the Array Theory of Trenchard More [130-138]. Although originally inspired by thoughts of extensions to APL, Array Theory has grown to become a discipline in its own right, with a syntax somewhat different than that of APL and with a flavor more akin to that of axiomatic set theory. However, some of the results of research in Array Theory have inspired further thoughts about extensions to APL, and the two fields remain closely linked (see especially Jenkins [84, 85, 87] and Singleton [174]).

Floating vs. Grounded Arrays. Although there is in general much agreement that nested arrays are the logical extension to APL data structures (for two exceptions see Orgass [146] and Orth [147]), one major issue still divides the community of researchers working in this area: the so-called "Floating vs. Grounded" array controversy. This controversy hinges on the definitions of two functions which are fundamental to the manipulation of nested arrays, and in particular on the effect of these functions on simple (i.e., non-nested) scalars. These functions are enclose, which encloses its argument into a (nested) scalar which may then become a component of another array, and its left inverse disclose, which extracts the enclosed item(s) of a nested array (note: the need to enclose arrays before inserting them into other arrays stems from the desire to preserve certain indexing identities; see Gull and Jenkins [56]. At least one proposal for nested arrays (that of Edwards [33]) does not make this requirement, but it seems to have gained comparatively little support in recent years).

In the "floating" theory of nested arrays, the enclose or disclose of a simple scalar is again a simple scalar, implying a sort of infinitely recursive nesting for these entities. In the "grounded" theory, the enclose of a scalar is different from that scalar, and the disclose of a scalar is undefined (though some systems employ a "permissive" disclose that returns the scalar unchanged). Although these may seem like minor points, they critically affect the behavior and flavor of the two systems: proposals for grounded arrays tend to have the flavor of current APL with an extra feature for enclosing (or encoding) arrays; proposals for floating arrays tend to embrace the nesting philosophy more completely, incorporating it into the very fabric of the language. The floating system must also allow heterogeneous arrays (mixed character and numeric data) because of this difference between the systems, as demonstrated by Gull and Jenkins [56] (for more on this subject, see the section on Heterogeneous Arrays). However, it should be stressed that the choice of a floating or grounded system does not affect certain other issues, such as the definition of certain operators (see Jenkins and Michel [83]) or the choice of a more or less conservative approach to the extension process.

As was mentioned above, major figures in the research community differ in their stances on the issue of floating versus

grounded arrays: Iverson [9, 73, 76] and Bernecky [9] (both of I. P. Sharp Associates) prefer the grounded system, whereas More [81, 130-138] and Brown [18-21](both of IBM), Jenkins [84, 85] (of Queen's University), and Smith [177](of STSC, Inc.) prefer the floating system (note especially that More's Array Theory employs the floating system definitions). It is interesting to note that at least two of these have "changed sides" on the issue during the course of their research: Bernecky from the grounded to the floating and then back again [192] and Jenkins from the grounded to the floating [84]. Singleton [174] concludes that although there exist translations of More's Array Theory from the floating to the grounded systems, none of the grounded versions preserve the simplicity and elegance of the original.

At this time there seems to be little hope for an easy resolution of the issue; adherents of both views seem committed in their beliefs (see Anderson [5] and [192]). If not resolved soon, the controversy threatens at best to delay the process of extension to nested arrays (or its easy acceptance) and at worst to produce two markedly different dialects of APL. In fact, the two major time-sharing bureaus have both implemented nested array extensions, one (I. P. Sharp Associates, Inc.) using the grounded system and the other (STSC, Inc.) using the floating system. To make matters even more confusing, the two implementations sometimes use the same symbols for entirely different meanings. Such diversification may be especially unwelcome at a point when the language is undergoing its first major standardization process [Wil].

II.C.1 Functions and Operators for Nested Arrays

In order to take full advantage of the power of nested arrays, it is desirable to define certain new functions and operators which act primarily on the nested attributes of data (some proponents of the grounded system would disagree with this; see especially Iverson [73]), as well as to define the manner in which existing functions and operators apply to the new structures. The conventional structural functions are normally extended in the obvious manner, i.e., they manipulate the (possibly enclosed) items of nested arrays instead of the scalar components of conventional arrays. Extensions of other functions and the addition of new functions and operators are described below.

Pervasiveness. Floating array proposals usually extend the primitive scalar functions to be pervasive, in the sense that they apply recursively at all levels of nesting until they reach simple scalar arguments. In the case of dyadic functions this definition requires that the arguments share a parallel structure (notwithstanding scalar extension). Proposals for grounded array systems usually require that nested items be explicitly disclosed before scalar functions are applied to them, although some proposals supply operators which when applied to scalar functions yield very similar results (see the section on Depth Operators and [56, 83]).

Level Modifying Functions. It is often desirable when manipulating nested arrays to modify the level of nesting of the arrays or their items. The most obvious examples of this concept are the enclose and disclose functions described above (usually denoted ⊂ and ⊃ or < and > in the floating and grounded systems, respectively). Ghandour and Mezei [47] provide the link and pair functions (⍪ and ⍪) to simplify construction of nested vectors (in some proposals this is effectively achieved in a syntactic manner with strand notation; see below and Iverson et. al. [77]). The restricted versions of raise and lower (see below) of Gull and Jenkins [56] enclose or disclose the items of their arguments, and are of special utility in the grounded system where it is often necessary to enclose the simple scalar items of an array before they can be applied to certain functions (this is unnecessary in the floating system because of the definition of simple scalars).

Nesting-Rectangularity Conversion. A related capability allows the rectangular aspects of structure (ordered by axis) to be re-expressed as nested aspects of structure (ordered by depth) or vice versa. For example, the conversion of a 2 by 5 matrix to a vector of 5 column-vectors or to a vector of 2 row-vectors. Bernecky and Iverson [9] define the disclose function on non-scalar arrays with enclosed elements so as to laminate together the disclosed arrays, which must be of a common shape. Gull and Jenkins [56] and Ghandour and Mezei [47] both define functions called raise and lower (↑ and ↓) to convert axes into a new level of nesting. These capabilities are included as the split and mix functions (monadic ↓ and) on STSC Inc.'s NARS™ system [23].

Depth Operators. One capability which is fundamental to the use of nested arrays is the ability to apply a function at different levels of nesting within a nested array. For example, consider the difference between reversing a vector of matrices and reversing each of the matrices within that same vector. In order to allow this type of application, most of the floating system proposals

provide an each (or itemwise) operator (¨) which applies its function argument to each of the (possibly nested) items of an array. In the grounded system proposals of Gull and Jenkins [56] and Jenkins and Michel [83], similar operators are supplied as well as others which apply their function arguments to the leaves or the scalar levels of nested arrays. Bernecky and Iverson [9] provide a similar facility through the use of a dual operator and their version of the disclose function.

Indexing. The extension to nested arrays allows an index to an array to be enclosed and thus allows the construction of arbitrarily shaped arrays whose items are enclosed indices into a given array. An indexing function which accepts such an array of indices and which returns a corresponding array of items selected by those indices is usually called choose indexing, and is denoted in a variety of ways (e.g., ⌷ , [] or ⌶). The choose function relieves the restriction of current indexing to rectangular blocks of components, allowing what Cheney [23] calls "scatter-point indexing". The choose function also allows the current semi-colon delimited index "lists" to be rationalized as the outer product concatenations of position vectors (see Cheney [23]). The index generator (monadic ⍳) can be extended along similar lines by allowing it to take a vector argument specifying the shape of its array result. Each component of the resulting array is an enclosed vector representing an index into the location occupied by that component (see [137, 83, 23]). Another capability which is useful in manipulating nested arrays is a form of indexing which specifies successively deeper levels of nesting with a nested vector. Functions which embody this form of indexing are called reach (⌾) by Ghandour and Mezei [47] and Gull and Jenkins [56], and pick (⊃) by More [136-137], Brown [19-20] and Smith [176-177].

Partitioning. Various proposals have been made to allow a vector or array to be split up (or partitioned) according to some specified criterion into an array holding groups of items from the original. Gull and Jenkins [56], Brown [20] and Smith [176-177] describe functions which partition vectors or arrays along some axis according to a boolean selector so that, for example, a sentence may be easily broken up into a vector of words by a boolean control indicating the positions of blanks in the sentence. Bernecky and Iverson [9] propose a more general partitioning operator which uses a numeric matrix control to allow an arbitrary function to be applied to overlapping partitions along a single axis. Benkard [6] describes two partitioning functions which allow either simple (boolean controlled, non-repeated) or overlapping partitions of vectors, matrices or higher rank arrays.

II.C.3 Type, Emptiness and Fill Elements.

Several inter-related issues concern the notion of type as applied to non-simple arrays, the representation of empty nested arrays and the definition of fill elements for nested arrays. Current APL retains type information for arrays based on whether the elements of the array are character or numeric, and retains both type and shape information for empty arrays (the shape vector of an empty array contains at least one zero element). Since all of the elements of simple arrays are scalars, current APL defines the fill element for an array to be a scalar zero or a scalar blank, depending on the type of the array (fill elements are used to "fill in" gaps in an array created by the take and expand functions).

All of the above concepts become less well-defined in a system with nested arrays and the issues involved in their definitions become further complicated by the related issues of heterogeniety vs. homogeniety and floating vs. grounded systems. In a homogeneous nested array system, the elements of an array must be either all numeric, all character or all nested arrays, whereas in a heterogeneous system the elements of an array may encompass any or all of these types (for more information on this issue, see the section on Heterogeneous Arrays). The relation between these various issues is explored thoroughly by Gull and Jenkins [57].

Ghandour and Mezei [47] propose that empty arrays carry information concerning their structure, but limit that structure to be of uniform length at each level. This proposal requires separate semantics for the application of different functions to empty arrays. Ghandour [48] demonstrates that a more general and consistent scheme results from preserving only the top level of shape information for empty arrays (i.e., all empty arrays are simple). A similar approach is followed by Jenkins and Michel [83].

In More's Array Theory [134], the type of an array is defined to be an array of the same structure but with all of the motes (simple scalars) replaced by their typical values (i.e., zeros for numbers and blanks for characters). The fill item for an array is then defined to be the prototype of that array, which is the type of its first item. Empty arrays carry information about their type and structure through their prototype, which is defined to be the prototype of the array from which they were derived. This scheme is

followed by several proposals and implementations which include floating nested arrays (see Brown [20], Cheney [23] and Jenkins [88]). It is also the scheme preferred by Gull and Jenkins [57] for all but homogeneous grounded systems.

II.C.4 Nested Array Input and Output

Some controversy surrounds the question of the input and output formats to be used for nested arrays. Many of the supporters of the floating system also support an input format called strand notation, while many of those who support the grounded system criticize it (see Iverson et. al. [77]). Using strand notation, a sequence of juxtaposed arrays is interpreted as a vector of enclosed items; for example, the expressions "*A B C*" and "*A* (⍳?*B*) (2×*C*)" would both be interpreted as three-element vectors. Proponents of the floating system see this notation as defaulting to the current notation for vector constants in the case where all items are simple scalars, because the floating system definition of enclosure does not affect simple scalars. Falkoff [38] claims that strand notation is not a true extension of the notation for vector constants because it fails to preserve some of the properties of that notation.

Those who criticize strand notation consider it to be an implicit function application and claim that it is not as clear, brief or useful as an explicit application of functions (usually the link and pair functions; see the section on Level Modifying Functions). Those who favor strand notation claim that it is a syntactic construct that is simpler and clearer than a set of explicit functions, but not necessarily incompatible with these functions. A related issue is the use of strand notation assignment (see the section on Assignment) which raises questions about name class determination and conversion. For a fuller discussion of the advantages and disadvantages of strand notation, see Falkoff [38] and Iverson, et. al. [77].

Another issue concerning nested arrays which is largely undecided is that of the format to be used in printing such arrays. Different proposals tend to stress either the content or the structure of the data in a variety of ways. Discussions of nested arrays in the literature often employ diagrams consisting of labeled trees [18, 56, 58, 123] or nested box drawings for the purposes of illustration. Both Jenkins and Michel [83] and Bernecky and Iverson [9] present proposals for functions which recursively format and pad nested arrays with blanks to achieve a regular format for output (a version of the latter proposal has been implemented on the SHARP APL system). Schmidt and Jenkins [166] describe schemes for providing either simple sketches or fully formatted versions of array "drawings", both of which use character symbols to draw boxes around nested items. STSC Inc.'s NARS system uses two different forms of output: the default form displays shape information and the elements of the array; use of the explicit quad form of output produces a fully formatted version which uses parentheses to highlight nesting (see Smith [178]).

II.D Heterogeneous Arrays.

In current APL, all of the elements of an array must be of the same type, i.e., they must be all character or all numeric; such arrays are said to be homogeneous arrays. Several proposals have been made to allow arrays to hold elements of varying types; these are called heterogeneous arrays. Although heterogeneous arrays are often discussed within the context of nested array proposals, their inclusion in a system is independent of the inclusion of nested arrays (however, the definitions of the floating nested arrays system all but necessitate the existence of heterogeneous arrays; see [56]).

Probably the single overriding factor which has delayed the extension to heterogeneous arrays is the difficulty of their internal representation: since most computers represent character and numeric data with codes of different lengths, the indexing and retrieval of elements of heterogeneous arrays is considerably complicated relative to that of homogeneous arrays.

Another difficulty is the definition of fill elements for heterogeneous arrays which are needed for applications of such functions as take and expand. Current APL defines the fill element for a homogeneous array to be zero for numeric arrays and blank for character arrays; neither choice seems preferable for heterogeneous arrays, since they may contain both types of data. Both Brown [18] and Haegi [58] suggest defining a scalar value which is neither numeric nor character in type to be used as the fill element for heterogeneous arrays. More [134] uses a fill element based on the type of the array's first element; this is the approach used in STSC Inc.'s NARS system implementation of heterogeneous arrays (see Cheney [23]). The choice of fill elements for arrays and the relation of this issue to issues concerning heterogeniety and types of nested array systems is discussed in Gull and Jenkins [57].

A final question involving heterogeneous arrays is that of their output format; although few proposals have discussed this issue in the context of

simple heterogeneous arrays, STSC Inc.'s NARS system seems to use a convention which separates characters and numbers horizontally with spaces in output (see Cheney [23]).

II.E Complex Numbers

For many years there has been discussion in the APL community about extending the numeric data type of the language to the set of complex numbers. Such an extension would be of great utility in many scientific and engineering applications which up until now have had to simulate complex arithmetic in APL with user-defined functions. The extension of numeric data to complex numbers is largely a proper extension because the complex numbers include the real numbers just as the real numbers include the integers, and thus the extension to complex numbers involves little change for users who are concerned exclusively with real or integer values (see Penfield [155] or McDonnell [120]).

Penfield has made several reports on the choices involved in a complex number extension [151-153], has reviewed the reactions of the community to these reports [154] and has made two detailed proposals for a specific set of extensions [155-156]. These proposals have now been implemented by I. P. Sharp Associates, Inc. on the production system version of SHARP APL [119, 120].

The major issues that have been examined with regard to complex numbers include the notation to be used for complex constants, the extension of arithmetic functions, the definition of principal values and branch cuts for complex functions, the application of comparison tolerance to complex functions and the definition of complex floor and ceiling functions (for more on the last two topics, see [39, 60, 111, 112]). Of these issues, only the last is still in contention, the major competing proposals being those of McDonnell [112] and Forkes [39].

II.F Infinite Values and Arrays

Mathematicians regularly deal in their work with infinite (and even transfinite) quantities; for example, in such constructs as summation over some index to infinity. Since APL was originally designed as an alternative mathematical notation, it seems reasonable that it should also have the capability to manipulate infinite values. Iverson [73] suggests the use of the underbar and overbar symbols (_ and ¯) to denote infinity and negative infinity, respectively. However, he uses the symbols only for limited purposes: to separate lists of values, to denote limits for the power operator and to specify an alternate fill element for the expand function. McDonnell and Shallit [117] discuss a variety of topics related to the use of infinite values in APL, including the creation and manipulation of arrays with infinite axes. They discuss the application of primitive scalar functions to infinite values, motivating their choices with accepted usage from mathematics, and stressing the difference between a truly infinite value and a value of "machine infinity" which results from representation limitations. They also stress here the difference between undefined (infinite) values and indeterminate values (for which several choices may make sense), and make suggestions for the internal representation of infinite values.

An extension of some mixed functions (e.g., monadic ⍳) to infinite values is seen to imply the creation of arrays with axes of infinite length. Such arrays can be represented as transformations on indices into the arrays; these representations can be further transformed to reflect the actions of function application. One very practical use of such arrays which is demonstrated is in effecting arbitrary-precision calculations and other "WHILE"-type constructions.

Shallit [172] continues this work, giving new examples of the use of infinite arrays both for implemented systems and in mathematical exposition, where they can be employed with fewer restrictions. He also discusses the diagonalization function defined in the previous work, giving it an APL symbol (⌀) and using it in exposition to present classical proofs in APL notation. The diagonalization function is used to transform arrays with infinite length axes into infinite-length vectors by selecting elements from successive diagonals.

II.G Sets

Several proposals have been made to include sets as a data structure in APL, both formally as a new recognized structural type and informally by defining APL functions representing traditional set operations to act on arrays representing sets. The difference between these two types of representation reflects the differences between sets and arrays as formal structures: sets lack many of the properties of arrays such as axes, well-ordering and preservation of repetitions (see More [136]). An array representation of a set, for example, would have to have its elements arranged in a specific order, whereas a set would not.

Soop [180] describes a way to construct sets through the reshaping of

arrays. He defines certain useful functions on sets and also discusses some problems with sets which result from their lack of ordering, specifically in the areas of display and selection. McAllister [110] compares three different ways of representing sets: as boolean selectors for some universe of elements, as integer encodings of these boolean values, and as vectors of elements without repetitions. This last type of representation is also described by Iverson [73], who defines many useful set functions on such vectors. This approach to set representation is notable in that it introduces no new data structures to the language and that the functions which it introduces are thus potentially extendable to arrays. Implementations of some of the functions of [73] have been done by Burroughs [193] and by STSC, Inc. on their NARS system [23]. For more information on these functions see the section on Set Functions.

II.H User-Defined Types

In current APL the concept of data type is quite simple; two types of data are recognized in the language, numeric and character, although there are usually more types represented internally in an actual implementation (e.g., bit, integer, real, etc.). These internal types are largely invisible to the user because of APL's use of type-generic arithmetic functions; i.e., functions whose arguments are automatically type-converted by the interpreter, if necessary. On the other hand, such type conversions can also be effected explicitly with functions such as ⌈ and ⌊. Because the interpreter can differentiate character and numeric types from context, this scheme frees the programmer from the necessity of making data type declarations for variables and formal parameters.

However, this freedom from concern with data types runs counter to some of the latest trends in mainstream programming language design. Many designers currently support not only the declaration of types for variables, but also the flexibility afforded by allowing users to define their own data types. The proponents of this concept of data abstraction cite several reasons for the use of these data typing facilities: they allow programmers to define the types of structures which are natural to an application; they provide a summary of the properties of objects of a given type; and they prevent functions from being erroneously applied to the wrong types of data.

Several proposals have been made to incorporate user-defined data typing facilities into APL. The APL-inspired languages X\APL (Braffort and Michel [16]) and ALICE (Jenkins [86]) provide means for users to define data types through a mechanism of tagged structures and, in ALICE, to define variant versions of functions to apply to these different types. Jenkins and Michel [82] demonstrate that different interpretations can be made when tags are associated with recursive data structures such as nested arrays. Hardwick [61] employs a record structuring facilities and typing to an application involving graphics data structures. Kajiya [90] discusses a new scoping mechanism, called downward scoping, and demonstrates how it can be used to obtain generic functions and thus a data "class" effect. In spite of the above proposals, no general scheme for introducing data abstraction facilities into APL has yet become widely accepted. This may be due to the fact that many people feel that strong data typing of any kind is foreign to the spirit of APL. For this reason, it may be some time before even an elegant proposal will be accepted by the APL community.

III. Functions and Operators

In order to manipulate its data arrays, APL contains a wide variety of primitive scalar and mixed functions. These functions are not assigned to any hierarchy of precedence, but simply apply to the results of the whole expression on their right and (if dyadic) to the argument to their immediate left (these rules can be modified through the use of parentheses). In order to augment these primitive functions, APL allows users to create their own functions defined through the composition of primitive functions and data values. Such defined functions follow the same syntax rules as primitive functions, and are therefore called simply by appropriate placement of their name in an expression. APL further augments the power of its primitive functions with entities called operators, which act upon data and functions to produce functions as results.

Even though these functions and operators provide a powerful functional facility in APL, many proposals have been made to enhance this facility by defining additional primitive functions and operators, by removing some of the restrictions placed on defined functions, and by formalizing operators and increasing their role in the language.

III.A Primitive Functions

The discussion of primitive functions below is divided up into sections which cover proposals for changes in the behavior of primitives in general, groups of related functions to which changes have been proposed and a section for miscellaneous proposals concerning

primitives. For information on arrays of functions, see the Miscellaneous Extensions section.

Rank, Uniformity and Symmetry. Some proposals would change not just a single primitive function, but would change the way in which all primitives behave in certain cases. Proposals concerning uniformity and symmetry affect the way in which structural and mixed functions apply to arguments of higher than "normal" rank. Iverson [73] defines the notions of argument rank and result rank for functions in order to clarify the notion of the "normal" rank of argument to which a function applies (a formal definition is referred to in Orth [148], though none is given in [73]). For example, the reversal function (⌽) has both argument and result ranks of one, whereas the derived function summation (+/) has an argument rank of one and a result rank of zero. Both of these functions are extended to arguments of higher rank by applying them along the last axes of those arguments, or along some other single axis of the array through the use of an axis specification. In some cases functions are also defined to apply to arguments of a rank smaller than their function rank, e.g., the degenerate cases of inner product and matrix inversion for vectors and scalars, or reversal for scalars.

In order to generalize the way in which functions apply to arguments of rank greater than their function rank, Iverson observes that functions extended in this way must be uniform in the sense that when applied to arguments of a given shape, they return results of a shape fully determined by the shape of the argument. For example, the reversal function when applied to a vector always returns a vector of the same shape as the original, whereas the summation function when applied to a vector always returns a scalar. Such uniform functions can be applied to arguments of high rank by splitting the arguments into collections of arrays along their last axes, applying the functions to the resultant arrays of appropriate rank, and then reassembling the results (which will share a common shape because of the uniformity requirement).

In some cases, these rules for uniform application would conflict with other proposals to extend certain functions. For example, under the rules of uniform application the grade functions (⍋ and ⍒) would apply separately to the individual vectors within a matrix, but another proposal (see Sykes [181]) would apply the grades to a whole array at once, interpreting positions on different axes as having different significances for ordering the final result. Uniform application also conflicts with the notion of symmetric function definitions.

Michel and Jenkins [83] propose alternative definitions for several primitive and derived functions, calling the functions so defined symmetric functions. These functions are symmetric in that they apply equally to all axes of an array and thus are not biased towards the last axes. For example, they define reduction so that the summation function (+/) would sum along all axes, returning a scalar result when applied to an array of any rank; similarly they define reversal so that it would reverse an array along all of its axes. In effect, the symmetric functions so defined have unbounded or infinite argument ranks (the same may be said of some current primitives such as ravel). This means that the symmetric functions require an explicit use of the axis operator in order to be applied to any limited number of axes of an array, whereas the uniform application rules require an explicit use of the axis operator only when the function is to be applied to axes other than the last. An issue closely related to those of uniform application and symmetric function definitions is the design of more sophisticated axis operators for APL; for more on this issue see the section on Axis Operators.

Scalar, Laminar and Rank Extension. APL's rules for scalar extension are a simple but useful way of allowing functions to be applied between scalars and arrays of higher rank where conforming arguments would normally be needed: the scalar argument is simply replicated to the appropriate shape before being applied to the array. For example, in the expression "3 + 4 4⍴0", the scalar 3 is extended to an array of shape 4 4 before it is added to the other argument. Scalar extension rules are currently used only with primitive scalar dyadic functions when applied between scalars and higher rank arrays, but proposals have been made to generalize this facility to be applicable in other contexts as well.

Several authors [89, 115, 121] propose a generalization of scalar extension that would allow one argument (of a dyadic scalar function) of any rank to be replicated along new axes in order to conform to another argument of higher rank. This facility would allow, for example, a vector to be added to each row or column of a matrix, or for a matrix to be added to the planes of a rank 3 array. Different proposals achieve this effect in different ways: Breed proposes an extension to the rules for applying dyadic scalar functions to arrays of differing rank (see Brown [18]). Jizba [89] proposes a new operator called the distributed product. Mebus [121] suggests using the expand function to allow the

replication of laminae of an array prior to function application (see below). McDonnell [115] proposes extending the axis operator to act on dyadic scalar functions (e.g., "*V*+[1] *M*" or "*V*+[2]*M* " to add a vector to a matrix). This last proposal has been implemented in Burroughs APL/700 [193] and on STSC, Inc.'s NARS system [23] (on the Burroughs system the axis information may be elided to default to the last axes of the array). Jenkins and Michel [83] note that the suggested default rules for their axis operator by-slice yield this interpretation for scalar functions.

Another generalization to scalar extension can be made by allowing a similar facility to be used in the application of mixed functions to arguments whose ranks do not agree for the purposes of the function application; this facility is called laminar extension by Mebus [121]. He notes that this facility is general enough to encompass the case of scalar function application and is currently implicit in the mixed functions encode and decode and in functions derived from the inner product operator (Jayasekera (see Keenan [91]) notes that encode and decode are anomalous for this reason). Mebus' proposal calls for the expand function to be extended to allow replication of laminae along axes of length one and along new axes created with a fractional axis specification. The generalization of scalar extension to higher rank arrays and to mixed functions can also be achieved through the use of more sophisticated axis operators; see the section on Axis Operators.

Mebus [121] notes that his proposal allows the creation of new axes of length one in an array even when replication is not specified. This facility is called rank expansion by Lucas [105], who decomposes the capabilities inherent in lamination into rank expansion, axis specification and function action along newly created axes. He continues by proposing several operators which would combine these capabilities in various combinations with a function specified for action along the new axes. Gilmore [51] proposes the shake function (from "shape" and "take") to allow explicit rank expansion without replication or function action, and to allow the "coalescence" of several axes into a single axis.

Indexing. Indexing is probably the most commonly used array function, and is indeed a necessity in standard programming languages which do not manipulate whole arrays. Unfortunately, indexing is also syntactically anomalous as it is currently defined, for it is a single function denoted by two widely separated symbols ([and]), and it employs the semi-colon as a non-functional separator. These anomalies might seem balanced by the familiarity of this notation but they lead to three major problems with indexing: in its current form, indexing is syntactically bound to the rank of the indexed array, and instances of indexing thus cannot be easily generalized to higher ranks; the index argument is not a simple APL array (some call it a list), and thus cannot be easily manipulated or assigned a name; and finally, it would be difficult to use indexing in its current form as the argument to an operator. On the other hand, the current form of indexing is a powerful facility, which encompasses at least three different capabilities (after Lewis [100]): simple indexing, which selects a single element of an array by specifying only one position per axis; combinational indexing, which selects an array of elements by specifying vectors of positions for each axis; and slice indexing, which allows the selection of all elements along an axis through the elision of position specifications for that axis. One form of indexing which is often desired but which is not available through the current facility is one in which a number of unrelated elements are selected by specifying the indices of each of the elements separately (this is called scatter-point indexing by Cheney [23]).

Several proposals have been made to add new forms of indexing to APL which would encompass the capabilities of the current form but under a more regular syntax; other proposals would add new forms of indexing entirely. Lewis [100] defines several successively more powerful systems of functions in order to realize all of the capabilities covered by current indexing. Iverson [73] defines the from function (denoted by ⌷) to treat the rows of its left argument as indices into an array right argument, thus allowing scattered points to be selected. Pesch [157] defines an operator (I) which, when applied to a nested array of position specifications, produces an ambivalent derived function that allows both indexing and indexed replacement. Several other proposals for indexing which employ nested arrays are discussed in the section on Nested Arrays; of these, Jenkins and Michel [83]note that their choose function could be implemented for non-nested indices in a manner similar to Iverson's from function.

Both Iverson [73] and Ghandour and Mezei [47] propose that negative indices be interpreted as counting backwards from the last position on an axis just as non-negative indices currently count forward from the first position. This facility, called complementary indexing by Bernecky and Iverson [9], allows references to be made relative to the last positions on an axis without actually

computing the length of that axis. This scheme can also be used for axis indices with an axis operator (see Ghandour and Mezei [47]).

Many proposals have also been made to generalize the index generator (⍳), usually to allow it to apply to vectors. Holmes [68] suggests an extension that would simplify the generation of arithmetic progressions. Many nested array proposals define an index generator extended to take vector arguments to produce an array of enclosed indices into an array of the shape specified by the vector argument (see the section on Nested Arrays); in particular, Jenkins and Michel [83] note that their index generator (called odometer) could be modified to produce non-nested arrays, the rows of which would comprise the generated indices, in a manner compatible with the non-nested version of their choose function. Brown [18] defines his index generator interval to accept negative values, but notes that it does not behave as desired for vector negative arguments.

Other Selection Functions. A variety of proposals have been made to extend the selection functions take, expand and compress. Abrams [2] has described the take function as "overburdened" in bearing the capabilities for both selection and expansion, suggesting that the "overtake" portion of the function be handled by a new function or by other primitives. However, several other proposals have been made which would increase rather than decrease the capabilities encompassed by the take function: Nater [143] suggests that both the take and drop functions be defined to act along the last axis of an array if given a scalar left argument, that they be allowed to take an axis specification, and that "first axis" versions of each be defined analogous to first dimension reduction, scan and reversal. Iverson [73] uses the variant operator to allow fill elements to be specified for both the take and expand functions. On STSC Inc.'s NARS system, the take function is extended to allow the specification of a fill element by means of a dyadic function derived through the composition of take with its selector argument. This approach is used uniformly to extend the expansion and compression functions to mesh and mask (see the section on Mesh and Mask).

There is some controversy over the status of the expand and compress functions: although originally defined as functions, Iverson [73] has suggested that they be considered operators, both to rationalize their relationship to the scan and reduction operators and to allow the mesh and mask functions of Iverson [70] to be defined through them under this interpretation.

Bernecky has defined a proper extension of compression called replication which allows non-negative integer values on the left to specify the number of times the corresponding element of the right argument is to be replicated in the result. The replication function has been implemented by I. P. Sharp Associates, Inc. and by STSC Inc. [11, 23] and has been extended on STSC Inc.'s NARS system to allow negative values on the left to specify replications of the right argument's fill element in the result. This system also extends expansion to take signed integer values on the left in a manner similar to the extended version of replication.

Several proposals have been made to allow other selection functions to be used on the left of assignment as indexing is currently used; for a discussion of these proposals see the section on Assignment.

Searching. Searching for elements in an array can be done in two ways: either for a specific element or for a pattern of elements. The former is exemplified by the index-finder (dyadic ⍳) of current APL, which searches within its vector left argument for the first occurrence of each of the elements of its array right argument. Several authors have proposed extending this definition to find first occurrences (in row-major order) of elements in arrays of rank greater than one (see Brown [18], Ghandour and Mezei [47] and Jenkins and Michel [83]), using either nested vectors or rows in the result to hold the vector indices returned. The find function defined by Ghandour and Mezei [47] finds the indices of all occurrences (not just the first) of the elements sought.

In current implementations of APL, if an element is sought, but no occurrence is found, the index returned is one greater than the length of the vector being searched. This result is thus not a legal index for the vector, and subsequent indexing with this value will produce an error. Brown [18] defines the position scalar (⍬) to be returned as the index for elements not found in the array being searched, following Iverson's early use of the null character [70].

The second type of searching mentioned above locates not individual elements but rather the pattern formed by a series (vector) or array of elements. This kind of pattern searching is embodied in the where function (⍷) described by Mercer [125], which returns a boolean vector with ones indicating the beginning of an occurrence of a pattern within the vector being searched. A similar function is implemented as "string search" (⌷*SS*) on STSC Inc.'s APL*PLUS® system. Falkoff [36] gives models for a search function

which takes arguments of any (but equal) rank, with an optional axis argument to allow searching for a low-rank array within one of higher rank, and another optional parameter to specify "don't-care" elements for patterns which are not solid infixes.

Sorting and Grading. Two kinds of proposals have been made to extend the grade functions (⍋ and ⍒). The first, suggested by Sykes [181], would extend the domain of the grade functions to higher rank numeric arrays by treating vectors within an array as composite values for grading purposes, or by independently grading parallel vectors within an array. The second proposal, made by Smith [179], is to define dyadic versions of the grade functions to grade character arrays. The left argument of these functions is an array describing an alphabet used for ordering, with differences in positions along different rows, planes, etc., having correspondingly different significance for ordering. The right argument is the array which is to be graded, and in Smith's proposal it is treated as a matrix by raveling together all but the first of its axes.

Set Functions. Although several proposals have been made to include sets in APL as a formal data type (see the section on Sets in Data Types and Structures), many functions can be defined which informally treat vectors (or arrays) as sets. An example is the membership function (∈) of current APL. In order to treat vectors more like sets, several authors have defined a function which ravels its array argument and eliminates all duplicate instances of its elements. Iverson [73] calls this function nub (∪) and defines an "ordered" counterpart that sorts the unique elements in ascending order, as well as distribution and ordered distribution functions which facilitate the reconstruction of an array from its nub. He also defines subset and containment propositions (⊂ and ⊃) and the familiar proper analogues to each (⊆ and ⊇). A function similar to nub but called ⍪nique is implemented on STSC Inc.'s NARS system [23].

Several authors define functions to obtain the union, intersection and set difference (∪, ∩ and ~) of two "sets". In some of these proposals, the functions are defined to ravel array arguments or to take their nub before operating on them.

Scalar Arithmetic Functions. A number of changes and additions have been proposed to the scalar arithmetic functions of current APL. McDonnell [113] proposes proper extensions of the logical functions and and or (∧ and ∨) to represent the least common multiple (LCM) and greatest common divisor (GCD) functions. His proposal has been implemented by I. P. Sharp Associates, Inc. on their SHARP APL system. McDonnell [114] also proposes a refinement to the division function, suggesting that the result of 0÷0 should be 0 rather than 1, for both practical and theoretical reasons. Others (DeKerf [30] and Eisenberg [34]) have suggested rather that the expression should produce a domain error, or that its result should be specifiable through a system variable.

The circular function has been criticized for the rather unprecedented way in which its numeric left argument selects a function from the family of functions associated with the symbol. Several additions and refinements to this family of functions have been proposed by Penfield [155] to aid in the manipulation of complex numbers (for more information on complex numbers, see the section on Complex Numbers in Data Types and Structures).

Matrix Functions. Many functions of interest in linear algebra are arithmetic functions that are not scalar functions but which rather utilize both the array structure and numeric contents of their arguments. Examples from current APL are the plus-times inner product (+.×) and the matrix divide function (⌹). Several proposals have been made to extend the domain of the latter to singular and rank-deficient systems using generalizations such as the Moore-Penrose pseudo-inverse [32, 67, 101]. Another matrix function inspired by linear algebra is the eigenproblem primitive (⍐) proposed by Jenkins [79], which finds eigenvalues or eigenvectors of its matrix argument. The common matrix determinant is a special case (denoted -.×) of the more general determinant operator defined by Iverson [72]. The definition of the operator allows non-square matrices and uses a monadic derived form of the inner product syntax.

Miscellaneous Primitives. Following are descriptions of several miscellaneous primitive function extensions which have been proposed (or implemented) by a variety of sources. The equivalent function (also called match or identical) defined by various sources [9, 23, 47, 56] compares its whole array arguments for equality in rank, shape and all elements. Some proposals for convenience define a similar function called inequivalent (see for example Cheney [23]). Although they are often defined in the context of nested arrays, these functions are also very useful in comparing simple arrays. Another function defined in the context of nested arrays, but which is useful with simple arrays is the type function (⍑) implemented on STSC Inc.'s NARS system: it returns (for simple, heterogeneous

arguments) a zero if the argument is numeric and a blank if it is character (see Cheney [23]).

The same system also allows a first-dimension version of catenate (⍪) defined as an analogue to the familiar first dimension reduction, scan and rotate functions (⌿, ⍀ and ⌽), and extends the reshape function to empty arrays by returning the reshape of the prototype of the array.

III.B Defined Functions

In order to supplement its wide variety of primitive functions, APL allows users to define their own functions, these being composed from both primitives and from other defined functions. These defined functions can then be used in much the same way as primitive ones: they are called by placing their name in an expression, between or in front of their arguments, and their results are returned as a value at the point of call. Unfortunately, there are at least two cases in which defined functions may not be used exactly as primitives in current APL: they may not be ambi-valent and they may not be used with operators. Proposals which have been made to remedy these problems, as well as some that allow new forms of function definition, are discussed below. For information on comments and statement separation within defined functions, see the Miscellaneous section.

Syntax. In many cases, two primitive functions are represented by a single symbol which is interpreted as the monadic or dyadic form based on the context in which the function is called. On most systems this property, called ambi-valence by Iverson [75], is not extendable to defined functions. However, on both the SHARP APL system and STSC Inc.'s NARS system, functions can be defined with both a monadic and a dyadic form. Thus, on these systems, defined functions can be written which simulate primitives whose monadic behavior supplies an elided argument, or whose monadic and dyadic forms perform completely different functions.

Another way in which defined functions differ from primitives is in their exclusion from the domain of operators. Several proposals have been made to extend the domain of operators to include defined functions (see Ghandour and Mezei [47], Iverson [73] and Jenkins and Michel[83]), and at least one implementation (STSC Inc.'s NARS system; see Cheney [23]) has actually realized this extension. Part of the difficulty of this extension lies with the problem of extending the domain of operators to mixed functions, since defined functions are often mixed, and since this fact may be difficult for an implementation to recognize. The NARS system implementation overcomes this difficulty by using its nested array facilities to define the results of the application of operators to mixed functions.

Canonical Representation. In order to define a function in current APL, a character matrix representation of the function is constructed and then used to "fix" the definition of the function (this is often done with the aid of a function editor). This character matrix form, called a canonical representation, consists of a header line indicating the function's syntax and name localizations and several lines representing statements which are to be executed upon function invocation. These statements are executed sequentially (unless the flow of control is modified by a branching statement) and when execution is complete the result is returned as the value of the function through the variable designated for this purpose in the function's header. Several criticisms have been raised against the canonical representation form of function definition: one author [Ell] notes the fact that it will not preserve simple indentations which would aid in readability. Other authors (see Holmes [68]) criticize its non-function-like ability to act without arguments, or to return no results, and thus its ability to cause "side-effects" (changes to the environment outside the function). But certainly the most controversial feature of the canonically defined function is its control structure, the branch arrow.

Control Structures. Ever since the doctrine of structured programming rose to popularity, APL has been criticized for its simple (and sole) control structure, the branch arrow (→). Critics complain that the branch is too easily misused and that its misuse leads to non-modular code; they seek instead the traditional sorts of control structures found in other languages: explicit iteration control, block formation, conditional statements, etc. In response to these criticisms, several proposals have suggested new control structures for APL which would be implemented in a variety of ways.

Foster [40] defines a statement structuring syntax which allows statements to be grouped and to be performed conditionally or repeatedly. In a later work [41] he describes a scheme which combines a restricted form of branching with a generalized function calling mechanism to achieve better potential for program modularity. Kelley et. al. [92, 93] describe processors for two versions of the APLGOL language which compile APL-like programs with structuring

keywords into normal APL code (APLGOL-2 [93] allows de-compilation and editing). Kemp [94] describes similar facilities for pre-processing structured APL programs.

Reeves and Besemer [160] define new control "primitives" (represented by ↑ and ↓) which can be used to construct several standard structural schemes. Ching [25] has recently described a new construct called a module (inspired by similar constructs in the MODULA language) which can be used to effect better structuring by improving scope control and readability. Oates [144] proposes a combination of local directly defined functions (see below) and a flow control mechanism to allow for more modular program design.

In reaction to this plethora of control structure proposals, Wiedmann [187] considers that the structured programming fervor has abated somewhat and that the current view of the community is that the control structures of traditional scalar languages may be inappropriate for APL.

Direct Definition. Although most current APL systems offer only the canonical form for defining functions, many of them supply software which simulates a form called direct definition, which was first defined by Iverson [71]. This form of function definition takes a single APL expression in the variable parameters α and ω and interprets the explicit result of the function to be the value of that expression given when the parameters are assigned the values of the function's arguments. Any variables which are assigned within the expression are automatically localized, and thus directly defined functions are free from side effects. The direct form of definition also allows a simple conditional construct: a propositional (boolean valued) expression determines which of two alternative expressions will define the actual result of the function. These three expressions are separated by colons with the proposition in the middle: if the proposition evaluates to 0, the left-hand expression defines the result; if it evaluates to 1, the right-hand expression is used. Given the ability to reference a function within its own definition, this construction allows a simple way of defining recursive functions.

A number of proposals have been made (and at least one has been implemented) to supply APL with an extended version of direct definition through an operator which takes as its argument a character string denoting the expression to be used for the direct definition, and which returns the corresponding function as its result. Iverson [73] defines such an operator as a special case of composition between character arguments or character and null arguments. This early definition is somewhat different from more recent proposals in that it allows abstraction with respect to one or two variables as well as the familiar α-ω form (Iverson has since argued against this type of abstraction; see "Direct Definition" in [Asilomar]. Iverson and Wooster [78] define an operator (∇) which may take one or two character strings as its arguments to return an ambi-valent derived function (one argument may be replaced by the null symbol in order to obtain a monovalent result). Their definition provides for statement separation and control of sequential execution of these statements, as well as reference to the function within its own definition. A very similar operator has been implemented on STSC Inc.'s NARS system as described in Cheney [23]. Metzger [128] discusses a facility which he calls extended direct definition which allows such constructs as statement sequences, explicit globalization of variables, a conditional ("WHILE") loop structure, a loop initialization feature, allowance for early termination of loops and a "CASE"-like construction.

III.C Operators

Besides arrays of data and the functions used to manipulate them, APL has a third class of objects called operators. Operators are similar to functions in that they take arguments and return results, but different from functions in that their arguments are functions themselves (or data) and their results are functions; these resultant functions are called derived functions. Examples of operators are reduction (as in $\times/$), inner product (the "." in $\wedge.=$) and the axis operator (as in $\phi[3]$). Operators are only now being fully explored and were in fact fairly late in becoming recognized as a separate class of objects (see Falkoff and Iverson [Fal]).

For information on direct definition operators, see the section on Direct Definition in Defined Functions.

Syntax and Valence. Most of the recent proposals to formalize operators define them to be monovalent (i.e., either strictly monadic or strictly dyadic) in order to remain consistent with the syntax of current APL and also to cut down on the number of parentheses needed in expressions (see [75]). However, Iverson [73] suggests that the null (or jot $\circ$) be used as an empty argument to achieve effectively ambivalent operators (as in $\circ.\times$ as opposed to $+.\times$). Note also that most proposals allow the derived function results of operators to be themselves ambi-valent (e.g., both "$\phi[3]$ *A*" and "1 $\phi[3]$ *A*"). There is also much agreement that operators should have higher

precedence than functions so that, for example, the operator in $R+.\times XY$ acts on the times function before the function itself acts on the array.

There is less agreement as to whether monadic operators should take their arguments on the left or on the right; several proposals claim that the argument should occur on the left (as in +/), thus following an earlier precedent that the syntax of monadic operators should mirror that of monadic functions. Adherents to this view usually also define operators to have long left scope (i.e., to take the entire operator sequence to their left as their left argument), again mirroring the syntax of functions (for example, the expression "+.×.÷" is interpreted as "(+.×).÷"). Proposals which do not follow these rules include some earlier works (see Ghandour and Mezei [47], Gull and Jenkins [56] and Jenkins and Michel [83]) and most works treating (or inspired by) Array Theory (see More [130-138], Jenkins [86-88]).

In order to facilitate the construction of operator expressions using these rules, parentheses are allowed to surround expressions which result in (derived) functions.

Domain and Range. In current APL, operators are largely limited to scalar primitive functions for their arguments (the exceptions being for some mixed functions, e.g., $\phi[2]$). Proposals for formalizing operators often suggest that these restrictions be removed so that mixed functions, derived functions (the results of operator expressions) and even defined functions would be allowed as operator arguments. Of course, the discussion of the scope and binding of operators in expressions such as "+.×.÷" implies that derived functions would be allowed as arguments to further operators. Operator sequences of this type have been suggested by several proposals (see Ghandour and Mezei [47], Iverson [73] and Jenkins and Michel [83]) but have been implemented only on STSC Inc.'s experimental NARS system (see Cheney [23]). This system also allows some operators to be applied to functions which return no results (see Smith [178]).

It is difficult to understand how some operators would be applied to some mixed functions because the results of these functions are of in general of rank greater than zero and not of uniform shape; thus they could not easily be assembled into a result array of the appropriate shape. This problem is solved in some proposals [20, 21, 23, 47, 83] through the introduction of nested arrays and thus through the incorporation of nested array functions into the definitions of the operators themselves. Orth [148] considers this a bad solution to the problem, and gives examples of some difficulties that arise when using this method.

The same difficulty which accompanies the extension of operators to mixed functions is probably the largest single difficulty in extending operators to defined functions, since most systems do not retain information on the scalar or mixed properties of defined functions, and since many defined functions are mixed. Two other difficulties with allowing defined functions as operator arguments are the association of an identity element with a function for use with the reduction operator, and the calculation of the inverse of a defined function for use with the power or dual operators.

Some proposals have suggested that the range of operators be extended to include operators themselves (see Iverson [73, 74]) although there has been little discussion of the semantics of such extensions. The APL-inspired language ALICE (see Jenkins [86]) defines a hierarchy of objects based on their order. The order of an object is defined to be 0 for arrays, 1 for functions, and higher values for ALICE functionals, which are operators generalized to act upon and return results of mixed orders. A syntax for applying operators to operators is given by Georgeff et. al. [45, 46], based upon the needs of established parsing methods. Some have questioned the application of operators to exclusively data arguments, especially the rationalization of compression as an operator which takes a left boolean argument and returns the appropriate selection function as a result (discussed by Iverson [73]).

Reduction and Scan. Iverson [73] describes extensions of the reduction and scan operators to allow dyadic functions to be derived from either of these. In the case of reduction, the derived function's integer left argument specifies the length of a "moving window" over which the reduction is performed; thus the expression "$_2 -/V$" returns the pair-wise differences between elements of the vector. For negative left arguments of the derived function, the sense of the arguments is reversed. The dyadic reductions derived from scan are defined similarly.

One problem associated with extending the domain of reduction to new types of functions (mixed, derived or defined) is the question of how to define identity elements for cases when the derived functions are applied to empty arguments. Hoskin [69] describes a scheme which provides "pseudo-identity" elements for some primitives. Brown and Jenkins [21]

describe progress towards defining identity element expressions for certain functions and classes of functions, but also demonstrate that no identity expressions exist for certain other functions.

Combinational Operators. The inner and outer product operators of current APL have been called combinational because of the way in which their results depend upon different combinations of their elements. Brown [18], Ghandour and Mezei [47] and Jenkins and Michel [83] all re-define the outer product with a new syntax, replacing the two symbols (∘.) with one, in seeming criticism of the notion of a two-symbol primitive operator. Keenan [91] has criticized the inner product as being too specialized in that, e.g., an internal scan cannot be specified in place of the internal reduction. He also suggests that the definition of inner product be refined so that the last axes of both arguments be eliminated from the results rather than the last axis of the left argument and the first axis of the right as is currently the case (the former definition is more compatible with the last axes bias of uniform application, while the latter is more consistent with the traditional definition of matrix multiplication in linear algebra).

Iverson [72] has defined two new operators for the monadic derived cases of the inner and outer product symbols. The first is a generalized version of the determinant which takes any two scalar primitive functions as its arguments, so that the special case -.× is the familiar determinant function (he also describes an extension to non-square matrices). This operator has recently been implemented on SHARP APL. The monadic derived case for the outer product symbol (∘.) is called the function table operator and produces a function table of the shape given by the function's argument by performing outer products between index-generated vectors. This operator has been implemented on STSC Inc.'s NARS system [23].

Also implemented on STSC Inc.'s NARS system is a dyadic operator called convolution (⍣) which produces a dyadic derived function that performs a "moving-window" inner product while reversing the selected portion of the left argument. The operator can be used to find the products of polynomials in coefficient-vector form (using +⍣×) and to perform string searches [23].

Axis operators. The bracketed axis specification used with many functions was not initially considered an operator (see Falkoff and Iverson [Fal]), but has since been widely recognized as such. Undoubtedly this delay in recognition was in part due to its anomalous syntax relative to other operators: although it takes its function argument to the left, it takes its data argument (axis information) between two separate symbols ([and]). Several proposals have been made to extend the axis operator or to define new axis operators to replace or augment the current one.

One of the most common extensions to the axis operator is to allow vector axis specifications for at least some functions (e.g., ×/). Brown [18] defines such extensions for many functions using the bracketed axis specification syntax but expresses them in terms of "indexed functions" rather than explicitly as functions derived from an operator. He uses nested array functions to define a general method of extending functions to indexed versions: the arrays are first split along the specified axis or axes into subarrays; these subarrays are enclosed and the function is applied to all subarrays separately; and finally the enclosed subarrays are disclosed and reassembled into one array. Many subsequent proposals for axis operators define them in similar terms. Ghandour and Mezei [47] define an axis operator (:) and supply definitions for its results when applied to a variety of functions, often with vector axis specifications and with complementary axis indices (see the section on Indexing).

Jenkins and Michel [83] criticize both of these early proposals for axis operators as being too function dependent, noting that in both schemes separate definitions must be supplied for each function extended. In order to supply a more function-independent semantics, they first define an axis operator called by-slice in terms of nested array functions, and then give new definitions for many primitives which they call symmetric definitions (see the section on Rank, Uniformity and Symmetry). They attempt to supply default rules for the less general axis operator and non-symmetric functions of current APL in terms of their proposal, but find this impossible because of inconsistencies in the current interpretations of certain functions derived with the axis operator. The by-slice operator takes 2 or 3 data arguments (all potentially vectors) to specify the axes along which the argument (or arguments) are to be sliced, and the axes along which the results are to be placed. The syntax used is that of a bracketed semicolon list, the semicolons separating the vector slice specifications.

Iverson [73] defines two axis operators called nuax and coax(⍤ and ⍥) which allow functions to be applied along a certain axis (if of rank 1) or axes (if of higher rank), or to be applied along

the complementary set of axes to those specified (coax). He emphasizes that the result of these operators is different from that of axis specification since the latter specifies both argument and result axes whereas the former specifies argument axes only and always places the results along the last axes. Bernecky and Iverson [9] re-define these two operators (as forms of the on operator, denoted by ⍤) in terms of nested arrays, allowing two enclosed vector right arguments to specify left and right argument axes, but still forcing all results to be placed along the last axes. They also define a separate operator called along (⌸) that splits an array into a collection of subarrays along some axis and applies its function argument to this collection before reassembling the subarrays.

Keenan [91] criticizes the by-slice and nuax operators as being "multi-adic" and suggests in their place a combination of several facilities. He first defines uniform functions following Iverson [73] and suggests that they be applied to the appropriate last axes of an array. He then defines the unit rank operators (left,⊢ , and right ⊣) to limit their function arguments to apply to the specified ranks, and the endspose function (⍉) to move the specified axes of an array into the last positions. In this way he spreads the capabilities of the axis operators over several facilities, none of which need be multi-adic.

The experimental NARS system of STSC Inc. does not currently support any new axis operators, but has extended the familiar bracketed scalar axis specification to take and drop, dyadic scalar functions and derived functions produced through the each operator. Orth [148] has criticized the lack of more sophisticated axis operators on the NARS system, as he feels that the support of such operators is an important issue in language extension.

Compositional Operators. One of the fundamental capabilities which is necessary to facilitate the construction of operator sequences is that of functional composition. Composition provides a means of binding functions together so that they may be applied as a unit as arguments to other operators or to data arguments. Several forms of composition are widely recognized, the simplest of these being the composition of a dyadic function with a single data argument to produce a monadic function similar to the dyadic one, but with one argument fixed. The supplied data argument may be composed on either side of the function to produce fixed-left and fixed-right cases.

Composition between functions is somewhat complicated by the fact that the argument functions may be either monadic or dyadic, as may be the composite function produced. Thus composition may be defined to combine its function arguments and and the data arguments of its derived result in a number of ways. To help separate these cases from each other, both the SHARP APL and STSC Inc. NARS system implementations of composition provide several cases through the use of different composition operators (see Bernecky and Iverson [9] and Smith [178]). A special form of composition called the dual operator is defined by Iverson [73]. This dyadic operator produces a derived function which is a composite of its right and left function arguments and the inverse of its right function argument. Iverson notes that this operator provides an extension of the concept of duality as expressed, for example, in DeMorgan's laws (∧ and ∨ are dual with respect to ~).

Keenan [91] has criticized composition as being a special case of function definition and thus as insufficiently general for consideration as an operator. Orth [148] also notes that composition cannot be used to easily define certain functions, but seems to regard it rather as an adjunct to functional abstraction as embodied in the direct definition operator.

One difference between the composition operators as they have been implemented on SHARP APL and on STSC Inc.'s NARS system has been stressed by both Orth [148] and Bernecky and Iverson [9]. The difference lies in the manner in which a composite function derived from two mixed functions is applied to its array arguments; in particular, the composite function defined on the NARS system behaves exactly as would the two mixed functions applied sequentially, whereas on the SHARP APL system the two composed functions are applied as a whole to the units appropriate to the right function argument. For example, consider a function derived from the domino and reversal functions as it applies to a matrix argument. On the SHARP APL system the function would split the matrix into row vectors (as appropriate units for reversal), apply both reversal and domino in sequence to each of the vectors so obtained, and then reassemble the resultant vectors into a whole matrix result. On STSC Inc.'s NARS system, the matrix as a whole would be reversed along the last axis and then the domino function would be applied to the whole reversed matrix to yield the final matrix result. Orth [148] in particular considers this an important difference between the two implementations and favors the SHARP APL approach.

Power Operators. The power (or fold) operator is defined in several proposals (see Ghandour and Mezei [47], Iverson [73] and Cheney[23]). This operator is dyadic, taking a monadic function and an integer scalar (say N) as its arguments, and is defined to return a monadic derived function which applies the given function to its argument N times sequentially. For example, if we use ⍣ to denote the power operator then the expression "(F⍣3) A" is equivalent to the expression "F F F A". If the function argument has a computable inverse, then the derived function can be defined for negative integer powers as the corresponding power of the inverse of F. The power limit operator defined by Iverson [73] applies its function argument repeatedly until the results of two successive applications are equal. This operator thus provides a facility similar to the "WHILE" control structure of other languages. STSC Inc.'s NARS system [23] features both of these power operators, as well as the power series operator which accumulates the results of a series of increasingly higher power inner products between a square matrix and itself until two successive results are equal. Smith [177] describes a number of useful applications of this operator, including paragraph formation, mini-max problems and other problems involving the transitive closure or least-cost traversal of directed graphs. A dyadic derived form of the power series operator has also been added to the NARS system [178] to allow a vector of coefficients to be specified as the derived function's left argument.

Mesh and Mask. Iverson describes two functions called mesh and mask in his original work on APL [70] that have not been included in current APL, probably because they require three arguments. However, Iverson [75] describes how the expand and compress functions could be interpreted as operators, freeing them for new dyadic derived forms which could be used to denote mesh and mask. McDonnell [118] has suggested an extension to this scheme which allows positive or negative integer vectors (as opposed to boolean vectors) to be used as arguments to the mesh and mask operators. These integers would be interpreted as specifying replication factors, with negative factors selecting from the left argument (of the derived function) and positive factors selecting from the right. Zero-valued factors would select either from neither argument (mask) or select the fill element of the right argument (mesh). Versions of the mesh and mask operators have been implemented on STSC Inc.'s NARS system, but with a syntax slightly different from Iverson's (they are expressed as dyadic forms derived from the composition of the selection vector with the expand and compress functions rather than as dyadic derived forms of expand and compress operators; see Cheney [23]).

Another operator called mesh but having different semantics is defined by Jenkins and Michel [83]; see the section on Assignment.

Miscellaneous Operators. A number of miscellaneous operators are defined by Iverson [73]: the scalar operators are defined from the scalar functions as a type of composition between the given scalar function and the operator's two arguments. For example, the scalar operator ⍤ is defined so that the expression "A F⍤G B" is equivalent to "(A F B) +A G B" and similarly for monadic F and G. The derivative and difference operators are defined from the derivative operator and the related difference quotient expression from calculus. The commutator operator (⍨) commutes the sense of a function's arguments (so that A -⍨ B ↔ B - A) and is especially useful in building certain operator sequences (this operator has been implemented on STSC Inc.'s NARS system under the name commute; see Cheney [23]).

The identity operator yields the identity function of its function argument in the sense that × and * are the identity functions of + and × respectively (see Iverson [73] for more information). The variant operator (⍠) is used to obtain variant versions of functions which depend on some implicit argument such as ⎕IO or ⎕CT; thus ⍳⍠K yields a vector whose origin is K. The domain operator (⍶) derives from a function a proposition which determines whether or not the propositions arguments are in the domain of the function. The valence operator (⍶) may be used to fix its function argument as either specifically monadic or specifically dyadic through expressions such as "+⍶1" or "+⍶2".

User-defined Operators. Several proposals have been made which suggest that users should be able to define their own operators, just as they can define functions in current APL. The APL-based language ALICE allows the definition of functionals, which may be operators or even higher or mixed order objects (see Jenkins [86]). Georgeff et. al. [45, 46] describe a syntax which could be used for operator abstractions, noting that under their parsing schemes separate operators would have to be used for monadic and dyadic operator abstraction.

IV. Evaluation and the System Environment

Although its data structures and the functions used to manipulate them form the core of the APL language, in order to be of practical use in an implementation they must be embedded in a system which will allow them to be applied to solving real

problems. The APL system achieves this by allowing users to evaluate expressions; to read, write and store data; to assign meaningful names to data and functions; to manipulate and query the systems environment; and to control and monitor the actions of the system itself. APL is notable among programming languages for its interactive nature and for its self-contained system environment which shields users from outside operating systems. This feature makes APL particularly well-suited to those who desire the use of a computer but who do not wish to immerse themselves in the intricacies of a large operating system which encompasses several processors. On the other hand, this self-contained nature also makes it difficult for more sophisticated users to interface APL with other processors in a natural way.

Other areas where APL's system environment is found to be lacking include: difficulties in combining functions and data into easily interfaced packages, a lack of control over name scopes and bindings, problems with system manipulation from program control and with the automation of processes, and inflexibility in handling errors. Although less research has been done to formalize this area of concern than has been done in the areas of data structure and functional extensions, many proposals have been made that would help solve these problems.

IV.A. Names, Naming and Data Access

In order to store data (and functions) for use in later calculations, APL allows names to be assigned to these entities through the use of the assignment arrow (←). Sets of named entities are gathered together into collections called workspaces, which typically hold the functions and arrays that combine to handle some particular applications problem. Named data may also be shared between different users or shared across the environmental boundaries of function execution. The proposals discussed below suggest extensions to these facilities which would allow more general forms of assignment, which would generalize the concept of collections of named data and which would allow more flexibility in specifying name scopes.

Assignment. The assignment arrow is an exceptional "function" in APL because of its unique properties; unlike other primitive functions, assignment does not return a result (or at least does not print its returned result) when it occurs as the root (or principle) function of an expression, although it does return a result when used in the middle of an expression. Assignment is the only primitive function (aside from system functions) which causes side effects; i.e., changes in the system environment. Finally, assignment is the only function which does not evaluate one of its arguments, but rather acts upon an unevaluated name.

Assignment does not always take a simple name as its left argument: the indexed form of assignment (e.g., "$A[I]←B$") allows an expression involving a selection function and a name to be used as the target of assignment. This has the effect of assigning a value not to a named array itself, but to some location within a named array. Several proposals [18, 47, 68, 83] have suggested that this facility be extended to a more general one in which expressions involving named arrays and any selection function (not just indexing) be allowed on the left of assignment. Brown [18] achieves this through the manipulation of arrays in the name domain on the left of assignment using selection functions. Jenkins and Michel [83] demonstrate that expressions of this kind may be achieved through a purely syntactic transformation of expressions involving an explicit operator called mesh (⍀). One advantage to their approach is that the whole modified array is returned as a result (i.e., not just the modified portion), and it may be assigned to any variable and not just to the named array. Simple assignment in current APL is limited to taking a single name as its left argument, but STSC Inc.'s NARS system (see Cheney [23]) allows multiple names to be assigned simultaneously using strand notation. This form of assignment takes a list of names on the left and a vector of equal length (or a scalar) on the right, and assigns elements of the vector (or the extended scalar) to the corresponding names. Strand notation is still somewhat controversial, however, and strand notation assignment is particularly controversial (see Iverson et. al. [77]). Brown's definition of the name domain (see [18]) to the left of assignment also allows the specification of multiple names. Another proposed generalization of assignment would allow expressions of the form "$I←I+1$" to be shortened by allowing the assignment arrow to act somewhat like an operator, taking a function left argument and producing a function which modifies the value of its named left argument. For example, the incrementation expression above would be written "$I+←1$". This form of assignment has been implemented on Burroughs' APL/700 [193] and also on STSC Inc.'s NARS system (see Smith [178]).

Iverson [73] defines a new form of assignment (⍎) which is used to assign a name to a function-valued expression (i.e., an expression whose evaluation results in a function). Bernecky and Iverson [9] use the assignment arrow of current APL to assign names to derived

functions (i.e., the function results of operators). In combination with a direct definition operator (see the section on Defined Functions), assignment facilities of this type allow the assignment of a name to a defined function without recourse to the current function fixing methods.

Collections of Named Data. Current APL supports a two- (and sometimes three-) tiered system for storing and manipulating collections of named objects. The first tier of this system is the library, which is a collection of workspaces which is referenced by a number, and which typically contains a single user's data. The second tier of the system is the workspace, a collection of named functions and arrays which is itself referenced by name and which typically holds the functions and data which combine to solve some applications problem. Some current APL systems still support a third tier of this system called the group, which is a collection of named functions, arrays and other groups, and which is itself referenced by name; most systems, however, are withdrawing their support for the group facility. All of these types of collections are manipulated in current APL through the use of system commands; system commands are extra-linguistic inquiries and directives that may be entered in APL's immediate execution mode.

This system for accessing collections of named objects presents one major problem for the development of general-purpose applications packages: because of the extra-linguistic nature of the system commands, there is no way for these objects to be manipulated under program control. Some systems have attempted to solve this problem by allowing system commands to be executed under program control indirectly through the execute function. Several other systems have attempted to solve this problem by replacing their system command set with a set of system functions and variables that lie within the scope of the language and are thus able to be used under program control. For example, many systems now support the system function □*NL*, which allows the name usages in the current environment to be examined under program control. For more on the move from system commands to system functions and variables, see the section on the System Interface.

Another solution to the problem of manipulating named data under program control is the package data type implemented on the SHARP APL system (see Berry [11, 12] and [191]). A package is a collection of named functions and arrays (and possibly other packages) which has no external structure but which may be used to store, retrieve and, in general, exchange its contents. Packages are manipulated with system functions and are in general not in the domain of other primitive functions; this is largely due to the fact that they have no external structure (and thus are not suitable arguments to structural functions) and are not allowed to be items of an array.

Several proposals have discussed generalizations of the collections of named entities which exist in current APL. Ryan [163] briefly describes objects called name contexts which generalize properties of workspaces and function execution environments and which allow greater control over name bindings. Murray [140] defines similar objects called namespaces which provide facilities for gathering named data and functions into collections and for specifying the kinds of interactions that may occur between them. Finally, Crick [29] has discussed a very broad scheme to generalize many APL entities (arrays, functions, libraries, workspaces, groups, etc.) into a single type of entity called generalized objects. His scheme calls for the use of nested arrays that may be indexed with character strings and that provide better facilities for sharing and access control, and also for a new data representation for functions. The generalization provided through this scheme makes possible a uniform syntax for the manipulation of libraries, workspaces, arrays, functions and other objects.

Data Sharing, Access and Security. APL systems are typically interactive, multi-user environments. It is often desirable in such an environment to allow different users to share the same stored data and functions, but it is also desirable that users be able to protect sensitive data from being accessed by unauthorized parties. In order to allow data to be shared between different users, most current APL implementations include a shared variable facility. With this facility, two users share access to a single variable, and may explicitly assign values to this variable and query its state through shared variable system functions. Falkoff [35] discusses some of the implications of the shared variable facility, including the rationalization of system variables and system input and output facilities (□ and ⍞) as being variables shared between the user and the system. Shared variables are also used in many implementations to allow APL users to communicate with outside processes and system facilities, especially with file system processors (see the section on Files in Data Types and Structures).

Shared variables have been criticized for being insufficiently general in that they allow sharing between only two users (or processors); Shastry [173] describes

a shared variable facility which allows sharing between multiple users.

A less explicit form of data sharing is available through APL's library system, which allows users to access the libraries of other users as well as the "public libraries" available on most systems. In order to provide better security for user libraries, most systems allow passwords to be attached to libraries and workspaces by their creator. Although the password system provides some measure of security, it is insufficiently powerful and flexible for the needs of most users (see Wheeler [185]). Wheeler [185] describes the use of access matrices to provide better control over access to libraries and workspaces. These matrices have long been used to provide better security within file processing subsystems and would under this proposal allow users to specify exactly who may read, write, copy, save, etc. into or out from a given workspace or library.

Another security device found on many APL systems is a facility for locking function definitions, which bars them from all further scrutiny. This facility is again quite restrictive, as it does not allow any differentiation as to who can access the function, barring even the function's creator from further inspection of the definition. In his proposal for generalized objects, Crick [29] requires that access matrices be extended to all arrays and their components, and thus to the generalized objects representing libraries, workspaces, arrays, functions, etc. This scheme provides full and uniform security facilities to all parts of a system.

Name Scope Control. In current APL, the calling of a defined function initiates a new local environment for the duration of function execution (an environment being a combination of system information and a set of bindings between variable names and objects). The function definition specifies a list of variable names whose values in the calling environment are to be blocked from access from within the local environment; the values of all other variables from the global (calling) environment remain accessible during function execution. This dynamic scoping mechanism is simple and easy to use, but its very simplicity can lead to problems with name scope control. For example (following Seeds et. al. [170]), a calling function or environment is unable to protect itself from a called function because the responsibility for specifying scopes rests with the called function. Another problem is that the inadvertent omission of names from the list of local variables can lead to unforseen (and difficult to trace) name conflicts. Seeds et. al. [170] have described an extended scope control facility which allows a variety of scopes to be declared linking (or barring linkage) of variable bindings between the calling, current and called environments. The proposed facilities allow different types of scope to be declared for each variable in the function header list and also allow a default scope to be specified for any variables not mentioned. This proposal and several other ideas for scope control mechanisms are discussed briefly by Gilmore [52].

Miscellaneous. Both Abrams [2] and Brown [18] have expressed a desire for a "call-by-name" facility in APL which would allow a name (and its binding in the global environment) to be passed to a function unevaluated. This is similar to the FEXPR facility in LISP and to the treatment of the left argument to assignment in current APL. As demonstrated by Abrams [2], the passage of a character string representing a name and the subsequent use of the execute function is an insufficient solution since the name may be shadowed in the local environment by a localization and redefinition of the variable. Brown [18] shows how his define function and definition-of operator can be used to effect a similar facility which allows the passage of unevaluated functions and expressions to a called function. He notes that this method associates the "call-by-name" property with the actual parameter to the function (as opposed to the formal parameter or to the function itself). He also demonstrates that this method does not solve the problem with passing a shadowed name.

Brown [18] and Holmes [68] both discuss ways to implement what are commonly known as overlays, i.e., variables which refer not to separate locations in memory but rather to locations shared by some other variable. For example, such a facility would allow the main diagonal of some named array to be named itself, so that changes in the value of one would be reflected in the other. Brown achieves this facility through the use of specification in the name domain to the left of assignment or through the use of his activation function. Holmes defines the aspect function to allow the indexing of arrays through "aspects" of the arrays, which may be shaped differently from the original; the ability to name these aspects realizes the overlay capability. Brown notes in his proposal that strong conformability requirements are placed on overlays, and that re-specification of the overlayed array may invalidate future specifications or references to the overlay. Holmes suggests several ways in which these problems may be overcome by special interpretations of non-standard aspects

such as empty or character arrays.

IV.B. The System Interface

Current APL provides several means for users to query and otherwise interact with the system environment. These facilities allow users to perform a variety of tasks: signing on and off of the system; managing libraries, workspaces and other collections of named data; setting various system parameters; querying the state of system execution; and canonically defining functions. Although early APL implementations largely kept these functions out of the domain of the language itself, current trends seem to be in favor of greater program control over these facilities (see Falkoff and Iverson [Fa2]).

System Commands and I-Beams. Early implementations of APL provided access to workspace management facilities, execution state information and other system interfaces largely through extra-linguistic entities called system commands. At that stage in the development of APL these facilities were still considered to be separate from the language itself and so they were provided in a form which allowed their use only directly by the user in immediate execution mode.

This limitation was imposed through the use of the right parenthesis as an escape character which preceded all system command keywords and which therefore signalled the need for extra-linguistic action. These keywords were followed by one or more parameters which, being again outside the scope of the language, could only be constants and not variables or expressions to be evaluated by the system.

The desire to allow at least some sort of communication with the system from within program control led to the implementation of the I-beam functions (⍎). These pseudo-primitive "functions" allowed a small set of system parameters and account and timing information to be set or queried. However, the action of these functions were controlled by rather arbitrary numeric encodings, and they were still quite limited in application; a more syntactically regular and powerful form of system interface was needed to truly bring system management into the domain of the language itself.

System Functions and Variables. With the advent of shared variables in IBM's APLSV release of APL came the rationalization of a system interface as a set of variables shared between the system and a user's workspace (see Falkoff [35] for more on the implications of shared variables). This led to the definition of a large number of system functions and system variables which were given reserved names beginning with the quad character (⎕). System functions are able to handle a wide variety of tasks including function definition, name class and name usage queries, object erasure, data formatting and timing, account and execution state information. System variables are used to set and query a number of system parameters including index origin, comparison tolerance and printing precision. In addition, all of these facilities are available under program control, and system variables can even be localized within a defined function just as other variables are.

Now that the use of system functions and variables has become widespread, they have begun to be used to rationalize access to a wide variety of implementation-dependent facilities such as file systems, event trapping mechanisms and certain pseudo-primitive functions which have yet to be fully accepted as part of the APL language. Nonetheless, a number of facilities which are available through system commands still have no equivalents among the current set of system functions common to most implementations.

A notable example is the case of workspace management facilities; most systems have yet to define system function equivalents to system commands such as ⎕*LOAD* and ⎕*COPY*. Crick [26] has praised the "clean" implementation of such functions on the University of Massachussets APLUM system (see Wiedmann [188]). Wheeler [185] has recently proposed a similar but even more complete set of system functions for workspace management.

Myrna [141] has expressed concern over the proliferation of system functions and variables and over the lack of uniformity in their names; he suggests some conventions for both the semantics and naming of system objects which might help to alleviate these problems.

IV.C Input and Output

One system facility which is certainly necessary for real computing tasks is the ability to read and write information to and from the user. Current APL offers a variety of ways to perform these tasks. Input may be made in immediate execution mode (where the given input is interpreted as an expression to be evaluated), in an evaluated input mode (where the input is evaluated and the result passed to the calling function) or in an unevaluated mode (where the character string actually read is returned to the calling function. Output may occur either by default (if the result of an expression is not assigned to a variable the result is

printed out) or through the use of the explicit output facilities □ and ⍞ . All of the output facilities mentioned above default to the same output formatting conventions, but the formatting of numeric output may be explicitly controlled with the dyadic primitive format function (⍕).

Several proposals have been made to provide new facilities which would allow better and more flexible control over the environment in which input is evaluated, the occurrence and format of output and the kinds of input and output facilities which are available.

Evaluated Input. Wells [184] has suggested that the expressions which are read during evaluated input mode should be evaluated in the global environment of the workspace rather than in the local environment of the calling function. This refinement would solve problems that result from conflicts between global and local definitions of variables referenced in the input which is to be evaluated.

Sink and Display Potential. STSC Inc.'s NARS system recognizes a property of functions called display potential which controls whether or not the result of the function will display by default when executed from within a defined function. The display potential of a defined function may be turned off by placing the name of the function's result in braces in the function header. The display potential of certain system functions such as □*EX* is defined to be turned off. This facility is especially useful when used with functions whose results are sometimes useful but often not needed. STSC Inc.'s NARS system also provides a primitive facility called sink (monadic use of ←) which prevents its argument from being displayed. Default display of the results is provided in both cases (i.e., sink and display potential off) if the function line being executed is traced.

Arbitrary Input and Output. Several implementations of APL now provide output facilities which allow arbitrary transmission codes to be sent to a terminal unedited by the system, and input facilities which allow unedited codes to be read from a terminal. These facilities are particularly useful for sending control code commands to intelligent terminals and graphics devices. Some implementations use system functions to provide these facilities (e.g., the □*ARBIN* and □*ARBOUT* functions of SHARP APL; see Berry [11]); others use primitive facilities similar to □ and ⍞ (e.g., the ⍠ facility of UNIVAC 1100 APL; see [195]). Myrna and Ryan [142] discuss the extension of APL's input and output facilities to better accommodate modern terminals directly in order to provide more convenience and flexibility.

Formatting. Current APL provides monadic and dyadic primitive format functions (⍕) to convert numeric data to character form and to allow control of column alignment and digit display. These facilities are of limited usefulness by themselves because of their simplicity, and several proposals have been made to extend their capabilities or to provide other, more powerful formatting functions. The original proposal for the dyadic format function by Seeds and Arpin [169] allowed for greater control over the type of formatting used by adding a third element to the left control argument of the function.

Many APL implementations now support a system function □*FMT* which provides a very versatile formatting facility that allows inserted text and decorations, qualified display fields and a variety of special format field types. However, this function has been criticized for its use of a "list" argument specified with semicolons to separate expressions for several arrays.

Falkoff [37] defines a pictorial format function which uses a character vector left argument to specify the form used to display its array right argument. The format picture argument uses blanks, text symbols and digit codes to specify a wide variety of format types, and is notable in that its length corresponds exactly to the width of the formatted result (scalar extension notwithstanding).

For information on the formatting of nested arrays, see the section on Nested Array Input and Output.

IV.D Control of Execution

It is often very useful to allow the users of a system to monitor and control the system's execution or to allow different processes within a system to monitor or control one another in an automated way. Early implementations of APL included little if any facilities to monitor, control or automate the execution of the system. However, with the growth of the language into larger and more diverse applications within commercial production environments, such facilities have become very desirable. The proposals discussed below concern refinements and additions to APL in the areas of debugging facilities (such as tracing and monitoring facilities), the automation of processes and explicit user control over the handling of errors and other events.

Stopping, Tracing and Monitoring. Often during the development of an application it is convenient to be able to monitor the execution of functions, and thus to determine whether or not they are performing as intended. Because of its dynamic, interactive nature, APL has

always been a particularly good environment in which to perform such debugging tasks.

In current APL, debugging facilities are supplied through the stop and trace vector controls: lines of a defined function are set for stopping or tracing by specifying the appropriate control vector to be equal to a vector of the desired line numbers. The name of the control vector is gotten by prefixing the function's name with one of the prefixes $S\Delta$ or $T\Delta$. Upon execution of the line of the defined function, the function's name and the line number will be printed and either function execution will be suspended for lines set to stop) or the value of the result of the expression on that line will be printed (for lines set to trace). Stop and trace control values cannot be queried or otherwise obtained for use in calculations and are removed by assigning the empty vector to the appropriate control vector.

Several systems have regularized these facilities by implementing them as system functions called $\square STOP$ and $\square TRACE$, which in their dyadic forms take a function name and a vector of the line numbers to be affected as their arguments, and which in their monadic forms return the current line numbers which are set for the specified function (see for example the APLUM system; see Wiedmann [188]).

Another facility which would be helpful for debugging and especially for tuning algorithms would be one which would allow function execution time to be monitored. On the University of Massachussets APLUM system (see Wiedmann [188]) this facility is supported through the $\square LTIME$ system function which is defined to parallel the $\square STOP$ and $\square TRACE$ functions. Burroughs APL/700 implements a symmetric set of functions to set, reset and query settings for stopping, tracing and monitoring of function lines.

Kline [95] suggests a facility which would allow variable access to be stopped or traced; i.e., controls similar to the $S\Delta$ and $T\Delta$ controls for functions would halt execution or print a value upon every reference or re-specification of a variable. Samson and Ouellet [164] recognize all of these debugging facilities as being based on the trapping of certain events (e.g., the events of a function line being executed or of a variable being set), and thus subsume these facilities within the context of a more general event trapping facility.

Automated Execution. APL has traditionally been implemented as an interpreted interactive system which communicates directly with the user. Other programming systems have often been implemented in such a way as to allow their execution to be controlled automatically (such systems are often called "batch" systems). Abrams and Myrna [3] note that although most applications systems are most easily developed in an interactive mode, as an application matures and stabilizes an automated mode of execution becomes desirable. An overview of the automated execution facilities provided by two implementations (those of I. P. Sharp Associates, Inc. and STSC Inc.) is presented below. For further information on the facilities of these systems see Berry [11] and Abrams and Myrna [3].

Both of these implementations allow APL processes, or tasks, to be executed automatically by allowing the user to specify two files related to the task: one which represents a sequence of inputs for the task (a source file) and one which is specified to receive the output produced by the execution of the task (a sink file). Once these files are specified, a system function is used to specify other information about the task including identification, execution time limits, etc.

In general, two types of automated tasks can be run: those which run concurrently with the initiating (user's) task and those which are specified to be run automatically by the system at some later time. These two types of tasks are called non-terminal and batch tasks, respectively, by I. P. Sharp Associates, Inc. and detached and deferred tasks, respectively, by STSC Inc.

Event Trapping. Certain expressions in APL can be impossible to evaluate for a variety of reasons: the expression may be ill-formed or contain references to variables which have not been assigned a value; functions may be applied to arguments which are outside the function's defined domain or to arguments whose structures do not conform in an allowable way; and in some cases there may not be sufficient system resources available to evaluate an expression. In all of these cases, current APL systems suspend evaluation of the expression and print an error message to the user indicating the type of problem encountered and the point in the expression where the problem occurred. Current APL also allows the user to suspend evaluation manually by generating an interrupt signal from the terminal. In both of these cases, control is returned to the user and evaluation continues in immediate execution mode within the environment that was current when the error or interrupt occurred.

Although this kind of action by the system is desirable under certain circumstances (for example, during casual

exploration in immediate execution mode, or when an application is being developed and debugged), it is often desirable to allow other kinds of actions to be taken by the system, as specified by the user or programmer (for example, in security-sensitive situations or in applications where APL's role is not meant to be visible). Several implementations now support facilities for specifying alternative system action upon the occurrence of certain events; such facilities are often called event trapping mechanisms.

In general, three kinds of capabilities constitute an event trapping mechanism: a means for the system to specify the type of event which has occurred, a means for the user to specify the alternative action which is to be taken and (sometimes) a means for the user to simulate normal error handling with a specified error message. In most implementations the system specifies the type of event that has occurred by setting a system variable to contain a character matrix representing the normal error message. This information can then be used by a user-specified error handler to respond appropriately to different types of errors.

Several different methods are used in different implementations to allow specification of the action to be taken upon the occurrence of a trappable event: Hewlett Packard's APL\3000 uses the dyadic system function ⎕*EMOD* to specify an expression to be executed when an event of a specified type occurs (see Marcum [106]). The University of Massachussets APLUM system (see Wiedmann [188]) uses the system function ⎕*TRAP* to force a branch to a specified line of the executing function. I. P. Sharp Associates, Inc.'s SHARP APL uses a system variable, also called ⎕*TRAP*, to specify which of several different kinds of action will be taken upon the occurrence of events of the correspondingly specified type (see Berry [11]). Both STSC Inc.'s APL*PLUS system (see Gilmore and Puckett [72]) and a proposal by Samson and Ouellet [164] use system variables to specify latent expressions which are to be executed upon the occurrence of different events.

Both the SHARP APL and APL*PLUS systems and the latter proposal allow users to simulate error message interrupts by specifying the message with a ⎕*SIGNAL* or ⎕*ERROR* system function.

The proposal by Samson and Ouellet [164] suggests generalizations of several capabilities over some previous proposals, including a larger variety of trappable events and greater control over manipulation of the execution stack. This proposal also allows access to the context of evaluation which was current when an event occurred through system variables which hold the name of the currently executing function (⎕*F*) and the values of its left and right arguments (⎕*X* and ⎕*Y*).

V. Miscellaneous Extensions

Several proposals are discussed below which are difficult to classify because they do not seem to fit into any of the above categories or because they seem to fit well into more than one of these categories.

Comments. APL currently allows canonically defined functions to be documented through the use of comments (denoted by the comment symbol ⍝); the comment symbol is placed at the beginning of a line and all subsequent characters in that line are ignored for the purposes of function execution. Many implementations now allow comments to be placed at the end of a line, following an executable statement, and some allow this usage in immediate execution mode. Mengarini [124] has suggested that APL be extended to include a more functional type of commentary called formal comments (denoted by ⍞). These comments are simply boolean-valued expressions (propositions) that describe conditions which are desired or expected at some point during a function's execution; if these conditions are found not to hold (i.e., if any zeros occur in the result of the proposition) a formal error is reported and function execution is suspended. Thus this type of comment provides both documentation of a programmer's intent and a means for verifying that a function is executing as intended.

Statement Separation. Several APL implementations now allow a number of statements to be entered on a single line, separated by a statement separator symbol (usually the diamond symbol ⋄). The statements entered in this way are executed sequentially, just as they would be if entered on separate lines (except for some differences concerning branching). Most implementations that allow the use of the statement separator allow its use both in immediate execution mode and within canonical function definitions, although others define it only as a less powerful aid to function definition, interpreting each statement as a separate function line (see Crick [26]).

Crick [27] describes a more powerful syntactic construct called recipe idiom, which he claims allows APL statements to be written and read in a more natural fashion. Recipe idiom allows several expressions to be joined by a statement separator and provides a way to pass the results of these expressions on as arguments to subsequent expressions using

a stack facility.

Arrays of Functions. Several proposals have discussed the possibility of allowing arrays to hold functions as their elements. Iverson [75] mentions this idea in the context of concatenation and selection operators that would be used to manipulate such arrays. Cherlin [24] expresses a desire for this capability in new implementations, but does not describe any plans for its design. Brown [18] gives a proposal for creating arrays of functions and also describes rules for applying such arrays to data array arguments.

Alternate Token Forms. Since most terminals in common use do not support the full APL character set, many APL implementations allow special code-forms constructed from available characters to be interpreted as APL symbols. The schemes used to construct these code-forms encompass a wide variety of techniques ranging from mnemonic single-symbol substitutions to escape character sequences and complete spellings of the symbol names.

Crick [28] has described a complete and consistent proposal for for an ASCII notation for APL. The proposal includes ASCII symbol and keyword equivalents for all APL functions as well as a special construct for binding arguments to operators. He motivates adoption of the proposal from several points of view, including standardization, the broader acceptance of APL and the economics of printing terminals.

Graphics Capabilities. A common computer application which APL does not support directly is the creation, manipulation and drawing of graphics structures. Several proposals have been made to incorporate more primitive support for graphics in APL.

Bork [13, 14] has suggested that a primitive graphics facility be included in the language in the form of primitive input and output facilities which plot points and specify such output controls as scaling and transformation parameters. Galbraith [43] describes an implementation of such facilities. Hardwick [61] uses an extensible version of APL to define various graphic structures and the functions that manipulate them. This last approach avoids the addition of primitive facilities designed specifically for graphic objects and also provides for extensions to other user-defined data types.

VI. Conclusions

In this section I hope to draw some general conclusions about the past, present and future of APL language extension. The topics covered will include a discussion of the changing circumstances surrounding the language design effort and how they affect that effort; the identification of some major themes in current research and their relation to mainstream language design theory; and some questions about the ultimate goals of APL language extension.

APL's early design phase was characterized by a number of circumstances which influenced its character and which were beneficial to its development (following Falkoff and Iverson [Fa1, Fa2]): the design was carried out by a small group which included all parties involved in the research and implementation effort; all decisions were arrived at by Quaker consensus so that disagreements were resolved before any new feature was implemented; and finally, the design itself was motivated solely by the aesthetic, theoretical and practical concerns of the designers, relatively free from the influence of other languages and from commercial imperatives, so that major design changes could be made without concern for conformance to popular trends or for their effect on compatibility.

Since this time many changes have taken place in the circumstances surrounding APL language design: a large number of groups and individuals are independently involved in the research and implementation of extensions to the original language on systems supported by a wide variety of commercial and academic institutions. Although there is a good deal of communication between these parties, it is necessarily less frequent and less effective than it was among the members of the single small group that performed the original design. Often the research efforts and even the implementations of two groups diverge sharply in the absence of consensus agreement. Finally, the growth of the popularity and thus the visibility of the language has placed pressure on implementors to enhance its commercial appeal, to conform to the popular trends of mainstream language design and to remain highly compatible with previous versions of the language.

Some of these changes may not necessarily have a negative effect on the language; for example, the growth in the number of different research projects aimed at language design could easily benefit the extension effort by providing it with a broader, more diverse base of ideas upon which to draw. The commercial success of the language is certainly crucial in providing support for language extension research and implementation. Finally, the influence of other languages could also be of great benefit to APL in

its present stage of development; although in the past APL has often seemed far removed from the mainstream of language design research, a comparison of the major trends which are current in these two areas reveals some striking similarities.

The two most obviously important extensions which are currently being incorporated into APL are the generalization of arrays to nested arrays and the formalization and generalization of operators. Another important set of extensions which may not be so obvious is the move toward the subsumption of APL's naming and execution environments into the domain of the language itself. This trend is indicated by the increasing favor of system functions and variables over system commands and the incorporation in the language of ever more powerful facilities for scope control, environmental manipulation and controlled execution. A major unifying influence on trends in this area may be Michael Crick's proposal for generalized objects which would bring together workspaces, data, functions and even processes under a single construct (see Crick [29]).

How do these trends relate to trends in the design of other programming languages? For example, as Giloi observes [Gil], language designers outside of APL such as Edsger Dijkstra are finally beginning to recognize the power of treating arrays as whole entities. APL was also clearly a major influence on John Backus' concept of functional programming (see Backus [Bal]). Now that operators are becoming fully recognized in APL, it is possible that research in functional programming may converge in many respects with research on operator extensions and generalizations. Finally, the unification of APL entities brought about through Crick's generalized objects is very similar to the unification of objects in object-generic languages like Smalltalk. Experience with these languages may help determine whether or not such extensions might be appropriate for APL.

A question raised by Abrams [2] seems particularly relevant in the light of these comparisons: at what point do we stop extending APL and start designing an entirely new language? As he points out, attempting to extend the language indefinitely while preserving weaknesses or simple stylistic differences inherent in the original design may result in a clumsy, constrained or inconsistent language. Should we stop adding to APL and start designing new languages which better embody its basic principles or which exploit entirely new concepts? Can we continue to extend the language freely when faced with the compatibility requirements of a rapidly growing base of serious users?

Since Abrams first posed these questions, several languages have been implemented which are based on APL, but which occasionally generalize some of its capabilities and which depart from its unique style in other respects. Examples of such languages are the extensible languages X\APL and ALICE (see Braffort and Michel [16] and Jenkins and Michel [86], respectively), the Array Theoretic language NIAL (see Jenkins [88]) and the structured language APLGOL (see Kelley et. al. [92, 93]). Are the changes which these languages introduce for the better or the worse, or are they simply different? And are they truly incompatible with the basic assumptions of APL's design?

One controversial issue which currently threatens the unity of the language and its community of supporters is that of the definitional system to be used for nested array generalizations. The two major time-sharing services which offer APL have taken firm stances on their incompatible routes of extension in this area. Each of these services offers APL to a large and growing number of customers who use the language in serious production environments. As Anderson [5] notes, if one of these systems eventually changes in regard to its choice on this matter, the decision could have drastic consequences for the users of that system. Will the pressure of this possibility manage to preserve the differences between the systems and eventually create two dialects of APL?

One positive indication that at least some of the differences between different implementations of APL can be settled is the progress being made towards an international standard for the language. Whether such standards efforts can settle larger differences remains to be seen, but they have performed a great service by managing to resolve many of the irregularities which have been incorporated in the language up to this time.

Another promising sign for APL is the continuing evidence of support at the deepest levels for the original design principles of uniformity, generality and brevity of expression. Although some disagreements do exist over particular extensions, a strong, common sense of aesthetics based on these principles seems to exist in the community as a whole. This common sense of aesthetics is reflected in the parallelism of motives and methods that exists among the best of the extension proposals; these proposals strive to eliminate anomalies and special cases, to find new ways of generalizing seemingly disparate concepts into unified wholes, and to maintain a good correlation between simple notation and the powerful

concepts which it can express. Whatever course the language takes, close attention to these most fundamental principles will certainly continue to guide its form and character toward the highest ideals of power, simplicity and utility.

Acknowledgements

Phillip Abrams' paper "What's Wrong with APL?" was a major inspiration for my interests in APL language extension and thus for this paper helps; hopefully this paper to answer the related question " ... and how do we correct it?". The paper was begun during my employment at Michigan Technological University (though independently of my work there) and was first drafted as part of a special topics course at the University of Michigan under Prof. Uwe Pleban. The final revision and copy were written and prepared at Sperry Univac Corp., Roseville.

I would especially like to thank Prof. Pleban for his advice and support and for clarifying conversations on many topics, and Tom Smith of Sperry Univac for his support of the project. I would also like to thank many others whose interest and comments helped to shape my perception of the issues involved and to improve the overall quality of the paper; these include Nancy Sprague, Brenda Liimatta, Chip Morningstar, Bob Smith and Al Michels. Many people at Sperry Univac provided invaluable aid in producing the final copy, including Max Feuer, Tony Bjerstedt, Greg Nault, Chris Prengel, Sherri Harnden and particularly Dan Nissen and Jim Young. The conference referees and program committee (particularly Prof. Janko) were extremely understanding and encouraging in their support of the paper. I am also deeply indebted to many friends and especially to my parents for their personal support and encouragement while I was writing the paper.

Finally, I am of course indebted to the many researchers whose work provided the basis for this paper; I can only hope that I have done them justice in my treatment of their contributions to APL.

References

Bal -- Backus, J. Can Programming Be Liberated from the Von Neumann Style? Communications of the ACM, v21#8, ACM, pp. 613-641.

Bol -- Bork, A. Limitations of APL as a Language for Student-Computer Dialogues. APL Quote Quad, v5#4, ACM, Winter 1974.

Ell -- Elliot, Maurice. Decision Support Systems: The Power and Problems of APL. APL Users Meetings Proceedings, I. P. Sharp Associates, Inc. 1980, pp. 363-368.

Fal -- Falkoff, A. D.; K. E. Iverson. The Design of APL. The IBM Journal of Research and Development, v17#4, pp. 324-334.

Fa2 -- ---; ---. The Evolution of APL. SIGPLAN Notices #13, ACM, August 1978.

Gil -- Giloi, W. K. How Modern Is APL? APL80 International Conference on APL, North Holland Pub. Co., 1980, pp. 291-297.

Mcl -- McDonnell, E. E. Registry of Extensions. APL Workshop at Asilomar, APL Quote Quad v10#1, ACM, Sept. 1979.

Mc2 -- McDonnell, E. E. The Four Cube Problem: A Study in BASIC, APL and Functional Programming. APL Press, 1981.

Wil -- Wiedmann, Clark. Progress on an ANSI Standard for APL. APL81 Conference Proceedings, ACM, pp. 335-340.

A Bibliography of Extensions to APL and Related Topics

The bibliography which follows is an attempt to gather together references to the published research on extensions to APL with related material such as research in Array Theory. The entries are arranged alphabetically by the authors' last names, and chronologically within the entries for a given author. Works with multiple authors are listed under the name of the first author given by the publication and are not cross-referenced. Some of the entries refer to published reports of informal presentations and discussions held at APL workshops, etc.; these entries are generally restricted to cases where the material in question has not appeared elsewhere in published form.

Following many of the entries are notations of (roughly) three kinds: those which clarify the content of a work where its title is sufficiently vague or general; those which indicate changes and developments in an author's views over the course of several works; and those which note relationships among the works of one or more authors. These notations are included primarily to provide assistance

to those wishing to pursue certain topics in greater depth and are not meant to be conclusive summaries or careful historical analyses of any author's works.

For many of the same reasons which are stated in the introduction to the paper, this bibliography is undoubtedly inaccurate and incomplete in some areas (note especially that is limited to English language works). Again, it is hoped that further work in this area may eventually correct these deficiencies and perhaps result in an expanded version of the sort envisioned in [Mc1].

Two types of abbreviation are used in the bibliography: the first is the abbreviation of publication and report series titles; the second is a set of topic keys which are used (in lieu of the more extensive notations described above) to clarify the scope of a work, and which follow the entry, enclosed within parentheses.

Publication Abbreviations

APL IV -- Proceedings of the Fourth International APL User's Conference (Atlanta), June 1972.

APL V -- Proceedings of the Fifth International APL User's Conference (Toronto), May 1973.

APL73 -- APL Congress 73 (Copenhagen), North Holland Pub. Co., 1973.

APL6 -- Proceedings of the Sixth International APL User's Conference (Anaheim), ACM, 1974.

APL75 -- Proceedings of APL75 (Pisa), ACM, 1975.

APL76 -- Proceedings of APL76 (Ottawa), ACM, 1976.

APL79 -- APL79 Conference Proceedings (Rochester), ACM, 1979 (also APLQQ v9#4).

APL80 -- APL80 International Conference on APL (Leeuwenhorst), North Holland Pub. Co., 1980.

APL81 -- APL81 Conference Proceedings (San Francisco), ACM, 1981 (also APLQQ v12#1).

Asilomar -- "APL Workshop at Asilomar", conference report in APLQQ v10#1, Sept. 1979.

Minnow2 -- "Another Workshop Held at Minnowbrook", conference report in APLQQ v11#1, Sept., 1980.

IBMJRD -- The IBM Journal of Research and Development, IBM Corp.

APLQQ -- APL Quote Quad, ACM (journal).

IBM/Cm -- Research Report, IBM Scientific Center, Cambridge, Mass.

IBM/Ph -- Research Report, IBM Scientific Center, Philadelphia, Penn.

QUTR -- Queen's University Technical Report, Department of Computing and Information Science, Kingston, Ontario.

Topic Key Abbreviations

AEx -- Automated Execution
ArF -- Arrays of Functions
AT -- Array Theory
Cpx -- Complex Numbers and Functions
CS -- Control Structures
CT -- Comparison Tolerance ("Fuzz")
DDf -- Direct Definition of Functions
Emp -- Empty Arrays
EvT -- Event Trapping (Error Handling)
Fmt -- Formatting Primitive
Idx -- Indexing
IO -- Input and Output
Lam -- Laminar Extension
MP -- Multiprocessing (Co-routining)
NA -- Nested Arrays
Nam -- Names and Naming
Ops -- Operators
Pkg -- Packages
PrF -- Primitive Function Definitions
Prt -- Partitioning Functions and Operators
Set -- Sets and Set Functions
SCF -- System Commands and Functions
SLA -- Selection to the Left of Assignment
Srt -- Sorting (Grading) Primitives
Typ -- Types for Data
Unf -- Uniformity, Rank and Symmetry of Functions

Bibliography

1. Abrams, Phillip S. An Interpreter for Iverson Notation. Tech. Rep. CS47, Comp. Sci. Dept., Stanford University, Stanford, Calif. 1966. (NA)
   First appearance of nested array ideas.

2. ---. What's Wrong with APL?. APL75. (Nam, SCF)
   Broad survey of APL's weak points with some suggestions for changes.

3. ---, John W. Myrna. Automatic Control of Execution: An overview. APL79, 141-147. (AEx, EvT)

4. Alfonseca, M.; M. L. Tavera. Extension of APL to Tree-Structured Information. APL76, 1-23. (SCF)

5. Anderson, William L. APL 81: A View from the Sidelines. APLQQ v12#2, 4-5, Dec. 1981.

6. Benkard, Phillip J. Adding and Using Structure in General Arrays. APL81, 35-41. (NA, Prt, PrF)
   Gives two dyadic partition functions and compares these to operator proposals.

7. Berke, P. Data Design with Array Theory. IBM/Cm G320-2123, 1978. (AT)

8. ---. Tables, Files and Relations in Array Theory. IBM/Cm G320-2122, 1978. (AT)

9. Bernecky, Bob; Kenneth E. Iverson. Operators and Enclosed Arrays. APL Users Meetings Proceedings, I. P. Sharp Associates, Inc., 1980. (Ops, DDf, Prt, NA)
   Continues work of Iverson [73, 75] with some important changes.

10. ---. Representations for Enclosed Arrays. (NA)
    Compares three storage techniques.

11. Berry, Paul. SHARP APL Reference Manual. I. P. Sharp Associates, Inc., March 1979. (AEx, EvT, Pkg, SCF)

12. ---. How the Package Data-Type Has Affected Programming. APL Users Meetings Proceedings, I. P. Sharp Associates, Inc., 1980. (Pkg, Nam)
    Gives motivations, uses, implications of packages. Compares to nested arrays.

13. Bork, Alfred M. Graphics in APL. APL IV, 33-36.

14. ---. Correspondence. APLQQ v6#3, Fall 1975.
    Suggests that graphics capability be primitive in the language.

15. Bouricius, W. G.; N. R. Sorensen. An Informal Introduction to a Language and a Data Base. Structures and Operations in Engineering and Management Science, Tapir Publishers, Norway, 1981.

16. Braffort, Paul; J. Michel. X\APL: An Experimental Extensible Programming System. Tech. Report, Mathematique Universite, Paris XI, Orsay, France.

17. Breed, Lawrence M. Definitions for Fuzzy Floor and Ceiling. APLQQ v8#3, 17-23, March 1978. (CT)

18. Brown, James A. A Generalization of APL. Doctoral Thesis, Dept. of Systems and Information Sciences, Syracuse University, New York 1971. (NA, Ops, PrF, Man, Idx, SLA, Lam, ArF, MP)
    An early inspirational work suggesting numerous extensions in many areas.

19. ---. Evaluating APL Extensions. Proceedings SEAS78 Conference, 1978.

20. ---. Evaluating Extensions to APL. APL79, 148-155. (Cpx, NA, Ops)
    Suggests a classification system and guidelines for extension proposals.

21. ---; M. A. Jenkins. The APL Identity Crisis. APL81, 62-66. (Emp, Ops)
    Discusses definition of reduction on empty (esp. nested) arguments.

22. Burrill, J. H. Jr. Some Notes on Handling Errors in APL. APLQQ v9#3, 44-47, March 1979. (EvT)

23. Cheney, Carl M. APL*PLUS Nested Arrays Reference Manual. STSC Inc., 1981. (NA, Ops, PrF, DDf, Idx, IO, Prt, Set)
    Describes numerous extensions available on STSC Inc.'s experimental NARS system.

24. Cherlin, Mokurai. Nested Arrays: Potent New Structure. APL Market News #8, p. 6. (ArF, Ops)
    Praises STSC Inc.'s NARS system with some reservations; mentions further extensions.

25. Ching, Wai-Mee. Introducing Modules into APL. APLQQ v11#4, 12-17, June 1981. (CS)
    Describes new facility for modular packaging of functions and expressions.

26. Crick, Michael F. C. Variations on APL Flat Major. APL In Practice, John Wiley and Sons, 1980. (PrF, SCF, Set, Ops)
    Surveys some extensions currently implemented on various systems.

27. ---. Right to Left OR Left to Right! APLQQ v10#3, 21-26, March 1980.
Introduces "recipe idiom" to break down APL expressions using separator punctuation and a stack.

28. ---. An ASCII Notation for APL. APLQQ v11#1, 18-25, Sep. 1980.
Motivates and describes ASCII keyword equivalents for APL symbols and constructions.

29. ---. Should APL Be a Declining Language? APL81, 83-88. (NA, Pkg, SCF, Idx, Nam)
Describes broad scheme to generalize APL objects (arrays, functions, workspaces, etc.) and thus to eliminate declensions from the language.

30. DeKerf, Joseph L. F. The Story of 0÷0. APL80, 133-135. (PrF)
Suggests that a DOMAIN ERROR should result from 0÷0.

31. Deturck, D. M.; D. L. Orth. A Derivative Algorithm for APL. APL Users Meeting Proceedings, I. P. Sharp Associates, Inc., 1980.

32. Dubrulle, A. A. An Extension of the Domain of the APL Domino Function to Rank Deficient Linear Least Squares Systems. APL75. (PrF)

33. Edwards, E. M. Generalized Arrays (Lists) in APL. APL73, 99-105. (NA)
Early nested arrays proposal, differs from both floating and grounded systems.

34. Eisenberg, Murray. Zero Divided by Zero: Zero or One -- or Neither. APLQQ v11#1 9-10, Sep. 1980. (PrF)
Suggests that a DOMAIN ERROR should result from 0÷0.

35. Falkoff, Adin D. Some Implications of Shared Variables. APL76, 11-148. (SCF)

36. ---. A Note on Pattern Matching: Where Do You Find the Match to an Empty Array? APL79, 119-122.

37. ---. A Pictorial Format Function for Patterning Decorated Numeric Displays. APL81, 101-106. (Fmt)

38. ---. More on Strand Notation. APLQQ v12#2, 7-8, Dec. 1981.

39. Forkes, Doug. Complex Floor Revisited. APL81, 107-111. (Cpx, CT)
Presents alternate definition to McDonnell [112] and motivations.

40. Foster, Garth H. What Lies Beyond the Branch Arrow? APL75. (CS)

41. ---. On the Locus of Flow Within and Among Secondary Functions. APL79, 333-339. (CS, MP)
Proposes restricted form of branching and a more general function calling mechanism.

42. Fritz, Eddie; et. al. Debugging Aids. Asilomar. (Evt)
Discussion of stopping and tracing facilities.

43. Galbraith, D. S. Primitive Functions for Graphics in APL. APLQQ v7#1, 27-36, Summer 1976.
Proposes 3 primitives for graphics I/O, graphic environment modification and storage.

44. ---. A Conditional Read Function for APL. APLQQ v8#3, 24, March 1978. (IO)
Describes experimental read facility which gets input only if available.

45. Georgeff, M. P.; Fris, I.; Kautsky, J. The Effect of Operators on Parsing and Evaluation in APL. Computer Languages, Vol. 6, 67-78, 1981. (Ops)

46. ---; ---; ---. Parsing and Evaluation of APL with Operators. APL81, 117-124. (Ops)
Gives modified grammars to parse new operator expressions.

47. Ghandour, Ziad Jamil; Jorge Mezei. Generalized Arrays, Operators and Functions. IBMJRD V17#4, 335-352. (NA, Ops, PrF)
Proposes a rich set of extensions for a floating nested array system.

48. ---. A Simple Approach to the Empty Generalized APL Arrays. APL76, 178-188. (NA, Emp)
Proposes an alternative scheme for handling empty arrays than is given in [47].

49. Gilmore, John C. Tree Theory in APL. APLQQ v8#3, 37-38, March 1978. (NA)
An enjoyable spoof of nested array proposals.

50. ---; Thomas H. Puckett. A Latent-Expression Exception-Handling System. APL79, 244-248. (EvT)
Describes STSC Inc.'s event trapping facility which uses character matrix error identification.

51. ---. Axis Generation and Coalescence. Asilomar. (PrF, Lam)
Describes shake primitive (shape-take) to extend arrays with singular axes.

52. ---; et. al. Name Scopes. Asilomar. (Nam)
Discusses different proposals for name scope control.

53. Grossman, Richard. Programmed Error Recovery for APLSV. APLQQ v7#3, 7-11, Fall 1976. (Evt) Describes error-trapping system using latent expression-like construct and character matrix error representation.

54. Gull, W. E.; Michael A. Jenkins. A Contribution to the Development of Recursive Data Structures in APL. QUTR 75-38, 1975. (NA, Ops, PrF)
Early proposal for grounded nested arrays system.

55. ---; ---. Recursive Data Structures and Related Control Mechanisms in APL. APL76, 201-210. (NA, Ops, PrF)
Continues work from [54] with applications.

56. ---; ---. Recursive Data Structures in APL. Communications of the ACM, v22#2, 79-96. (NA, Ops, PrF, Emp)
Compares various nested array proposals and continues work of [54, 55] with some changes.

57. ---; ---. Decisions for 'Type' in APL. 6th Annual Principles of Programming Languages Conference proceedings, 190-196. (Typ, NA)

58. Haegi, Hans R. The Extension of APL to Treelike Data Structures. APLQQ v7#2, 8-18, Summer 1976. (NA, PrF, Ops, Idx)
Proposes what is actually a nested arrays system (using a canonical tree representation), stressing indexing. Also discusses composition as a generalization of reduction.

59. Hagerty, Patrick E. More on Fuzzy Floor and Ceiling. APLQQ v8#4, 20-24, June 1978. (CT)
Replies to [17] suggesting some changes.

60. ---. Floor. Asilomar. (CT, Cpx)
Presents revised opinions from [59] with regard to the complex domain.

61. Hardwick, Martin. Graphical Data Structures in APL. APL81, 129-136. (PrF, NA)
Describes nested extensible graphics structures and functions for manipulating them.

62. Harris, L. R. A Logical Control Structure for APL. APL73, 203-210. (CS)

63. Harris, Thomas, J. Event Variables--ON Conditions for APL. APL75,177-180. (Evt)

64. Hartigan, Bruce J. AP19 - A Shared Variable Terminal I/O Interface for APL Systems. APL81, 137-141. (IO)

65. Haspel, Chuck. More APL Symbols. APLQQ v6#4, p. 2, Winter 1976.
Gives further overstruck symbol possibilities (see [150]).

66. Hassitt, A.; L. E. Lyon. Array Theory in an APL Environment. APL79, 110-115. (NA)
Describes APL-like system for testing models of Array Theory.

67. Von Hohenbalken, B.; W. C. Riddel. A Compact Algorithm for the Moore-Penrose Generalized Inverse. APLQQ v10#2, 30-32, Dec. 1979.

68. Holmes, W. N. Of Noughts and IF's and Matrices -- Some Comments on APLQQ[9;2;]. APLQQ v10#3, 7-11, March 1980. (PrF, Idx, Lam)
Makes suggestions for axis specification for scalar functions, extended index generation for arithmetic sequences, and others.

69. Hoskin, Zeke. Redefining Reduction Along an Empty Axis. APLQQ v11#3, 17-18, March 1981. (Emp, Ops, PrF)
Describes scheme which provides pseudo-identity elements for some primitives.

70. Iverson, Kenneth E. A Programming Language. John Wiley & Sons, Inc., New York, 1962.
The original work describing APL as a mathematcal notation; uses a notation very different from current APL and includes some functions not yet incorporated into the language.

71. ---. Elementary Analysis. APL Press, 1976.

72. ---. Two Combinatoric Operators. APL76, 233-237. (Ops)
Describes operators for a generalized determinant and for function table generation.

73. ---. Operators and Functions. IBM Research Report #7091, IBM Corp., April 1978. (Ops, PrF, NA, DDf, Idx, Unf, Set)
Presents a rich and diverse set of ideas for extensions; later works expand on only some portions of this one.

74. ---. The Derivative Operator. APL79, 347-354. (Ops)
Discusses a derivative operator and its use in exposition. Gives functions for use in experimentation.

75. ---. The Role of Operators in APL. APL79, 128-133. (Ops, ArF)
Discusses syntax of operators and other general issues. Gives a number of examples of operators.

76. ---. Operators. Transactions on Programming Languages and Systems v1#2, 161-176, ACM, October 1979. (Ops, DDf)
Discusses operator concept and gives some proposals. Some differences from [73, 75].

77. ---; R. Smith; J. A. Brown. On Strand Notation. APLQQ v11#3, 3-8, March 1981. (NA, PrF)
A lively discussion of Strand notation; Iverson against it and Smith and Brown in favor of it.

78. ---; Peter K. Wooster. A Function Definition Operator. APL81, 142-145. (DDf, Ops, CS)
Proposal includes facility for control of multiple statement execution.

79. Jenkins, Michael A. The Design of an APL Primitive for the Eigenproblem. APL75. (PrF)

80. ---. The APL Workshop Session on Extensibility. APLQQ v8#2, 14, Dec. 1977.
Describes the development of ideas on extensibility and an implementation at Universite Laval.

81. ---; Trenchard More. The APL Workshop Session on General Arrays. APLQQ v8#2, 12-13, Dec. 1977. (NA)
Discusses the differences between the various systems.

82. ---; Jean Michel. On Types in Recursive Data Structures: A Study from the APL Literature. Proceedings of the 5th Jerusalem Conference on Info. Tech., 523-538, August 1978. Also QUTR 77-59, Dec. 1977. (Typ, NA)

83. ---; ---. Operators in an APL Containing Nested Arrays. APLQQ v9#2, 8-20, Dec. 1978. Also QUTR 78-60. (NA, Ops, Idx, Unf, Lam, PrF, SLA)
Presents powerful depth and axis operators for use with nested arrays, as well as other functions and operators. Discusses symmetry in primitives and other issues. Uses a new formal notation for discussion of arrays.

84. ---; et. al. General (Nested) Arrays. Asilomar. (NA)
Jenkins reports his "conversion" to the floating nested array scheme.

85. ---. On Combining the Data Structure Concepts of LISP and APL. QUTR 80-109, Sept. 1980. (AT, NA)
Derives a fundamental equation of Array Theory which strongly suggests use of the floating system.

86. ---; Jean Michel. ALICE: An Extensible Language Based on APL Concepts. QUTR 80-104, Nov. 1980. (NA, Ops, Typ)
Generalizes many APL concepts and adds a typing facility to achieve a powerful extensible language.

87. ---. A Development System for Testing Array Theory Concepts. APL81, 152-159. (AT, NA)
Describes the implementation and use of the NIAL interpreter.

88. ---. The Q'Nial Reference Manual. Queen's University, Kingston, Nov. 1981.

89. Jizba, Z. V. Distributed Product. APLQQ v6#1, 37, Spring 1975. (Lam, Ops)
Proposes the distributed product operator to allow vector/matrix operations.

90. Kajiya, James T. Generic Functions by Non-standard Name Scoping in APL. APL81, 172-179. (Typ, Nam, MP)
Achieves generic functions without specific data typing through hierarchical namespaces and a coroutine mechanism.

91. Keenan, Douglas J. Operators and Uniform Forms. APL79, 355-361. (Unf, Ops, Lam, PrF)
Describes extension of uniform functions (forms) to higher rank arrays. Defines unit rank operators to limit function ranks.

92. Kelley, R. A. APLGOL, An Experimental Structured Programming Language. IBMJRD, v17#1, 69-73, January 1973. (CS)
Describes an extended APL with ALGOL-like control structures.

93. ---; John R. Walters. APLGOL-2: A Structured Programming Language System for APL. APL6, 275-280. (CS)

94. Kemp, Franklin. Design of a Structured APL. APLQQ v9#1, 11-13, Sep. 1978. (CS)
Describes an APL preprocessor to handle DO and IF constructions.

95. Kline, Edward M. Variable Control. APLQQ v4#4. (EvT)
Suggests a stop/trace type of control to print information on variables each time they are accessed.

96. Lathwell, Richard H. APL Comparison Tolerance. APL76, 255-258. (CT)
Discusses motivations for comparison tolerance and derives definitions for tolerant functions.

97. ---. Some Implications of APL Order-of-Execution Rules. APL79, 329-332.
Examines conventions for determining order of execution and finds them inadequate.

98. ---. SHARP APL Multiprocessing and Shared Variables. APL Users Meetings Proceedings, I. P. Sharp Associates, Inc., 1980. (MP, AEx)
Describes the combination of shared variables and a detached execution facility to achieve a multiprocessing environment.

99. ---. The SHARP APL S-Task Interface. SATN-39, I. P. Sharp Associates, Inc., June 1981. (AEx, MP)
Describes the SHARP APL detached task facility.

100. Lewis, G. R. A New Array Indexing System for APL. APL75. (Idx)
Suggests the breakdown of indexing into more fundamental capabilities. Also suggests a hierarchical set of APL-like languages.

101. Lezotte, D. C.; J. J. Hubert. The Generalized Inverse. APLQQ v7#2, Summer 1976.
Discusses the Moore-Penrose inverse of non-square, possibly singular matrices and demonstrates its usefulness.

102. Lim, A. L.; G. R. Lewis. Towards Structured Programs in APL. The Computer Journal, v18#2 140-143. (CS)

103. Link, Donald A.; Martin W. Gardner. Deferred Execution: An ACE of an Application. APL79, 1-7. (AEx)
Describes the design and implementation of STSC Inc.'s facility for specifying the deferred execution of tasks.

104. Lowney, Geoffrey; Alan Perlis. Does APL Need Arbitrary Nesting? Minnow2. (NA)
Suggests that two levels of nesting in nested arrays suffice for most applications.

105. Lucas, Jim. Beyond Laminate: Generalizing Creation of New Dimensions and Function Action Along Them. APL81, 195-198. (Lam, Ops)
Defines an operator to allow rank expansion with possible function action along the created dimensions.

106. Marcum, Alan M. Secure Application Environments in APL\3000. APL79, 257-263. (AEx, EvT)
Describes a combination of exception handling and state indicator interrupt/return mechanism to achieve greater security.

107. ---. Multiple Execution Environments in APL. APL80, 105-111. (MP, CS)
Describes the implementation of multiprocessing facilities on Hewlett Packard's APL\3000.

108. Martin, G. A. The Solutions of Linear Systems in APL: Towards an Extension of Matrix Divide. APL80, 113-121. (PrF, Cpx)

109. Mayforth, Rick. APLUM - APL at the University of Massachusetts. APLQQ v6#1, Spring 1975. (SCF, EvT)
Describes several enhancements made to APL at the U. of Mass. including: execute primitive, tracing and locking facilities, and system function equivalents of system commands.

110. McAllister, B. I. Representation and Manipulation of Finite Sets. APLQQ v10#4, 8-12, June 1980. (Set)
Presents 3 possible representations for sets (vector, boolean, integer-encoded boolean) and their uses.

111. McDonnell, Eugene E. Integer Functions of Complex Numbers with Applications. IBM/Ph #320-3008, Feb. 1973. (Cpx)

112. ---. Complex Floor. APL73, 299-305. (Cpx, CT)

113. ---. A Notation for the GCD and LCM Functions. APL75, 240-243. (PrF)
Proposes extension of ∨ and ∧ to the GCD and LCM functions.

114. ---. Zero Divided by Zero. APL76, 295-296. (PrF)
Presents reasons for changing the result of 0*0 from 1 to 0.

115. ---. Sauce for the Gander (or Adding a Vector to a Matrix). APLQQ v9#3, 64-66, March 1979. (Lam, Ops, PrF)
Discusses various proposals for laminar extension and suggests extending the domain of the axis operator to scalar functions.

116. ---. Fuzzy Residue. APL79, 42-46. (CT, Cpx, PrF)
Discusses tolerant versions of residue and their effects on complex arguments and the representation function.

117. ---; Jeffrey O. Shallit. Extending APL to Infinity. APL80, 123-132. (PrF)
Discusses the motivations for, representation of, and use of infinite values and arrays with infinite axes.

118. ---. Mask and Mesh. Minnow2. (PrF, Ops)
Proposes the implementation of mesh and mask functions with the selector vectors extended from boolean to signed integer.

119. ---. An Implementation of Complex APL. APLQQ v11#3, 19-22, March 1981. (Cpx, PrF)
Describes the SHARP APL enhancements for handling complex numbers (follows Penfield [155]).

120. ---. Complex Numbers. SATN-40, I. P. Sharp Associates, Inc., June 1981. (Cpx, PrF)

121. Mebus, George. Laminar Extension: An Overlooked Capability and the Search for its Proper Home. APL79, 36-41. (Ops, PrF)
Suggests that laminar extension be available through the expansion function/operator. Discusses compatibility with the by-slice operator of [83].

122. Mein, Wm. J. Data Structure Extensions to APL: A Survey. M. Sc. thesis, Dept. of Comp. and Info. Science, Queen's University, Kingston, 1975. (NA, Ops, PrF)

123. ---. Toward A Data Structure Extension to APL. APL76, 308-313. (NA, Ops, PrF, Idx)
Reviews proposals for nested arrays, identifying critical issues and minimal capabilities.

124. Mengarini, William. Formal Commenting in APL. APLQQ v7#1, 11, Summer 1976. (EvT)
Describes formal comment expressions (signalled by "∩") which return a FORMAL ERROR if any element of the result is false.

125. Mercer, R. L. WHERE - ⍵. APLQQ v4#3, 18, April 1973. (PrF)
Describes a primitive which searches for occurrences of the right argument pattern in the left argument.

126. ---. Extensions of APL to Include Arrays of Arrays. Tech. Report COINS 701, Univ. of Mass., Amherst, 1976. (NA)

127. ---. A Based System for General Arrays. APLQQ v12#2, 18-21, Dec. 1981.
Describes an alternative to the floating and grounded array systems that purports to solve some problems of both.

128. Metzger, Robert C. Extended Direct Definition of APL Functions. APL80, 143-148. (DDf, CS)
Motivates and describes the addition of control structure and multi-statement capabilities to direct definition proposals.

129. Mezei, Jorge E. Uses of General Arrays and Operators. APL6, 334-348. (NA, Ops)

130. More, Trenchard. Axioms and Theorems for a Theory of Arrays. IBMJRD v17#2, 135-175, March 1973. (AT, NA)
Presents the formal axiomatics of Array Theory.

131. ---. Notes on the Axioms for a Theory of Arrays. IBM/Ph #320-3017, May 1973. (AT, NA)

132. ---. Notes on the Development of a Theory of Arrays. IBM/Ph #320-3016, May 1973. (AT, NA)

133. ---. A Theory of Arrays with Applications to Data Bases. IBM/Ca #G320-2016, Sep. 1975. (AT, NA)

134. ---. Types and Prototypes in a Theory of Arrays. IBM/Ca #G320-2112, May 1976. (AT, NA)

135. ---. On the Composition of Array-Theoretic Operations. IBM/Ca #320-2113, May 1976. (AT, NA)

136. ---. The Nested Rectangular Array as a Model of Data. APL79, 55-73. (AT, NA)
Overview of Array Theory, its development and the choices it makes on key issues.

137. ---. Nested Rectangular Arrays for Measures, Addresses and Paths. APL79, 156-163, (AT, NA, Idx)
Demonstrates the generalization of measures, addresses and paths for arbitrary arrays (i.e., not just simple integer vectors).

138. ---. Notes on the Diagrams, Logic and Operations of Array Theory. Structures and Operations in Engineering and management Science, Tapir Publishers, Norway, 1981. (AT, NA)

139. Murray, Ronald C. On Tree Structure Extensions to the APL Language. APL73, 333-338.

140. ---. Namespaces: Semipermeable Membranes for APL Applications. APL Applications. APL81, 220-226. (Nam, SCF)
Describes the addition of namespace and interface structures, and functions to manipulate them, to achieve packages with highly controllable name sharing.

141. Myrna, John. Names for System Functions. Minnow2. (SCF)
Notes proliferation of system functions and suggests naming conventions.

142. ---; Jim Ryan. New Directions in Terminals. Minnow2. (IO)
Notes the need for APL to more fully recognize the abilities of powerful new display terminals.

143. Nater, Feico. APL\360 Enhancements. APLQQ v6#1, Spring 1975. (PrF, SCF)
Suggests scalar extension, axis specification, and first axis versions for ↑ and ↓. Also suggests SAVE system function.

144. Oates, Richard H. Iota Flow with Direct Local Functions. APLQQ v11#3, 9-17, March 1981. (CS, DDf)
Describes the use of a forking function and directly defined local functions to improve program structure.

145. O'Dell, Michael D. APL/XAD: An Extension of APL for Abstract Data Manipulation. APL6, 405-413.

146. Orgass, Richard J. The 1E6?1E6 APL Workshop: Another Overview. APLQQ v8#2, 8-11, Dec. 1977. (NA, Nam)
Describes nested arrays as both too general and too restricted a solution to APL's data structure problem. Also gives a description of namespaces.

147. Orth, Donald A. A User's View of General Arrays. IBM Research Report #RC 8782, IBM Corp., Apr., 1981. (Ops, NA)
Examines the utility of adding nested arrays to APL (using either the grounded or floating systems) and concludes that they may be undesirable in the language.

148. ---. A Comparison of the IPSA and STSC Implementations of Operators and Nested Arrays. APLQQ v12#2, 11-18, Dec. 1981.

149. Penfield, Paul Jr. Proposed Notation and Implementation for Derivatives in APL. APL V, 12-1 - 12-5. (Ops)

150. ---. APL Symbols. APLQQ v6#1, Spring 1975.
Discusses aesthetic choices in picking overstruck symbols; suggests several symbols and names for them.

151. ---. Notation for Complex "Part" Functions. APLQQ v8#1, Sep. 1977. (Cpx, PrF)
Presents 5 proposals for notation of real, imaginary (parts), magnitude and phase functions. Prefers the circular fucntion proposal.

152. ---. Extension of APL Primitives to the Complex Domain. APLQQ v8#2, Dec. 1977. (Cpx, PrF)
Second in a series of articles on extension of APL to the complex domain.

153. ---. Design Choices for Complex APL. APLQQ v8#3, 8-15, March 1978. (Cpx, PrF)
Third paper in a series; covers miscellaneous issues (notation, polar form, default interpreter, etc.).

154. ---. Complex APL - Comments from the Community. APLQQ v9#1, 6-10, Sept. 1978. (Cpx, PrF)
Reviews comments received in reply to the author's series of papers on complex number extensions to APL [151-153].

155. ---. Proposal for a Complex APL. APL79, 47-53. (Cpx, PrF)
Culmination of the author's explorations into a complex number extension.

156. ---. Principle Values and Branch Cuts in Complex APL. APL81, 248-256. (Cpx, PrF)
Presents choices for values of complex functions where such values are ill-defined or non-unique.

157. Pesch, Roland H. Indexing and Indexed Replacement. APL81, 258-261. (Idx, NA)
Proposes an indexing operator for use with nested array indices; monadic and dyadic derived functions provide for both selection and replacement.

158. Puckett, Thomas H. Improved Security in APL Applications Packages. APL6, 438-441.

159. ---. New Mexico State University Enhancements to APL\360. APLQQ v4#2, Jan. 1973. (EvT, SCF)
Describes an implementation of an event trapping facility and an execute function extended to act on system commands.

160. Reeves, A. P.; J. Besemer. Special Control Structures for APL. APLQQ v9#2, 23-31, Dec. 1978. (CS)
Derives several control structure patterns using new function-like control facilities.

161. Robichaud, Louis P. A. *\APL, An Extensible APL System. Centre de Traitement de L'Information, Universite Laval, Quebec, August 1977.

162. Ryan, James. Generalized Lists and Other Extensions. APLQQ v3#1, June 1971.

163. ---. Name Contexts. Minnow2. (Nam)
Mentions the name context idea for generalizing access to named objects and its facilities for scope control and data sharing.

164. Samson, Denis; Yves Ouellet. Convivial Error Recovery. APL81, 271-279. (EvT)
Surveys various facilities for event control, noting common qualities and generalizing these. Describes an implementation based on these findings.

165. Sarachik, P. E.; Ü. Özgüner. An APL Algorithm for Finding the Generalized Inverse of a Matrix. APLQQ v9#3, 39-43, March 1979.

166. Schmidt, Fleming; Michael A. Jenkins. Array Diagrams and the NIAL Approach. QUTR #81-131 Nov. 1981. (AT, NA, Fmt)
Describes and discusses a scheme for displaying nested arrays in both sketched and fully formatted forms.

167. Seeds, Glen M. APL Character Mnemonics. APLQQ v5#2, Fall 1974.
Suggests ANSI FORTRAN symbol equivalents for all APL characters.

168. ---. Fuzzy Floor and Ceiling. APLQQ v5#4, Winter 1974. (CT, PrF)
Suggests changes to definitions of tolerant definitions of floor and ceiling.

169. ---; A. Arpin. A Numeric-Controlled Formatter. APL76, 388-391. (Fmt)
Describes a formatting function with numeric controls for width, precision and format type.

170. ---; ---; M. LeBarre. Name Scope Control in APL Defined Functions. APLQQ v8#4, June 1978. (Nam)
Proposes a scheme for specifying 5 types of name scope and proves the exhaustiveness of this set in a general situation.

171. ---. Tolerant Representation. APLQQ v11#2, 15, Dec. 1980. (CT, PrF)
Motivates and describes a tolerant version of the representation (encode) primitive.

172. Shallit, Jeffery O. Infinite Arrays and Diagonalization. APL81, 281-285.
Discusses applications of infinite arrays in programming and exposition. Defines 2 diagonalization functions and discusses their implementation.

173. Shastry, S. K. A Generalized APL Shared Variable System. APL75.

174. Singleton, Sheila M. An Investigation of More's Array Theory. QUTR #80-99, April 1980.
Describes More's Array Theory and demonstrates that there is no translation of the floating system into grounded terms which can preserve the elegance and simplicity of the former.

175. Smith, Bob. A Programming Technique for Non-Rectangular Data. APL79, 362-369. (Prt, Ops)
Gives motivations, definition and applications for a partitioning operator for non-nested arrays. APL functions for simulating the operator are also presented.

176. ---. Nested Arrays: The Tool for the Future. APL In Practice, Wiley, 1980. (NA, Ops, PrF)
Brief, non-technical discussion of motivations and advantages of nested arrays.

177. ---. Nested Arrays, Operators and Functions. APL81, 286-290. (NA, Ops, PrF)
Describes some features and applications of STSC Inc.'s experimental NARS system.

178. ---. NARSNEWS Supplements to NARS Reference Manual. Available through STSC Inc.'s NARS system.
Describes extensions and modifications made to STSC Inc.'s NARS system since the publication of the reference manual, including new composition operators, modify assignment, new types of indexing, etc.

179. Smith, Howard J. Jr. Sorting - A New/Old Problem. APL79, 123-127. (Srt, PrF)
Discusses history of alphabetic sorting and presents APL functions which generalize the grade functions to allow higher rank arrays and specification of complex collating sequences.

180. Soop, Karl. Thoughts on Sets in APL. APLQQ v11#1, 10, Sep. 1980. (Set, PrF)
Describes a representation for sets in APL and functions defined on such sets.

181. Sykes, Roy. Multi-Rank Grade. Asilomar. (Srt)
Discusses extension of grade functions to higher rank arrays by grading subarrays as composite values.

182. Thompson, Norman D. Some Geometrical Consequences of Complex APL. APL80, 137-142.

183. Vasseur, J. P. Extension of APL Operators to Tree-like Structures. APL73, 457-464.

184. Wells, J. M. Quad Functions in APL\360. APLQQ v6#1, 38, Spring 1975. (IO)
Asks if quad input should not be evaluated in a global naming environment, as opposed to the current local environment.

185. Wheeler, James G. Improved Sharing of APL Workspaces and Libraries. APL81, 327-334. (Nam, SCF)
Discusses a re-design of APL workspaces and libraries to allow sharing and access control, and an expanded set of system functions with which to manipulate them.

186. Wiedmann, Clark. APL Problems with Order of Execution. APLQQ v8#3, 25-29, March 1978.
Raises some questions about APL order of execution and suggests the adoption of a consistent set of rules.

187. ---. Whither (Wither?) Control Structures? APLQQ v9#2, 21-22, Dec. 1978. (CS)
Suggests that control structures may be unnecessary and undesirable in APL.

188. ---. APLUM Reference Manual. Control Data Corp., Sep. 1979.

189. Wilhelmi, G. Formal Differentiation Using APL. APLV, 11-1 - 11-9.

190. N-Tasks and B-Tasks. SATN-4, Rev. 2, I. P. Sharp Associates, Inc., April 1978. (AEx)
Describes SHARP APL facilities for running detached and deferred tasks under program control.

191. Package - A New Variable Type. SATN-14, Rev. 2, I. P. Sharp Associates, Inc., August 1978. (Pkg, Nam)
Describes SHARP APL package data type used to aggregate named functions and data.

192. General Array Systems -- A Panel Discussion. APLQQ v12#2, 5-6, Dec. 1981. (NA, Ops)
Presents a summary of the panel discussion on nested arrays that was held at APL 81.

193. Burroughs APL/700 Users Reference manual. Burroughs Corp., March 1977.

194. APLSF Programmer's Reference Manual. Digital Equipment Corporation, Maynard Mass., May 1977.

195. APL 1100 Level 7R2 Programmer Reference. Sperry Corp., 1981.