10-1-1986

JMSL - a language derived from APL

John Mark Smeenk

Rochester Institute of Technology
School of Computer Science and Technology

JMSL: A Language Derived from APL

by John Mark Smeenk

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Jim Hammerton
Professor James Hammerton

Guy Johnson
Professor Guy Johnson

Rayno Niemi
Professor Rayno Niemi

RWW Taylor
Professor Robert Taylor

October 1, 1986

Title of Thesis: JMSL--A Language Derived From APL

I, John Mark Smeenk hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

John Mark Smeenk

Date: October 3, 1986

Abstract

A new APL-derived language called JMSL is presented which modifies seven aspects of APL so that many current and potential APL users could benefit from a language which is easier to learn, read, write, and maintain.

JMSL uses ASCII tokens instead of APL symbols to remedy interfacing, extensibility, and readability problems with APL. JMSL revises and extends APL built-in capabilities to provide greater expression and improved symbol-meaning correspondence. JMSL includes a new notation for nested arrays (a powerful data structure which combines the array processing of APL with the tree processing of LISP). JMSL provides hierarchical directories (similar to PASCAL or PL/I records) to allow structures to be indexed by name. JMSL modifies the traditional APL library/workspace storage interface by unifying the syntax of system commands in a way which allows UNIX-like directory storage. JMSL provides high-level control structures similar to those found in block-structured languages, including an event-handling mechanism. JMSL amends the APL scope rules to alleviate problems with side effects and object localization.

Some areas of future work are discussed, and a description of JMSL syntax and semantics is included.

Key Words and Phrases

APL; interactive computing; functional programming; directory; nested arrays; control structure; scope; programming language; mathematical programming; recursion; JMSL

Computing Review Subject Codes

Primary: Programming Languages.
Secondary: Procedure- and Problem-Oriented Languages.

Table of Contents

1. Preliminary Information.....1

1.1. Title and Acceptance Page.....2

1.2. Abstract.....3

1.3. Key Words and Phrases.....3

1.4. Computing Review Subject Codes.....3

1.5. Table of Contents.....4

2. Proposal.....5

3. Introduction and Background.....8

4. Aspect 1 (ASCII Tokens Instead of APL Symbols).....12

5. Aspect 2 (Revision/Extension of Built-in Capabilities)..21

6. Aspect 3 (Inclusion of Nested Arrays).....33

7. Aspect 4 (Hierarchical Directories).....43

8. Aspect 5 (Temporary/Permanent Storage Interface).....54

9. Aspect 6 (High-Level Control Structures).....69

10. Aspect 7 (Amended Scoping Rules).....78

11. Conclusions.....83

11.1. Summary.....83

11.2. Directions for Future Work.....85

12. Bibliography.....86

13. Appendix: JMSL Syntax and Semantics.....90

13.1. JMSL Syntax.....90

13.2. JMSL Semantics.....96

Proposed Work

My thesis is this: Many people, particularly those who are not primarily interested in machine efficiency or compatibility to existing systems, would benefit if certain aspects of APL were changed; these people would benefit because APL would be easier to learn, read, write, and maintain.

I propose to discuss in depth the issues below:

APL (A Programming Language) is a well-known programming language which has evolved for over 20 years and continues to evolve. Dr. Kenneth Iverson invented a mathematical notation for communicating algorithms in 1964; it was quickly modified for computer use and standardized as APL/360. Use of APL/360 suggested further modifications, and these have most recently been standardized by APL's user community in the 1983 APL Draft Standard. The Draft Standard describes a rather limited language which reflects the fact that many implementations differ on many details; the APL literature as summarized by the ACM Special Interest Group's APL Quote Quad Journal reflects many proposals and implementations of features which extend the APL described in the Draft Standard. Today APL faces new commercial challenges in the form of the rising popularity of microcomputers, new programming languages, and new operating systems.

Many people, particularly those who are not primarily interested in machine efficiency or compatability to existing systems, would benefit if certain aspects of APL were changed; these people would benefit because APL would be easier to learn, read, write, and maintain. These aspects are the APL character set, the built-in capabilities, the data structure and data manipulations, the object collection capabilities, the temporary/permanent storage interface, the control structures, and the name scoping. (Unless otherwise indicated, APL refers to the version of APL presented in the 1983 APL Draft Standard.) The term JMSL (John Mark Smeenk's Language) will denote the language resulting from a specific modification of APL to provide the seven features below. The APL character set should be replaced by the ASCII character set. This would help alleviate problems APL has with interfacing, extensibility, and readability. Historical schemes have focused on transliteration schemes and keyword token schemes to solve these problems. JMSL uses a keyword token scheme using the ASCII character set.

APL built-in capabilities should be revised and extended. Currently APL suffers from multiple meanings and uses per symbol, unnecessary built-in capabilities, inconveniently designed built-in capabilities, and missing built-in capabilities. APL literature points out many of these problems and solutions to them. JMSL draws on much of the literature in its resolution of the problems.

APL should enhance the power of the array data structure by providing an extension called nested arrays along with appropriate ways to manipulate these nested arrays. APL currently cannot conveniently store and manipulate heterogeneous (mixed numeric and character) data; APL also cannot easily store and manipulate tree structures of arbitrary depth. Historical approaches have included file systems, groups, packages, and Carrier Arrays; nested arrays (arrays which contain other arrays as elements) have been discussed in depth in the literature and implemented (in different forms) on several systems. Since nested arrays (with functions and operator to manipulate them) have proved to be powerful, JMSL provides nested array capabilities and ways to manipulate them.

APL should be extended to provide hierarchical directories. Current APL cannot show hierarchical relationships among named collections well, which means that objects cannot be meaningfully grouped to arbitrary levels, and also that information cannot be hidden appropriately. Historical solutions to this problem have included groups, packages, directories, and index-by-name proposals. Since directories with index-by-name capabilities show hierarchical relationships very well, JMSL provides them.

APL should improve its primary/secondary storage interface. APL as defined in the Draft Standard uses only workspaces to transfer data between primary and secondary storage; this solution is not efficient enough for some types of data transfer, does not address the problem of storing data which exceeds the (usually fixed) workspace size, does not allow multiple users to share data securely, and does not facilitate manipulation of permanent objects.

Historical approaches focused on file systems including transparent files, revisions to workspace commands, shared variables, and proposals to create and manipulate permanent directories. JMSL chooses an approach which reworks APL system commands so they may be efficiently used to modify permanent directories.

APL should add more high-level control structures. APL's only control structure is branching to a line, which leads to difficulty in specifying selection and looping control structures; APL also lacks a mechanism for event handling. Solutions presented in the literature are APLGOL (a fusion of APL expressions with ALGOL control structures), specialized control structures, and event handling. The JMSL solution to these problems involves use of special tokens and indentation to delimit control structures, automatic initialization of loop variables combined with a form of assignment called incremental assignment, and special constructs for event handling.

Finally, APL should amend name scoping rules. APL fre-

quently has problems with side effects due to an excessive number of global variables, the relative difficulty in localizing functions, the fact that each local variable must be declared in the function header, and the difficulty in identifying which variables are global. Historical solutions are programming disciplines to avoid and/or document global variables, dynamically defined functions, and revisions to local/global scoping conventions. JMSL solves these problems by allowing more than two arguments to user-defined functions, allowing local functions to be specified in function code, eliminating shadowing of global objects (thus eliminating the need to declare local objects), and requiring global objects to be permanent objects prefixed by a special token.

If these modifications were made to APL, it is likely that many users would see JMSL as a better programming tool than APL. There are many directions for future work.

As part of the proposed work, a reasonably formal description of JMSL will follow, including a Backus-Naur Form description.

Modifications to Proposal

Automatic initialization of loop variables was not included due to an unforeseen default case which rendered the idea nearly useless. A modified form of Backus-Naur Form was used to describe JMSL syntax.

INTRODUCTION: APL HAS EVOLVED FOR OVER 20 YEARS AND CONTINUES TO EVOLVE.

APL began as a mathematical notation developed by Dr. Kenneth E. Iverson, a Harvard professor. This mathematical notation, designed to facilitate the presentation of topics in data processing, was developed by Iverson and others in a purely academic setting. In 1961, this developmental activity culminated in the 1961 publication A Programming Language [1]. The notation was used in 1962 to describe the instruction sets of the IBM/360 family of computers in a formal manner. This work, documented in "A Formal Description of System/360" [2] used a modified form of the notation presented in A Programming Language. In 1964, work commenced at IBM on a machine implementation of APL. Machine implementation of APL necessitated many further changes to APL syntax and semantics; the result was the APL/360 language which became widely used in the late 1960's.

The implementation described in the APL/360 User's Manual [3] represented an early de facto standard. At first, IBM had the only implementation of APL; soon, however, other companies developed APL implementations. One aspect in which APL/360 was somewhat deficient was in system interfacing; later versions of APL developed shared variables and/or component file systems to enable APL to store large quantities of data and to interface with the non-APL world. A number of companies extended the set of primitives provided in APL/360 by implementing new system variables and functions whose names consisted of a ⎕ followed by one or more letters (e.g. ⎕IO).

By the late 1970's, there existed many versions of APL on many systems, with numerous implementation differences. Some companies followed the APL/360 standard fairly rigorously; others enhanced the power of APL significantly. Throughout the 1970's, and continuing today, the APL community shared implementation ideas, overviews of applications, algorithms, and other technical details through conferences and through APL Quote Quad [4], a quarterly journal of the Association for Computing Machinery's SIGPLAN Technical Committee on APL.

In recent years, the APL community perceived a need for a formal APL standardization undertaken through the International Standards Organization. Several years of discussion culminated in the Fifth Working Draft Standard for Programming Language APL, published in APL Quote Quad in December, 1983 [5]. The language described by this document will be termed "Standard APL" in this thesis. In many respects, Standard APL is a limited description of APL capabilities. Since implementations of APL vary widely, the committees had to document, insofar as possible, those language features common to

the majority of implementations without restricting possibilities for future growth. Thus Standard APL, like many standards, reflects a snapshot of the "past" of APL.

Three commercial mainframe APL implementations that have a broad base of users are especially noteworthy in that these implementations surpass Standard APL. These implementations are I.P. Sharp Associates' (IPSA's) SHARP APL system, Scientific Time Sharing Corporation's (STSC's) APL*PLUS and Nested Array System (NARS) systems, and IBM's APL2 system. (STSC's users have traditionally employed the APL*PLUS system; NARS is STSC's experimental system.) These implementations are incompatible with each other in many ways; serious differences of opinion exist in the APL community about a number of issues. Reference [6] records that when major proponents of these three systems were asked if they were willing to work together towards a single standard APL in the future, the question was unanswered.

Today, APL faces new commercial challenges in the form of the increased popularity of microcomputers, new programming languages (particularly ADA), and new operating systems (such as UNIX and its derivatives). Spreadsheet programs are replacing APL software in some business organizations. Myrna [7] discusses the commercial challenges in some detail:

> It appears that APL is used by only 5 to 10 percent of the programming population that might use it. After 15 years of commercial availability, many APL proponents find this a disconcerting percentage.

It is hard to quantify how many programs are written in FORTRAN, COBOL, PL/I, and similar languages when the applications' development and maintenance characteristics are better suited for APL. Myrna notes that the "opportunity cost" in developing APL applications is less than the cost of comparable applications developed in other languages. Myrna also notes that one reason many applications are not developed in APL is that the comparatively limited "experience base" of APL programmers is not conducive to supporting APL applications. Presumably Myrna's point is that APL is a worthwhile language which has not yet gained sufficient "critical mass" in popularity, and if APL were made easier to use, more programmers would use it.

Myrna continues with some rather daunting factors weighing against APL. They are summarized here because they illustrate the present commercial status of APL.

--APL is not widely taught in schools (since APL is relatively expensive, APL is unavailable on many low-cost microcomputers, APL is inaccessible to some computer science students, and qualified instructors are in short supply).

--APL has a weak tie with people's base of experience (since they cannot use familiar terminals, program editors, development utilities, and non-APL applications).
--APL has unique language characteristics which people find cryptic (such as APL's syntax and character set).
--APL has a premium cost (since APL interpreters require extensive development, fewer systems use APL, APL requires considerable storage, APL terminals are expensive, and APL applications frequently must be optimized for efficient execution).
--APL implementations often lack desirable facilities (such as full-screen editing, lower-case letters, and application development/maintenance systems)
--APL has historically not been standardized.
--APL is hard to justify in its initial purchase.

Myrna discusses solutions to some of these hindrances to APL popularity. Most of the hindrances are commercial or historical, rather than linguistic in nature and will not be addressed further.

APL literature is replete with proposals to rectify what one or more writers consider to be deficiencies of APL. Most of these ostensible deficiencies are of a linguistic nature. The sco e of these proposals (well-summarized by Ruehr [8]) suggests that APL needs some major modifications in order to remain viable in the long run. These modifications would logically be based on Standard APL.

The purpose of this thesis is to demonstrate that many current APL users and many potential APL users would benefit if certain aspects of Standard APL were changed; these users would benefit because the resulting language would be easier to learn, read, write, and maintain.

Certain classes of APL users would not benefit from the major changes to be discussed: those interested in compatibility with existing APL systems, those for whom execution efficiency is an over-riding concern, and those more interested in APL as a mathematical notation than as a production programming language. However, if Myrna's estimate that APL is grossly under-utilized is correct, there exists a potentially huge class of users who are interested in developing applications in a language which remedies widely-perceived flaws of APL.

The aspects of APL to be changed from Standard APL which will be discussed below are:

1. the character set
2. the built-in variables, functions, subscripting, operators, and input/output capabilities
3. the data structure and data manipulations
4. the object collection capabilities

5. the temporary/permanent storage interface
6. the control structures
7. the name scoping

Each aspect will be treated in detail; the problems and solutions proposed in the literature will be discussed, followed by the description of a specific solution to these problems. The specific modifications to each of the seven aspects are features of a language derived from APL which will be termed JMSL (for John Mark Smeenk's Language). A reasonably formal description of JMSL is included in the appendix.

ASPECT: USE OF ASCII TOKENS INSTEAD OF APL SYMBOLS

PROBLEM: THE APL CHARACTER SET CAUSES PROBLEMS WITH INTERFACING, EXTENSIBILITY, AND READABILITY.

The Standard APL character set is a combination of capital letters, underlined letters, digits, punctuation symbols, mathematical symbols, Greek letters, and novel symbols. This makes APL programs appear strikingly different from programs in other languages. There are so many symbols that a substantial percentage are usually implemented as overstruck characters, with a single character typed in and printed as a 3-character sequence (e.g. 'X' 'backspace' '_' for the character 'X').

Many users are inconvenienced to some degree by the APL character set. There are three major areas of concern: input/output interfacing, extensibility, and ease of programming.

Nearly all modern computer systems use either the ASCII or EBCDIC character set as a basis for all input/output. Extra interfacing is required to support APL characters. Rubber keyboard overlays printed with APL symbols, stickers to attach to each key, or keys printed with APL symbols are required to facilitate program entry. A special read-only memory chip is frequently needed to allow a screen to display APL characters since backspacing on most terminals is destructive (does not allow overstrikes). A special typing element, or some change to dot-matrix programming is required for printers; special protocol is needed for those printers not designed to allow backspacing. Usually a special character sequence such as 'control O' is needed to permit the user to switch between APL and ASCII as desired. These interfaces require significant amounts of human attention in practice, and if any of these interfaces fail, bizarre input/ output can result.

These problems have been overcome (with varying degrees of success) by all APL installations, but as Crick [9] and Myrna [7] point out, the interfacing costs may be prohibitive for many users. Moreover, existing facilities become harder to use because extra complexity has been introduced. When first introduced, APL needed to provide its own editor, the line-oriented ∇ editor, because existing editors could not handle APL characters. The ∇ editor is still used today, although it is very weak in comparison with screen-oriented editors that have been designed or modified for APL use. Myrna [7] observes that users frequently lose the ability to use existing facilities (editors, preprocessors, files created by other languages) because interfaces to these facilities are too time-consuming or too costly to provide. A fundamental difficulty is the fact that users who frequently switch

between APL and ASCII typing modes experience a slowdown in both modes since these users must constantly remember which character set is in use.

Extensibility is also a problem inherent in the APL approach to using single-character symbols for concepts. Crick [9] asks:

> Given that one defines a new composite character (e.g. ε̈) where does it fit in the collating sequence? How does one pronounce it? (epsilon-dieresis?).

Current data transmission techniques transmit either 7 or 8 bits at a time, so an upper limit of 128 or 256 symbols seems reasonable. Users would have difficulty typing in a system that features many more symbols than this (as Chinese typography shows). Standard APL uses 94 printable characters, plus additional overstrike sequences. Any number of characters may be overstruck, but in practice, overstrike sequences are limited to two printable characters per print position (except for '0' 'backspace' 'U' 'backspace' 'T', an escape sequence). As Ney [10] points out, many of these overstrikes do not make visual sense. STSC has already defined 150 system variables and functions of the form ⎕NAME in their APL*PLUS implementation; it is doubtful that sufficient overstrikes exist to translate these ⎕NAMEs in any meaningful fashion. IPSA took a different approach and defined fewer system variables and functions; but to provide similar overall capabilities, IPSA resorted to combining functionality in the same fashion that Standard APL combines 15 functions into dyadic ⍟.

This is related to APL's widely recognized problems with readability (and consequently with maintainability). It is inconsistent that APL denotes one concept with a Greek symbol (⍴) and another concept with a 2-letter abbreviation (⎕IO). One consequence of conserving symbols by defining several meanings for many of them is that new APL users frequently experience confusion in separating these meanings. Moreover, the conciseness of the APL character set seems to encourage one-liners such as the "APL pornography" cited by Gilman and Rose [11]:

```
R←(+/X×Y)÷((+/(X←X-(+/X)÷⍴X)*2)×+/(Y←Y-(+/Y)÷⍴Y)*2)*.5
```

which calculates the correlation coefficient between two sets of data.

Any language can be abused, but APL seems especially prone to misuse (as many maintenance programmers can attest). Crick [9] suggests that some users simply prefer a language based on written words rather than on visual symbols. Perhaps programmers using words (which are oriented towards natural

language) which would document algorithms more clearly than programmers using symbols (which are oriented towards machine efficiency or mathematical language).

A related difficulty with the APL character sets is that of producing a professional paper about APL which is easy for humans to read. In recent years, a number of visually appealing papers about APL have appeared, but many APL users do not have access to the appropriate text processing systems.

PROPOSED SOLUTIONS: TRANSLITERATION SCHEMES AND KEYWORD TOKEN SCHEMES.

There have been two main approaches to the above problem: transliteration schemes, which replace Standard APL characters with ASCII/EBCDIC character sequences, and keyword token schemes, which replace APL concepts with ASCII/EBCDIC character sequences).

Many implementations provide a transliteration scheme to assist users who wish to use APL from an ASCII or EBCDIC terminal which does not support the APL character set. Cain [12] gives a short example of what a "literal" translation from an APL keyboard to an ASCII keyboard looks like:

```
    G correlate list;a;b;q
[1]   space 1
[2]   L{'correlation matrix for variables '
      , 3 0 Nlist
            J
[3]   space 2
[4]   q{x[;list]
[5]   a{ 8 3 Na%(bJ.Xb{ 1 1 0a{(0a)+.X
            J                 \   \
      a{q_(Rq)R(+/[1]q)%(Rq)[1])*0.5
[6]   L{'variable', 8 0 Nlist
                          J
[7]   space 1
[8]   L{(4 0 N((Rlist),1)Rlist),[2]
            J
      (((Rlist),5)R' '),[2]a
[9]   space 2
   G
```

Figure 1. APL Transliteration.

Most programmers would consider this too indecipherable to be practicable. Crick [9] cites some of the more readable approaches to transliteration, such as Burroughs' [13] use of:

A(IS)''(RHO)(RHO)T for A←''⍴⍴T

but goes on to suggest that this is inadequate. Haynes [14]

describes Digital Equipment Corporation transliteration schemes; he describes an approach based on prefixing ASCII letters with @ and another approach based on 1- and 2-letter sequences. But he concludes that "the direction is toward plain language".

Crick [9] established a keyword token approach; the details are best illustrated by his example:

```
    ∇ COMPILE FNS;FN;T;I;M;MN;W;L;A;B;C;CD;D;E;MM
[1]   ⍝ OPERATE ON LINES OF FNS (SEP. BY COMMAS) IN FOLLOWING FORM
[2]   ⍝ A(B:C:D)E WHERE          (LEADING ⍝ OPTIONAL)
[3]   ⍝     A = HEADER
[4]   ⍝     B = CONDITION
[5]   ⍝     C = CODE INCLUDED IF CONDITION=TRUE
[6]   ⍝     D = CODE INCLUDED IF CONDITION=FALSE
[7]   ⍝     E = TRAILER
[8]   ⍝ ALTERNATE FORM: B = VARIABLE NAME (NO COLON)
[9]   ⍝ THERE MAY BE MANY CONDITION CLAUSES ON A LINE WITH NESTING
[10]  ⍝ GET NEXT FUNCTION
[11]  L5:FN←((T←FNS⍳',')-1)↑FNS ◇ →(0=⍴FN)↑0 ◇ FNS←T↓FNS
[12]   M←⍎□CR FN ◇ MN←1↑⍴M ◇ W←1↓⍴M ◇ I←1 ◇ MM←MN⍴1 ⍝ MASK
[13]  L10:→((I←I+1)>MN)↑L90
[14]   →('('∈L←M[I;])↓L10
[15]  ⍝ ANALYSE LINE L AND RECONSTITUTE IT
[16]  L20:A←((T←L⍳'(')-1)↑L ◇ L←T↓L
[17]   B←(L/(T←L⍳':)')-1)↑L ◇ L←(L/T)↓L
[18]   →(∧/T)↑NEXT ◇ L←A,(▼⍤1 B),L ◇ →L40
[19]   CD←((T←(+\-/L∘.='}{')⌿1)-1)↑L ◇ E←T↓L
```

Figure 2a. APL Original (truncated).

```
    define compile FNS;FN;T;I;M;WIDTH;LINE;A;B;C;CD;D;E;MASK
[1]  # Operate on lines of FNS (sep. by commas) in following format
[2]  # A(B:C:D)E where       (Leading # optional)
[3]  #    A = Header
[4]  #    B = Condition
[5]  #    C = Code included if condition is TRUE
[6]  #    D = Code included if condition is FALSE
[7]  #    E = Trailer
[8]  # Alternative form: B = Variable name (no colon)
[9]  # There may be many condition clauses on a line with nesting
[10] # Get next function
[11] L5: FN=((T=FNS find ",")-1)take FNS \ Go 0 if 0 eq shape FN \ FNS=T drop FNS
[12]  M=⍎rep FN \ LEN=1 take shape M \ WIDTH=1 drop shape M \ I=1 \ MASK=LEN shape 1
[13] L10: Go L90 if (I=I+1) gt LEN
[14]  Go L10 if not "(" member LINE=M[I;]
[15] # Analyse LINE and reconstitute it
[16] L20: A=((T=LINE find "(")-1)take LINE \ LINE=T drop LINE
[17]  B=(⍋in.of(T=LINE find ": )")-1)take LINE \ LINE=(⍋in.of T)drop LINE
[18]  Go NEXT if le.of T \ LINE=A,(form do B),LINE \ Go L40
[19]  CD=((T=(+.scan -.of.cols LINE eq.outer "}("find 1)-1)take LINE \ E=T drop LINE
```

Figure 2b. APL Keyword Translation (truncated).

This approach has been implemented on the IBM PC by STSC's APL*PLUS/PC, although the details vary from the above scheme.

JMSL SOLUTION: KEYWORD TOKEN SCHEME USING ASCII TOKENS.

To answer the problems described above, JMSL uses a keyword token scheme using the ASCII character set. (ASCII and EBCDIC are the only commonly used character sets, and since each printable character of ASCII is also a printable character of EBCDIC, anything written in ASCII is upwards-compatible regarding character set.)

The JMSL keyword scheme differs from other schemes mentioned, both for stylistic reasons and to better accomodate features described in subsequent sections. The naming convention permits (1) programmers to clearly show multiword variable names, (2) users to type exclusively in lower-case (if only built-in capabilities are used) or exclusively in upper-case (if only programmer-defined capabilities are used), and (3) a clear distinction between system capabilities and user-defined capabilities.

Although assigning one basic meaning to each ASCII symbol and avoiding repeated-character tokens (cf. use of * and ** in FORTRAN) are desirable, the richness of the APL character set in comparison with the ASCII character set and the additional features of JMSL make full realization of this goal nearly impossible to achieve without sacrificing readability.

Programmers would have little difficulty converting Standard APL programs to JMSL programs; perhaps a filter program could be written to assist such conversions. Since JMSL has capabilities which Standard APL does not have, converting from JMSL to APL would be more difficult.

The major points of the JMSL keyword scheme are as follows:

1. APL constants are translated literally to JMSL, substituting the ASCII characters " e - for the APL ' E ¯.

2. APL user-defined variable names are represented in JMSL by names consisting of an upper-case letter optionally followed by one or more upper-case letters, lower-case letters, and digits.

3. APL user-defined function names are represented in JMSL by names consisting of &, !, or % followed by an upper-case letter, optionally followed by one or more upper-case letters, lower-case letters, and digits. An initial & signifies a niladic function, an initial ! signifies a monadic function, and an initial % signifies a dyadic function. Standard APL requires each user-defined function to be monovalent (strictly niladic, strictly monadic, or strictly dyadic) and so does JMSL.

4. APL symbols representing primitive functions, system variables, and system functions are represented in JMSL by

names consisting of a lower-case letter optionally followed by one or more upper-case letters, lower-case letters, and digits. If a primitive or system function has both monadic and dyadic uses, each use is given a separate built-in name in JMSL. However, the following uses of functions in APL have JMSL symbols as well as JMSL built-in names:

Standard APL function	JMSL symbol
Monadic ⍴ ,	# _
Dyadic + - × ÷ * ⍴ ,	+ - * / ^ @ ,
Dyadic < = > ≤ ≥ ≠	< = > <= >= <>

Table 1. JMSL Symbols for Certain APL Functions.

Not all variables and functions are translated literally (see next section and appendix). In particular, monadic - in APL (e.g. -Y) is not provided in JMSL; its meaning is expressed using dyadic - (e.g. 0-Y) since JMSL uses - in a monadic-like way as a sign in numeric constants. Although the JMSL symbols <= >= and <> each require two characters, there is no ambiguity since > and = do not have monadic meanings.

5. APL symbols signifing input/output are translated according to the following table (for further information, see next section and appendix).

APL symbol	JMSL function
⎕←Y	pr Y (monadic function; result is Y)
⍞←Y	pr Y (same as above)
⎕ (not followed by ←)	input (niladic function with result)
⍞ (not followed by ←)	ask (niladic function with result)

Table 2. Comparison of APL vs. JMSL Input/Output.

6. APL (←) is translated to JMSL (:). The interpreter can insert a space after JMSL (:) if none was typed. APL labels are translated by separating them from any code on the same line and preceding them by the word (label) so as not to conflict with this symbolism for assignment. Example:

APL	JMSL
LABEL9:Y Z+2	label "LABEL9" Y: Z+2

This leads to functions with more lines; however, labels are used far less frequently in JMSL than in APL (see the section on control structures).

7. APL [; ( ) .] are transliterated to JMSL [; ( ) .] without change.

8. APL operators are translated as follows (see next section and appendix for details):

APL Operator	JMSL Translation
f/[Z]Y and f⌿[Z]Y	various built-in functions such as JMSL (sum Y) for APL (+/Y); for some APL choices of f there is no equivalent JMSL built-in function for (f/Y); an axis must be explicitly provided if the rank of Y is greater than 1
f\[Z]Y and f⍀[Z]Y	the cover function used for the corresponding reduction preceded by the operator c~; for example, JMSL (c~sum Y) for APL (+\Y); for some APL choices of f there is no equivalent JMSL translation for (f\Y); an axis must be explicitly provided if the rank of Y is greater than 1
X ∘.g Y	X g[] Y; the empty [] suggests "every scalar of X" g "every scalar of Y"
X f.g Y	X f[g] Y, where f is replaced by a corresponding built-in JMSL function (e.g. X sum[*] Y for matrix multiplication); for some choices of f there is no equivalent JMSL translation for APL(X f.g Y).

Table 3. Comparison of APL vs. JMSL Operators.

9. APL (⍝) is translated by JMSL(==); since JMSL does not use = monadically, there is no syntactic ambiguity. Anything following JMSL == on a statement line is considered comment.

10. Although it is not part of Standard APL, many implementations let users separate multiple statements on a line using the APL ◇ character. Statements are interpreted from left to right; if the ◇ occurs within a comment it is ignored. JMSL recognizes the utility of this and allows users to separate multiple statements per line with JMSL(!).

11. All other APL conventions, such as order-of-execution and the convention that multiple blanks are syntactically equivalent to a single blank except within literal constants, are unchanged in JMSL unless otherwise noted.

12. The ASCII symbols $ ` ' { } \ and ? are available for JMSL features to be described elsewhere.

One major implication of this change to the APL character set is that the standard ASCII sequence would be the logical choice of collating sequence.

This keyword token scheme has some disadvantages:

1. Users would lose visual conciseness. This is a disadvantage of any keyword scheme. The sample program from Crick (above) shows how printing in compact-print mode could partially compensate.

2. Users would be less likely to think in symbols. This is another disadvantage of a keyword token scheme; however, there is no reason why users could not use symbols (perhaps a variation of APL symbols) on paper or on a blackboard--as long as the users could convert the symbols to JMSL equivalents.

3. Users would lose typing compactness. Some users would not care; others could be accomodated using function keys (as suggested by Crick [9]) or with special "power typing" modes (as implemented by STSC's APL*PLUS/PC). These special typing modes could also be used to accomodate users who find upper-case/lower-case distinctions difficult to type; alternatively, user-defined functions with exclusively upper-case characters which behave like the built-in system functions could be loaded en masse from a public library.

4. Users would lose the richness of the APL character set. Many users would not care. In any event, many ASCII-based systems provide mechanisms to allow programmers to define their own characters. Alternatively, existing APL hardware techniques could be employed since there is an APL near-equivalent for each printable ASCII character.

5. Users would be less able to communicate algorithms internationally. However, algorithms must be described in prose as well as in a programming language; at a minimum, meanings of each user-defined variable and function must be described. Thus no programming language can really communicate algorithms to users of varying linguistic backgrounds. Non-English speaking users of JMSL could substitute their own keywords for the English keywords specified in the appendix; programs to translate JMSL with English keywords to JMSL with non-English keywords would be simple

to write.

The advantages inherent in these changes to APL would offset the disadvantages:

1. Users would not need to buy and maintain special hardware and software interfaces; users already familiar with one typing technique and editor would not need to learn new ways to type and edit programs.

2. It would be easy to add new primitives to JMSL by defining new built-in names.

3. In JMSL, there would be a greater one-symbol-one-meaning correspondence than in APL, fostering readability.

ASPECT: REVISION AND EXTENSION OF BUILT-IN CAPABILITIES.

PROBLEM: SOME APL SYMBOLS HAVE MULTIPLE MEANINGS AND USES.

Many APL symbols have more than one meaning; the meanings are distinguished by context. Most programming languages share this characteristic to some degree. However, sometimes people who learn APL find APL especially difficult since they must pay considerable attention to context in order to parse APL statements.

Standard APL functions are not consistently defined with respect to the relation between monadic and dyadic uses:

Standard APL Functions (F)	Monadic/Dyadic Uses
+ - ÷ * ⍟	F Y is the same as X F Y for some constant X
× ○ ⌊ ⌈ | , !	F Y is somehow related to X F Y
⍳ ⍷ ⍒ ⍋ ⎕NL	F Y is the same as X F Y for at least some values of Y if a suitable value of X (depending on Y) is chosen
?	F Y is a primitive scalar function, X F Y is not
⎕STOP ⎕TRACE ⎕SVO ⎕SVC	F Y is a query, X F Y is a modification
~ ⍸ ⍤ ⍣ ⎕DL ⎕EX ⎕NC ⎕FX ⎕CR ⎕SVQ ⎕SVR	F Y is defined, X F Y is not
∨ ∧ ⍱ ⍲ < ≤ = ≥ > ≠ ↑ ↓ / ⌿ \ ⍀ ∈ ⊤ ⊥	F Y is not defined, X F Y is

Table 4. Monadic vs. Dyadic Uses of APL Functions.

This lack of uniformity requires users to take special care in separating monadic and dyadic meanings of functions. The fact that many symbols have two meanings leads to a greater chance of error when novice APL users parse a line of code.

Another serious difficulty is that APL uses the symbols /[Z] ⌿[Z] \[Z] and ⍀[Z] as both functions (compress and expand) and operators (reduction and scan).

One APL function, dyadic ○, packs 15 different meanings into a single function, using a left argument to select the desired meaning. The correspondence between the values of the left argument and the functions to be selected is not easily remembered.

Several other functions combine multiple meanings in a more subtle way. Although APL emphasizes generality, sometimes this generality contributes to readability problems. Monadic ⍴ is used for the shape of any array and for the special case of a vector, whose shape is commonly known as "length". Monadic ! is used for the widely known factorial function and for the more esoteric gamma function, even though in mathematics the two are written with separate symbols. The ⊥ function is used to (1) convert a number from another base system, (2) evaluate a polynomial, (3) pack several arrays into a single array to conserve storage, and (4) perform other calculations; similar observations can be made about ⊤. Clearly and are versatile functions, but it is not always easy to discern which meaning is intended. In these cases, the generality of APL does not contribute to the readability and maintainability of code. For example, in the APL code fragment ((!X)=⍴Y) it would be helpful for a reader to know that "factorial" and "length" were intended, since this would document the fact that X must be an integer and Y must be a vector.

The APL input/output symbols ⍞ and ⎕ also have several meanings. The ⍞ symbol is used to obtain numeric constants and also to obtain evaluated input; this requires some programs to do special processing to ensure that only valid numbers are entered. Moreover, both ⍞ and ⎕ signify output if immediately followed by the ← symbol, and otherwise signify input.

PROBLEM: SOME APL BUILT-IN CAPABILITIES ARE UNNECESSARY.

Several APL capabilities contribute to readability problems and are unnecessary.

1. Some of APL's mathematical functions do not parallel the traditional mathematics as found in most technical publications; these functions should not be provided at all. ⍟Y and *Y and ○Y are more clearly expressed in APL as 1⍟Y and E*Y and PI×Y, where E and PI are suitably defined. From a readability viewpoint, it seems likely that most readers would take an extra mental step to insert the default 1 or E or PI; this suggests providing built-in constants rather than these built-in monadic functions. The Pythagorean functions (4○Y and ¯4○Y and (0○Y) are more clearly expressed in APL as ((Y*2)-1)*.5 and ((Y*2)+1)*.5 and (1-Y*2)*.5).

2. Many programmers find a variable index origin (⎕IO) to

be a nuisance. Changing ⎕IO affects subscripting, monadic ?, dyadic ?, ↑, ↓, monadic ⍳, and dyadic ⍳, but does not affect function line numbering, monadic ⎕STOP, dyadic ⎕STOP, monadic ⎕TRACE, or dyadic ⎕TRACE. Most programmers habitually think in one origin or another and find it cumbersome to have to think in the other ⎕IO system. Allowing ⎕IO to vary does not enhance readability, particularly since ⍳ and subscripting are such integral parts of APL that changing ⎕IO may lead to unexpected consequences. (For instance, →⍳1 exits the function if ⎕IO is 0 but branches to the first line if ⎕IO is 1).

PROBLEM: SOME APL BUILT-IN CAPABILITIES ARE INCONVENIENTLY DESIGNED.

Some APL capabilities could be improved in their design.

1. The order of arguments in APL(X|Y) is surprising to some users since this idea is symbolized by (Y mod X) in mathematics, (Y MOD X) in PASCAL, and (Y%X) in C. (X÷Y sets a precedent for ignoring the APL convention of CONTROL DYADICFUNCTION DATA, at least in basic mathematical functions.)

2. Some domains and ranges in APL are not as useful as they could be:

(a) the domains of < > ≤ ≥ ↑ and ↓ do not accept literal arguments; some APL implementations surpass Standard APL by specifying a collating sequence called the atomic vector (⎕AV) which can be used in defining comparisons. (Example: 'CAB' ↔ 3 1 2, where ↔ means "is equivalent to")
(b) monadic ⍳ is not defined for vector arguments; this function could reasonably be extended to produce an array whose rows consist of indices (in ravel order) of an array with the same shape as the vector argument to the function. (Example: ⍳2 3 ↔
2 3 2 1 1,1 2,1 3,2 1,2 2,2 3)
(c) dyadic ⍳ is not defined for left arguments which are not vectors, so it is not easy to determine the first appearance (in row-major order) of particular data elements within a higher-order array. (Example: (2 3⍴⍳6) 4 ↔ 2 1)
(d) ⎕NC returns numeric codes, but literal codes would be more legible.
(e) 0÷0 is somewhat arbitrarily defined to be 1; new users would expect the expression to yield an error.

3. The Standard APL operators add great power to the language; their utility and readability could be improved by some design changes.

(a) Analysis of typical usage of the reduction operator yields the following results:

APL Reduction	Analysis
+/Y ×/Y ∧/Y ∨/Y ⌈/Y ⌊/Y	appear frequently in code; very useful; meanings are fairly clear
-/Y ÷/Y ≠/Y	appear infrequently in code; somewhat useful; meanings (alternating sum, alternating product, parity of a boolean vector) are less clear
∧/Y ∨/Y ⍱/Y !/Y ⍲/Y */Y ⍲/Y ≤/Y =/Y ≥/Y >/Y	appear rarely in code; practically useless; confusing (nand, nor, and relational reductions do not work as users might expect)

Table 5. Analysis of APL Reductions.

The 11 useless reductions potentially lead to obscure, tricky code. The observation that 9 useful reductions are outnumbered by 11 useless reductions suggests that reduction is not designed optimally. The purpose of reduction is to perform aggregations on vectors of data which yield a scalar result. But there exist numerous aggregations such as mean, median, and standard deviation which cannot be expressed by a simple reduction. These aggregations should be facilitated in JMSL.

(b) When the scan operator is analyzed similarly, the same types of results are obtained because the concept of scan is based on the concept of reduction. Perlis and Rugaber [15] cite three highly idiomatic uses of relational scans on boolean vectors:

⍲\	Leave only the first (leftmost) 1
≤\	Leave only the first 0 turned on
≠\B	Create a vector of running even parity on B. Also, convert reflected Gray code to binary.

These expressions are of dubious utility and are error-prone to use. Their actions would be more intelligibly replaced by functions.

(c) The outer product operator is useful for all of its functional arguments since it can, at least, be used to create mathematical tables. The ∘.g notation for this operator is syntactically anomolous, as several writers have noticed.

(d) The inner product operator is based on reduction, so many of the 441 possible inner products are practically useless. APL inner product notation is similar to APL outer

product notation; although the two notations are mathematically clear, they obscure the fact that outer product is designed to work with scalars, while inner product is designed to work on vectors using reduction.

(e) The axis operator in Standard APL is only used in conjunction with reduction, scan, compress, expand, concatenate/laminate, and reverse/rotate. It would be very useful to extend the axis operator to other functions. Holmes [16] suggests that allowing monadic ⍴ to take an axis operator would enable users to condense APL expressions such as ((⍴MATRIX)[1]) to (⍴[1] MATRIX), but the approach is not uniform for all functions. Jenkins and Michel [17] note that the choice of a default axis (the last axis if no axis operator is specified) leads to asymmetry in APL. Eliminating the default axis operator for arrays of rank higher than 1 would make APL more symmetric and readable. (This is particularly desirable if an ASCII-based notation is employed.)

The Standard APL operators use inter-related concepts such as axes and reduction. It would be desirable to unify these operators notationally. It would also be desirable to allow operators to modify user-defined functions in a uniform fashion.

PROBLEM: APL IS MISSING SOME CAPABILITIES WHICH SHOULD BE BUILT-IN.

There are a large number of capabilities which could be usefully added to APL; if these capabilities were available, users would not have to go through any trouble to define them.

1. It is a little surprising that APL, a mathematically-based language, contains so few mathematical functions. The utility of APL would be increased if the most fundamental aspects of many branches of mathematics were included:

Area of Mathematics	Missing Facilities
Complex Variables	complex datatype functions to manipulate complex data
Trigonometry	cofunctions (sec, csc, etc.) inverse cofunctions (arcsec, etc.) hyperbolic cofunctions (sech, etc.) inverse hyperbolic cofunctions (arcsech, etc.) way to convert between angles/grads/radians

Area of Mathematics	Missing Facilities (continued)
Statistics	mean, median, and mode functions biased and unbiased variance functions biased and unbiased standard deviation functions standard error about the mean deviation and range functions function to "bucket" data into given intervals
Linear Algebra	determinant function (linear algebra) rank function ways to create upper-triangular, lower-triangular, and identity matrices function to test for singular matrix function to return elements along the major/minor diagonal of an array
Boolean Algebra	twelve logical scalar dyadic functions to complement ∧ ∨ ⍲ and ⍱ three logical scalar monadic functions to complement ~
Set Theory	function to ravel an array and remove duplicate elements, converting it into a setlike data structure functions to find the cardinality of such a set functions for set union, intersection, difference, equality, and inequality functions to test whether X is a (proper) subset of Y or a (proper) superset of Y
Number Theory	function to test whether N is even or odd function to test whether N is prime or composite function to test whether X divides Y evenly function to return the divisors of integer N function to perform "clock" remaindering (i.e. the Standard APL function 1+Y|X-1)

Area of Mathematics	Missing Facilities (continued)
General Mathematics	functions to round N up or down to the nearest integer
	functions to return square root or principal Xth root of Y
	function to return the fractional part (characteristic part, mantissa part) of Y
	function to return the roots of a polynomial given a vector V of coefficients
	functions to create arithmetic and geometric sequences
	function to calculate geometric mean of some data
	function to calculate reciprocal-sum-of-reciprocals (used in electrical engineering)
	ways to test whether data is ascending or descending or uniform, and whether the sequence is monotonic or not
	function to return the number of permutations of X things taken Y at a time
	function to "remap" an array of values from a given interval in one system to projected values in a given interval of another system
	common constants (e, pi, true, false, etc.)

Table 6. Mathematical Features Missing In Standard APL.

2. Standard APL could benefit from some capabilities to manipulate literal data:

(a) common literal constants: matrix of weekday names, matrix of month names, non-printing character scalars (such as ⎕TCBL in some versions of APL), vectors of all upper-case (or lower-case or digit or punctuation) characters
(b) way to query whether a literal argument is all lower-case, all upper-case, all mixed case, or composed only of punctuation and digits
(c) function to capitalize (or decapitalize or capitalize only the first letter within) all sequences of letters in a literal argument

3. Standard APL could also use additional capabilities to enhance structural manipulations:

(a) common structural constant (⍬⍴0)
(b) functions to perform (~X∈Y) and (~X)/Y
(c) function (known as "replicate" which works like X/Y but

allows non-negative integers in X to specify the number of times the corresponding element of Y is to be repeated in the result

(d) function to perform a dyadic shift analogous to dyadic rotate

(e) way to test whether two arrays are identical or not

(f) way to test whether an argument is literal or numeric, returning a boolean result

(g) function to return a blank if Y is literal or 0 if Y is numeric

(h) function to provide report-quality formatting (c.f. FMT)

(i) functions to left-/right-/center-justify an array whose subarrays are separated by some delimiter(s), returning a higher-order array

(j) functions to shed trailing/leading/leading-and-trailing occurrences of some delimiter(s) from an array

(k) function to compress successive occurrences of some delimiter(s) to a single occurrence

(l) functions to return and respecify the rank of any array

(m) ways to query the number of elements in an array and to test whether an array is (non)empty or (non)singleton

(n) functions to return the first/last element of an array

(o) functions to create sequences with equivalent values to those created by the BASIC (FOR I=X TO Y STEP Z); these functions could be defined for vector arguments to facilitate creation of lists of array indices

(p) functions to sort a vector in ascending or descending order, and to shuffle a vector in pseudo-random order

(q) functions to supplement dyadic ⍳; X Y returns the first position a data value is found, rather than the last position; 0 could be returned instead of 1+⍴X when an element of Y is not found in a search for the first/last occurrence.

(r) function to facilitate laminating together many arrays of common shape

(s) function to return a matrix of all indices into an array, and a similar function (for database queries) which would return only those indices where its boolean array argument was 1

(t) modification (called "complementary indexing") to subscripting (both in arrays and in the axis operator) to permit negative subscripts (so that -1 indicates last axis or element of axis, -2 indicates second last axis or element of axis, etc.) and the definition of constants to aid legibility in indicating axes. (Example: Y[-1;-1] ↔ Y[1↑⍴Y;1↓⍴Y])

4. APL input/output also could benefit from some modifications. The following forms of output would be useful:

(a) form of numeric input which accepts only a constant numeric vector

(b) form of literal input which accepts a single character without requiring a carriage return terminator (c.f. ⎕GET

in APL*PLUS)
(c) a form of output which does not automatically generate a newline and which is treated separately from input (unlike ⍞ input, but similar to the PASCAL WRITE construct)
(d) monadic function without result which prevents its argument from being displayed by APL's default output rule (c.f. ⎕SINK in APL*PLUS)

5. APL assignment could benefit from the incremental assignment capability found on those versions of APL which allow expressions of the form (X+←Y) to signify (X←X+Y). Any dyadic function could be used instead of +; the notation helps humans to conceptualize the fact that a variable is being incremented, and also helps computers to minimize data movement when the expression is evaluated.

6. Finally, APL could gain added power from two new operators:

(a) a moving-window operator to permit "running" aggregations of subvectors (Example: something similar to
2 +/1 2 3 4 5 6 ↔ 3 5 7 9 11)
(b) a partitioning operator to permit partitioned aggregations of subvectors (Example: something similar to
2 +⌸1 2 3 4 5 6 ↔ 3 7 11)

The preceding observations about APL capabilities do not include every problem Standard APL has, or every enhancement ever proposed or implemented. Many people could reasonably disagree about whether some capability is truly necessary (or truly unnecessary). However, the list does illustrate that Standard APL has not fully exploited its potential, especially in areas such as mathematics and structural manipulation. Development in these areas would make JMSL more powerful, readable, and easily utilized.

PROPOSED SOLUTIONS: THERE HAVE BEEN MANY PROPOSED SOLUTIONS TO THESE PROBLEMS.

The proposed work on these problems is extensive. A selection from the work is reviewed here. Some of the others are referenced. Ruehr [8] surveys extensions to APL with a fairly comprehensive bibliography of APL literature. Crick [9] discusses modifications suggested by using an ASCII character set. Eisenberg and Peelle [18] discuss ways that current APL surprises APL learners. Perlis and Rugaber [15] and Smith [19] describe APL idioms which suggest built-in capabilities. Penfield(Jr.) [20,21] provides a very detailed discussion of issues pertinant to complex numbers. Numerous other writers have made contributions relating to the above list of problems.

Many of these problems have been recognized, solved, and implemented on systems which provide capabilities beyond those specified in Standard APL. These solutions are fairly well-

known and can be found in reference manuals for the three systems discussed in the introduction. A few problems or desired capabilities are suggested by mathematics or by other programming languages.

The desirability of defining a significantly larger number of built-in capabilities does not seem to appear in APL literature, although idiom collections (for example, [22] and [23]) are suggestive. Writers either propose a few new capabilities at a time or else extoll the fact that APL already has a plethora of capabilities.

Several of the listed issues are controversial and merit additional discussion.

1. A number of references suggest a fixed ⎕IO; Iverson, Pesch and Schueler [24], Brown [25], and Holmes [16] prefer 0 for ⎕IO, while Jochman [26] and Smith [19] prefer 1.

2. The discussion of shortcomings related to operators is not reflected in APL literature. The trend among innovative implementations and in recent APL literature is to define many extremely general new operators (Iverson [27] and many others), to modify syntax to permit operator sequences such as +.×/A,B,[1.5]C (Brown [28]), to permit multiple axes to be specified within the axis operator (Orth [29]), and to permit user-defined operators (Orth [29]). This innovation with operators, while valuable, tends to favor highly abstract thinkers at the expense of concrete thinkers who may have to maintain programs containing these innovations. There is little research relating to the question of how much generality is optimal in a programming language--the issue is style-related and thus highly subjective--but some of the innovation with operators is so new that it has yet to prove its utility. For example, NARS includes a new operator ⍤ named "convolution". It is about as easy to learn and as general as inner-product, but its practical uses seem limited to (1) string searching and (2) polynomial multiplication. Two specific functions ⎕SS and ⎕PM would seem to be indicated rather than an operator. Operator sequences are fairly difficult to parse, both for humans and for APL interpreters. Functions which are modified along several axes at a time can also be difficult to conceptualize. User-defined operators add complexity to APL; it seems likely that ultimately only a few operators will ever be used in practice (just as only a few of the 441 inner products are ever used in practice).

3. The APL literature does not explicitly mention any need for functions to perform "clock" remaindering, remapping, or bucketing; nor does it support any need for string constants and functions. These capabilities are suggested by common situations in production environments such as the need to formfeed after N lines or the need to create

a histogram.

Many of these and related capabilities are almost too obvious to state. Nevertheless, they contribute to standardization of code and aid in conceptualization of problems--after a programmer has spent a trivial amount of time learning these functions, it becomes easier to mentally "chunk" the concepts needed to design and communicate more complicated programming ideas. (Polivka [30] describes a "chunk" as a unit of information stored in human memory and notes that chunking of information permits a larger amount of information to be retained.)

Traditionally, many of the missing capabilities have been provided through programming idioms or through cover functions (user-defined functions which hide the details of program code). These solutions are difficult to standardize for many users. Idioms are a major factor in making APL difficult to read, particularly since different users may use different expressions to represent the same concept. Cover functions are not available in the workspace and must be explicitly loaded from a utility workspace. Moreover, idioms and cover functions are typically not optimized for speed in the underlying machine code.

JMSL SOLUTION: REVISE/MODIFY/EXTEND APL PRIMITIVES.

In many cases, to recognize and state the problem is to suggest a solution. The problem of multiple meanings and uses is straightforwardly solved by defining a separate symbol for each meaning. The problem of unnecessary built-in symbols is straightforwardly solved by removing these capabilities. The problem of inconveniently designed built-in capabilities is easily resolved by modifying the Standard APL definitions or syntax.

The missing capabilities present a somewhat greater challenge. Most of the details are described in the appendix. Design decisions particularly worth noting are the fact that JMSL does not contain a ⎕IO variable (⎕IO is permanently set to 1 since most people, even technical people, count this way). Operators are treated in a way which (1) makes it reasonably clear when an operator is being used, (2) disallows ambivalent operators, (3) does not include user-defined operators, (4) permits user-defined functional arguments, and (5) discourages complicated operator sequences.

The list of problems suggests an overall direction for JMSL: provide more primitives to make JMSL programming more readable and less idiomatic. This overall direction is more important to JMSL than any individual improvement, since in order for an APL-like language to gain widespread acceptance, the language must be far more legible than Standard APL. Standardized capabilities are a major step towards this acceptance.

This overall approach has some disadvantages:

1. New learners must learn a larger language. This is undeniably true, but whether it is a disadvantage depends on one's philosophy of how large a vocabulary a programming language should have.

2. Providing many built-in capabilities is less efficient to implement than providing a minimal set. This is not necessarily a serious difficulty if less-commonly used capabilities are kept in a public library, to be automatically loaded into a workspace as needed. A number of the capabilities could be provided as macros within the interpreter, so that functions such as the "clock" function would simply expand into APL code which is almost as efficient as a machine-coded function would be.

3. Ambivalent functions (functions which can be used both monadically and dyadically) are not provided. Most APL users have no trouble using them, so eliminating them from the language may seem unnecessary to some users. This elimination does simplify APL, in the sense that Standard APL allows ambivalent primitive functions but not ambivalent user-defined functions. However, the primary reason for removing ambivalent functions is to support the concept of one-symbol-one-meaning.

4. Operators are significantly less concise and powerful than operators in some of the enhanced versions of APL. The reasons for this have been described above.

The advantages in modifying APL as described above would offset the disadvantages:

1. Primitive capabilities which are part of Standard APL would become easier to learn, use, and read.

2. Capabilities added to APL would be available for those who need them and could be ignored by those who do not. (Maintenance programmers could not ignore the added capabilities, but they would almost certainly find a clearer explanation of these capabilities in a JMSL reference manual than in documented Standard APL code which provides similar capabilities, since few programmers document as clearly as reference manual writers.)

ASPECT: INCLUSION OF NESTED ARRAYS AND WAYS TO MANIPULATE THEM.

PROBLEM: APL CANNOT CONVENIENTLY STORE AND MANIPULATE DATA OF HETEROGENEOUS TYPE AND SHAPE.

Standard APL has a single data structure, the array. Arrays must be of uniform type and rectangular shape. This prevents arrays of mixed type (with both literal and numeric data) from being combined and manipulated as a single array. This also prevents arrays of non-conformable shapes from being combined and manipulated as a single array. There exist ways to work around these limitations when working with arrays of non-conformable type/shape (converting characters to integer equivalents and vice versa; padding arrays to conformable shapes) but frequently these methods obscure the algorithm at hand. Manipulation of arays with heterogeneous type and shape is desirable since frequently a diverse collection of arrays must be treated as a group. For instance, in a database, a list of names of varying lengths may need to be combined with a list of numbers, so that the result may be treated as a single collection of data. APL does not provide a straightforward mechanism for this.

PROBLEM: APL CANNOT CONVENIENTLY STORE AND MANIPULATE STRUCTURES OF ARBITRARY DEPTH.

Another limitation of APL's array data structure is that there is no way to create and manipulate tree-like structures. LISP is an example of a language that provides ways to process data structures of arbitrary depth but APL does not. These tree-like structures are important in many applications such as manufacturing bills of lading, organizational charts, and the recursive data structures which frequently appear in artificial intelligence work.

PROPOSED SOLUTIONS: APPROACHES INCLUDE HETEROGENEOUS ARRAYS, FILES, GROUPS, PACKAGES, CARRIER ARRAYS, AND SEVERAL FORMS OF NESTED ARRAYS.

The two problems mentioned above are actually separate, but because both address limitations in APL's data structure, they are frequently discussed together.

Arrays which are heterogeneous in type have been proposed and implemented. These arrays are created by expressions similar to (X←2,3,'ABC',5) which on STSC's NARS system creates a 6-element vector which can be reshaped, concatenated with similar vectors, and structurally manipulated in other ways. The difficulty with this approach is that since the type of an array is no longer uniform, it is more difficult to implement array manipulations in hardware. Perhaps more important-

ly, this solution does not address the difficulty of manipulating objects of non-conformable shape.

APL component files allow heterogeneous objects to be stored, regardless of type and shape. However, files store the data outside the active workspace, so that a file must be created to store the data, even for temporary data. Moreover, component files have not been designed to allow manipulation on more than one item of data at a time, whereas APL's power lies chiefly in the ability to manipulate many items of data simultaneously.

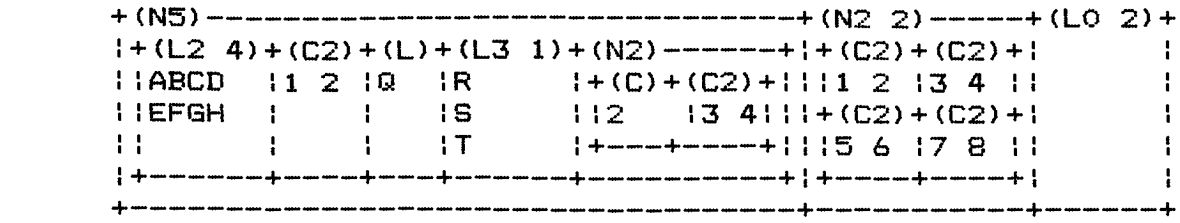
APL groups and packages suffer from similar limitations. (For a description of these well-known APL constructs, see the next section.) Although both groups and packages permit the storage within the workspace of objects with varying type and shape, both approaches lack a sufficiently powerful way to manipulate many items of data at a time. Groups, being system commands, can not be used or modified under program control. Packages can be modified under program control but are not amenable to structural manipulations such as concatenate or reshape.

An interesting solution to the problem that arrays must be of conformable shape is the idea of Carrier Arrays presented by Lowney and Perlis [31]. This idea extends APL by permitting a form of general "ragged arrays" which allows, for example, a list of words of varying lengths to be stored and manipulated. Perhaps Carrier Arrays could be combined with arrays of heterogeneous type in order to completely solve the problem of storing and manipulating data of heterogeneous type and shape. However, none of the solutions presented thus far addresses the problem of manipulating structures of arbitrary depth.

There are a number of proposals and implementations which solve both problems using the data structure known as the nested array. A nested array is an array which is composed of other arrays, which may in turn be nested. As this recursive definition suggests, nested arrays may be used to build structures of arbitrary complexity. Since the arrays used in nested arrays may be literal and/or numeric arrays of any shape, data of heterogeneous type and shape may be stored and created in a unified fashion.

The APL community has not agreed on basic definitions regarding nested arrays, so several conflicting systems of nested arrays have developed.

The nested arrays in any nested array system may be conceptualized using a diagrammatic technique employing outline notation. A rather complicated example illustrates a nested array P:



Terminology differs among various nested array systems. The terms defined below reflect concepts common to most systems.

1. A simple array or non-nested array is a Standard APL array.

2. A nested scalar is a new type of scalar which may be conceptualized as an outline together with its contents; the contents are either a nested array or a simple array.

3. A nested array is a rectangularly ordered collection of nested scalars. (Examples are nested scalars, nested vectors, nested matrices, and nested higher-order arrays.)

4. An array is a simple array or a nested array.

5. An enclosed scalar is a nested scalar.

6. The enclose of an array Y is a nested scalar whose contents are Y.

7. The disclose of a nested scalar Y is the contents of Y.

8. A scalar X is an element (or member) of an array Y if X matches one of the collection of scalars comprising Y.

9. X includes Y if Y is an element of X.

10. A scalar X is an item of an array Y if the enclose of X is an element of Y.

11. X holds Y if Y is an item of X.

Accordingly, P is a nested vector with 3 elements. The first element of P holds a nested 5-element vector; the items of this 5-element vector are a literal matrix of shape 2 4, a numeric vector of shape 2, a literal scalar, another literal matrix, and a nested vector which includes a nested numeric scalar and a nested 2-element vector. The second element of P holds a nested array which is a nested matrix of shape 2 2. The disclose of each nested scalar comprising this matrix is a simple array; each of these simple arrays has the same type (numeric) and shape (2). The third element of P is a nested literal matrix of shape 0 2.

There are several systems of nested arrays; the two most

popular will be discussed in some detail.

The grounded approach to nested arrays has been adopted by IPSA's SHARP APL. In this system, an array may be enclosed using the monadic function ⟨. Y is distinct from ⟨Y for all Y.

Nested scalars may be structurally manipulated in ways analogous to numeric or literal scalars, using APL functions such as ⍴ , ⍵ etc. In particular, dyadic ⍴ and dyadic , may be used to create nested arrays out of nested scalars. Functions such as dyadic ≀ and ∈ may be used with nested arrays, but the implied comparison in these functions is modified from a comparison based on equality to a comparison based on exact match (of both nesting structure and contents). A new function to perform a test for exact match is provided. Nested arrays may be subscripted, and subscripted assignments such as A[B]←C may be performed.

An array may be disclosed using the monadic function ⟩. For a nested scalar Y, Y is ⟨Q for some item Q, so ⟩Y is simply Q. In this case, Y and ⟨⟩Y and ⟩⟨Y are all the same. However, if Y is a simple array, there is no outline to remove; SHARP APL defines ⟩Y to be Y in this case. If Y is a nested non-scalar array, ⟩Y is defined only if each item of Y has homogeneous type and shape; the shape of the result is (shape of Y),(shape of each item of Y).

Several matters regarding structural manipulation have been glossed over; these relate to a thorny issue in nested array systems: the type of a nested array. Some APL implementations preserve type information in all functional manipulations; on these systems 0⍴2 3 is the same as 0⍴0 but not the same as 0⍴'ABC'. Standard APL considers 0⍴0 to be exactly identical to 0⍴'ABC'; so does the SHARP APL system. This is an important consideration because this implies that 0⍴(('PQ'),4 should be equivalent to 0⍴0 and because this means that there is no natural "fill" element. (In APL, ' ' and 0 are used to fill the result of certain expressions such as 3↑'AB' ↔ 'AB ' and 1 1 0\1 2 ↔ 1 2 0. The result of (3↑('PQ'),⟨4) is not obvious.) SHARP APL does not define "overtake" and expand unless the right argument is of homogeneous type and shape.

Arithmetic functions are not defined for nested arrays, so ((⟨2),⟨4)+⟨(1 5) yields an error. However, the answer ((⟨3 7),⟨(5 9) may be obtained by using an arithmetic function in conjunction with one of several new operators. These new operators perform varying forms of function composition in ways that facilitate manipulation of one or two nested or non-nested arrays, by modifying functions to work on items of an array rather than on the elements of an array.

It is not easy to input data structures using the notation

of SHARP APL. A nested vector whose items are the first 4 integers is created as follows: (V←(⟨1),(⟨2),(⟨3),⟨4. SHARP APL provides a "link" function which allows V to be created by specifying 1⊃2⊃3⊃⟨4; this partly solves the problem but does not always show deeply nested structures clearly. The last piece of data (4) needs to be enclosed before it can be linked with other data, which is inelegant; as Benkard [32] points out, this is unavoidable and no function F can be defined in a consistent fashion to handle this problem. Moreover, if parentheses are omitted when an expression is input, the depth of the resulting nested array may be obscured.

Another form of notation suitable for the grounded approach is semicolon-bracket notation. Proposed (simultaneously with minor differences) by Chastney [33] and Falkoff [34], this notation assigns a monadic meaning to indexing. The first element of P might be denoted in semicolon-bracket notation as follows:

```
[[2 4⍴'ABCDEFGH';1 2;'Q';3 1⍴'RST';[2;3 4]]]
```

This notation is more concise than SHARP APL notation, and clearly depicts the nesting structure of the array. The notation is almost completely upwards-compatible from Standard APL but requires expressions in Standard APL such as X[Y][Z] to be rewritten (X[Y])[Z]. Both writers suggest that [] hold an "undefined" value. Falkoff suggests function headers which contain multiple assignments of the form ∇[X;Y;Z]←[L1;L2] F [R1;R2]. The idea of multiple assignments has conveniences and ramifications which will not be explored further in this section since it is not crucial to this discussion of nested arrays.

SHARP APL allows nested arrays to be output using diagrams similar to the outline notation presented above, although some information is omitted. Nested arrays may also be displayed without outlines being printed.

The floating approach to nested arrays has been adopted primarily by STSC's NARS system and IBM's APL2. Trenchard More's Array Theory as implemented by Queens University's NIAL (Nested Interactive Array Language) system also uses the floating approach. The NARS system will be used to illustrate the floating approach; it is similar in many respects to IBM's APL2. In most floating systems, an array Y may be enclosed using the monadic function ⊂Y. (The meanings of APL symbols are not consistent between the various implementations.) The major difference between the floating and the grounded approach is the definition of enclose. In the floating approach an enclosed scalar is the same as the scalar; the "outlines" only appear when a higher-order array is enclosed. If Y is scalar, ⊂Y and Y are identical. The NARS system provides ways to enclose along an axis, perform a partitioned enclose, and perform a partitioned enclose along an axis.

The NARS system allows some structural manipulations which the SHARP APL system does not. As mentioned earlier, the NARS system extends the domain of dyadic , to permit a non-nested array to have heterogeneous type. Subscripting and subscripted assignment are modified to permit "scatter-point" indexing (indexing which may select arbitrary elements of an array, as opposed to indexing which may only select elements along slices of an array) and also "choose" indexing (indexing which may select elements of arbitrary nesting depth, as opposed to indexing which may only select the elements at the outermost level of nesting). To facilitate these powerful forms of subscripting, the domain of monadic ⍳ is extended to generate index sequences consisting of nested vectors of positive integers. Additionally, a function to return the nesting depth of an array is provided.

In the NARS system, an array Y may be disclosed using the monadic function ⊃Y although if Y is not scalar, ⊃Y merely returns the disclose of the first element of Y. Another function works similarly to the SHARP APL disclose, but allows an axis specification which controls where the new axes created by the disclose are to be placed in the result. (Arrays must be of homogeneous shape in order to be disclosed, but they do not have to be of homogeneous type since non-nested arrays of heterogeneous type are permitted.) X⊃Y may be used dyadically to disclose selected elements at some arbitrary position and nesting depth rather than all elements at the outermost nesting level.

The issue of type is particularly difficult in the floating systems, since even non-nested arrays may be of heterogeneous type and since these systems choose to maintain a distinction between 0⍴0 and ''. The NARS system's definition of monadic ⊃ is designed to return the fill element of an array. The fill element in such expressions as 1 0 1\Y is given by ⊃Y; this implies that 1 0 1\2,'A' is 2,0,'A' but 1 0 1\'A',2 is 'A',' ',2. Because of this definition, X⍴Y may be defined even if ((0≠⍴,X)∧0=⍴,Y); in this case, the fill element of Y is used in the result instead of Y. A special function is provided to return the type of an array (which is the same as its fill element). In a tree containing diverse elements (such as P, above), the fill element may seem quite arbitrary.

In the NARS system, arithmetic operations may be directly performed on nested arrays without using any operator. The NARS system contains new operators for nested arrays which are similar in concept to the SHARP APL operators, although details differ. The NARS system allows user-defined operators, which can be used to create very complicated manipulations on nested arrays; a function definition operator which may be used to define functions directly from APL (rather than through an editor or use of ⎕FX) can be used to define functions manipulating nested arrays.

A major issue separating the floating approach and the grounded approach is the floating approach's use of a notation called Strand notation. Strand notation creates nested structure using parentheses to show the hierarchy and blanks to separate the heterogeneous data; for example,

((2 4⍴'ABCDEFGH') (1 2) ('Q') (3 1⍴'RST') ((2) (3 4)))

would be used to create the first element of P. Strand notation clearly shows the nested array structure, but is quite controversial and possibly ambiguous. The difficulty lies in the fact that parentheses are used both to show nesting structure AND order of execution; also, spaces are used to show separation of numeric vector constants AND separation of heterogeneous arrays AND separation between user-defined variables and functions. Use of Strand notation practically requires arrays (both nested and non-nested) to contain data of heterogeneous type. Thus 'A',3 is a non-nested vector in the floating systems, whereas 'A',3 is an error in the grounded system. (The grounded approach could be modified to permit arrays of heterogeneous type by changing the domains of the dyadic , function; however, since the definition of the grounded approach precludes Strand notation, the utility of this modification is negligible.)

Strand notation is used for output as well as for input in NARS and APL2.

The SHARP APL, NARS, and APL2 implementations share much in common; for example, certain advanced arithmetic/structural functions such as ⍕ ⊥ ? and ⍞ are not extended to nested arrays. SHARP APL seems easier for APL users to learn since the notation used is syntactically more like APL than Strand notation; however, the SHARP APL operators are confusing to many users since they are used ambivalently (both monadically and dyadically) and have similar symbols. NARS and APL2 seem very convenient to APL users who have mastered Strand notation since it is easier to input nested arrays. Differences between NARS and APL2 are minor, and Sykes [35] demonstrates evidence that the two systems are converging in philosophy.

In addition to the floating system and the grounded system, Mercer [36] has proposed a "based system"; this proposal has not received much comment and has not been implemented. The definitions of this system do not seem very intuitive.

JMSL SOLUTION: A GROUNDED APPROACH WITH "SEMICOLON-CURLY BRACE" NOTATION

JMSL uses a grounded approach to nested arrays. The grounded approach is used since it is more of an orthogonal extension to Standard APL than the floating approach; Eisenberg and Peelle [18] observe that new APL learners tend to cluster APL symbols visually in ways which might cause errors in

floating systems due to the unwitting use of Strand notation. The many users who would not care to learn about nested arrays are spared from any need to be particularly careful about blanks and parentheses in the grounded approach.

JMSL uses a two-symbol combination rather than a single symbol to denote enclosure: Y is enclosed using the notation {Y}. Since JMSL is a grounded system, Y is never the same as {Y}. The shorthand notation {X;Y;Z} may be used for {X},{Y},{Z}. This "semicolon-curly brace" notation allows {} to signify a vacant nested scalar (a data structure which does not yet hold any data); this is motivated by the semicolon-bracket notation discussed earlier. Shorthand notation may be used to indicate vacant nested scalars within arrays:

{},{"P";3 4},{},{} ↔ {;"P";3 4;;}

The meaning of ; within {} as a nested variable separator is distinct from the meaning of ; within [] as an index separator. The context provided by the paired delimiters surrounding ; distinguishes these meanings. (Ideally, distinct delimiters would be used, but ASCII has only a limited number of candidates.)

A characteristic of both semicolon-bracket notation and semicolon-curly brace notation is that since they are notations and not functions, they cannot unambiguously perform an enclose along an axis. For this reason JMSL provides an additional monadic enclose function which may take an axis specification.

Most of the JMSL structural functions are suggested by NARS and APL2. A monadic conditional-enclose function, which encloses its argument if the argument is not already enclosed, is suggested by SHARP APL proposals. Functions to test whether an array is homogeneous or heterogeneous seem useful. A function to re-specify the nesting depth of an array seems a natural complement to the depth function provided by NARS. For readability, monadic functions to test whether the argument is nested/non-nested seem beneficial. Since the SHARP APL meaning of ∈ is closer in meaning to the mathematical form {3}⊆{3,4} than to the mathematical form 3∈{3,4}, functions corresponding to the mathematical meanings of ∈ and ∉ are provided.

Disclose in JMSL is somewhat different from the forms of disclose provided by other systems. Disclose is symbolized Y'' where '' is a notation treated syntactically like Y[] is treated in Standard APL. Generally, Y'' is equivalent to the SHARP APL)Y described earlier. If any element of Y is {} then Y'' is a domain error. "Choose" indexing may be performed by specifying X'Y', where Y is a list of arrays which are to be interpreted as indices. For example, in JMSL, X'2 3;4' ↔ (X[2;3]'')[4] and X'2 2 2@1 2 3 4 5 6 7 8' ↔

2 2⍴X[1;2],X[3;4],X[5;6],X[7;8]. There are further subtleties to disclose in JMSL which are discussed in the appendix. The notation allows disclosed-assignments such as X'3';5; this idea has conveniences and implications which have not been pursued on existing nested array systems.

JMSL retains type information in nested arrays because this type information can be used to provide fill elements for overtake and expand. The fill element for nested arrays is {}. 0⍴0 and "" and 0⍴{} are all distinct. In a sense, JMSL is a strongly-typed language since each array is strictly numeric or strictly literal or strictly nested.

JMSL extends both primitive scalar arithmetic functions such as + and primitive mixed arithmetic functions such as ⍴ to nested arrays by defining these functions pervasively. For instance, X+Y can be extended to nested arrays by defining the result to be {X''+Y''} if either X or Y is nested. Two kinds of operators are provided: an "each" operator similar to one provided by all 3 of the major implementors of enhanced APLs, which modifies a function to work on the outermost level of items of a nested array, and a "pervasive" operator which modifies a function to recurse until the non-nested array of each element is encountered.

The input notation for JMSL has already been discussed. Several forms of display are defined. Default output prints nested arrays in a form similar to the well-known obsolete form of APL/360 output called mixed output; string delimiters and nesting delimiters are not printed. A form of output similar to Strand notation provides more detailed type and structural information. A form of output which depicts nested arrays within outlines similar to the outline notation at the beginning of this section is also provided to aid conceptualization. (The internal format of JMSL nested arrays is an implementation-dependent detail.)

Nested arrays have many implications for APL which are just beginning to be explored. One benefit of nested arrays is that they allow more than two arguments to be passed to a function, and they allow more than one result to be returned. Nested arrays provide tree-like structures similar to those used in LISP, while the (basically) infix notation of APL permits far fewer parentheses than LISP requires.

There are a few disadvantages to providing nested arrays:

1. Nested arrays require users to learn many new capabilities. However, the remarks of the previous section apply; nested arrays and related features are simply standardized tools which may be learned or ignored as developers choose, and which are standardized and documented for maintenance programmers.

2. Nested array processing leads to a less efficient system. This has not proved to be a serious problem on the major implementations of nested arrays.

The advantages offset these disadvantages:

1. Multiple arguments and multiple results can be passed to and from functions, preventing the need for many global variables.

2. Related information of differing types and shapes may be stored under a single name, a capability which is particularly useful in database applications.

3. Recursive programming applications requiring tree-like structures are facilitated.

ASPECT: INCLUSION OF HIERARCHICAL DIRECTORIES.

PROBLEM: APL CANNOT SHOW HIERARCHICAL RELATIONSHIPS AMONG NAMED COLLECTIONS WELL.

Languages such as PASCAL and PL/I provide records to describe data structures. The fields of records may be indexed by name, while the items of a nested array must be indexed by position. Standard APL does not have a construct analogous to a record which can be indexed by name under program control.

One consequence of Standard APL's inability to build structures indexed by name is that it is not possible to group related variable and function names, except through the choice of characters within the name of an object. (Label names are treated as "local constants" and are automatically grouped within functions.) Workspaces in production environments may contain over a hundred variables and functions, as Murray [37] notes; most programmers find keeping track of many objects which are classified by lexicographic order rather than by meaning to be a difficult chore.

Another consequence of Standard APL's inability to build structures indexed by name is that it is impossible to hide information appropriately by defining named hierarchies. This is a serious problem since programmers are not able to isolate variables and functions based on their usage within subsystems, as Murray [37] points out. When two or more subsystems must be interfaced, programmers must take particular care to ensure that one subsystem's variables and functions do not affect another subsystem's variables and functions in any way. In applications which naturally lead to a multi-level solution to problems (a classic example is the physical/logical/conceptual separation of database implementation), the variables and functions implementing low-, medium- and high-level capabilities are intermingled due to the fact that the APL namespace (a term used loosely to describe the set of all name-referent pairs) is flat rather than structured.

PROPOSED SOLUTIONS: APPROACHES INCLUDE GROUPS, PACKAGES, DIRECTORIES AND INDEX-BY-NAME PROPOSALS.

APL/360, documented in [3], allowed related variables and functions to be classified by name, using groups. Groups were implemented as system commands and allowed hierarchical storage of named objects (including other groups) within the workspace. A brief description of these commands shows that groups seemed to be designed with set theory in mind:

APL Group Command	Meaning of Command
)GROUP GNAME OBJ1 OBJ2	create a group GNAME with members OBJ1 and OBJ2. GNAME must be an undefined name; OBJ1 OBJ2 must be variables, functions, groups, or undefined names
)GROUP GNAME GNAME OBJ3	add new member OBJ3 to existing group GNAME
)GROUP GNAME	delete group GNAME (but members persist)
)ERASE GNAME	erase group GNAME (and all its members)
)GRP GNAME	list all members of group GNAME
)GRPS	list all groups

Table 7. Summary of APL Group Commands.

One problem with the group concept is that groups could only be manipulated by system commands and not under program control. Another problem with groups was that they could overlap; since an object could be a member of several groups simultaneously, anomolies arose in copying groups into a workspace. Since groups served as little more than a rather poor documentation aid (as Murray [37] noted), groups fell into disfavor among the APL community and were not included as part of Standard APL.

Attempting to remedy the deficiencies of groups, IPSA developed the concept of an APL package. Packages allow hierarchical storage of named objects (including other packages) within the workspace. Unlike groups, packages can be manipulated under program control using some new system functions IPSA developed to support packages. Some of these functions are briefly described in the following table.

APL Package Command	Meaning of Command
⎕PNAMES PKG	return literal matrix of names within package PKG
⎕PNC PKG	return integer codes corresponding to the name class of each member of PKG (2 = variable, 3 = function, etc.)
NAMELIST ⎕PSEL PKG	create a package with names from NAMELIST (a literal matrix of names) and referents from PKG
NAMELIST ⎕PEXP PNAMES	create package like PNAMES but excluding those whose names are found in NAMELIST
PKG1 ⎕PINS PKG2	copy names and referents of PKG1 into PKG2, possibly overwriting
⎕PACK NAMELIST	create a package containing names listed in NAMELIST and their corresponding referents in the visible environment
NAME ⎕PVAL PKG	value of referent for name NAME of PKG
⎕PDEF PKG	define names of PKG within visible environments with their corresponding referents from PKG

(adapted from [38])

Table 8. Summary of APL Package Commands.

These package manipulation functions do not interact with other SHARP APL primitive functions, since they manipulate names and referents, rather than structures and values of arrays. Retrieval of a given named object is straightforward at the outermost (most visible) level of hierarchy but is rather difficult if the name is not at the outermost level, as Crick [39] observes.

Crick [39], Taylor [40], and a number of others consider both groups and packages to be obsolete. Crick [39] was among the first to propose unifying the APL language by providing a directory data structure; further work has been performed on the concept, and Taylor and Whitney [41] describe directories as follows:

> Namespaces are everywhere used to distinguish objects from their context. APL provides symbols and a means for assigning values to them...Another way of distinguishing an object is by its position. With the introduction of [nested] arrays and the notion of symbol-object pairs a very natural path notation suggests itself. In this notation p_obj refers to the value paired with the symbol obj in the [directory] p...[W]e can also execute p_obj←X (for any value of X). Call any such object obj private to object p.

Benkard [42] describes non-positional indexing which uses a new type of object, a token (a short literal vector considered as a unit, essentially the keyword tokens discussed earlier). Taylor and Whitney [41] and others utilize similar tokenized strings. Benkard's description of how these tokenized strings might be used to perform updates of directory-like structures containing named objects is illustrative; it has influenced writers such as Taylor [40] who are interested in hierarchical naming. Benkard proposes that objects be indexed by position using normal APL subscripting rules, but also by (tokenized) names, with the names being maintained in lexicographic order to facilitate editing of these objects. Editing of objects stored in an external file which are indexed using a named key has been performed on a number of APL file systems (Xerox Corporation [43], Deerhake [44], and Wiland [45]).

JMSL SOLUTION: PROVIDE DIRECTORIES WITH INDEX-BY-NAME AND INDEX-BY-POSITION CAPABILITIES.

JMSL combines the directory structure defined above by Taylor and Whitney with updating similar to the index-by-name proposal of Benkard. In JMSL a directory is defined naturally as a nested array:

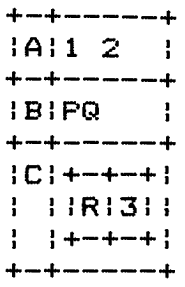
A nested array D is a directory if:

(1) The shape of D is 1 1,N,2 for some N
(2) {} is not an element of D
(3) Each item of D[;1] is a literal vector in the form of a valid variable name (e.g. "Name1")
(4) These names are unique and appear in sorted order

In short, a directory is simply a list of names along with their corresponding referents. These referents are arrays which may or may not be directories. Example:

```
      view D: 1 1 3 2⍴{"A";1 2;"B";"PQ";"C";{"R";3}}
```

(output on next page)



Since a directory is merely a special case of a variable, it may be manipulated with all the functions presented in the preceding two sections. For example, to query the names of the members of directory D, the JMSL statement (see D[1;1;;1]) would produce a somewhat hard-to-read display. To modify the referent for member A, (D[1;1;1;2]: {NewValue}) would suffice. (D: 1 1 1 2@D[1;1;3;] would delete the first 2 members of the directory.

JMSL defines a more effective way to manipulate directories using the symbol \. This \ notation is similar to the aforementioned p_obj notation, and is illustrated by the following table:

Use of \	Explanation
D\E	D is an existing directory; E is any defined name. D\E returns the item in the second column of D corresponding to the row whose item in the first column matches "E". If no match is found, an error is returned.
D\E: F	The item D\E is replaced by F within directory D; if the name "E" is not found, a new entry is inserted into the directory D (maintaining sorted order) with name "E" and referent F.
D\E\F	D\E\F is equivalent in all contexts to (D\E)\F. Further qualification of a directory can continue to arbitrary depth.

Table 9. Description of \ in JMSL.

Note that \ is considered bound to D and E for parsing purposes and is evaluated with higher precedence than other functions (e.g. D\E+1 signifies (D\E)+1 rather than D\(E+1)).

It is important that directories be able to store functions. To facilitate this, JMSL defines the concept of an "executable array", which is nothing more than a variable containing a listing of the lines of code which make up the visible repre-

sentation of a function. More precisely, a nested array EA is an executable array if:

(1) The shape of EA is 1 1,N,1 for some N)1
(2) {} is not an element of EA
(3) The first item of EA is a literal vector which resembles the syntax of an APL function header with the function name replaced by (&, !, or %); for example, "R: X % Y"
(4) Each subsequent item of EA is a literal vector (of presumably executable code)

An example of an executable array follows:

```
      view EA: 1 1 3 10{"&";"1+2";"3A"}
+---+
|&  |
+---+
|1+2|
+---+
|3A |
+---+
```

EA satisfies the above definition even though the last line of the code is syntactically invalid.

To execute an executable array, the programmer specifies &EA or !EA or %EA, depending on the syntax of EA. (&, !, and %) are similar to the concept of ' in LISP or in APL; they execute each line of code in the executable array, returning a result when done (if appropriate) and also erasing any temporary variables created along the way. Since the only way to call a JMSL user-defined function is to convert an executable array into a working function using one of these three special characters (e.g. 3 %EA 4), new users of JMSL distinguish user-defined functions from user-defined variables by the fact that the names of the former all begin with one of the three characters. (It is not necessary to prefix JMSL built-in functions with a special symbol in order to execute them.)

Note that the appropriate special character replaces the name of the function in the function header. The special character also replaces the name of the function in any self-recursive reference. Standard APL's use of the function name in function headers and self-recursive references leads to problems in copying function code. Ideally, a function copy should be performed with a statement such as FNEW FOLD; however, then the object FNEW would contain references to FOLD. (Syntax considerations complicate the matter in Standard APL, since FOLD could be the name of a niladic function.) Standard APL, since FOLD could be the name of a niladic function). Standard APL performs function copying in a manner which Bozinovic [46] calls "obtuse": programmers must convert FOLD to a literal matrix, copy the matrix using the assignment

statement used for variables, change the function header and self-recursive references of the new matrix, convert the new matrix back to a function FNEW, and erase any intermediate variables utilized. An example shows how awkward this process is:

```
      ∇FOLD[⎕]∇
      ∇FOLD
[1]   'PI IS 3.14159265358979'
[2]   ⍝ APPROXIMATELY

      FOLDMAT ⎕CR 'FOLD'
      FOLDMAT                          (optional step)
FOLD
PI IS 3.14159265358979
⍝ APPROXIMATELY
      ⍴FOLDMAT
3 24
      FNEWMAT FOLDMAT
      FNEWMAT[⎕IO;] 24 'FNEW'
      ⎕FX FNEWMAT
FNEW                               (indicates successful copy)
      )ERASE FOLDMAT
      )ERASE FNEWMAT
```

It is easier to copy functions in JMSL--the executable arrays are copied. Since the executable arrays do not contain the function name in the header or in self-recursive references, and since arrays are not automatically evaluated as APL functions are, problems of naming and syntax considerations are avoided. In JMSL, an assignment such as (FNEW: FOLD) suffices.

Self-recursion (F1 calls F1) occurs frequently in the type of mathematical problem-solving for which JMSL is suited. When copying F1 to a new name G1, it is desirable to have G1] behave the same way as F1; this is one reason for using a special character instead of the name F1 in JMSL function headers. Co-recursion (F1 calls F2 while F2 calls F1) occurs much less frequently in mathematical problem-solving. In the case of co-recursion, it is not possible to copy functions to new names with any assurance that they will still behave the same way as the originals; the recursive references must be changed by the programmer. JMSL permits co-recursion; this naturally implies that in a self-recursive function F1, a recursive reference to &F1 (or !F1 or %F1) may appear in the executable array for F1 (although outside of the header of F1). This type of self-recursive function F1 is grudgingly permitted but discouraged due to its lack of portability.

An example should clarify this discussion:

```
      view Factorial1
+-----------------------------------------------+
!R: ! N                                         !
+-----------------------------------------------+
!IF (N=0) THEN (R: 1) ELSE (R: N*!Factorial1 N-1)!
+-----------------------------------------------+

      view Factorial2
+-------------------------------------+
!R: ! N                               !
+-------------------------------------+
!IF (N=0) THEN (R: 1) ELSE (R: N*! N-1)!
+-------------------------------------+
```

Both versions of the factorial function work correctly (although pseudocode has been used instead of the control structures to be described in a later section). The second form is preferable to the first because it is more portable--if the executable array is renamed, the second form will continue to work properly, while the first form may not if Factorial1 is subsequently modified.

There are other possible definitions for an executable array than the one given. A non-nested literal matrix would require storage for blank padding. Crick [39] suggests that code be stored in a tokenized form to facilitate dynamic editing of functions at a token level rather than at a character level. Definitions could also vary regarding the degree to which the functional code would be checked for validity. In practice, the exact definition would not be a crucial issue--the stored form of an executable array would be unimportant most of the time, since executable arrays would normally be created and modified using an editor and run by prefixing with & or ! or %. The actual storage form of the executable array would be accessed only in unusual circumstances (just as an executable file on most computer systems is examined only rarely).

The purpose of (& ! %) and executable arrays is to allow functions to be manipulated as easily as other nested arrays. In particular, executable arrays may be stored in JMSL directories (e.g. D\E: EA) and run from the directory (e.g. 3+!D\E). Also, executable arrays facilitate the data transfers to be described in the next section, since functions and variables may be copied with equal ease. Another effect of the concept of executable arrays is to unify editing with the rest of JMSL, since an editor just becomes a function which would create/modify an (executable) array.

The directory concept requires JMSL users to view variables and functions in a slightly different manner than Standard APL users:

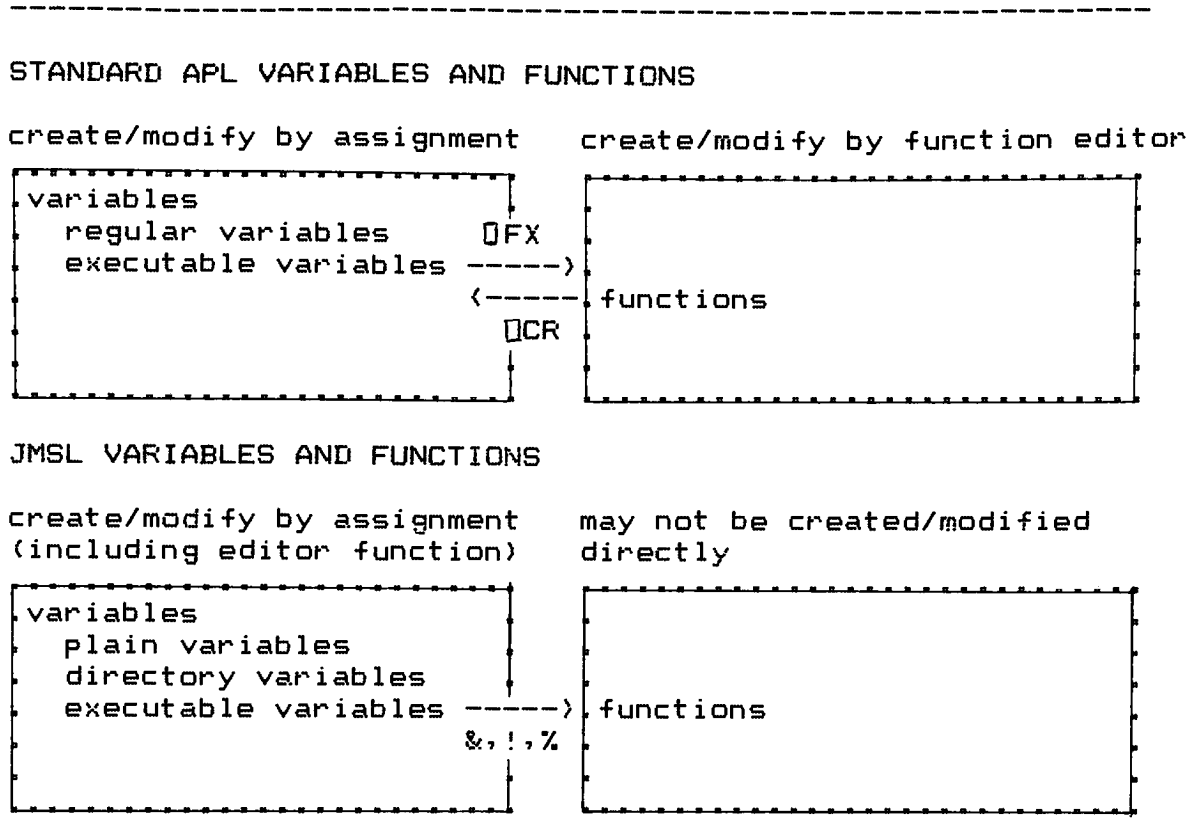


Figure 3. Standard APL/JMSL Views of Variables & Functions.

In these conceptual diagrams, 'regular variables' refers to all Standard APL variables which are not in the domain of ⎕FX, and 'plain variables' refers to all JMSL variables which do not satisfy either the definition of an directory or an executable array.

Since it would be desirable to query a directory to determine its contents, JMSL defines the functions (vars Y), (pvars Y), (dvars Y), and (evars Y). The argument Y for each function must be a directory. These functions return the names of all variables, all plain variables, all directory variables, and all executable variables which are entries of Y. The result of each of these functions is a nested vector of literal vectors, where each literal vector is a referent name with a trailing blank appended. With this format, novice users could easily read default output of the form (Name1 Name2) and more experienced users would be able to manipulate the actual result which had the form {"Name1 ";"Name2 "}. These experienced users would be able to write detailed query functions (e.g. a function to return only directory names starting with "P") if the need arose.

It should be clear that the above description of directories solves the difficulties associated with classifying names

and isolating subsystems from unforeseen interactions. There are a few important implications. The \ notation manipulates only a single directory entry at a time; it would be desirable at times to manipulate several entries at once. Also, the directories are only used to store objects within the active working area, but operating systems such as UNIX suggest that directory manipulations on permanently saved objects are feasible and useful. These two concerns are addressed in the next section.

A very important question is whether nested arrays can be implemented in an efficient enough fashion so that directories can quickly be accessed. A common manipulation such as (D\E: F\G) would typically require processing of at least two directory lookups and an assignment which could completely resize D. IPSA packages lend some support to the idea that directory manipulations are technically feasible to implement efficiently; however, although several writers have discussed adding directories to APL, there has not yet been an implementation. Only an actual implementation of directories can answer this question; the UNIX operating system might suggest some techniques, but the details are beyond the scope of this thesis.

Directories (along with the attendant concepts of executable arrays and (& ! %) have a few disadvantages:

1. JMSL users would have to prefix all invocations of user-defined functions with a special character. Crick [39] notes that such a prefixing convention might aid parsing for both humans and computers.

2. JMSL users would experience some chance of unexpected classification with the functions (pvars, evars, and dvars); a variable expected to be a plain variable might happen to fit the definition of an executable variable or a directory variable, and be classified as such. This problem would occur very rarely in practice, since 4-dimensional nested arrays occur infrequently. Standard APL avoids the ambiguity by storing an object's syntactic class with the object.

3. In JMSL, a function's name would not be documented in the header of its referent, so that if a function were renamed, a clue to the original purpose of the function would be lost. Standard APL variables and APL component files already suffer from this problem; the cure lies in making sure that referents are always associated with meaningful names, and/or providing documentation within the function.

4. Recursive functions in JMSL are not as simply explained as they are in APL. This is an inevitable complexity but not a hindrance in most programming situations.

The advantages of directories would seem to outweigh the disadvantages:

1. Users who need to classify named objects hierarchically (either by name or by position) could easily do so, while users who do not need this capability could ignore it. Hierarchical classification aids in documenting workspace organization and facilitates interfacing of subsystems.

2. The concept of executable arrays would unify JMSL in the areas of directory classification, editing, and function copying. The concepts of executable arrays supersede the traditional APL ∇ editor and the Standard APL functions ⎕FX and ⎕CR.

3. Users would obtain results from queries such as (pvars Y) and (evars Y). The Standard APL function ⎕NL gives similar results based on the entire working area, but the JMSL functions allow queries of specific directories.

ASPECT: THE APL TEMPORARY/PERMANENT STORAGE INTERFACE IS WEAK.

PROBLEM: APL WORKSPACES DO NOT ADDRESS THE PROBLEM OF STORING DATA WHICH EXCEEDS THE (USUALLY FIXED) WORKSPACE SIZE.

Although nowhere stated in the Standard APL documentation, APL implementations have traditionally employed workspaces of fixed size. This is partly due to the fact that many operating systems are geared towards fixed-size workspaces. Some operating systems, such as UNIX, permit multiple users to have variable-sized storage areas similar in concept to workspaces. This is probably the ultimate solution to the problem.

However, fixed-size active workspaces are a reality on many operating systems; these operating systems are insufficiently flexible to allow transferring data between primary and secondary storage invisibly to users. This problem seems likely to remain for some time, particularly with smaller systems (where it may be technologically infeasible to provide this capability at low cost). Pesch [47] states:

> I.P. Sharp and other large timesharing organizations have argued that [use of large virtual workspaces] imposes unacceptable limitations on user load and response time.

Fixed-size active workspaces may actually have some desirable properties in an interactive language such as APL. It is easy for users to make mistakes which are costly in data transfer; for example, if a user types 1E99 the system should not respond by trying to extend the active workspace to gigantic size.

However, assuming that active workspaces of relatively small fixed size must be lived with, the problem of storing huge quantities of data which exceed this fixed size is not addressed in Standard APL. This problem must be solved in order for APL to have any real commercial success.

PROBLEM: APL DOES NOT FACILITATE THE MANIPULATION OF PERMANENT OBJECTS STORED IN WORKSPACES.

In Standard APL, all permanent data is stored within named workspaces. One deficiency of workspaces is that they suffer from the same limitations as Standard APL variables and functions: the namespace of the workspace is flat and does not allow hierarchical structuring of the names. (Many APL implementations use a 2-tier structure, with numbered library accounts and named workspaces; Bozinovic [46] states that this is unlikely to withstand the test of time.)

Another deficiency of workspaces is that they may be manipulated only by system commands. System commands may not

be executed within functions, so it is impossible for any function to modify any permanent object within a workspace. Most programming languages allow programs to modify data stored permanently in files.

Even outside of functions, system commands do not permit permanent objects to be easily manipulated. Objects must be copied from a stored workspace into the active workspace, (possibly) modified, and then saved back into a (possibly different) workspace. The programmer must be careful to avoid WS FULL errors and name conflicts.

PROBLEM: APL WORKSPACES ARE NOT EFFICIENT FOR SOME KINDS OF DATA TRANSFER.

One problem with data transfer in Standard APL is the fact that objects are manipulated singly. For example, only a single object at a time may be extracted from a stored workspace using the system command)COPY WS X. More generally, there is no way to initialize several related variables using a single assignment, or to copy several functions at a time.

System commands do not permit objects to be renamed when they are copied in. There is no way to copy out single objects back to permanent storage, as Wheeler [48] notes. System commands do not even follow APL syntax. Crick [39] gives a table which illustrates just how irregular APL syntax gets.

Object	Copy the object	Erase the object	Lock the object
1. Array	*Y←X*	*)ERASE X* or *⎕EX 'X'*	*X:*[1] or *⎕LOCK 'X'*[1]
2. File[2]	Requires a non-trivial program	*FILENAME ⎕TIE N ⋄ FILENAME ⎕ERASE N*	*FILENAME ⎕TIE N ⋄ ACCESS ⎕STAC N*
3. Function	Edit header *or* *)COPY WS FUNCTION* or Non-trivial program	*)ERASE X* or *⎕EX 'X'*	*∇X∇* or *⎕LOCK 'X'*
4. Workspace	*)LOAD WS* or *)SAVE WS* or *⎕LOAD*[1] *⎕SAVE 'WS'*[1] or *)WSIS WS* renames	*)DROP WS* or *⎕DROP 'WS'*[1]	*)SAVE WS:PASS*
5. Group	*)GROUP X Y*	*)GROUP X* or *⎕EX 'X'*[1]	*⎕LOCK 'X'*[1]
6. Package[3]	*Y←X*	*)ERASE X* or *⎕EX 'X'*	*⎕PLOCK X*

[1] Found only on CDC APL Version 2 (APLUM)

[2] STSC has an 'F' after the quad in all file function names

[3] Found only on IPSA's APL (Sharp APL)

Figure 3. Irregularities in APL Syntax.

PROBLEM: APL WORKSPACES DO NOT ALLOW MULTIPLE USERS TO SHARE DATA SECURELY.

Since APL implementations vary on details regarding multiple users, Standard APL does not discuss the issues involved. Two issues of paramount concern in multi-user systems are resource conflict management and secure sharing of data among users. In particular, some way of preventing functional code from being indiscriminately accessed must be provided. Authorized sharing of data among colleagues must be facilitated, while unauthorized access to data must be hindered.

PROPOSED SOLUTIONS: SOLUTIONS INCLUDE FILE SYSTEMS, SHARED VARIABLES, REVISED WORKSPACE COMMANDS, TRANSPARENT FILES, AND PERMANENT DIRECTORIES.

File systems are widely used in APL to store permanent data which exceeds the workspace size. Two types are well-known: component files (such as those used on the STSC and IPSA implementations), and TOTAL files (used on the Burroughs 700 system). These file systems are similar in concept; the differences are not important here. Both file systems store APL arrays into an external file and define functions to perform the following manipulations:

--insert/modify/delete/append an array to/from a file
--query a file for information such as the number of records on file
--hold/release a file
--control security access to a file

Ruehr [8] sums up the limitations of this approach as follows:

> Although they solve many of the problems of arrays, files have also revoked many of their advantages: first, 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.

Shared variables are part of Standard APL; they have frequently been used instead of files to store data which exceeds the active workspace. Shared variables, however, are inconvenient in that they must be explicitly coupled and decoupled. Shared variables are designed for 2-way sharing rather than N-way sharing; on a multi-user system it is desirable for N users to be able to access and update a database.

Wheeler [48] discusses improved sharing of APL workspaces

and libraries. Wheeler describes access matrices which may be assigned to workspaces and libraries. These access matrices are similar in concept to the widely-used file access matrices (which encode user account numbers, permission codes, and passnumbers). Unfortunately, Wheeler describes three separate kinds of matrices rather than a single unified approach. Wheeler also describes system functions such as COPY and ERASE which perform actions analogous to the corresponding system commands (within the local environment of the function, so that dangerous side effects are avoided). A new command)STORE (and the analogous STORE) is described, which complements)COPY (and COPY) by saving selected objects from the active workspace into an existing saved workspace. In discussing security, Wheeler observes that APL implementations sometimes inhibit legitimate sharing of workspaces among colleagues (for example, users cannot query)LIB of another user's account on some systems), and he also observes that passwords do not contribute significantly to workspace security. These ideas are noteworthy; however, the overall implementation described by Wheeler is excessively complicated. Lucas [49] proposes a form of permanent storage called a "transparent file" which is more flexible than the file systems previously described. A transparent file contains objects with names of the general form:

⍞OBJECT⍞FILE⍞LIBRARYACCOUNT⍞PASSWORD.

The most significant feature of Lucas' preliminary proposal is that such objects may be manipulated by any APL functions, rather than by specialized file functions. These objects behave like huge variables as far as users are concerned. Lucas describes the advantages as follows:

> [Transparent files] provide a means of applying any and all APL functions to externally filed data without the need for separate read and write operations or explicit loops. File references could be treated as a separate datatype (or types) by the internal code of the primitive functions, which could then handle the segmented reading, processing, and writing internally. Though this would involve additional overhead in the interpreter I believe that in the long run the savings obtained through the reduction of design time and the elimination of iterated interpretation of explicit loops would more than compensate.

Lucas recognizes that transparent files raise a number of implementation challenges. Lucas defines new functions which hold, resize, and set security for transparent files. He notes that a variety of opinions exist about security and describes a rather comprehensive range of permission specifications. He observes that details such as opening/closing files and limitations on the number of currently open files should

be invisible to users, and that garbage collection and internal updates must be efficient. He points out that some mechanism must be provided in order to prevent sequential writings of the form FILE←FILE,NEXTDATAITEM from being grossly inefficient. He mentions a possible need for surrogate names for transparent files since his naming convention is a long one.

It seems likely that transparent files are technologically feasible. Pesch [47] describes one possible mechanism for manipulating huge arrays although this mechanism is not as transparent as the mechanism Lucas specifies. One limitation of transparent files is that the naming convention relies on a 2-tier (account/workspace) hierarchy of permanent storage. As the previous section suggested, hierarchical names are desirable for permanent storage as well as for temporary storage.

Crick [39] was the first to propose permanent directories instead of workspaces. The idea has received considerable attention, and Taylor and Whitney [41] describe their concept of permanent directories as follows:

> The Tree is an array: in fact, a [directory]. Everything in the Tree is an APL object...Any name which begins with an underbar (e.g. _user1_ws1) is resolved in the Tree. Every object in the Tree is potentially accessible, given permission by its owner, to any task. Although the Tree is conceptually a single object, it may in fact be stored in all sorts of ways: disk packs, other machines, perhaps even incoming data lines. The important aspect is the uniform notation for referencing and (where applicable) setting the objects.
>
> Note that names that don't begin with _ are resolved as usual in the task's symbol table.
>
> Typically many nodes of the Tree would comprise too much data to fit into the task's working area, but the notation allows one to distinguish pieces of it. When referencing, the entire name must specify an object small enough to fit in the task. For referencing and setting, the name can include regular indexing at the end. For example, _user1_file1[n] .

Taylor and Whitney's concept is similar to the Lucas' transparent files, with the addition of hierarchical naming for permanent objects. The last paragraph is illuminating--Taylor and Whitney do not require that the workspace be somehow expanded or that permanent objects be treated exactly as huge variables; instead, they imply that a uniform notation for setting and referencing all APL objects is more important

than the fact that some huge objects must be processed a piece at a time. The concept is upwards-compatible with variable-sized workspaces, of course.

Permanent directories which can be treated like temporary directories and manipulated with functions seems a promising direction for APL. The grand unification implicit in such a scheme eliminates specialized file functions and many workspace commands, as Taylor [40] and others note.

One feature which has not been discussed in the APL literature is the need for absolute and relative pathnames for permanent directories. UNIX allows both forms; this is quite useful, especially when several machines are connected into a network. Another area worthy of further exploration is the area of security. Crick [39], Taylor [40], and Lucas [49] each propose that every permanent object have its own unique access matrix. Crick would even assign access matrices to subscripted portions of permanent objects; this could be viewed by less security-minded implementors as wastefully inefficient and an inessential requirement.

JMSL SOLUTION: PROVIDE MECHANISMS TO MODIFY PERMANENT DIRECTORIES.

JMSL provides permanent directories to store data which may exceed the storage limit of a fixed-size working area. In general, these permanent directories are similar to the directories described in the previous section, except that (1) they may store huge objects which may be far larger than the working area, and (2) they store these objects permanently.

The directory concept and \ notation already discussed can readily be extended to these permanent objects. (Standard APL workspaces are little more than collections of named objects such as variables and functions, and on many implementations APL libraries are just collections of named objects which all happen to be workspaces; further tiers of classification can easily be imagined.)

JMSL uses a permanent directory called (root), which is conceptually nothing more than a gigantic array which contains permanent objects using the hierarchical directory scheme described in the last section. The leaf nodes are (possibly huge) permanent arrays, which may include executable arrays. This form of directory structure is convenient for hierarchical machine networks which may contain multiple users.

When logging onto a multi-user system, the JMSL user specifies an existing subdirectory of the permanent directory (root). In a very hierarchical system, the logon procedure might ask the user to specify a pathname from the root to the logon procedure might ask the user to specify a pathname from

the root to the subdirectory in which the user wanted to work. More typically, the logon procedure would group all users into a single subdirectory and automatically assume this subdirectory as a prefix, to be supplemented by an entered user id. In either case, an active working area is created as the newest entry of this subdirectory.

This active working area is called an active directory; it is similar to an APL workspace. Like an APL workspace, it is initially empty of data, except for system variables which have default values. The active directory is of fixed size, in contrast to other permanent objects which may be of variable size. (This is a reflection of the fact that on many computer systems, the constraints regarding primary storage are less flexible than the constraints regarding secondary storage; systems which have no limitations in this regard could declare the fixed size of the active directory to be a very large number.) The active directory becomes a newly-created entry of the subdirectory specified at logon time, and the active directory and its contents continue to be part of the specified subdirectory until the user logs off. When the user logs off. the active directory's entry is automatically deleted.

A user may refer to a saved subdirectory or executable array or other array by its path from the root (e.g. root\User1\Project7\X). A user may also refer to an object in an active directory by specifying a similar path from the root (e.g. root\User2\active\Y); each user's active directory is named (active). However, the user may more concisely refer to (root\User2\active\Y) as (Y); the JMSL system supplies the prefix (root\User2\active) based on the user's logon id. This path notation may be used to access any branch node or leaf node of the permanent tree (including any active directory and its contents, which are temporarily part of the permanent tree). The pathname from the root to the object is unique, and is called an absolute pathname.

Absolute pathnames may be used to perform operations analogous to loading, saving, and clearing an APL workspace. A user may create objects in an active directory and then clear the active directory of these objects by an assignment of the form (root\User2\active: new). The system constant (new) is a directory containing only JMSL system variables and their default values. The user could create further objects and then save the active directory with a specification of the form (root\User2\MondayWork: root\User2\active). The user could also save an empty directory. The next day, the user could continue working by loading the saved directory with an assignment of the form (root\User2\active: root\User2\MondayWork). At the end of the week, hierarchical classification of the saved directories could be accomplished by a sequence such as:

```
root\User2\Week1Work\MondayWork: root\User2\MondayWork
:                                :
root\User2\Week1Work\FridayWork: root\User2\FridayWork
```

In addition, the user would be able to manipulate portions of the permanent objects which fit within the active workspace; for instance, if X were a small variable from Monday's work, the user could modify its value with an assignment of the form (root\User2\Week1Work\MondayWork\X: NewValue). In particular, permanent objects could be manipulated within functions and passed as parameters, as long as the manipulations fit within the bounds of the active directory. Manipulations such as an update of a huge file would typically be performed in segmented fashion, reading part of the file into active memory, modifying this part, and writing back the revised part. A manipulation such as root\File1: root\File2 might involve two huge permanent objects but not necessarily any storage in the fixed-sized active workspace; the success or failure of this type of transfer could reasonably be left implementation-dependent.

The following table illustrates absolute pathnames for a small educational network of computers:

Node #	Network Tree	Absolute Pathname for all users
1	root	root
2	UnivA	root\UnivA
3	Stud1	root\UnivA\Stud1
4	active	root\UnivA\Stud1\active
5	Var1	root\UnivA\Stud1\active\Var1
6	UtilFns	root\UnivA\Stud1\active\UtilFns
7	Fn1	root\UnivA\Stud1\active\UtilFns\Fn1
8	Fn2	root\UnivA\Stud1\active\UtilFns\Fn2
9	Proj1	root\UnivA\Stud1\Proj1
10	Var2	root\UnivA\Stud1\Proj1\Var2
11	Stud2	root\UnivA\Stud2
12	Var3	root\UnivA\Stud2\Var3
13	Fn1	root\UnivA\Stud2\Fn1
14	UnivB	root\UnivB
15	Stud3	root\UnivB\Stud3
16	active	root\UnivB\Stud3\active
17	Var1	root\UnivB\Stud3\active\Var1
18	Fn1	root\UnivB\Stud3\active\Fn1
19	Proj3	root\UnivB\Stud3\Proj3
20	Var2	root\UnivB\Stud3\Proj3\Var2

Table 10. Absolute Pathnames in JMSL.

It is evident from this table that absolute pathnames be-

come lengthy even for a relatively small system. Moreover, if a system is expanded to include new levels near the top of the hierarchy (e.g. the case where the educational network becomes a part of a statewide network), all the absolute pathnames must be revised. The UNIX operating system employs pathnames which are relative to each user's point of view to alleviate the difficulties of absolute pathnames. JMSL uses a similar technique to allow users to ignore parts of the system which do not concern them.

In JMSL, a relative pathname begins with a user's active directory, which is termed $. The active directory's siblings within the permanent tree are referred to by names of the form $Sibling1, $Sibling2, etc. The parent of the active directory is termed $$, and the siblings of $$ have names of the form $$Sibling1, $$Sibling2, etc. The parent of $$ is named $$$, and further ancestors of $ are named with longer strings of $s. Ultimately, the root is referred to by the longest string of $s; the exact number of $s depends upon the user's position within the system hierarchy and may vary from user to user. From the nodes of the permanent tree described so far (all closely related to $), any other node of the permanent tree may be reached by qualification with \ (e.g. $$Sibling1\User2\Project5). The following table illustrates relative pathnames within the educational network mentioned above:

Node #	Relative Path for Student 1	Relative Path for Student 3
1	$$$$	$$$$
2	$$$	$$$UnivA
3	$$	$$$UnivA\Stud1
4	$	$$$UnivA\Stud1\active
5	Var1	$$$UnivA\Stud1\active\Var1
6	UtilFns	$$$UnivA\Stud1\active\UtilFns
7	Utils\Fn1	$$$UnivA\Stud1\active\Utils\Fn1
8	Utils\Fn2	$$$UnivA\Stud1\active\Utils\Fn2
9	$Proj1	$$$UnivA\Stud1\Proj1
10	$Proj1\Var2	$$$UnivA\Stud1\Proj1\Var2
11	$$Stud2	$$$UnivA\Stud2
12	$$Stud2\Var3	$$$UnivA\Stud2\Var3
13	$$Stud2\Fn1	$$$UnivA\Stud2\Fn1
14	$$$UnivB	$$$
15	$$$UnivB\Stud3	$$
16	$$$UnivB\Stud3\active	$
17	$$$UnivB\Stud3\active\Var1	Var1
18	$$$UnivB\Stud3\active\Fn1	Fn1
19	$$$UnivB\Stud3\Proj3	$Proj3
20	$$$UnivB\Stud3\Proj3\Var2	$Proj3\Var2

Table 11. Relative Pathnames in JMSL.

The same manipulations which may be performed by absolute pathnames may also be performed using relative pathnames; relative pathnames will nearly always be shorter and more convenient for common manipulations. Clearing the active directory can be performed by ($: new). Saving the active directory is done by ($MondayWork: $). Loading a saved directory is done by ($: $MondayWork). Hierarchical classification of directories may be done from a relative point of view; for example,

```
$Week1Work\MondayWork: $MondayWork
:                      :
$Week1Work\FridayWork: $FridayWork
```

Manipulation of permanent objects within the active workspace may also be performed with assignments such as ($Week1Work\MondayWork\X: NewValue).

The JMSL scheme must address some subtle issues which do not appear in APL implementations which use libraries and workspaces. The problem is that since JMSL arrays are easily manipulated, a user may attempt to modify directories and executable arrays inappropriately. Examples of these inappropriate manipulations are:

(1) $: 0 (or equivalently, root\UserMe\active: 0)
--- Problem: user's active directory no longer is a directory

(2) $$: 0
--- Problem: respecification of active directory's parent would destroy the active directory

(3) root: 0
--- Problem: respecification of the root would destroy all user-defined objects, including any active directories

(4) root\UserYou\active: 0
--- Problem: colleague's active directory is no longer a directory

(5) F: 0 (where F is the executable array for a currently executing function)
--- Problem: currently executing function is no longer a function

These manipulations are unusual and do not normally occur in programming practice; problems (2), (3), and (4) would usually be prevented by default security considerations implemented to protect each user's active workspace, but users could conceivably modify the safeguards. APL implementations have struggled with similar issues for years and have found several different solutions. JMSL standardizes the results of these actions by disallowing each of them; an error message is returned in each case. The guiding principles are:

(1) Active directories and their ancestor directories (including the root, of course) may not be implicitly or explicitly modified in any way which causes them to no longer be directories. Active directories may be erased only by logging off; ancestor directories of active directories may not be erased at all.

(2) No function which is currently executing may be modified in any way.

(3) JMSL system objects (such as system functions or constants) may not be modified or erased, except for system variables which may only be modified in ways consistent with their domains.

While the above scheme provides a way to store data which exceeds the capacity of the active directory, and also provides a way to manipulate permanent objects, the real utility of saved directories and active directories becomes apparent when the data transfers associated with these concepts are streamlined.

Multiple assignments may be performed in JMSL using the form exemplified by ({Var1;Var2;Var3}: Expression), where Expression must evaluate to a result which is conformable to a 3-element nested vector; Var1, Var2, and Var3 are assigned the corresponding items of this Expression. A number of details such as length conformability and repeated names are resolved in the appendix. In particular, ({Var1}: {}) is defined to erase Var1; if Var1 is not defined, the assignment is ignored. The result of multiple assignment may be used as part of other expressions or for further assignments. Similar versions of multiple assignment have been proposed by Falkoff [34] and implemented on NARS.

Shortcuts become convenient in conjunction with multiple assignment. $D\E\{X;Y;Z} is defined to be equivalent to {$D\E\X;$D\E\Y;$D\E\Z} (see appendix for further details). {X Y Z} is unambiguously defined as {X;Y;Z} in the case where X Y and Z are all valid user-defined variable names. {X Y}: D\{X Y} may be abbreviated as {X Y}: D\.

The chief merit of the multiple assignments and the shortcuts is that if a dyadic 'merge' function is suitably defined to merge its right argument (a directory) into its left argument (another directory) resulting in a composite directory, JMSL can replace APL system commands with more powerful assignments which do not require a special syntax. (The commands shown here are STSC commands which are among the most powerful extensions to the Standard APL system commands.)

STSC APL*PLUS SYSTEM COMMANDS	JMSL EQUIVALENT
)CLEAR	$: new
)COPY WS OBJ1 OBJ2	{Obj1 Obj2}: $Direc\
)COPY WS	$ merge: $Direc
)CONTINUE HOLD	$Continue: $
)DROP WS	{$Direc}: {}
)ERASE OBJ1 OBJ2	{Obj1 Obj2}: {}
)FNS	evars $
)GROUP GPNAME OBJ1 OBJ2	Direc\{Obj1 Obj2}: $\
)GROUP GPNAME	$ merge: Direc
)GRP GPNAME	vars Direc
)GRPS	dvars $
)LIB	dvars $$
)LOAD WS	$: $Direc
)OFF HOLD	{$}: {}
)SAVE WS	$Direc: $
)SAVE	(no equivalent; must specify a directory)
)STORE WS OBJ1 OBJ2	$Direc\{Obj1 Obj2}: $\
)STORE WS	$Direc merge: $
)VARS	pvars $
)WSID WS	(no equivalent)
)WSID	(no equivalent)
)XLOAD WS	&$DriverFn (where the first action of the function is to copy in any needed objects)

Table 12. APL*PLUS System Commands and JMSL Equivalents.

The JMSL equivalents are of roughly comparable length to these system commands. The JMSL notation allows many more general capabilities:

INVALID APL*PLUS COMMAND	JMSL EQUIVALENT
)VARS WS	pvars $Direc
⍴)VARS	#pvars $
)SAVE WS1←)LOAD WS2	$Direc1: $Direc2
)EXPORT UNIVC WS FRIEND PKG Y←2	$$$UnivC\Friend\Dir\Y: 2
OBJ1←)COPY WS OBJ2	Obj1: $Dir\Obj2

Table 13. JMSL Capabilities With No APL*PLUS Equivalents.

Permanent directories also replace the usual manipulations on APL component files; here the IPSA system (a typical implementation) is compared:

IPSA FILE FUNCTION	JMSL EQUIVLENT
⎕LIB acctnum	dvars $$
⎕NAMES (of open files)	(not needed)
⎕NUMS (of open files)	(not needed)
filename ⎕CREATE filetie	$File: 0⍴⍥StartVal (optional)
filename ⎕RENAME filetie	$NewFileName: $OldFileName
filename ⎕ERASE filetie	{$File}: {}
⎕UNTIE filetienum	(not needed)
⎕SIZE filetienum	(#$File is a near-equivalent for (⎕SIZE filetie)[2]-1)
newsize ⎕RESIZE filetie	(not needed)
⎕READ filetie,compnum	$File[Component]
newval ⎕REPLACE filetie,compnum	$File[Component]: NewValue
newval ⎕APPENDR filetie	(not needed)
newval ⎕APPEND filetie	$File: $File,NewValue
numcompstodrop ⎕DROP filetie	$File: NumToDrop drop $File

IPSA packages could also be compared, but the utility of the JMSL scheme should be apparent by now.

Conceptually, security in JMSL is implemented using a security processor which keeps track of the security for permanent directories. Since moving an object or group of objects into another directory is a trivial matter, it seems unnecessary to provide the subscripted level of security proposed by Crick [39]. The security processor stores the absolute pathnames of each permanent directory, the absolute pathname of the users who may access each of these permanent directories, the permission codes specifying what each user may do, and a list of objects currently held within the directory which may not yet be accessed. The UNIX system suggests that 'read', 'write', and 'execute' privileges are the only necessary privileges; perhaps 'hold' and 'change permission' privileges could be added to this list, but the latter two have proved unnecessary on the UNIX system, so JMSL does not include them. The security could be changed with the following group of functions:

```
UserPath may    PermObj
UserPath mayr   PermObj   (note that read privilege implies
UserPath mayw   PermObj   that another user can copy an execu-
UserPath mayx   PermObj   table array and subsequently execute
UserPath mayrw  PermObj   the copy, so for all practical pur-
UserPath mayrx  PermObj   poses the function pairs mayr & mayrx
UserPath maywx  PermObj   and mayrw & mayrwx are equivalent)
UserPath maywx  PermObj
UserPath mayrwx PermObj
```

The security privileges are cumulative, so that ($Pat mayrwx $File) followed by ($Lee mayrwx $File) allows both Pat and Lee to use $File; to retract the privileges, the owner of $File must specify ({$Pat;$Lee} may $File). Security

may be queried by saying (perm $File), which would return a nested matrix containing rows of colleagues' directories followed by a 3-element permission code (e.g. "r-x"); of course, if a colleague had no form of permission for $File, there would be no listing. As with the (vars), (evars), (dvars), and (pvars) functions mentioned in the previous section, the result of (perm) is a nested array, but novice users would not need to be concerned about the nesting since the result would be rarely used in expressions. The pathnames of the permanent directories would be stored internally in the security processor as absolute pathnames, but returned as literal vectors which would display the pathname from a relative point of view. JMSL provides the functions (abspath) and (relpath) to convert between literal vectors representing absolute pathnames and relative pathnames.

There are several implications to the above scheme. The description presents a minimal number of features; almost certainly, an actual implementation of JMSL would soon recognize a need for further functions. The security processor presented in a manner which is as orthogonal to the rest of JMSL as possible, since different implementations have different security requirements. A relatively small multi-user system which implemented the capabilities described in the above scheme would scarcely need a bulletin-board/mail system.

There are some disadvantages to the JMSL solution:

1. Not every STSC command translates neatly into JMSL; in particular,)PCOPY and)PSTORE are problematic. These commands behave like)COPY and)STORE but fail gracefully if a name would be overwritten. The problem is that they would require another symbol for assignment such as (X?Y) but ASCII has only a limited number of symbols. Since many APL installations do not have these commands, JMSL ignores the issue.

2. It would be desirable to have a renaming facility which would perform the equivalent of (NewName: OldName ! {OldName}: {}) in JMSL, but this would be as problematic as)PCOPY or)PSTORE. None of the major APL implementations provide a rename capability.

3. JMSL does not return information such as a permanent directory's time of most recent update or the active directory's source (c.f. WSID). Perhaps this would be missed by some users. IPSA's inclusion of a command)QLOAD which suppresses the usual message about a workspace's source and time of most recent update suggests that this information is often a hindrance. A number of interactive languages use working areas but do not use a concept analogous to WSID; these languages require users to think more carefully about loading/saving work, but have one less concept to learn.

4. JMSL does not automatically retain audit trail information such as an object's time of last update; some component file systems automatically save such information for each component. This information is frequently ignored by programmers and is often examined only in case of some abnormality such as recovery from a system crash. JMSL users who need audit trail information could explicitly include this information when writing other data to a permanent object.

The advantages of the JMSL solution seem to outweigh the disadvantages.

1. Permanent directories allow the storage of data which exceeds a fixed workspace size and are upward-compatible with a variable workspace size.

2. Permanent directories allow hierarchical classification of names and allow permanent objects to be manipulated even inside functions without the possibility of name conflicts. The notational scheme is compatible with the notation of the previous section and generalizes to hierarchical machine networks which need to employ both absolute and relative pathnames.

3. Multiple assignments are useful in their own right for initializations. They may be used in conjunction with directories to replace the capabilities traditionally performed by system commmands, file functions, shared variables, and packages. The resulting capabilities maintain an APL-like syntax. The added power of multiple assignments combined with permanent directories compensates for the fact that a few non-crucial system commands cannot be readily implemented.

4. Security is added in a manner which has proven in UNIX to be unobtrusive to use and efficient to implement. Security considerations have minimal impact on users who do not need to be particularly security-conscious, while providing adequate protection for users who need to be more careful. Since executable arrays may be saved permanently and assigned execute-only priveleges, there is no need for the locked functions traditionally employed by APL implementations.

ASPECT: APL LACKS HIGH-LEVEL CONTROL STRUCTURES.

PROBLEM: APL BRANCHING LEADS O DIFFICULTY IN SPECIFYING SELECTION AND LOOPING CONTROL STRUCTURES.

Most computer programming languages have control structures to delineate flow of control in selections (such as if/then/else or case/of) and in looping (while/for/repeat). APL is limited to a single control structure, the branch arrow. It is widely recognized that APL branching is insufficient for production programming; Samson [50] cites 10 references to support this view. However, the subject of exactly how to introduce control structures into a language designed to minimize explicit looping such as APL is a controversial one. This may explain why Standard APL does not yet have a control structure other than branching, even though structured programming has been a recognized discipline since Nicholaus Wirth decried "goto abuse" in 1968.

PROBLEM: APL LACKS A MECHANISM FOR EVENT HANDLING.

Some programming languages such as ADA and PL/I have mechanisms to handle events. The term "event" is usually (though not always) synonymous with "error". Especially in production environments, errors frequently cannot be anticipated. Even when they can be anticipated, a test for the error is frequently wasteful in time. (A classic example is testing a matrix to make sure it is not singular before finding the inverse of the matrix. The test for singularity is approximately as time-consuming a process as actually calculating the inverse.) If errors cannot be anticipated, it is still desirable to handle them in some way. In a complicated data storage algorithm, a 'WS FULL' error might signal that garbage collection should be performed; if the garbage collection fails to yield enough free storage, the application should die gracefully with a message to users more along the lines of 'YOU NEED MORE SPACE, PLEASE CALL J. PROGRAMMER AT 555-1234' than an APL error message. Standard APL does not have any mechanism for event handling.

PROPOSED SOLUTIONS: APPROACHES INCLUDE APLGOL, SPECIALIZED CONTROL STRUCTURES, AND EVENT HANDLING MECHANISMS.

Not surprisingly, many proposals have been published which attempt to solve these problems. One solution to the control structure problem, APLGOL, fused APL statements with an ALGOL control structure. Sniedovich [51] gives a good example comparing Standard APL and APLGOL:

Figure 1. The function DEAL

```
    ∇ Z←A DEAL B;I;J
[1]   →SHORT IF A<⌊B÷16
[2]   Z←(⍴Q-1)+⍳B
[3]   →END IF A=0
[4]   I←0
[5] LOOP:J←1+I+(ROLL B-I)-Q
[6]   I←I+1
[7]   Z[I,J]←Z[J,I]
[8]   →LOOP IF A>I
[9] END:→0,0⍴Z←A↑Z
[10] SHORT:Z←⍳0
[11] OUTER:→0 IF A=⍴Z
[12] INNER:I←ROLL B
[13]   →INNER IF I∈Z
[14]   Z←Z,I
[15]   →OUTER
    ∇
```

Figure 2. APLGOL version of DEAL

```
PROCEDURE Z←A DEAL B,I,J
   IF A≥⌊B÷16 THEN
      BEGIN
         Z←(⍴Q-1)+⍳B;
         IF A=0 THEN RETURN Z←A↑Z;
         I←0;
         REPEAT
            J←I+1+(ROLL B-I)-Q;I←I+1;
            Z[I,J]←Z[J,I];
         UNTIL  A≤I;
         RETURN Z←A↑Z;
      END
   ELSE
      BEGIN
         Z←⍳0;
         WHILE A≠⍴Z DO
            REPEAT I←ROLL B; UNTIL  ~I∈Z;
            Z←Z,I;
         END
      END
END PROCEDURE
```

Figure 4. Comparison of APL and APLGOL.

The APLGOL version does not require labels, clearly shows whether a keyword is a control structure keyword or a user-defined name, and uses indentation to aid readability. However, APLGOL uses a reserved-word approach and the excessive number of control structure tokens (notably PROCEDURE, RETURN, BEGIN, END, and ;) detract from readability.

Numerous other control structures have been proposed. Samson [50] in one of the latest papers on the subject of control structures, sums up the situation admirably:

> The need for control structures in APL having been stressed by so many people [3 4 5 6 10 11 12 13 14 15] over the last 15 years, my concern here is not to show once again that the lack of a decent way of expressing constructs such as "if then else", "do while", "case of" or "repeat" in APL is both a programming nuisance and a severe lack of expressivity in the language.

Samson's concern is to present what he terms "yet another control structure". Before presenting it, he motivates it with the following:

> ...all of the extensions seen so far in the literature or in implementations suffer from one of the five following weaknesses:
>
> 1- They use keywords such as "do", "while" or "then". Besides causing problems with normal APL homonyms--that could be cured with ⎕DO etc--the new constructs do not fit into APL

syntax. APLGOL is the best-known extension of this type.

2- They work for scalar "control arguments" only and they can't apply a (sequence of) operations to a set of arguments according to a set of conditions.

3- They do not yield an APL result.

4- They emulate control constructs by putting into action a lot of machinery that is completely alien to the construct. An example of such an extension--although not presented as a control structure extension per se--is Iverson's and Wooster's function-definition operator, in which (a function using) a "if X then A else B" is expressed as

```
('∇'T:A◇F:B◇→⍎(F,T)[⎕IO+⍵]') X
```

which does a pretty good job of drowning the intent by use of catenation, addition, and indexing.

5- They are inherently inefficient, e.g.
'FOO I' WHILE '0<(I←I-1' or
⍎¯7 4[⎕IO+CONDITION]↑'THIS:OR THAT'

Samson's points are accurate and may be responded to as follows:

1- Keywords are not a crucial problem (new symbols could be developed or a keyword token approach could be adopted). The problem that new constructs do not fit APL syntax is somewhat more serious; whether the new constructs are functions or not, their use should mesh with the philosophy of the language.

2- Control structures are needed primarily because array processing cannot do everything--at times a step-by-step approach is required. Array-oriented control structures which process blocks of statements at a time are hard for users to comprehend.

3- Control structures do not yield results in other languages and it is not clear that they should yield APL results.

4,5- It is quite true that control structures should look like control structures and that they should be efficient to read and execute.

Van Batenburg [52] also gives some criteria for judging control structures:

> --The control structures show the scope of their effect as fast and as easily as possible
>
> --The user functions [which implement control structures] are relatively small [in number] in order to present comprehensively the main objectives of the program [i.e. not detract from readability]

Van Batenburg's first criteria is desirable, but the second criteria presupposes that user functions will be used to provide control structure and that it is desirable to allow one symbol to take on several different meanings.

There have also been numerous proposals for event handling mechanisms in APL. IPSA, STSC, and IBM have implemented event handling by adding new variables and functions. This approach seems to be based on the fact that APL traditionally favors a functional approach rather than a control structure approach. If control structures are to be provided in APL, it seems reasonable to consider event handling within this context. The existing implementations seem to be somewhat ad hoc: IPSA's ⎕TRAP variable does not reflect APL syntax; STSC modifies comments to become syntactically significant; IBM employs an auxiliary input/output stack processor in VSAPL and uses a rather weak ⎕EA function in APL2.

JMSL SOLUTION: CONTROL STRUCTURES WITH INDENTATION, INCREMENTAL ASSIGNMENT, AND AN EVENT-HANDLING CONTROL STRUCTURE AND PARAMETER.

JMSL implements control structures similar to those used in PASCAL or APLGOL; however, most keywords are of short, uniform length and the approach is not free-form. Disregarding other aspects of JMSL such as character set for the moment, the JMSL translation of the APL program above is:

```
fd Z←A DEAL B
   if A≥⌊B÷16
   t? Z←(⍳⍵-1)+?B
      A=0?? Z←A↑Z ⋮ return
      I←0
      do J←I+1+(ROLL B-I)-⍳⍵ ⋮ I←I+1
         Z[I,J]←Z[J,I]
         A>I?? redo
      Z←A↑Z ⋮ return
   f? Z←?0
      as A≠⍴Z
         do I←ROLL B
            I∈Z?? redo
         Z←Z,I
```

The keywords (if do as) which begin blocks are each 2 characters long; so are the syntactic elements (t? f?). Scope of blocks is shown by left-margin indentation rather than by tokens such as BEGIN or ; or END. This choice requires programmers to indent with great care--but in practice most programmers in block-structured languages already do indent with great care. The JMSL interpreter can help by indenting each line to the nearest indentation margin and by inserting a space if a space is required by syntax.

The sequence of control is iteration (as in most languages), although JMSL provides the same recursion capabilities that Standard APL provides. All statements to be performed iteratively as a group are indented to share a common left margin.

The JMSL forms of selection are presented along with PASCAL equivalents. In this context, {} are used to indicate that a construct may be repeated 0 or more times, and [] are used to indicate that a construct is optional.

	JMSL CONTROL STRUCTURE	PASCAL EQUIVALENT
	==========	==========
1.	Expression?? Statement	IF Expression THEN Statement;
2.	if Expression t? Statement1 {Statement2} [f? Statement3 {Statement4}]	IF Expression THEN BEGIN Statement1 { ; Statement2} END [ELSE BEGIN Statement3 { ; Statement4} END];
3.	if Expression f? Statement1 {Statement2} [t? Statement3 {Statement4}]	IF NOT(Expression) THEN BEGIN Statement1 { ; Statement2} END [ELSE BEGIN Statement3 { ; Statement4} END];
4.	on Expression Value1? Statement1 {Statement2} {Value2? Statement3 {Statement4}} [other? Statement5 {Statement6}]	CASE Variable OF ValueList1: BEGIN Statement1 { ; Statement2} END; {ValueList2: BEGIN Statement3 { ; Statement4} END;} [OTHERWISE: BEGIN Statement5 { ; Statement6} END;]

Notes:

1. Standard APL provides only ⍎(X)/'Y' to handle the IF-THEN construct.

2. The JMSL constructs (t?) and (f?) may not be replaced by such constructs as (0?) or (Variable?) within an "if" sequence.

3. This JMSL form complements construct 2; it is useful in translating programs which contain gotos to programs which do not contain gotos.

4. Standard PASCAL disallows the OTHERWISE portion, but certain extensions to PASCAL permit this variation. The JMSL construct permits an expression after "on", while PASCAL requires a variable after "CASE". JMSL permits only single arrays to serve as case selectors, while PASCAL allows a list of case selectors. The JMSL expression and each case selector array are evaluated dynamically. If any of the case selector arrays are identical to the result of the expression, the statements corresponding to the first (and only the first) match are executed. If no case selector array matches, the statements corresponding to the "other" selector are executed; if the "other" selection is not provided, an error occurs. Indentation with the JMSL construct is less uniform than indentation with other constructs due to the varying lengths of the case selectors; if each case has a single statement the lack of uniformity is not a problem, and in any event, the first statement of each case may be left blank.

Looping forms are as follows:

5.
```
as Expression              WHILE Expression DO
   Statement1                BEGIN Statement1
   {Statement2}                 { ; Statement2} END;
```

6.
```
do Statement1              REPEAT Statement1;
  {Statement2}                   { Statement2; }
  redo                     UNTIL Condition;
```

7.
```
ea Variable: Expression    FOR Variable := Var1 TO Var2 DO
   Statement1                BEGIN Statement1
  {Statement2}                 { ; Statement2} END;
```

Notes:

5. The word "as" may be less clear than "while" but is 2 letters long and thereby maintains the indentation scheme.

6. The equivalence here between the JMSL construct and the PASCAL construct is imperfect; "redo" does not need to be the final statement (although structured programming would suggest this). The "redo" statement may appear at any time; whenever a "redo" is encountered, control jumps backwards to the statement following the first preceding "do". If no "redo" is placed within a "do" block, the "do" block acts more like a BEGIN/END block in PASCAL than like a REPEAT/UNTIL block. If "redo" appears outside of any "do" block, an error message ensues.

7. The JMSL expression Expression is evaluated once and the

iteration variable Variable successively becomes each element of the resulting array (in ravel order). The statements within the block are successively iterated. If the resulting array is empty, no statements of the block are executed. The abbreviation "ea" (for "each") may be less clear than "for" but is 2 letters long.

JMSL uses a control structure to define local functions:

8. fd LocalFnName R: X % Y Statement1 {Statement2}	FUNCTION(X: TYPE; Y: TYPE): TYPE; BEGIN STATEMENT1 { ; STATEMENT2 } END;

These two forms are roughly equivalent and the analogues to niladic and monadic functions and functions without result are obvious. Unlike Standard APL functions but like PASCAL functions, JMSL functions may contain local functions. Local functions may themselves contain local functions. The action of this control structure is easily described; it is the same as saying:

```
LocalFnName: (1 1,(N+1),1)@{"R: X % Y";"Stmt1";"Stmt2";...."StmtN"}
```

LocalFnName may be subsequently run, modified, or erased once it has been defined. The outermost function surrounding The local function does not include the 'fd LocalFnName' line. (The treatment of the scope of variables within these local functions is clarified in the next section.) The keyword "fd" is an abbreviation for 'function definition'.

JMSL contains five other control structures which are single statements:

9. exit	(no equivalent)
10. halt	GOTO 999; {Statements ;} 999: END.
11. label "LabelName" [Statement]	LABELNUM: Statement; [Statement]
12. jump "LabelName"	GOTO LABELNUM;
13. return	(no equivalent)

Notes:

9. JMSL "exit" transfers control out of the current block of statements wherever it is encountered. If "exit" appears in the outermost block control transfers into immediate-mode execution (or, like most control structures, if executed in direct mode leads to a "not in direct mode"

error).

10. JMSL "halt" performs the same action as APL →

11. JMSL labels work essentially like PASCAL labels, except that JMSL labels are literal constants for readability.

12. JMSL "jump" works like PASCAL goto. The intricacies of the APL branch arrow (such as branching to a line number out of a program or branching absolutely to a line number) are avoided. The label must be a literal constant, so that it may be evaluated statically.

13. Performs the same action as APL →0; automatically implied by the last line of a function.

Event handling is done in JMSL by extending the "do" structure presented earlier:

```
6b. do Statement1
      {Statement2}
      {at Array1
          Statement3
        {Statement4}}
```

When any type of error or interrupt occurs within the body of the "do" block, each array following a subsequent "at" is evaluated dynamically. These arrays must evaluate to strings which can be compared to the name of the error (e.g. "VALUE ERROR") and are typically constant strings. The statement block corresponding to the first (and only the first) array which matches the error is executed. The string "ANY ERROR" may be used after an "at" to match any error. The "redo" statement may still be placed anywhere within the statement(s) of a "do" block or within any statement(s) of one of the error handlers. If an error occurs which is not handled (including errors within an error handler) the error is propogated to surrounding "do" blocks; if no ultimate error handler exists for an error, the error is displayed as it would normally be displayed. This propogation is done via the system variable "signal" which contains the error message. If a user wishes to simulate a user-defined error, the user can simply assign something like:

```
signal: "My New Error"
```

which will cause an error to be simulated. These user-defined errors may be handled by "at" error-handlers, and if no error-handler is encountered, the error message "My New Error" will be displayed.

These control structures are largely an orthogonal addition to JMSL and they have few implications to other features of JMSL (other than obvious ones such as the fact that the choice

of keywords depends on the permissible characters of the language). Users must be somewhat more careful in indenting statements, but interpreters could be suitably modified to eliminate this problem for statements outside of any control block.

The addition of control structures to APL presents a few disadvantages:

1. Absolute branching (e.g. 5) and dynamic branching (e.g. →LABEL-3 or →(LABEL1,LABEL2)[I]) are not allowed. These APL techniques are highly suspect anyway and would not be missed by many experienced APL programmers.

2. The easy availability of control structures could lead to a decline in execution efficiency due to excessive use of looping. In many instances, the user is more concerned with programming efficiency than with execution efficiency. However, it is undeniable that it is easy to write APL code which is extremely inefficient, and the JMSL control structures make it even easier for a programmer to write grossly inefficient code. Another danger is the indiscriminate use of event handling. The cure is better training; most programming languages courses and texts stop with the bare minimum of techniques, but particularly in an APL-like language, subsequent training regarding style, efficiency, documentation, programming methods, and model programs is desirable.

3. Users would have to type more carefully. Even for programmers whose typing skills need improvement, typing is easier than programming and debugging.

However, the advantages seem to offset the disadvantages:

1. Users would gain the ability to express selection, looping, and event handling ideas in a natural, readable way.

2. The control structures are easy to edit, especially when compared to free-form languages such as PASCAL and C.

ASPECT: APL SHOULD AMEND SCOPING RULES.

PROBLEM: APL FREQUENTLY HAS PROBLEM WITH SIDE EFFECTS DUE TO AN EXCESSIVE NUMBER OF GLOBAL VARIABLES.

Standard APL allows functions to accept at most two arguments and to return at most one result. Many applications require functions to accept more than two arguments and/or to return more than one result. In these applications, programmers frequently resort to using global variables to pass the surplus arguments/results. An alternative is to concatenate variables which are essentially dissimilar and pass them as a single argument or result. Another alternative is to write global variables to a file. None of these techniques follows accepted structured programming practices.

PROBLEM: APL MAKES IT RELATIVELY HARD TO LOCALIZE FUNCTIONS.

In Standard APL, a function may be made local to another function only by dynamically creating a literal matrix of function lines, converting this matrix to a function using ⎕FX, and localizing the name of this new function within the function header. Since this is wasteful of both human time and computer time, APL programmers frequently omit such localizations. This leads to a large number of global utility functions in a workspace. If the utility functions are called by several application functions, this is not particularly bad. Frequently, however, a utility function is written expressly for a single application function; in this case it makes more sense for the utility functions to be easily and clearly localized within the application function.

PROBLEM: APL REQUIRES EACH LOCAL VARIABLE TO BE DECLARED IN A FUNCTION HEADER.

APL programmers find it a nuisance to search through a function to localize all variables. Every time a function has a new local variable added or an old local variable deleted, the function header must be changed. Consequently, some APL programmers simply ignore the rules of structured programming and do not localize all variables, so that some or all of these variables are considered to be global to the function. Even "conscientious" APL programmers sometimes mistakenly omit a localization; this in combination with APL's dynamic scoping rule can lead to extremely subtle bugs, as Seeds, Arpin, and LaBarre [53] note. This abuse of APL is unfortunately widespread and suggests that APL merits changes in name scoping.

Ching [54] finds fault with APL's dynamic scoping rules, which Ching describes as a dynamic version of the scope rules for block structure in ALGOL. Ching states:

> A major deficiency of block structure for modular construction of programs is that the visibility control on variables, i.e. the scope rule, of the block structure is a one-way street; while inner-block variables are not visible in its outer block, all variables in an outer block are accessible to all its inner blocks. Hence, code in some very small subunit--an implementation detail subject to modification during construction--can change the more stable parts of the program's data structure, including all global variables.

Seeds, Arpin, and LaBarre [53] criticize APL's dynamic scoping rules for similar reasons. Interestingly, Iverson and Falkoff [55] record that APL's scoping rule was originally a static rule which was later changed to the dynamic rule employed by Standard APL.

PROBLEM: APL DOES NOT HAVE A CONVENTION WHICH MAKES GLOBAL VARIABLES STAND OUT FROM LOCAL VARIABLES.

Even when names are appropriately localized, it is not always immediately obvious whether a function has any possible side effects to its environment. This is another consequence of dynamic scoping. APL programmers must document any side-effects with comments; if no documentation is available, the maintenance programmer may have a difficult time ensuring that no unforeseen shadowing is occurring.

PROPOSED SOLUTIONS: APPROACHES INCLUDE PROGRAMMING WORKAROUNDS AND REVISIONS TO LOCAL/GLOBAL SCOPING CONVENTIONS.

Few implementations deviate from Standard APL regarding name scoping; the practical solution to these problems has been the adoption of rather tedious programming practices: designing functions in ways which minimize the creation of global variables, rigorously localizing variables (including dynamically defined functions) and documenting side effects with comments.

Nested arrays solve the problem of passing an arbitrary number of arguments to and from a function. Strand notation is particularly effective here, as several APL implementations based on the floating approach allow function headers of the following form:

```
∇R1 R2 R3←(X1 X2 X3) FN Y1 Y2 Y3
```

Bozinovic [46] proposes a form of local function which is similar to PASCAL local functions; Bozinovic describes the advantages as follows:

> By nesting block definitions, local functions can

be defined without the need to 'fix' them at execution time. This will eliminate present penalties for modular programming: numbers of global functions or difficulty of debugging and editing dynamically created local functions.

Bozinovic describes these local functions in conjunction with several other ideas such as multitasking; while these other ideas are not appropriate in this context, the points Bozinovic makes about local functions are valid.

Kajiya [56] and Seeds, Arpin, and LaBarre [53] make proposals to change APL's dynamic scoping rules in ways which would alleviate the problems of variable localization and side-effect documentation. Kajiya's proposal is syntactically complicated because it is designed to support an abstract data-typing mechanism; the proposal of Seeds, Arpin, and LaBarre is overly general.

JMSL SOLUTION: ALLOW MULTIPLE ARGUMENTS AND RESULTS IN DEFINED FUNCTIONS, ALLOW LOCAL FUNCTIONS, AND ELIMINATE DYNAMIC SHADOWING.

Since JMSL has nested arrays, the solution to the problem of passing multiple parameters is obvious. Although a function header such as (R: ! Y) could be used in the case where Y is a nested array with 3 items and R is a nested array with 4 items, JMSL takes a cue from the Strand notation implementations and permits function headers such as the following:

{R1;R2;R3;R4}: ! {Y1;Y2;Y3}

This use of multiple assignment saves programming time since the programmer does not need to take any extra steps decomposing a nested input parameter into several local variables, or composing several local variables into a nested output parameter. (However, these extra steps must be taken if the number of input or output parameters is unknown or variable.) Single assignments may be mixed with multiple assignments in a function header, and the shortcut regarding multiple variable names permits the ; to be omitted since only user-defined variable names may appear in the function header. This permits function headers such as ({R1 R2}: X % {Y1}).

A vacant nested scalar ({}) may not appear within a function header since {} is not a user-defined variable name. However, it is possible that a user might pass {} to a header argument as part of the implied multiple assignment; in this case, the variable which is to be "erased" is considered undefined within the function. Inside, the function can detect this using an error handler. Symmetrically, if a variable within a multiple result is not defined within the function, {} is passed back to the calling statement (possibly causing an erasure there). Although this rule might require extra

validation checking in some functions, the rule does permit optional parameters to be passed to and from functions.

JMSL allows local functions. The control structure was presented in the previous section; since this control structure is nothing more than a mechanism for creating local arrays which will be subsequently executed, the scope rules for local functions are covered by the same scope rules which govern their surrounding functions.

JMSL does not require local variables to be declared in function headers. JMSL does not use the same dynamic scoping rule APL uses to distinguish between local variables and global variables. In fact, JMSL does not even have global variables which may be dynamically shadowed. (Shadowing occurs whenever a local variable name 'masks' the referent of the same name in a more global environment. In APL, suppose (1) F contains local variables L1 and L2, (2) F calls G, and (3) G contains local variable L1. Then L1 is shadowed but L2 is not, since L1 refers to a different value within F than within G, but L2 refers to the same value within both F and G.)

There are three reasons why APL uses global variables at all:

(1) Standard APL syntax requires one or more globals in order to pass more than two input parameters or more than one output parameter.

(2) For a large array, the overhead involved with creating a local copy of the array is excessive.

(3) Use of global variables eliminates the necessity of explicitly listing the globals as input/output parameters.

The first reason has already been disposed of by using nested arrays within function headers. The second and third reasons may be disposed of by saving a single copy of the array to be passed from the calling function on the permanent tree. This saved array may then be referred to by an absolute or relative pathname within the function. This solution may seem awkward if the array is a temporary one created dynamically, but in this case the function may be revised to accept an extra input parameter or alternatively the temporary array may be explicitly erased.

This solution also solves the problem of documenting the fact that a side effect is occurring. Side effects are automatically flagged by the presence of a relative pathname beginning with ($) or an absolute pathname beginning with (root). A function which does not contain any pathname starting with (root) or ($) does not have side effects. It is possible for a function to modify its active directory (c.f. $\Var: 0) but this side effect is automatically visible to

anyone examining the code. It is also possible for the function to create, modify, or erase an object on the permanent tree (providing, of course, it has access to the object). It is not possible for any function to modify the local variables of its calling function.

The JMSL solution to the problems of Standard APL scoping has some implications. One aspect of Standard APL scoping noted by Seeds, Arpin, and LaBarre [53] and Kajiya [56] is that information within an inner block is not accessible to an outer block. This observation is true of JMSL as well; this can be considered a positive aspect in facilitating modular programming or a negative aspect in hindering information transfer. It is difficult to predict how the revised scoping rule would affect the efficiency of a JMSL interpreter; in particular, it is difficult to know whether local functions could be created efficiently.

There are some disadvantages to the JMSL scheme:

1. There is a small amount of programming effort involved in writing a global object to permanent storage and then later erasing it. If a programmer writes all global objects to a single permanent directory, they may all be erased at once. But the possiblity exists that the programmer may forget to erase a global variable; this would linger permanently on the tree until the programmer performed housekeeping.

2. The rules regarding multiple assignment and local functions imply that JMSL is more complicated than APL. These features are optional and may be ignored by anyone except maintenance programmers.

The advantages of the JMSL scheme seem to outweigh the disadvantages:

1. Users are able to pass an arbitrary number of parameters in and out of functions in an easy-to-read fashion.

2. Users are able to conveniently define local functions and thereby facilitate workspace management.

3. Users do not need to declare all local variables in a function header.

4. It is easy to detect when side effects occur.

SUMMARY: THE 7 CHANGES PROPOSED ABOVE WOULD LEAD TO GREAT IMPROVEMENTS IN APL IN A PRODUCTION ENVIRONMENT.

The preceding seven sections have described seven major areas of Standard APL which could be improved. These points are recapitulated here:

1. Since the Standard APL character set causes problems with input/output interfacing, extensibility, and ease of programming (in terms of readability and maintainability), JMSL uses a keyword token scheme using ASCII tokens. The disadvantages of the scheme are loss of visual conciseness, tendency for users to think less symbolically, loss of typing compactness, loss of an expressive character set, and loss of ability to communicate algorithms internationally. The advantages of the scheme are ease of interfacing and editing, extensibility with regard to new primitives, and improved readability (due to a clearer symbol-meaning correspondence).

2. Since Standard APL symbols have multiple meanings and uses, and some built-in capabilities are unnecessary or inconveniently designed or missing, JMSL reflects many revisions to Standard APL capabilities. The disadvantages of the numerous revisions are increased linguistic complexity, possible loss of execution efficiency, lack of ambivalent functions, and limitations on the power of operators. The advantages of the revisions are improved efficiency of programming (due to the greater number of standard capabilities), ease of learning/maintaining programs, and increased programming power for those requiring it.

3. Since Standard APL cannot conveniently store/manipulate data of either (1) heterogeneous type and shape or (2) arbitrary depth, JMSL uses nested arrays which reflect the definitions of the grounded approach and which adapt functions found on existing nested array systems to a new notation. The disadvantages of nested arrays in JMSL are increased linguistic complexity and possible loss of execution efficiency. The advantages of nested arrays in JMSL are improved modularity of functions, ability to store related disparate information within a single array, and improvements in recursive programming.

4. Since Standard APL cannot show hierarchical relationships among named collections well, JMSL provides directories with index-by-name capabilities which may store executable arrays. The disadvantages of these directories are the need to prefix each function name with a special character, difficulty distinguishing directories and executable arrays from variables not intended as such, lack of automatic documentation of a clue to a function's purpose by its name,

and increased complexity of recursive functions. The advantages of directories are improved documentation of workspace organization, better interfacing of subsystems, and improved editing and function copying.

5. Standard APL needs ways to store data which exceeds a fixed workspace size, facilitate manipulation of permanent objects stored in workspaces, streamline certain data transfers, and allow multiple users to share data securely. Consequently, JMSL uses mechanisms to modify permanent directories rather than workspaces; they replace Standard APL system commands and the traditional component file functions. The disadvantages of the JMSL mechanisms are lack of overwrite protection, lack of a rename facility, lack of update information about a directory, and lack of audit trail information about directory components. The advantages of the JMSL mechanisms are generalizability to hierarchical machine networks, unification of syntax, and a convenient security system.

6. Since Standard APL branching is insufficiently powerful for selection and looping control structures, and since Standard APL has no mechanism for event handling, JMSL uses control structures with indentation, incremental assignment, and an event-handling mechanism. The disadvantages of these additions to JMSL are restrictions on branching, possible decline in programming style, and increased care in typing. The advantages of these additions are readable ways to express selection, looping, and event handling ideas, ease in incrementing, and ease in editing.

7. Since Standard APL leads to difficulties in undesirable side effects (due to global variables), localizing functions, localizing variables, and indicating whether a variable is local or global, JMSL allows multiple parameters within defined functions, local functions, and freedom from dynamic scoping. The disadvantages of the JMSL scheme are the need to explicitly erase temporary objects which are stored permanently and increased linguistic complexity. The advantages of the JMSL scheme are ease of passing any number of parameters in and out of functions, ease in defining local functions, freedom from declaring global variables in a function header, and ease in detecting side effects.

These features would interact in many ways. Although some of these features (such as the distinct treatment of self-recursion and co-recursion, or the restrictions on manipulating active directories) are not as "mathematically pure" as the APL philosophy might suggest, the practical utility of the definitions should make up for the more complicated definitions.

FUTURE WORK: THERE ARE MANY OTHER POSSIBLE MODIFICATIONS TO JMSL

There are a large number of directions for future work. A number of areas of research in APL are beyond the scope of this thesis but indicate other aspects of Standard APL which could be modified and incorporated into JMSL. Functional overloading, array theory, abstract data-typing, compilation, operating system interfacing, machine language interacing, full-screen management, relational database capabilities, pattern matching, alternate parsing rules, and concurrency are some of the exciting research areas in APL.

Even within the framework of JMSL presented, there exist many areas for developing system functions. These areas include editing, workspace management, documentation aids, debugging aids, date functions, and the like. Error recovery has not been described because the question of how to restart an executing function is not clear within the context of high-level control structures. Other JMSL capabilities, such as operators, seem worthy of further development activities. It may be desirable to provide many explicit error messages which do not point to the error with a caret, instead of relatively few generic error messages such as 'DOMAIN ERROR' or 'SYNTAX ERROR'. Although a named datatyping facility such as ADA and PASCAL provide may not be appropriate for an interpreted language, constraints may prove useful as a debugging tool in JMSL. (For example, X:::{0⟨i⌈⟨=30⌉;n⌈5;⌉} might indicate that X must be a 2-element nested vector whose first item must be a vector of up to 30 whole numbers and whose second item must be a nested matrix of 5 rows. Once this declaration was made, any assignment causing X to deviate from this definition would cause an error.)

It would be desirable for many fields (among them science, mathematics, business and linguistics) to integrate in an interdisciplinary manner in a single large programming environment. To some extent this has already occurred in FORTRAN and perhaps in the UNIX environment; however, more of this integration is possible than has occurred to date. To accomplish this goal, a large number of JMSL functions could be combined from many fields. These functions include scientific conversion functions, advanced mathematical functions, natural language processing functions, graphics functions, and sound functions.

A necessary development before any of these other ideas could be pursued is a demonstration that JMSL can be implemented in an efficient manner. Another equally necessary development is a demonstration that JMSL can be learned and maintained in an efficient manner.

BIBLIOGRAPHY

[1] Iverson, Kenneth. A Programming Language. John Wiley and Sons, 1962.

[2] Falkoff, A.D., Iverson, K.E., and Sussenguth, E.H. A Formal Description of System 360. IBM Systems Journal v. 4 #4, 198-262, October 1964.

[3] Pakin, Sandra. APL/360 Reference Manual. USA: Science Research Associates, Inc. 1972.

[4] APL Quote Quad, Association for Computing Machinery (journal).

[5] Draft Standard APL. APL Quote Quad v. 14 #2, complete issue, December 1983.

[6] Standards, General Arrays Debated. APL Quote Quad v. 13 #2, 3-5, December 1982.

[7] Myrna, John W. Increasing the Use of APL. APL Quote Quad v. 13 #3, 17-21, March 1983.

[8] Ruehr, Karl Fritz. A Survey of Extensions to APL. APL Quote Quad v. 13 #1, 277-314, September 1982.

[9] Crick, Michael F.C. An ASCII Notation for APL. APL Quote Quad v. 11 #1, 18-25, September 1980.

[10] Ney, Paul. APL Overstrikes, Optical Illusions, and Software Ergonomics. APL Quote Quad v. 13 #4, 7-8, June 1983.

[11] Gilman, Leonard and Rose, Allen J. APL: An Interactive Approach (2nd ed.). John Wiley & Sons, Inc., New York, 1976, p. 54.

[12] Cain, Michael. More on ASCII APL. APL Quote Quad v. 14 #1, 12, September 1983.

[13] Burroughs APL/700 Users Reference Manual (March 1977), 2-4.

[14] Haynes, Peter. More on ASCII APL. APL Quote Quad v. 13 #2, 10, December 1982.

[15] Perlis, Alan J. and Rugaber, Spencer. Programming With Idioms in APL. APL Quote Quad v. 9 #4A, 232-235, June 1979.

[16] Holmes, W. Neville. Of Noughts and IF's and Matrices--Some Comments on APLQQ[9;2;]. APL Quote Quad v. 10 #3, 7-11, March 1980.

[17] Jenkins, M.A. and Michel, Jean. Operators In An APL Containing Nested Arrays. APL Quote Quad v. 9 #2, 8-19, December 1978.

[18] Eisenberg, Murray and Peelle, Howard A. APL Learning Bugs. APL Quote Quad v. 13 #3, 11-16, March 1983.

[19] Smith, Adrian. APL, A Design Handbook for Commercial Systems. John Wiley & Sons, Inc. New York, 1982, pp. 64-89,167-173.

[20] Penfield, Paul Jr. Proposal for a Complex APL. APL Quote Quad v. 9 #4A, 47-53, June 1979.

[21] Penfield, Paul Jr. Principal Values and Branch Cuts in Complex APL. APL Quote Quad v. 12 #1, 248-256, September 1981.

[22] The APL Handbook of Techniques. IBM S320-5996.

[23] Finnish APL Association. FinnAPL Idiom Library. ISSBN 951-99397-9-2. Helsinki, 1982.

[24] Iverson, Kenneth E., Pesch, Roland, and Schueler, J. Henri. An Operator Calculus. APL Quote Quad v. 14 #4, 213-218, June 1984.

[25] Brown, J.A. In Defense of Index Origin 0. APL Quote Quad v. 9 #2, 7, December 1978.

[26] Jochman, D. Thoughts Concerning the Index Origin. APL Quote Quad v. 9 #2, 6, December 1978.

[27] Iverson, Kenneth E. The Role of Operators in APL. APL Quote Quad v. 9 # 4A, 128-133, June 1979.

[28] Brown, J.A. Evaluating Extensions to APL. APL Quote Quad v. 9 #4A, 148-155, June 1979.

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

[30] Polivka, Raymond P. The Impact of APL2 on Teaching APL. APL Quote Quad v. 14 #4, 263-269, June 1984.

[31] Lowney, Geoffrey and Perlis, Alan. Does APL Need Arbitrary Nesting? "Another Workshop Held at Minnowbrook", conference report in APL Quote Quad v. 11 #1, 4-8, September 1980.

[32] Benkard, J. Philip. Adding and Using Structure in General Arrays. APL Quote Quad v. 12 #1, 35-41, September 1981.

[33] Chastney, Philip R. The Hunting of the Snark. APL Quote Quad v. 13 #1, 72-77, Sepember 1981.

[34] Falkoff, Adin D. Semicolon-Bracket Notation: A Hidden Resource in APL. APL Quote Quad v. 13 #1, 113-116, September 1981.

[35] Sykes, Roy A. Jr. Discussion of APL2 and NARS Systems. APL Quote Quad v. 14 #1, 4-6, September 1983.

[36] Mercer, Randall. A Based System for General Arrays. APL Quote Quad v. 12 #2, 18-19, December 1981.

[37] Murray, Ronald C. Namespaces--SemiPermeable Membranes for APL Applications. APL Quote Quad v. 12 #1, 220-226, September 1981.

[38] Sharp APL Pocket Reference (release of November 1982). I.P. Sharp Associates Limited, USA.

[39] Crick, Michael C.F. Should APL be a Declining Language? APL Quote Quad v. 12 #1, 83-88, Sept. 1981.

[40] Taylor, Stephen. Function and Context. APL Quote Quad v. 13 #3, 285-289, March 1983.

[41] Taylor, Stephen and Whitney, Arthur. The One Tree (Breaking Out of the Workspace). APL Quote Quad v. 14 #4, 339-341, June 1984.

[42] Benkard, J. Philip. Nonpositional Indexing for a Relational Data Base. APL Quote Quad v. 13 #1, 32-39, September 1982.

[43] Xerox APL Language and Operations Reference Manual for Xerox 560 and Sigma 6/7/9 Computers. June 1975.

[44] Deerhake, Stephen. An APL-Based Multiple-Key File System. APL Quote Quad v. 13 #3, 77-83, March 1983.

[45] Wiland, Geoffrey. A Primitive-Function APL Keyed-File System. APL Quote Quad v. 9 #4A, 393-396, June 1979.

[46] Bozinovic, Dragan. Extending APL: What More Can A Programmer Ask For? APL Quote Quad v. 13 #1, 49-56, September 1982.

[47] Pesch, Roland. Large Arrays and Files. APL Quote Quad v. 13 #1, 249-253, September 1982.

[48] Wheeler, James G. Improved Sharing of APL Workspaces and Libraries. APL Quote Quad v. 12 #1, 327-334, September 1981.

[49] Lucas, Jim. Transparent Files in APL (A Preliminary Proposal). APL Quote Quad v. 13 #1, 201-206, September 1982.

[50] Samson, Denis. A Proposal for Control Structures in APL. APL Quote Quad v. 14 #4, 279-284. June 1984.

[51] Sniedovich, Moshe. Interactive Multilevel Definition of APL Functions. APL Quote Quad v. 13 #1, 339-345, September 1982.

[52] van Batenburg, F.H.D. New Control Structures in APL? APL Quote Quad v. 13 #2, 15-20, December 1982.

[53] Seeds, G.M., Arpin, A., and LaBarre, M. Name Scope Control in APL Defined Functions. APL Quote Quad v. 8 #4, 15-19, June 1978.

[54] Ching, Wai-Mee. Introducing Modules into APL. APL Quote Quad v. 11 #4, 12-17, June 1981.

[55] Falkoff, Adin D. and Iverson, Kenneth E. The Evolution of APL. APL Quote Quad v. 9 #1, 30-44, September 1978.

[56] Kajiya, James T. Generic Functions by Nonstandard Name Scoping in APL. APL Quote Quad v. 12 #1, 172-179, September 1981.

APPENDIX: JMSL Syntax and Semantics

JMSL SYNTAX:

JMSL syntax is described using a metalanguage similar to BNF.

Key: ::= means "is defined as"
[] means "optional"
{} means "repeated 0 or more times"
() means "repeated 1 or more times"
<|> means "choice of alternatives"
! means "vertical description; indentation of left margin syntactically significant"
& means "continued from previous line due to print width limitations"
'' means "description not easily expressed in the metalanguage"

```
Bl ::= 'a blank character space'

Indnt ::= Bl Bl Bl

Bls ::= (Bl)

SpecChar ::= <!|@|#|$|%|^|&|*|(|)|_|+|-|=|~|`|{|}|[|]|:|;|'|
             &   "|\|<|,|>|.|?|/>

DigitChar ::= <0|1|2|3|4|5|6|7|8|9>

UpperCaseChar ::= <A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
                  &   U|V|W|X|Y|Z>

LowerCaseChar ::= <a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|
                  &   u|v|w|x|y|z>

AlphChar ::= <UpperCaseChar|LowerCaseChar>

QuoteableChar ::= <AlphChar|DigitChar|SpecChar|Bl>

PrintableChar ::= <QuoteableChar|">

NonPrintableChar ::= <nul|soh|stx|etx|eot|enq|ack|bel|bs|ht|
                     &   lf|vt|ff|cr|so|si|dle|dc1|dc2|dc3|dc4|
                     &   nak|syn|etb|can|em|sub|fs|gs|rs|us|del>

LitSca ::= " <QuoteableChar|" [NonPrintableChar] "> "

LitList ::= " (<QuoteableChar|" [NonPrintableChar] ">) "

BoolSca ::= <0|1>

WholeSca ::= (DigitChar)
```

```
WholeList ::= WholeSca {Bls WholeSca}

IntSca ::= [-] WholeSca

RealSca ::= IntSca [.] [WholeSca] [e IntSca]

ComplexSca ::= RealSca [j RealSca]

NumSca ::= ComplexSca

NumList ::= NumSca {Bls NumSca}

NestedSca ::= { [Array] }

NestedList ::= { [Array] {; [Array]} }

Array ::= {[WholeList @] <LitList|NumList|NestedList>|" "}

UserVarName ::= UpperCaseChar {<AlphChar|DigitChar>}

SysName ::= LowerCaseChar {<AlphChar|DigitChar>}

SysVarName ::= <delta|prec|seed|time|colseq|angmeas|root|
             &  signal>

VarName ::= <UserVarName|SysVarName>

NiladSysFnName ::= <input|ask|enter|get|
                 &  e|pi|true|false|
                 &  z|n|new|
                 &  ascii|uc|lc|dc|sc|ac|nc|
                 &  days|mths|
                 &  rows|cols|pils|
                 &  userid>

NiladUserFnName ::= & [PathName]

NiladicFn ::= <NiladSysFnName|NiladUserFnName>

ScawiseMonadSysFnName ::= <sin|cos|tan|asin|acos|atan|
                        &  sinh|cosh|tanh|asinh|acosh|atanh|
                        &  csc|sec|cot|acsc|asec|acot|
                        &  csch|sech|coth|acsch|asech|acoth|
                        &  mon0|nnot|not|mon1|
                        &  identfn|ln|unit|mag|floor|ceil|
                        &  gammafn|fact|roll|re|im|angle|
                        &  even|notevn|odd|notodd|
                        &  prime|notprime|compos|notcompos|
                        &  round|xround|sqrt|
                        &  sign|trunc|frac|charac|mant|
                        &  cap|xcap>
```

```
AggregMonadSysFnName  ::= <sum|altsum|prod|altprod|all|any|
                           |  parity|most|least|
                           |  mean|gmean|median|mode|xmode|gmean|
                           |  var|bvar|stdev|bstdev|stderr|dev|
                           |  range|
                           |  rsr|
                           |  asc|ascm|desc|descm|unif|notunif|
                           |  case>

ScaAxisableMonadSysFnName ::= <depth|hide|enc|vac|notvac>

VecAxisableMonadSysFnName ::= <ScaAxisableMonadSysFnName|
                               |  AggregMonadSysFnName|
                               |  homog|heter|
                               |  polyrt|
                               |  capwords|
                               |  seq|xseq|
                               |  sort|xsort|shuffle|rev|
                               |  card|
                               |  #|shape|qty|
                               |  empty|notempty|
                               |  single|notsingle|
                               |  _|ravel|
                               |  disc>

MonadSysFnName ::= <ScawiseMonadSysFnName|
                    |  AggregMonadSysFnName|
                    |  ScaAxisableMonadSysFnName|
                    |  VecAxisableMonadSysFnName|
                    |  pr|wr|see|view|show|sink|
                    |  vars|
                    |  class|
                    |  dvars|evars|pvars|
                    |  len|rank|
                    |  ex|trans|delay|
                    |  count|
                    |  matinv|det|larank|utri|ltri|idmat|
                    |  sing|notsing|diag|xdiag|
                    |  aprog|gprog|
                    |  divisors|
                    |  type|fill|simp|notsimp|
                    |  ind|
                    |  set|
                    |  where|
                    |  abspath|relpath>

MonadUserFnName ::= ! [PathName]

MonadFnName ::= <MonadSysFnName|MonadUserFnName>

CmpdMonadFn ::= {<e~|v~|c~>} MonadFnName
```

```
CmpdMonadFn ::= {(e~|v~|c~)} MonadFnName

MonadFn ::= (CmpdMonadFn|
             CmpdMonadFn [ ]|
             CmpdMonadFn [ Expr ]) Bls
            'Trailing Bls required only if MonadFn ends in
             an AlphChar or DigitChar'

ScawiseDyadSysFnName ::= ((|lt|)|gt|(=|le|)=|ge|=|eq|(>|ne|
                          +|plus|-|minus|*|times|/|divide|
                          ^|power|root|log|
                          dyO|and|nimpby|dylf|
                          nimp|dyrt|neqv|xor|or|
                          nor|eqv|nxor|ndyrt|imp|
                          ndylf|impby|nand|dy1|
                          min|max|div|mod|clk|perm|comb|
                          divides)

VecAxisableDyadSysFnName ::= (eqs|neqs|subs|
                              psubs|supers|psupers|
                              seek|xseek|seekO|xseekO|
                              take|drop|rot|shift|
                              pick|keep|lose|pad|
                              in|notin|elt|notelt|is|isnot|
                              just|xjust|center|
                              ,|append)

DyadSysFnName ::= (ScawiseDyadSysFnName|
                   VecAxisableDyadSysFnName|
                   union|inter|diff|
                   merge|
                   @|reshape|rerank|redepth|
                   join|
                   base|unbase|
                   polyfn|
                   pack|unpack|
                   encode|decode|
                   matdiv|
                   fmt|
                   deal|
                   dtrans|
                   bucket|remap|
                   to|xto|
                   shed|xshed|trim|
                   may|mayr|mayw|mayx|mayrw|maywx|mayrx|mayrwx)

DyadUserFnName ::= % [PathName]

UserFnName ::= (NiladUserFnName|MonadUserFnName|DyadUserFnName)

DyadFnName ::= (DyadSysFnName|DyadUserFnName)

CmpdDyadFn ::= {(el~|er~|vl~|vr~)} ((w~|p~)
                                    CmpdMonadFn|DyadFnName)
```

```
DyadFn ::= Bls ⟨CmpdDyadFn|
         &       CmpdDyadFn [ ]|
         &       CmpdDyadFn [ Expr ]|
         &       CmpdMonadFn [ CmpdDyadFn ]⟩ Bls
         'Leading Bls required only if DyadFn begins with
          LowerCasechar and Trailing Bls required only if
          DyadFn ends with an AlphChar or DigitChar'

VarNames ::= { VarName {⟨Bls|;⟩ VarName} }

UserVarNames ::= { UserVarName {⟨Bls|;⟩ UserVarName} }

Path ::= {⟨{$} ⟨VarName \⟩|\⟩

PathName ::= ⟨Path VarName|($)⟩

PathNames ::= Path VarNames

Specification ::= [DyadSysFn] : Bls

SingleAssign ::= PathName Specification Expr

SubscrSingleAssign ::= PathName [ ExprSeq ] Specification Expr

DiscSingleAssign ::= PathName ' ExprSeq ' Specification Expr

MultAssign ::= PathNames Specification Expr

Assign ::= ⟨SingleAssign|SubscrSingleAssign|DiscSingleAssign|
         &   MultAssign⟩

Copy ::= PathNames Specification PathName \

ExprSeq ::= [Expr] {; [Expr]}

IndexableExpr ::= ⟨Const|PathName|PathNames|NiladFn|( Expr )|
                &   { ExprSeq }⟩

SubscrExpr ::= IndexableExpr [ ExprSeq ]

DiscExpr ::= IndexableExpr ' ExprSeq '

MonadExpr ::= MonadFn Expr

DyadExpr ::= Expr DyadFn Expr

Expr ::= ⟨Assign|Copy|IndexableExpr|SubscrExpr|DiscExpr|
           MonadExpr|DyadExpr⟩

Comment ::= == {PrintableChar}

SimpleStmt ::= [⟨Expr|AlterCtrlStmt⟩]

MultiStmt ::= SimpleStmt { | SimpleStmt} [Comment]
```

```
Block ::= !Stmt
          [!Block]

CondStmt ::=  {Expr ?? Bls} MultiStmt

IfTrueStmt ::= !if Bl Expr [Comment]
               !t? Bl Block
              [!f? Bl Block]

IfFalseStmt ::= !if Bl Expr [Comment]
                !f? Bl Block
               [!t? Bl Block]

OnSelecList ::= !Expr ? Bl CondStmt [Comment]
               [!Indnt Block]
               [!OnSelecList]

OnStmt ::= !on Bl Expr [Comment]
           !OnSelecList
          [!other ? Bl CondStmt [Comment]
           !Indnt Block]

AsStmt ::= !as Bl Expr [Comment]
           !Indnt Block

AtSelecList ::= !at Bl Expr [Comment]
                !Indnt Block
               [!AtSelecList]

DoStmt ::= !do Bl Block
          [!AtSelecList]

EaStmt ::= !ea Bl UserVarName : Bls Expr [Comment]
           !Indnt Block

NilFnHdr ::= [UserVarNames : Bls] &

MonadFnHdr ::= [UserVarNames : Bls] ! UserVarNames

DyadFnHdr ::= [UserVarNames : Bls] UserVarNames % UserVarNames

FnHdr ::= <NilFnHdr!MonadFnHdr!DyadFnHdr>

FdStmt ::= !fd Bl UserVarName [Comment]
           !Indnt FnHdr [Comment]
           !Indnt Block

LabelStmt ::= label Bls LitConst

JumpStmt ::= jump Bls LitConst

AlterCtrlStmt ::= <return!halt!exit!redo!JumpStmt> [Comment]

Stmt ::= <CondStmt!IfTrueStmt!IfFalseStmt!OnStmt!AsStmt!
      &   DoStmt!EaStmt!FdStmt!LabelStmt>

Program ::= 'any immediate-mode Expr that invokes a UserFnName'
```

JMSL SEMANTICS:

The description for JMSL semantics follows the guidelines below:

1. Not every term is explained in depth. It is presumed that a knowledgeable APL implementer would be able to "read between the lines". Degenerate cases (such as empty arrays) are not elucidated since there are too many of them. A knowledgeable APL programmer should be able to infer an appropriate way to handle these cases based on consistency considerations and existing implementation strategies.

2. The metalanguage used to describe JMSL syntax is used here informally to shorten some semantic descriptions; no confusion should result from this shortcut. Additional metalanguage symbols are provided to indicate various APL-like languages: J ↔ JMSL, S ↔ SHARP APL, A ↔ Standard APL, N ↔ NARS, P ↔ APL*PLUS.

3. System functions are explained using certain letters to indicate types of domains and ranges. Codes: B ↔ Boolean, I ↔ integer, R ↔ real, C ↔ complex, L ↔ literal, D directory, N ↔ nested. If multiple letters appear in an argument or result, the union of the domains is indicated. Operators are explained using X, Y, and Z since the types are less important to the concepts.

4. If a domain does not include N, the function is assumed to be pervasive. Thus if Foo is described as having syntax (L: Foo R), then when N is nested, the significance of (Foo N) is J(v~Foo N).

5. In cases where a domain is extended from numeric to literal, the conversion is done by converting literal scalars to their ordinal position in J(colseq), applying the function to the resulting integers, and converting this result back to the corresponding literal scalars.

LitSca and LitScaList

Literal constants are written similar to Standard APL constants. Nonprintable JMSL characters are included in the constant using an escape sequence delimited by ""s. This idea is consistent with the Standard APL practice of doubling 's within a literal constant, and allows definition of nonprintable characters as constants rather than as niladic functions with result.

Array

A JMSL array is similar to a SHARP APL nested array, ex-

cept for the inclusion of {} as a nested scalar. (The action of {} could be imperfectly simulated on the SHARP APL system by considering it to be equivalent to some bizarre constant such as ((1 2 1p'⍽5F').) Unlike SHARP APL, ⍎(0⍴0) and ⍎("") and ⍎(0⍴{}) are distinct types.

SysVarName

--JMSL--	--Semantics--
⟨delta⎮prec⎮seed⎮time⟩	⍕((⟨⎕CT⎮⎕PP⎮⎕RC⎮⎕TS⟩)
colseq	some permutation of ⍎(ascii); default value is ⍎(ascii); implicit parameter specifying a literal collating sequence for any function which relationally compares literal arrays
angmeas	positive real scalar; default value is ⍎(pi); implicit parameter specifying angular measure of semicircle (typically reset to 180 or 200 for degrees or grads) for any trigonomeric function
root	directory; default depends on current storage of system; constantly being modified by assignments to subdirectories
signal	literal scalar or vector; default value is ⍎(""); implicit parameter in every function; if reassigned to a nonempty value but not handled by ⍎(at) block, halts execution and is displayed.

NiladSysFnName

--JMSL--	--Semantics--
CLN: input	⍕(⎕) but prints ⍎("?") instead of ⍕(:)
L: ask	⍋(⎕)
C: enter	⍎(input) but accepts only complex scalar or vector constants
L: get	⍒(⎕INKEY)
R: ⟨e⎮pi⟩	⍋(⟨*⎮○⟩ 1)
B: ⟨true⎮false⟩	⍋(⟨0⎮1⟩)
C: z	⍎(0⍴0)
N: n	⍎(0⍴{})
N: new	directory consisting of all JMSL system variable names and their default values
L: ascii	⍕(⎕AV) but ASCII characters
L: ⟨uc⎮lc⟩	⍎(⟨"A" to "Z"⎮"a" to "z"⟩)
L: ⟨dc⎮sc⟩	⍎(⟨"0" to "9"⎮ " !@#$%^&*()_+-=~`{}[]:;'⎮\<,.?/"""⟩)

```
L: ⟨ac|nc⟩

N: days
N: mths
N: ⟨rows|cols|pils⟩
L: userid
```

```
⍎(⟨uc,lc|
  ascii lose ascii in uc,lc,dc,sc⟩)
⍎({"Monday";"Tuesday";...;"Sunday"})
⍎({"January";"February";...;"December"})
⍎(⟨¯1|¯2|1⟩)
vector whose value (e.g.
  "root\Users\Pat") is absolute path-
  name to parent of user's active dir-
  ectory; value varies from user to
  user but each user perceives his/her
  userid as constant
```

ScawiseMonadSysFnName

```
--JMSL--
C1: ⟨sin|cos|tan⟩ C2
C1: ⟨asin|acos|atan⟩ C2
C1: ⟨sinh|cosh|tanh⟩ C2
C1: ⟨asinh|acosh|atanh⟩ C2
C1: ⟨csc|sec|cot⟩ C2
C1: ⟨acsc|asec|acot⟩ C2
C1: ⟨csch|sech|coth⟩ C2
C1: ⟨acsch|asech|acoth⟩ C2
B1: ⟨mon0|nnot|not|mon1⟩ B2
CLN1: identfn CLN2
C: ⟨ln|unit|mag⟩ C
R: ⟨floor|ceil|gammafn⟩ R
N: fact I

I: roll I
R: ⟨re|im|angle⟩ C
B: [not]even C

B: [not]odd C

B: [not]prime C

B: [not]compos C

I: [x]round R

C1: sqrt C2
I: sign R
I: trunc R
R1: frac R2

I: charac R
R1: mant R
L1: [x]cap L2
```

```
--Semantics--
⍕(⟨1|2|3⟩ ∘.C2)
⍕(⟨¯1|¯2|¯3⟩ ∘.C2)
⍕(⟨5|6|7⟩ ∘.C2)
⍕(⟨¯5|¯6|¯7⟩ ∘.C2)
⍎(÷⟨1|2|3⟩ ∘.C2)
⍕(⟨¯1|¯2|¯3⟩ ∘.÷C2)
⍕(÷⟨5|6|7⟩ ∘.C2)
⍕(⟨¯5|¯6|¯7⟩ ∘.÷C2)
⍕(⟨0|B2|~B2|1⟩)
⍎(CLN2)
⍕(⟨⊗|×|⟩ C)
⍕(⟨⌊|⌈|!⟩ R)
⍕(!I) where I is a nonnegative
  integer
⍕(?I)
⍎(⟨9|10|11⟩∘.C)
1 if C is [not] even; 0 other-
  wise
1 if C is [not] odd; 0 other-
  wise
1 if C is [not] prime; 0 other-
  wise
1 if C is [not] composite; 0
  otherwise
round R up[down] to nearest
  integer
principle square root of C2
⍎(×R)
whole part of R, retaining sign
fractional part of R2, retain-
  ing sign
characteristic of R
mantissa of R
change all lower[upper] case
  letters of L2 to upper[lower]
  case
```

AggregMonadSysFnName

--JMSL--	--Semantics--
C1: ⟨sum|altsum⟩ C2	⍎(⟨+/|-/⟩ C2)
C1: ⟨prod|altprod⟩ C2	⍎(⟨×/|÷/⟩ C2)
B1: ⟨all|any|parity⟩ B2	⍎(⟨∧/|∨/|≠/⟩ B2)
RL1: ⟨most|least⟩ RL2	⍎(⟨⌈/|⌊/⟩ RL2)
C1: [g]mean C2	arithmetic[geometric] mean of C2
R1: median R2	median of R2 (arithmetic mean of middle two values if ⍎(odd #R2))
CLN1: [x]mode CLN2	most frequent scalar of CLN2 (first[last] appearing value in case of tie)
R1: [b]var R2	unbiased[biased] variance of R2
R1: [b]stdev R2	unbiased[biased] standard deviation of R2
R1: stderr R2	standard error about the mean of R2
R1: ⟨dev|range⟩ R2	⟨deviation|range⟩ of R2
C1: rsr R2	0 if ⍎(0 in R2); ⍎(1/sum 1/R2) otherwise
B: asc[m] RL	1 if each scalar of RL≤[⌽] the next scalar; 0 otherwise
B: desc[m] RL	1 if each scalar of RL≥[⌽] the next scalar; 0 otherwise
B: [not]unif RL	1[0] if each scalar of RL= the next scalar; 0[1] otherwise
L1: case L2	⍎("nlum"[1+(L2 in lc)+2×L2 in uc])

In case argument is empty, return:

sum altsum any	0
prod altprod all parity	1
most	-∞ or colseq[1] (based on argument type)
least	∞ or colseq[128] (based on argument type)
ALL OTHERS	Error

If argument is not a vector, return ⍎((#Arg)⍹AggregFn _Arg)

ScaAxisableMonadSysFnName

--JMSL--	--Semantics--
I: depth CLN	⍁(≡CLN)
N: ⟨hide|enc⟩ CLN	⍎(⟨⍉|⟨⟩ CLN)
B: [not]vac CLN	⍎({⍬} [not]in CLN)

VecAxisableMonadSysFnName

--JMSL--	--Semantics--
B: homog CLN	1 if ⍎(CLN'') exists; 0 otherwise
B: heter CLN	⍎(not homog CLN)

C1: polyrt C2	roots of polynomial with coefficients C2 in "sorted" order (real part=major key; imaginary part=minor key)
L1: capwords L2	for each of the longest possible substrings of alphabetic characters in L2, make the first character uppercase and any subsequent characters lowercase
I: ⌈x⌋seq RL	⍎(⍋⌈⍋⌋ RL)
RL1: ⌈x⌋sort RL2	⍕(RL2[⌈x⌋seq RL2])
CLN1: shuffle CLN2	CLN2 with scalars returned in random order
CLN1: rev CLN2	⍎(rev CLN2)
I: card CLN	⍕(#set CLN)
I: ⍳#⍴shape⍴ CLN	⍎(⍴⍴⍴ CLN)
I: qty CLN	⍎(prod #CLN)
B: ⌈not⌋empty CLN	⍕(⌈not⌋ 0=qty CLN)
B: ⌈not⌋single CLN	⍕(⌈not⌋ 1=qty CLN)
CLN1: ⍳,⍴ravel⍴ CLN2	⍎(⍳,⍴,⍴ CLN2)
CLN1: disc CLN2	⍕(CLN2⍳'')

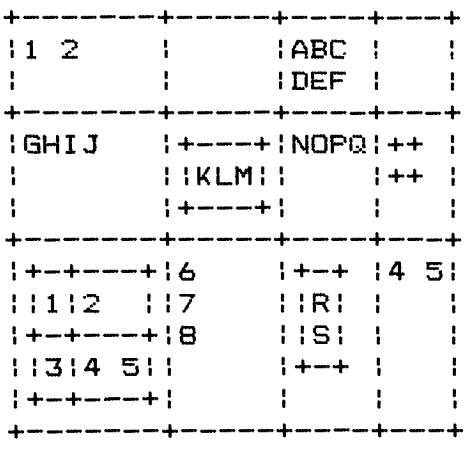
MonadSysFnName

The following five algorithms are difficult to describe but easy to grasp from examples. The first four functions return a literal matrix; the fifth returns a literal vector. The outputs emphasize various aspects of structure, type, shape, and values. Default output follows the display rules for ⍕(pr).

```
      CLN: {1 2;;2 3@"ABCDEF";0@0;"GHIJ";{"KLM"};"NOPQ";{}}
      CLN: 3 4@CLN,{2 2@{1;2;3;4 5};3 1@6 7 8;2 1@{"RS"};4 5}

       pr CLN
1 2      ABC
         DEF
GHIJ KLMNOPQ
1 2  6   R    4 5
3 4 57   S
     8
       wr CLN
(same output as pr, without automatic carriage return at end)
       see CLN
{1 2    ;         ;"ABC" ;    }
                    DEF
{"GHIJ" ;{"KLM"};"NOPQ";{} }
{{1;2  };6        ;{"R"} ;4 5}
  3;4 5  7        ;  S
         8
```

view CLN




```
--JMSL--
sink CLN
N: vars D
L: class CLN

N: ⟨del|elp⟩vars D

I: len CLN

I: rank CLN
⌈CLN1:⌋ ex L
CLN1: trans CLN2
R1: delay R2
I: count RL
R1: matinv R2
R1: ⟨det|larank⟩ R2

I1: ⟨utri|ltri|idmat⟩ I2

B: ⌈not⌋sing R
CLN1: ⌈x⌋diag CLN2

C1: ⟨a|g⟩prog C2

I: divisors R

L: type CLN
CLN1: fill CLN2
B: ⌈not⌋simp CLN
I: ind CLN
I: set CLN

I: where B
```

```
--Semantics--
does nothing; no result
⍎(D[1;1;;2] el⍕," ")
"d" if CLN is a directory, "e" if
  CLN is an executable array, "p"
  otherwise
⍎(((class D[1;1;;2])="⟨del|elp⟩")
  keep vars D)
⍎(#CLN); error if CLN is not a vec-
  tor
⍎(##CLN)
⍋(⍋L)
⍋(⍒CLN2)
⍲(⌷DL R2)
⍎((1|colseq[1]) to RL)
⍲(⌹R2)
linear algebra ⟨determinant|rank⟩
  of matrix R2
⟨upper-triangular|lower-triangular|
  identity⟩ matrix of shape
  ⍎(2⍴I2)
⍎(⌈not⌋ 0=det R)
major⌈minor⌋ diagonal scalars of
  CLN2
⟨arithmetic|geometric⟩ progression
  of C2[1] terms starting with
  C2[2] with common ⟨difference|
  ratio⟩ C2[3]
vector of all positive integers
  which evenly divide R
⍎("cln"[{0⍴0;"";0⍴{}} seek 0⍴CLN])
⍎(0⍴CLN2)
⍎(⌈not⌋ 0=depth CLN)
⍎(count #CLN)
ravel CLN and eliminate duplicate
  elements
⍎(B keep[cols] count B)
```

```
L1: ⟨abs|rel⟩path L2        convert literal vector describing
                              absolute or relative pathname
                              to ⟨absolute|relative⟩ pathname
```

CmpdMonadFn

Let MFN be an abbreviation for MonadicFnName.

```
--JMSL--        --Semantics--
Z: e~MFN X      if X is not nested, return ⍎(Z: MFN X);
                otherwise for each scalar S of X,
                            Tmp: ⍎({MFN S''})
                            (corresponding scalar of Z): Tmp
                          return Z
                Note: "e~" is the "each" operator.

Z: v~MFN X      if X is not nested, return ⍎(Z: MFN X);
                otherwise for each scalar S of X,
                            Tmp: ⍎({v~MFN S''})
                            (corresponding scalar of Z): Tmp
                          return Z
                Note: "v~" is the "pervasive" operator.

Z: c~MFN X      ⍎(((#X)@e~MFN (_X)[enc[] count qty X])'')
                Note: "c~" is the "cumulative" operator.
```

MonadFn

Let CMF be an abbreviation for CmpdMonadFn.

```
--JMSL--          --Semantics--
Z: CMF X          ⍎(CMF X)
Z: CMF[] X        ⍎((e~CMF enc[] X)'')
Z: CMF[Expr] Y    ⍎(e~CMF enc[Expr] X) if CMF is (disc);
                    ⍎((e~CMF enc[Expr] X'') otherwise
```

where the above semantic descriptions use the next two:

```
Z: enc[] X        for each scalar S of X
                    T: {S}
                    (corresponding scalar of Z): T
                  return Z

Z: enc[Expr] X    for each vector V of X along axis (Expr)
                    T: {V}
                    (corresponding scalar of Z): T
                  return Z
```

ScawiseDyadSysFnName

```
--JMSL--                                  --Semantics--
B: RL1 ⟨<|lt|<=|le⟩ RL2                   A(RL1 ⟨<|<≤|≤⟩ RL2)
R: RL1 ⟨>|gt|>=|ge⟩ RL2                   A(RL1 ⟨>|>≥|≥⟩ RL2)
B: CL1 ⟨=|eq|<>|ne⟩ CL2                   S(CL1 ⟨=|=|≠|≠⟩ CL2)
C1: C2 ⟨+|plus|-|minus|*|times⟩ C3        S(C2 ⟨+|+|-|-|×|×⟩ C3)
```

```
C1: C2/C3                                  S((C2+(C2=0)^C3=0)÷C3)
C1: C2 divide C3                           J(C2/C3)
C1: C2 ⟨^|power|root|log⟩ C3               S(C2 ⟨*|*|*÷|⍟⟩ C3)
B1: B2 ⟨dy0|and|nimpby|dylf⟩ B3            A(⟨0|B2^B3|B2)B3|B2⟩)
B1: B2 ⟨nimp|dyrt|neqv|xor|or⟩ B3          A(⟨B2<B3|B3|B2≠B3|B2≠B3|
                                             B2∨B3⟩)
B1: B2 ⟨nor|eqv|nxor|ndyrt|imp⟩ B3         A(⟨B2*B3|B2=B3|B2=B3|~B3|
                                             B2≥B3⟩)
B1: B2 ⟨ndylf|impby|nand|dy1⟩ B3           A(⟨~B2|B2≤B3|B2*B3|1⟩)
RL1: RL2 ⟨min|max⟩ RL3                     A(RL2 ⟨⌊|⌈⟩ RL3)
C1: C2 mod C3                              S(C3|C2)
C1: C2 clk C3                              S(1+C3|C2-1)
C1: C2 div C3                              S(C2÷C3+C3=0)
I1: I2 perm I3                             A((!I2)×I2!I3)
I1: I2 comb I3                             A(I2!I3)
B: C1 divides C2                           S(0=C2|C1)
```

VecAxisableDyadFn

```
--JMSL--                                   --Semantics--
B: CLN1 [n]eqs CLN2                        J([not] (set CLN1) is set CLN2)
B: CLN1 [p]subs CLN2                       J((all CLN1 in CLN2) [and (card
                                             CLN1)(card CLN2])
B: CLN1 [p]supers CLN2                     J(CLN2 [p]subs CLN1)
I: CLN1 seek CLN2                          S(CLN1⍳CLN2) but extend domain of
                                             CLN1 to arrays of any rank, re-
                                             turning J(1+#CLN1) for any sca-
                                             lar of CLN2 not in CLN1
I: CLN1 xseek CLN2                         J(CLN1 seek CLN2) but return last
                                             occurrence within CLN1 instead
                                             of first
I: CLN1 [x]seek0 CLN2                      J(CLN1 [x]seek CLN2) but return
                                             J((#CLN1)@00) for any scalar
                                             of CLN2 not in CLN1
CLN1: I ⟨take|drop⟩ CLN2                   S(I ⟨↑|↓⟩ CLN2)
CLN1: I rot CLN2                           S(I⌽CLN2)
CLN1: I shift CLN2                         J((0-#CLN2) take I drop CLN2)
CLN1: I pick CLN2                          S(I/CLN2) where / is extended to
                                             replicate
CLN1: B ⟨keep|lose⟩ CLN2                   S(⟨B|(~B)⟩ /CLN2)
CLN1: B pad CLN2                           S(B\CLN2)
B: CLN1 is[not] CLN2                       S([~] CLN1 CLN2)
CLN1: CLN2 [x]just CLN3                    J((CLN3 [x]seek CLN2)-1) shift
                                             CLN3)
CLN1: CLN2 center CLN3                     J((floor (CLN3 seek CLN2)-1)/2)
                                             shift CLN3)
CLN1: CLN2 ⟨,|append⟩ CLN3                 J(CLN2 ⟨,|,⟩ CLN3)
```

DyadSysFnName

```
--JMSL--                                   --Semantics--
CLN1: CLN2 union CLN3                      J(set (set CLN2),set CLN3)
CLN1: CLN2 ⟨inter|diff⟩ CLN3               J(((set CLN2) in CLN3 ⟨keep|lose⟩
                                             set CLN2))
```

	overwriting referents of D2
CLN1: I ⍠reshape⍠ CLN2	⍎(I ⍠⍴⍠ CLN2)
CLN1: I rerank CLN2	promote or demote rank of CLN2 to rank I
CLN1: I redepth CLN2	enclose or disclose CLN2 to depth I
CLN1: CLN2 join CLN3	⍮(CLN2,[.5] CLN3) if ⍮(#CLN2) is ⍮(#CLN2); ⍮(CLN2,[1]CLN3) otherwise
I1: I2 base I3	convert I3 to base I2 completely
I1: I2 unbase I3	⍮(sum I2*I3^((qty I2)-1) ⌿to 0)
C1: C2 polyfn C3	evaluate scalars of C3 as arguments to polynomial with coefficients C2
I1: N1 pack N2	items of N1 store sets of all possible values for corresponding items of data in N2; encode data to array whose shape matches the common shape of items of N2
N2: N1 unpack I	unpack data stored in I based on packing specifications in N1
R1: R2 ⍠encode⍠decode⍠ R3	⍋(R2 ⍠⍳⍠⊥⍠ R3)
R1: R2 matdiv R3	⍋(R2⌹R3)
R1: L fmt R2	⍎(L ⍕FMT R2)
I1: I2 deal I3	⍋(I2?I3)
CLN1: I dtrans CLN2	⍎(I⍉CLN2)
I: RL1 bucket RL2	for each row R of RL1, return the scalar ⍮(sum (R[1](=RL) and RL2(=R[2])
C1: C2 remap C3	rescale data in C3 from scale with lower and upper limits C2[1 2] to scale with lower and upper limits C2[3 4]
RL1: RL2 to RL3	⍋((RL2-1)+⍳⌈0⌊1+RL3-RL2) if RL2 and RL3 are both real scalars; extend to vectors as the array of indices generated by nested ⍮(ea) loops calculating ⍮(RL2[i] to RL3[i])
RL1: RL2 xto RL3	⍮(0-(0-RL3) to 0-RL2)
RL1: RL2 ⍠x⍠shed RL3	⍮(RL2 ⍠x⍠just RL3) but drop leading⍠trailing⍠ occurrences of RL2 insead of rotating
RL1: RL2 trim RL3	⍮(RL2 shed RL2 xshed RL3)
D may⍠r⍠⍠w⍠⍠x⍠ CLN	assign user D permission to ⍠read⍠⍠write⍠ and/or ⍠execute⍠ object CLN

CmpdDyadFn

Let CMF be an abbreviation for CmpdMonadFn, let DFN be an abbreviation for DyadFnName, and let F be either a CFM or a DFN.

```
--JMSL--          --Semantics--
Z: Y er~F X       if X is not nested, return ⍎(Z: Y F X);
                  otherwise for each scalar S of X,
                            Tmp: ⍎({Y F S''})
                              (corresponding scalar of Z): Tmp
                            return Z
                  Note: "er~" is the "each-of-right" operator.

Z: Y el~F X       if Y is not nested, return ⍎(Z: F X);
                  otherwise for each scalar S of Y,
                              Tmp: ⍎({S'' F X})
                              (corresponding scalar of Z): Tmp
                            return Z
                  Note: "el~" is the "each-of-left" operator.

Z: Y vr~F X       if X is not nested, return ⍎(Z: Y F X);
                  otherwise for each scalar S of X,
                              Tmp: ⍎({Y v~F S''})
                              (corresponding scalar of Z): Tmp
                            return Z
                  Note: "vr~" is the "pervasive-right" operator.

Z: Y vl~F X       if Y is not nested, return ⍎(Z: Y F X);
                  otherwise for each scalar S of Y,
                              Tmp: ⍎({S'' v~F Y})
                              (corresponding scalar of Z): Tmp
                            return Z
                  Note: "vl~" is the "pervasive-left" operator.

Z: Y w~CMF X      Tmp: ⍎(enc[rows] ((count 1+(qty X)-Y)+[]
                                                 count Y)-1)
                  return Z: ⍎(((#X)⍝e~CMF Tmp)'')
                  Note: "w~" is the "window" operator.

Z: Y p~CMF X      Tmp: ⍎(enc[rows] (((qty X)/Y),Y)⍝X)
                  return Z: ⍎((e~CMF Tmp)'')
                  Note: "p~" is the "partition" operator.
```

DyadFn

Let CMF be an abbreviation for CmpdMonadFn, and let CDF be an abbreviation for CmpdDyadFn.

```
--JMSL--          --Semantics--
Z: Y CDF[] X      Z: ⍎(((#Y),#X)⍝{})
                  for each scalar S in enc[] Y,
                    for each scalar T in enc[] X,
                        Tmp: ⍎({S CDF T})
                        (corresponding scalar of Z): Tmp
                  return Z

Z: Y CDF[Expr] X  Z: ⍎((Y er~CDF enc[Expr] X)'')

Z: Y CMF[CDF] X   Z: ⍎((CMF (enc[rows] Y) DF[] enc[pils] X)'')
```

VarNames and UserVarNames

{VarName1 VarName2 VarName3} means {VarName1;VarName2;VarName3} when found to the right of assignment. Similarly, {UserVarName1 UserVarName2} means {UserVarName1;UserVarName2} when found to the right of assignment.

Path and PathName

Let V1, V2, and V3 be abbreviations for three VarNames.

--JMSL--	--Semantics--
V1\V2	V1 must be a directory (a nested array of shape 1,1,N,2 for some N, such that V[1;1;;1] is a set whose items are literal vectors of VarNames and V[1;1;;2] has no vacant elements). V2 must be a defined or undefined VarName; let L be a literal vector whose value is this name V2. Return ⍎(V[1;1;V1[1;1;;1] seek0 L;2])
V1\V2: Value	⍎(V[1;1;V1[1;1;;1] seek0 L;2]: Value) but if ⍎(V1[1;1;;1] seek0 L) is 0, perform ⍎(V: V,[3] 1 1 1 1⍀{L;Value}) instead of yielding an error
V1\V2\V3	Return (V1\V2)\V3 in all contexts
$	Active directory in which user is working
⍞($)	Ancestors of active directory ($$=parent, $$$=grandparent,...⍞($)=root)
$V1	Sibling of ancestor directory
\V1	Within a function, indicates V1 of immediate scope rather than variable V1 local to the function

Specification and SingleAssign

--JMSL--	--Specification--
PathName: Value	⍋(PathName←Value)
PathName:DyadSysFn Value	⍎(PathName: PathName DyadSysFn Value)

SubscrSingleAssign and SubscrExpr

⍎(PathName[ExprSeq]) and ⍎(PathName[ExprSeq]: Expr) evaluate exactly as the corresponding APL expressions would evaluate, except for two minor points: (1) If an Expr within an ExprSeq evaluates to a negative integer less than or equal to the number of elements N along this axis, the result is

increased by N+1. (2) If ExprSeq evaluates to a nested array, the result is pervasive (so that ⍎(X[{Y}]) is evaluated as ⍎({X[Y]})).

DiscSingleAssign and DiscAssign

--JMSL--	--Semantics--
X⍠⍠	If X is nonnested, X⍠⍠ is X. Otherwise, X must be a nonempty nested array such that ⍎(0=vac X) and the contents of each scalar of X have the same type T and shape S. X⍠⍠ yields an array of type T and shape ((#X,S) containing the contents of the scalars of X.
X⍠Y⍠	Y must be an array whose rows consist of routes into X. For each row R of Y, the corresponding scalar S of the result of X⍠Y⍠ is ⍎(X[R[1];R[2];...;R[#R]]⍠⍠). If Y is nested, X⍠Y⍠ is equivalent to ⍎({X⍠(Y⍠⍠)⍠}).
X⍠Y;;Z;⍠	X⍠Y⍠⍠⍠Z⍠⍠ which evaluates as (((X⍠Y⍠)⍠⍠)⍠Z⍠)⍠⍠
X⍠⍠: Expr	⍎(X[]: hide Expr)
X⍠Y⍠: Expr	Y must be an array whose rows consist of routes into X. For each row R of Y, perform the following: ⍎(X[R[1];R[2];...;R[#R]]: hide Expr).
X⍠Y;;Z;⍠: Expr	(X⍠Y⍠⍠⍠Z⍠⍠: Expr), which evaluates as follows: Tmp1: X⍠Y⍠⍠⍠Z⍠ Tmp1⍠⍠: Expr Tmp2: X⍠Y⍠⍠ Tmp2⍠Z⍠: Tmp1 Tmp3: X⍠Y⍠ Tmp3⍠⍠: Tmp2 X⍠Y⍠: Tmp3

MultAssign

Let N be the number of VarNames in a multiple assignment of the form ⍎(Path {VarName1 VarName2 ... VarNameN}: Expr). The VarNames must be unique. If ⍎(hide Expr) is a scalar, the right side of assignment is extended to match the N VarNames on the left (e.g. ⍎(N⍴hide Expr)). If the resulting right side does not have shape N, no assignment is performed and an error is signalled. Otherwise, the right side is evaluated and the contents of each scalar is assigned to the corresponding Path VarName. (Note that the order of assignment does not matter.) The result of the multiple assignment is ⍎(N⍴hide Expr). If a scalar of the right side is {}, there

are no assignable contents; in this case, the corresponding ⌡(Path VarName) is erased.

Copy

A Copy has the form (Path VarNames: PathName \) and behaves exactly like ⌡(Path VarNames: PathName\VarNames).

MultiStmt

Each SimpleStmt comprising the MultiStmt is executed in left to right order.

Block

Each Stmt is executed in top to bottom order. If the first character of Block is a blank, a problem with indentation exists and an error is signalled.

CondStmt

The expressions in {Expr ?? Bls} are evaluated in left to right order. If an Expr evaluates to 0, no further expressions are evaluated and the rest of the CondStmt is ignored. If an Expr evaluates to a non-Boolean value, an error is signalled. If each Expr evaluates to 1, the MultiStmt following the list of expressions is executed.

IfTrueStmt and IfFalseStmt

If Expr evaluates to 1, the (t?) Block is executed (if such a block is present). If Expr evaluates to 0, the (f?) Block is executed (if such a Block is present). If Expr evaluates to a non-Boolean value, an error is signalled.

OnStmt

If any Expr in the OnSelectList matches the Expr following (on), the CondStmt following the corresponding (?) is executed and the subsequent Block (if present) is executed. If no Expr in the OnSelectList matches the Expr following (on), an error is signalled unless an (other) portion is specified, in which case the CondStmt and optional Block corresponding to (other) is executed.

AsStmt

If Expr evaluates to a non-Boolean value, an error is signalled. If Expr is 0, the Block is ignored. If Expr is 1, the Block is executed and the process is repeated until Expr is 0.

DoStmt

The Block following (do) is executed. If ⍎(signal) becomes nonempty within the block, control branches to the first (at) handler whose LitConst matches (signal); (signal) is reset to ("") and the Block of the (at) handler is executed. If no (at) handler is found for (signal), the signal is propagated to surrounding (do) blocks within the function and to any calling functions.

EaStmt

The Expr following (ea) is evaluated once; if Expr is empty, the Block is ignored. Otherwise, for each scalar in ⍎(_Expr), UserVarName is assigned to this scalar and the Block is executed. Respecification of UserVarName within Block results in an error. Upon completion of the loop, even if terminated early by an (exit) or respecification of (signal), UserVarName retains its latest value. UserVarName retains its defined or undefined value before the (ea) loop if Expr is empty.

FdStmt

A variable UserVarName (whose value is an executable array) is created local to the current function. The first line of this executable array is the FnHeader, and subsequent lines of this executable array are taken from Block. Statements are not executed in any way; a FdStmt is more a lexical assignment than an executable statement. As in other forms of assignment, any prior value of UserVarName is overwritten.

LabelStmt

Labels mark the target of branches, much like ⍎(LAB: statement). Labels must appear within blocks and must be unique to the current block.

JumpStmt

A JumpStmt transfers flow of control to the label specified by the LitConst. An error is signalled if no corresponding label exists within the same block as the JumpStmt. A JumpStmt behaves like ⍎(→LAB) but the target label may not be numeric or variable.

AlterCtrlStmt

--JMSL--	--Semantics--
return	⍎(→0); last line of function has an implicit return, as in Standard APL.
halt	⍎(→)

exit — transfers control to statement following current (do), (at), (as), or (ea) block; error if there is no lexically-surrounding (do), (at), (as), or (ea).

redo — causes execution of the innermost surrounding (do) or (at) block to cease; control is continues at the most recent (do) block; error if there is no lexically-surrounding (do) or (at).

Program

A program is the execution of a user-defined function, as in Standard APL. Typically an application would be called using a user-defined niladic function without result (e.g. &Main).

Scope of Variables:

A user variable V1 in immediate mode persists until erased (explicitly by multiple assignment or implicitly by logging off or by reassigning the active directory). A user variable V2 in a function persists until erased or until function execution terminates; such a variable is considered local to the function. System variables and functions are always available and may never be erased. Since (root) is a system variable, changes to (root) persist after the user logs of; changes to (root) are documented by the presence of an absolute or relative path. Within an executing function, a user variable V1 from immediate mode may be accessed or assigned by specifying (\V1); without the preceding (\), V1 would refer to a local variable V1. Local variables are inaccessible outside the function in which the occur, except when passed as parameters in function headers. To save a local variable or immediate variable, an assignment must be made to (root) using an absolute or relative path. The scope rules for JMSL resemble those of FORTRAN with a directory-structured COMMON area, rather than the dynamic scoping rules of Standard APL.

Function Execution:

A UserFn executes the same way as a Standard APL function. The first line of the PathName being executed is the FnHeader; if the calling syntax does not match the header syntax, an error results. Otherwise, the Block of the function is executed. Local variables are erased upon termination of the Block. If an isolated (&) or (!) or (%) appear within the Block without a subsequent pathname, the name of the currently executing function is supplied. This permits self-recursive functions which do not reference the name of the function.

Modes of Operation:

Standard APL encompasses three distinct modes: immediate mode, edit mode, and execution mode. JMSL has similar modes,

but both edit mode and execution modes are included in immediate mode.

Immediate mode in JMSL acts as in Standard APL--the system prompts the user with a six-blank indentation, the user types a line, the system executes the line (possibly returning an error message) and repeats the cycle indefinitely.

Edit mode in JMSL is provided by one or more implementation-defined functions which modify a JMSL array. For example, an implementation on a UNIX-based system might interface to the existing 'vi' editor by defining a JMSL function (Vi) which is monadic with result; the argument to (Vi) would be would be a literal vector which would evaluate to the absolute or relative pathname of an executable array, and the result of (Vi) would be a new version of this executable array. Similar JMSL functions could be written for the 'ed' and 'ex' functions on the UNIX system, and new editing functions could be created which would be suited to the special needs of JMSL. Edit mode in JMSL would be entered by executing an editing function in immediate mode.

Execution mode in JMSL is entered originally from immediate mode. A function is executed when a line typed in immediate mode contains the name of the function, or when a line of an executing function contains the name of a function.