APL procedures (user defined operators, functions and token strings)

Article *in* ACM SIGAPL APL Quote Quad · May 1986

DOI: 10.1145/22008.22034

1 author:

Rob Hodgkinson
MarketGrid Systems Pty Limited
2 PUBLICATIONS **3** CITATIONS

APL PROCEDURES
(USER DEFINED OPERATORS, FUNCTIONS AND TOKEN STRINGS)

Robert Hodgkinson
I.P.SHARP ASSOCIATES
55 Elizabeth Street
Sydney NSW 2000
Australia

ABSTRACT

This paper describes some central aspects of an APL implementation on a Hewlett Packard Minicomputer. The development of these ideas led to an elegant, consistent underlying structure for all procedures, where a procedure is defined as a structured sequence of APL expressions, instances of which are niladic functions, ambivalent functions, monadic operators and dyadic operators. Further to this idea, the introduction of two new functions (tokenise and detokenise) and a single hyperoperator (∇) gave rise to the following features;

- Ability to manipulate functions and operators as APL objects
- 'Extended Assignment' applied to all APL objects
- Ability to store preset (or initialised) values into the header of any procedure
- Make direct use of the (usually restricted) facet of tokenising and detokenising in APL to generate token strings, which may be applied by the programmer to form individual variants of *⎕FX*, *⎕CR* and/or ∇ editing.

These extensions have been superimposed upon a basic imprint of SHARP APL.

INTRODUCTION

The notion of a hyperoperator has been raised in previous papers [1-3] however, this was usually only as a passing notion and was not explored to any significant level as a means of generating and manipulating existing APL objects. The following paper describes in detail the purpose and functionality of the ∇ hyperoperator within the APL implementation on the Hewlett Packard 1000 A series minicomputers (henceforth referred to as SHARP APL/HP). The user level blueprint of this APL interpreter was SHARP APL, since a major objective of its implementation was to execute SHARP APL code without alteration.

As the implementation proceeded it became evident that the single hyperoperator approach was able to produce function arrays (in fact arrays of procedures), as well as provide the basic framework upon which 'The One Tree' [4] was designed. This paper concentrates on the language concepts which were introduced in 'The One Tree' rather than the concept of the storage and execution of these objects. For this reason, the definitions introduced in 'The One Tree' have been included as Appendix A with some minor alterations. Familiarisation with these definitions would be advised before proceeding.

All examples contained herein assume Index Origin 0 and all enclosed arrays are displayed according to the setting of the system variable *⎕PS*←¯1 ¯1 0 ¯4 in SHARP APL, which places pairs of vertical bars around each cell and separates adjacent cells by an additional two columns.

HOUSING A PROCEDURE

The premise of commonly constructed user-defined procedures (henceforth only referred to as procedures) is to incorporate a basic atom (or building block) into the interpreter with which the required framework may be constructed. In SHARP APL/HP this atom is the extended enclosed array, which is defined as an enclosed array whose cells may not only be a referent to data [5 6] but may also be a referent to either of a symbol name, primitive or procedure. Thus any APL object may be an an element of an extended enclosed array in SHARP APL/HP, primarily because all objects are based on the one common data structure.

This premise allows tokens (which are non-simple cells) to be treated as bricks in the housing of the procedure. Token strings (formed by applying the tokenise primitive, monadic ⍎, to a character string to produce a non-simple vector of tokens) become the lines of the procedure. Token strings are intangible to the APL user in other current APL implementations, since they may not be accessed or viewed, let alone manipulated or utilised. The closest perception the current APL user has of them is the character string from which they are formed before being invisibly transformed into executable procedure lines (specifically function lines in the majority of current APL implementations).

The extended enclosed array also forms the basis of the next prerequisite for the housing of the procedure, wherein the floor plan or points of entry (input parameters) and exit (result) are

defined, along with any other localised objects which may be required. The procedure header is in fact an enclosed package of objects (SHARP packages are defined in [7]). The order of the first three objects defines the result, left and right arguments respectively, although the actual use of the last two is dependent upon the intended syntactic class of the procedure.
The data representation of any procedure is the enclosed package header catenated to the tokenised lines which comprise the procedure, as evidenced in the following examples;

```
      DR1←(<⎕PACK'R'),<T'R←⍳10'
      ⍝ DR1 might be a niladic function
```

The extended semicolon notation (;;;) defined in Appendix A is now used to produce the following line of APL which is functionally equivalent to the line of APL above;

```
      DR1←(⎕PACK'R';T'R←⍳10')
```

This notation will be used where possible within the paper for convenience. Other examples of data representations might therefore be;

```
      DR2←(⎕PACK'R A B';T'R←A+B')
      DR3←(⎕PACK'R ⍺ ⍵';T'R←⍺')
```

DR3 is the data representation of a procedure that returns its left argument as its result. Conceptually, it could be either an ambivalent function, monadic operator, or dyadic operator, although the data representation for all three would be identical in this instance. If it were to be an ambivalent function then the execution would be as yet unclear since this is only a data representation and it has not yet been defined. The step of actually defining the procedure allows ⍺ to be assigned a default preset value. This topic is discussed in detail in the subsequent paragraph entitled 'UTILISING THE PROCEDURE HEADER'.
A data representation may now be converted to an object of the required syntactic class by application of ∇. The left argument specifies the required class (Appendix A) whilst the right argument specifies the data representation. In SHARP APL/HP, ∇ may be used to directly produce procedures or data. The precedence of ∇ is similar to ← in that it has long right scope, however the treatment of niladics is different; *R←NIL* returns the result of the niladic function *NIL*, whereas *R←2∇NIL* returns the data representation of the niladic function without executing it. Since ∇ has a similar precedence to ←, future extensions such as transforms [2] may be made without affecting the existing structure, should subsequent work justify their incorporation into the APL language. The single hyperoperator scheme proposed also averts the complexities introduced with multiple such objects, although it should be emphasised that no more than one hyperoperator is required to achieve the required functionality.

EXTENDED ASSIGNMENT

It should be noted that so far, the creation of a procedure has not required specification of its name. This suggests the ability to execute nameless procedures by simply;

```
      3 (3∇DR2) 6 7 8
9 10 11
```

$ allows for self reference of a nameless procedure in SHARP APL/HP, although this may be changed to Δ in light of comments in [1] and, more recently, in [8].
Extended assignment [8 10] is used to attach any object to a name as in;

```
      3 (PLUS←3∇DR2) 6 7 8
9 10 11
```

or we could assign the primitive directly;

```
      PLUS←+
```

Function assignment has been mentioned in several papers (three examples are [1 3 9]) although it was only fully exploited by Iverson and Whitney [10] wherein it could apply to any APL object (which is referred to as extended assignment). The first hurdle encountered when any object may return an assignable result is the display. It is envisaged that these objects be displayed as follows...

```
      +
+
      /
/
      MP←+.× ⋄ MP ⋄ DOT←. ⋄ DOT
+.×
.
```

although this has not yet been implemented in SHARP APL/HP. Further examples illustrate extended assignment applied to procedures by using DR3 to produce three different procedures;

```
      LHA←3∇DR3 ⍝ A Dyadic function
      SHOW 2∇LHA ⍝ SHOW in Appendix B.
|R ⍺ ⍵| ⍝ A 2 elt enclosed column vector
|R←⍺|
      3 LHA 6 ¯7
3
      CLHA←LHA ⍝ Take a copy
      SHOW 2∇CLHA
|R←⍺ CLHA ⍵|
|R←⍺|
      ×(4∇DR3) 6 ¯7 ⍝ Result is left arg
1 ¯1
      3 +(5∇DR3)× 6 ¯7 ⍝ And again
9 ¯4
```

As a last example, here is a model of the monadic EACH operator as defined in APL2 but written in SHARP APL/HP. DEF may be found in Appendix B;

```
      CR⍙EACH
|R F|
|R←F¨>|
      EACH←4∇DEF CR⍙EACH
      SUM←PLUS/ ⍝ To enhance readability
      SUM EACH 3 4 5⊃9 10 11 12⊃⍳20
|12|  |42|  |210| ⍝ 'Boxed' as per ⎕PS
```

NOTE: At this stage the reader may note that ∇ does not provide a particularly neat mechanism to write and edit procedures, as lengthy functions would become tedious and cumbersome to enter and then edit. However the real beauty of editing in

SHARP APL/HP is difficult to convey because it utilises a full screen driver via the expression ⍞←⍞. Appendix B explains the full screen mechanism and also describes a simple editor (three functions DEF EDIT and SHOW) which utilise ∇,⍕ and ⊥.

In order that this paper concentrate on the practicality and structure of procedures, these utility functions will be used in preference to typing the APL expressions as we have done so far (with the exception of EACH above, which used DEF). In this author's view, this is how APL should be used (and is currently used for applications work) so that the language provides the building blocks and the programmer the ideas for the furnishings. Having the APL run as an independent program which communicates to specific screen drivers allows for unlimited terminal tailoring without burdening the APL interpreter other than to pass it the APL commands to be executed and receive back the results.

UTILISING THE PROCEDURE HEADER

A prerequisite for 'The One Tree' is the ability to localise any collection of objects within a procedure so that they only become visible to the user when the procedure successfully begins executing. The extended naming (or underbar) notation defined in [4] allows easy setting, viewing and retrieval of these objects. Although the underbar scheme has not yet advanced from the design stage, all of the functionality and environmental advantages may be gained simply by packaging the required objects into the header of the procedure, using the notation described in this paper.

SHARP APL packages (as defined in [7]) are collections of named objects whose values were 'snapshot' at the time the package was created via the ⎕*PACK* command. The most obvious use of these initialised objects is as private localised objects which have a predefined value, as opposed to existing implementations wherein locals, with the exception of the argument(s), have no initial value. The header package is actually an extended enclosed array of objects which already exist in the workspace but are not visible outside the procedure. Therefore all that is required for the interpreter to create these objects upon procedure entry is to stack their old values and attach their names to the new values. This has the advantage of practically no execution overhead and, thanks to joint representation [5] of extended enclosed arrays, no workspace lost to the user upon procedure entry.

An illustration of the practical application of these locals was described to me by Peter Cinque [11] regarding running a full screen manager (such as AP124) in the workspace concurrently with the application. The only functions visible in the workspace (before running the application) are the full screen cover functions and the application itself. When a request is made to read a screen the data is actually stored in the header of the full screen READ function which was called, so that any other full screen commands may access the data by extracting it from the header itself using SHARP APL package commands [7]. This avoids cluttering of the visible workspace with global variables and also avoids name clashes between the full screen manager and the application.

Although the most significant use of preinitialised locals is to define private objects, thereby avoiding name conflicts, they also assist in the modular design of procedures. User-defined operators are an instance of where this technique may be applied, as various derived functions may be initialised into the header so that one may be selected and returned as the result of the operator. The syntactic class of the argument might determine the result as is the case with ⍤ in SHARP APL. *A⍤B* may represent either composition (*A* and *B* are functions) or an application of *A* on rank *B* cells (*A* is a function, *B* is data) or a 'cut' of *A* partitions of function *B* (*A* is data, *B* is a function).

A SHARP APL/APL model of such an operator would consist of only the lines to select the appropriate derived function, all three of which would be preinitialised into the header of the operator. Each derived function would contain the code to loop appropriately, executing some function *FOO* within the loop (composition would require a second function *GOO*). Thus after the operator had selected the appropriate function it need only initialise *FOO* in the header (and *GOO* for composition) with *A* or *B* before returning this new function as its result. The new function is then totally independent and therefore extended assignment applies naturally to user defined operators and their associated derived functions.

The reduction in name clashes from using initialised locals as private objects is obvious, however the more subtle advantage is that at any point in time, the only objects visible would be globals or those objects visible within the header of any procedure on the current stack. This means that procedure execution becomes self-contained and avoids 'cluttering' of the named objects within the workspace. Of course such an environment would also be tedious for large scale systems without the benefit of improved object manipulation and editing.

A sophisticated full screen editor has been developed for SHARP APL/HP by Peter Cinque [11] which allows definition of a procedure tree and simple traversing of the tree by object name, to edit or manipulate any of the localised objects, to any depth in the tree.

The nature of private objects typically addresses problems found in larger applications, making demonstration of their usefulness difficult in a paper such as this, however the following examples may help to clarify their application, if only on a small scale (Note that DEF may be found in Appendix B)

The first example is a function to check the access of a requested user. The result is 1 for ACCESS OK and ¯1 for NOT OK with an appropriate message being printed;

```
      CR⎕ACCESS ⍝ Enclosed column vector
|R NAMES ENTRY IF| ⍝ NAMES ENTRY are args
|→0 IF (⊃ENTRY)∈NAMES ⍝ DEFAULT RESULT|
|'DISALLOWED'|
|R←¯1|
      NAMES ⍝ Use as default left arg
|ROB|  |STEVE|  |ARTHUR|  |GARY|  |LAURIE|
      SHOW 2∇IF ⍝ IF used in ACCESS
|R ⍺ ⍵|
|R←⍵/⍺|
      R←1 ⍝ Set the default result in R
      ACCESS←3∇DEF CR⎕ACCESS ⍝ Package it
      )ERASE NAMES IF R ⍝ Not needed now
```

```
      ACCESS 'ROB'
1             ⍝ Default result used here
      ACCESS 'BILL'
DISALLOWED
¯1
```

R already contains a value in the function ACCESS so that it may be used as a default result, which can be particularly useful for recursive definitions, as in the following model of the Fibonacci function (FIB), where it should be noted that a default left argument is utilised as well;

```
      CR∆FIB ⍝ Enclosed column vector
|R ⍺ ⍵ IF|
|→0 IF ⍵<3 ⍝ USE DEFAULT RESULT|
|R←R,+/(-⍺)↑R←⍺ $ ⍵-1 ⍝ $ IN FOR RECURSION|

      R←0 1⍪⍺←2 ⍝ Preset result,left arg
      ⎕NC 'IF' ⍝ IF defined as in ACCESS
2
      FIB←3∇DEF CR∆FIB ⍝ Pack it
      )ERASE R ⍺ IF ⍝ globals now gone
      FIB 14 ⍝ default to sum pairs
0 1 1 2 3 5 8 13 21 34 55 89 144 233
      3 FIB 14 ⍝ now sum triples
0 1 1 2 4 7 13 24 44 81 149 274 504 927
```

ARRAYS OF PROCEDURES

Arrays of functions have been addressed in [1-3] and also their useful application [12-14] showing a genuine value in such objects. Once again the extended enclosed array provides the mechanism upon which arrays of functions have been both designed and implemented in SHARP APL/HP. It was mentioned before that the cell of an extended enclosed array could be a referent to a primitive or procedure. ∇ provides the facility to return this referent (enclose of a procedure) and also to convert the referent back to the original object (disclose of a procedure) in a manner similar to that proposed by Bernecky [1]. A procedure array may thus be formed by...

```
      PA←(0∇+),(0∇-),(0∇×),(0∇÷),0∇FIB
```

Again, the extended (;;;) notation as defined in Appendix A will be used for convenience so that the above expression is identical to the following...

```
      PA←(+;-;×;÷;FIB)
      ⍴PA
5
      3 (1∇PA[2]) 5 6 7
15 18 21
```

Operators behave in a similar fashion although their useful application remains to be explored...

```
      OPLIST←(⍤;⍥;¨;.)
      3 4 5 (1∇PA[0])(1∇OPLIST[3])×1 2 1
16
```

The significant difference here from Bernecky's approach is that execution of a procedure array is not implicit. ∇ is used to explicitly transform an element of a function array back to an executable procedure or primitive. This explicit approach has been preferred to the implicit approach as it allows the more effective use of function arrays by operators. An illustration of this would be the way in which a function array might be used as an argument to an operator, which would return an explicit derived function that would then apply to data. This contrasts significantly with the notion of 'Functional Application' [1] wherein execution is implied and automatic if the user catenates the data onto the function array and does not assign the final object.

By merging the functionality of procedure arrays with the advantages of initialised local objects, we may write simple efficient models of any procedure required. One instance of this would be to write a monadic operator which applies a function array to its arguments on rank 0 as in the following;

```
      3 4 5 (+;-;×)APPLYE 6 7 8
9 ¯3 40
```

However a more useful approach might be to apply the function array in an outer product fashion as follows;

```
      3 4 5 (+;-;×)APPLYO 6 7 8
 9 11 13
¯3 ¯3 ¯3
18 28 40
```

which can easily be used to generate arithmetic tables;

```
      1 2 3 (∘.+;∘.-;∘.×)APPLYO 1 2 3
 2  3  4
 4  5  6
 5  6  7

 0 ¯1 ¯2
 1  0 ¯1
 2  1  0

 1  2  3
 2  4  6
 3  6  9
```

The definition of both of these operators is given in Appendix C. Another illustration of their practicality is evident from the modelling of control structures proposed by Eusebi [13] an example of which follows...

```
      3 4 5 (PLUS;-;×)CASE 0 1 0⊢6 7 8
¯3 ¯3 ¯3
      3 4 5 (PLUS;-;×)CASE 3 0 1⊢6 7 8
 9 10 11       ⍝ multiples also, due to
 9 10 11       ⍝ the test by / in CASE
 9 10 11
18 28 40
```

Again, the definition is given in Appendix C, along with the following example for an Inverse operator...

```
      10 ⍟INVERSE 100
2             ⍝ ⍟ is inverse of ⍟
```

ATTRIBUTES

The previous example shows how easily an Inverse operator may be implemented under this scheme. An alternative approach was suggested by Garry Gould

[15] wherein no centralised table of functions and their inverse is stored, but rather a SETINVERSE dyadic operator is applied to the function and its inverse, producing a new derived function in which the inverse is stored in the header as an attribute. A monadic operator, GETINVERSE, may then be applied to the function to extract and return the actual inverse function when required.
This approach introduces the notion of the procedure header also catering to the effective implementation of function attributes. Extensive work has been recently undertaken in the area of these attributes (or parts) by Iverson, Pesch and Schueler [16 and related papers mentioned therein]. The seven parts defined in this paper (rank, coherence, shape, surrogate, inverse, dyad, derivative and identity) could be stored as initialised locals. However it is the belief of this author that rank is more easily utilised by the direct application of the rank operator. The remaining six attributes could then be stored within the function header. Models of such extensions have already been experimented with and the simplicity of implementing them under SHARP APL/HP has been more than encouraging. It is hoped to pursue this particular area in much greater detail in a subsequent paper, in order to give a more detailed treatment of this area of work.

SUMMARY

The objective of this paper is to illustrate how a simple, unifying, centralised data type enhances the power and versatility of APL. This basic atom, or building block, becomes the basis of tokenised lines and, consequently, procedures. The system QUAD functions ⎕*FX* and ⎕*CR* may be discarded (although still supported) in preference for an explicit and less mysterious mechanism for procedure construction and editing. This same tool may be utilised by the programmer for individual tailoring, should a more sophisticated and elaborate editor be required. The procedure header under this scheme provides an improved programming environment, whilst proving also to be a practical solution to the storage and manipulation of function attributes. When extended assignment is combined with these extended enclosed arrays a simple and efficient implementation of arrays of procedures results. Aside from ∇ and token strings, none of the language concepts described herein are novel. The point to stress is that the use of ∇ and extended enclosed arrays provides APL with a common denominator with which to treat all of these constructs.

FUTURE WORK

Display in SHARP APL/HP will shortly be extended to allow printing of all of these objects. It is envisaged that follow up papers will be written, particularly in the areas of Function Composition and Attributes where there is much to be resolved. So much material was covered in this paper that I also hope to write more detailed presentations for various subtopics. The separation of APL from the terminal dependent driver is one such area, as also is the internal structure and implementation of the extended enclosed array and yet another is the practical use of operators in writing APL applications. One last direction is to further investigate and refine the notation used to define operators as compared to APL2 for example, wherein the programmer may only define the derived function produced by the operator as opposed to the operator itself.

ACKNOWLEDGEMENTS

The ideas incorporating the central design of SHARP APL/HP are those of Arthur Whitney, who has an uncanny ability to resolve such things as enclosed arrays, procedures and arrays of procedures down to the lowest common denominator...in this case, the extended enclosed array. Thanks are also due to Laurie Gellatly, Garry Gould and Peter Cinque for their continuing work in this area and useful suggestions. Also to Stephen Taylor without whose introductory work this paper would have been far more difficult to write. Finally, to the APL 86 referees for their constructive comments and ideas.

REFERENCES

1. R.Bernecky, "Function Arrays", APL 84 Proceedings, pp 53-56.
2. J.P.Benkard, "Syntactic Experiments with Arrays of Functions and Operators", APL 84 Proceedings, pp 41-51.
3. J.A.Brown, "Function Assignment and Arrays of Functions", APL 84 Proceedings, pp 81-84.
4. S.J.Taylor and A.T.Whitney, "The One Tree", APL 84 Proceedings.
5. R.Bernecky and K.E.Iverson, "Operators and Enclosed Arrays", (I.P.Sharp, Proceedings of the User Meeting, 1980).
6. R.Bernecky, "Representations for Enclosed Arrays", APL 81 Proceedings, pp 42-46.
7. P.C.Berry, SHARP APL Reference Manual, I.P.Sharp Associates, 1981.
8. K.E.Iverson, "A Dictionary of the APL Language", I.P.Sharp Associates, 1985 Draft.
9. K.E.Iverson and P.K.Wooster, "A Function Definition Operator", APL 81 Proceedings.
10. K.E.Iverson and A.T.Whitney, "Practical Uses of a Model of APL", APL 82 Proceedings.
11. P.G.Cinque, personal communications.
12. J.P.Benkard, "Structural Experiments with Arrays of Functions", APL 85 Proceedings, pp 166-172.
13. E.Eusebi, "Operators for program control", APL 85 Proceedings, pp 181-189.
14. E.Eusebi, "Operators for Recursion", APL 85 Proceedings, pp 190-194.
15. G.Gould, personal communications.
16. K.E.Iverson, R.Pesch, J.H.Schueler, "An Operator Calculus", APL 84 Proceedings, pp 213-218.

APPENDIX A - DEFINITIONS ADAPTED FROM 'THE ONE TREE'

APL objects are of 5 syntactic types numbered here 2 to 6:

2 Data	2 3 4
3 Ambivalent Function	-
4 Monadic Operator	/
5 Dyadic Operator	.
6 Niladic Function	⎕*AI*

Symbols can name any APL object. The object is attached by Extended Assignment using ←.

Punctuation is provided by ()[]→←⍎⍕◇;:⍝

Extended Enclosed Arrays are comprised not only of referents to data (as in SHARP APL) but also of referents to punctuation, symbol names and objects.

Monadic ⍑ is the tokenise primitive. It takes a character string and produces a vector of objects, symbols and punctuation (extended enclosed array); that is, a tokenstring. It reports a domain error if there is an error in punctuation (eg unbalanced parentheses). For example, the result of ⍑'*KING*←''*RICHARD* '',⍕3' is a non simple vector of length 6. The first enclosed cell (or referent) is a symbol name, the second is punctuation, the third is a character string, the fourth and fifth are primitives whilst the last is a numeric scalar. Note that there are far more executable tokenstrings than there are executable character strings. Every executable character string can be converted to an executable tokenstring via ⍑, but (<2 3⍴⍳6),(⍑'+'),⍑'*MAT*' is an example of an executable tokenstring which (currently) has no character string equivalent. The major power here comes from the ability to place any object in a tokenstring.

A Procedure is any user-defined object of type 3,4,5 or 6. Its data representation is:
package⊃tknstring⊃tknstring⊃...⊃><tknstring

⍬ (jot) is a non simple scalar; but disclosing it generates a result error; that is to say, >⍬ ↔ ⊣⍬. This is useful in packages to represent symbols with no value.

∇ (del) is the one Hyperoperator for syntax class changes. It behaves syntactically like assignment, that is, it has long right scope, except that the result of 2∇*NIL* returns the data representation of the niladic function *NIL*, not the result of executing it. ∇ has seven possible left arguments:

0∇*X* encloses any object X. For example, 0∇+ produces a scalar referent to a function. Note that enclose (<⍵) is a special case of 0∇⍵.

1∇*X* discloses the scalar referent X (A symbol or punctuation returns a result error). Thus the result of this can be of any syntactic type. Disclose (>⍵) is again a special case of 1∇⍵, where ⍵ is an enclosed scalar.

2∇*X* yields the data representation of procedure x

3∇*X* yields an ambivalent function from data representation x

4∇*X* yields a monadic operator from data representation x

5∇*X* yields a dyadic operator from data representation x

6∇*X* yields a niladic function from data representation x

At the APL user level, packages are equivalent to SHARP APL packages [6], however their representation is far more obvious. A package is an extended enclosed array consisting of alternating referents to symbol names and data. Packages can be formed via a packaging primitive [6] or using ⍑ and ∇. For example ... *PKG*←(⍑'*FOO*'),⍬,(⍑'*A*'),0∇*GOO* will return a package containing two named objects, FOO and A, in which FOO is undefined and the value of GOO is attached to the symbol A.

Procedures are formed by packaging the relevant names, tokenising the relevant expressions, catenating them together and then applying ∇. For example...

```
FOO←3∇P⊃L1⊃L2⊃L3⊃><L4 ⍝SHARP APL
FOO←3∇P L1 L2 L3 L4  ⍝Strand not'n
```

where, since FOO is an ambivalent function (3∇), *P* is a package with at least three variables (result, left argument, right argument followed by any locals) and L1, L2, L3 and L4 are four tokenstrings: the four lines of the function. As another example, if we have the rank operator, then ⍑ on vectors is a close parallel to ⎕*FX*, except that the package (header line) is, appropriately, separate. The form and execution of all procedures is the same irrespective of syntactic class, except for argument passing. The first name in the package is the result name. Thereafter, for example in a function, the next symbol is the left argument; then the right argument; then the locals. Note that as this is a package it is necessary for all these symbols to have initial values, though they may be undefined; that is, represented by jot. The values of argument symbols are overwritten by the parser but the rest survive, yielding default results, default left argument (in the case of an ambivalent function called monadically) plus the benefits of initialised locals.

Extended semi colon notation (;;;) has been implemented to allow a shorthand for much of this work. In general, the expression (EXP1;EXP2;A;;⍟) is analogous to (0∇*EXP*1),(0∇*EXP*2),(0∇*A*),⍬,0∇⍟. This notation is already in common use within several APL dialects (SHARP APL included) within the right hand argument to ⎕*FMT* [6], where the result, although unassignable, is implicitly a string of data objects. The definition of this notation in terms of extended enclosed arrays serves also to simplify the definition of ⎕*FMT* as its argument is now merely an enclosed array. An attempt to format, by ⎕*FMT*, an extended enclosed array consisting of other than data referents produces 'DOMAIN ERROR'.

APPENDIX B - FULLSCREEN EDITING TOOLS.

SHARP APL/HP has been implemented in such a way that all the Input and Output mechanisms have been divorced from the language core and are written into 'Screen Driver' programs which communicate with the APL program core. This allows APL to communicate with any device and also take full advantage of terminal attributes which may be available within that device. A basic teletype (TTY) driver allows APL to communicate with any teletype terminal but with no special features; ie only character input, output and translation are performed by the driver with results passing back and forward to the APL program. More advanced drivers (eg For Human Designed Systems Concept terminals) are also available which make use of such features as Line Transmission (allows the programmer to (re)enter any line in the screen's memory) and Full Screen editing (allows local editing on the screen then using a SEND key to transmit the screen contents to the driver). The full screen interface with the APL is illustrated by the following expression....

```
A←⍞⍝⍞←3 4⍴'WAKEREADBOOK'
```

Here, the character array (rank 2) signals full screen output (screen is cleared before printing) then the system waits on ⍞ input, at which time the screen contents are passed back to the APL and assigned to the variable A. Trailing blanks on the screen are also suppressed. Were the character array being output empty (eg Shape of 0 0) then full screen mode would still be triggered, however the contents of the screen would not be cleared first, allowing APL expressions to be edited on the screen and placed directly into procedures. To facilitate editing, the following three procedures are used in this paper;

```
      ⎕CR'SHOW'
R←SHOW ⍵;HDR
⍝ *** SHOW: CONVERTS A DATA REPRESENTATION
⍝ ... INTO A DISPLAYABLE REPRESENTATION.
⍝ ***   ⍵: ENCLOSED VECTOR. IT IS THE DATA
⍝ ... REPRESENT'N PRODUCED BY 2∇PROCEDURE
⍝ ***   R: ENCLOSED COLUMN VECTOR WHERE
⍝ ... EACH CELL IS SIMILAR TO A ROW OF ⎕CR
⍵←,⍵ ⍝ Ravel the representation
HDR←1↓(,HDR≠' ')/,' ',HDR←⎕PNAMES>⍵[0]
R←↑HDR⊃⍳¨>1↓⍵ ⍝ Must detokenise the lines

      ⎕CR'EDIT'
R←EDIT ⍵;⎕PS
⍝ *** RESULT: NEW DISPLAYABLE REPRESENT'N
⍝ ... AFTER FULL SCREEN EDITING.
⍝ ***       ⍵: OLD DISPAYABLE REPRESENT'N
⍝ ... BEFORE FULL SCREEN EDITING.
⎕PS←¯1 ¯1 0 1 ⍝ Turn off boxing for ⍞
R←⍞⍝⍞←⍵ ⍝ R now contains a char matrix.
R←<⍤1 R ⍝ Enclose each row
R←↑R ⍝ Result is a column vector.

      ⎕CR'DEF'
R←DEF ⍵;HDR
⍝ *** DEF: CONVERT DISPLAYABLE REPRESENT'N
⍝ ... INTO TOKENISED FORM READY FOR ∇.
⍝ ***   ⍵: ENCLOSED COLUMN VECTOR. _IRST
⍝ ... CELL IS THE PACKAGE NAMES AND
⍝ ... THE REMAINDER ARE THE LINES, EACH
⍝ ... CELL BEING AN ENCLOSED CHAR VECTOR.
⍝ ***   R: DATA REPRESENT'N READY FOR ∇
⍝ ... FIRST CELL IS AN ENCLOSED PACKAGE
⍝ ... OF THE HEADER, THE REMAINDER ARE THE
⍝ ... ENCLOSED TOKENISED LINES.
⍵←,⍵ ⍝ Ravel the representation
HDR←⎕PACK >⍵[0] ⍝ Package the header vars
R←HDR⊃⍳¨>1↓⍵ ⍝ Attach tokenised lines
```

APPENDIX C - REPRESENTATIONS OF PROCEDURES USED.

```
      SHOW 2∇APPLYE
|R F CRΔAPPLYE DEF| ⍝ CRΔAPPLYE is below
|⍝ *** A MONADIC OPERATOR WHOSE FUNCTION|
|⍝ ... ARG IS F. R IS PRESET TO ⍳0, READY|
|⍝ ... FOR CRΔAPPLYE TO BE DEFINED.|
|⍝ *** PRESET LOCALS: CRΔAPPLY-REPRSNT'N|
|⍝ ... OF THE FUNCTION RETURNED BY THIS|
|⍝ ... OPERATOR. DEF-THE PROCEDURE DEF'N|
|⍝ ... UTILITY AS DEFINED IN THE APL86|
|⍝ ... PAPER, APPENDIX B.|
|⍝ *** NOTE: F 'EXISTS' IN APPLYE, SO,|
|⍝ ... WILL BE AUTOMATICALLY PACKAGED BY|
|⍝ ... DEF INTO THE RESULT FUNCTION.|
|R←3∇DEF CRΔAPPLYE|

      CRΔAPPLYE ⍝ Col vector represent'n
|R ⍺ ⍵ F SF| ⍝ F set within APPLYE
|⍝ *** APPLY FUNCTION ARRAY F OVER ⍺,⍵.|
|⍝ ... PRESET LOCALS: F(FNARRAY),R(⍳0).|
|⍝|
|⍺←,⍺⍝⍵←,⍵ ⍝ Ravel the data arguments|
|SF←⍴F ⍝ Save the shape of the FNARRAY|
|F←,F ⍝ Now ravel the FNARRAY.|
|⍝ *** LOOP TO CALL F MONADICALLY OR|
|⍝ ... DYADICALLY DEPENDING UPON ⎕NC '⍺'.|
|LP:|
|R←R,<⍤(4×0=⎕NC'⍺')↓'⍺[0](1∇''''⍴F)⍵[0]'|
|⍺←1↓⍺⍝⍵←1↓⍵ ⍝ Drop completed data cells|
|→(0∈⍴F←1↓F)↓LP ⍝ Continue if more to do.|
|R←>SF⍴R|

      SHOW 2∇APPLYO
|R F CRΔAPPLYO DEF| ⍝ CRΔAPPLYO is below
|⍝ *** THIS OPERATOR SAME AS 'APPLYE'|
|⍝ ... EXCEPT FOR THE PRESET FUNCTION|
|⍝ ... REPRSNT'N. SEE 'APPLYE' FOR INFO.|
|R←3∇DEF CRΔAPPLYO|

      CRΔAPPLYO
|R ⍺ ⍵ F SF|
|⍝ *** APPLY FUNCTION ARRAY OVER ARGS ⍺,⍵|
|⍝ ... PRESET LOCALS: F(FNARRY),R(⍳0).|
|⍝|
|SF←⍴F ⍝ Save shape of the FNARRAY|
|F←,F ⍝ Now ravel it|
|⍝ *** LOOP TO CALL F MONADICALLY OR|
|⍝ ... DYADICALLY DEPENDING UPON ⎕NC '⍺'|
|LP:R←R,<⍤(0=⎕NC'⍺')↓'⍺(1∇''''⍴F)⍵'|
|→(0∈⍴F←1↓F)↓LP ⍝ Test for end|
|R←>SF⍴R ⍝ Reshape result.|
```

```
      SHOW 2∇CASE
|R F G CR∆CASE DEF| ⍝ CR∆CASE given below
|⍝ *** DYADIC OPERATOR ; FUNCTION ARGS|
|⍝ ... IN F,G.  SIMILAR TO 'APPLYE'; BOTH|
|⍝ ... F,G ARE PACKAGED INTO THE RESULT.|
|⍝|
|R←3∇DEF CR∆CASE|

      CR∆CASE
|R ⍺ ⍵ F G|
|⍝ *** FUNCTION TO EXECUTE FN ARRAY F|
|⍝ ... ACCORDING TO REPLICATION VECTOR G|
|⍝ ... ⍺,⍵ ARE THE DATA ARGUMENTS|


|⍝ ... PRESET LOCALS: F,G WITHIN CASE.|
|⍝|
|→((0∈⍴F)∨0=+/G)⍴0 ⍝ If nothing to do.|
|F←G/F ⍝ Replicate the function array.|
|⍝|
|⍝ *** NOTE: NEXT LINE RUNS 1ST FUNCTION|
|⍝ ... ONCE TO SET TRAILING RESULT SHAPE|
|R←0⌿(¯2↑1 1,⍴R)⍴R←⍺(1∇''⍴F)⍵|
|⍝|
|⍝ *** LOOP TO CALL FUNCTION IN ARRAY F|
|⍝ ... WITH ARGUMENTS ⍺,⍵.|
|LP:R←R,⍺(1∇''⍴F)⍵ ⍝ Run next function|
|→(0∈⍴F←1↓F)↓LP ⍝ Test for end|


      SHOW 2∇INVERSE
|R F FNS FNI|
|⍝ *** MONADIC OPERATOR: RETURNS INVERSE|
|⍝ ... OF F OR SIGNALS DOMAIN ERROR.|
|⍝|
|⍝ *** SET UP FUNCTIONS AND INVERSES|
|⍝ ... NOTE: FNS,FNI COULD BE PRESET.|
|FNS←(⍟;⍣;+;-;÷;⌽;⍉;⊢;⊣;<;>;⌹;~;⊤;⊥)|
|FNI←(⍣;⍟;+;-;÷;⌽;⍉;⊣;⊢;>;<;⌹;~;⊥;⊤)|
|'NO INVERSE' ⎕SIGNAL (~(F←0∇F)∈FN)/11|
|⍝ *** FIND F IN FNS WITH ⍳, THEN|
|⍝ ... THEN DO PROCEDURAL DISCLOSE (1∇)|
|⍝ ... OF CORRESPONDING INVERSE IN FNI|
|R←1∇FNI[FNS⍳F]|
```