APL Extensions - A User's View

Maurice H. Jordan

British Airways
Information Systems Support(S42)
PO Box 10-Heathrow Airport
Hounslow Middlesex TW6 2JA
England

ABSTRACT

There has been much heated argument about extensions in APL. This paper reflects 5 years' experience with one brand of extensions (STSC's nested array system). Useful and irritating features are discussed.

Facilities available are compared with other implementations - APL2, Dyalog, and IPSA. Topics covered include event handling, file systems, strand notation, indexing, the each dual and rank operators, and interfaces to other languages. The paper is illustrated with examples drawn from code produced internally, and from VECTOR competitions.

INTRODUCTION

British Airways have been using APL heavily since 1982, both for information centre work and for airline planning models written by the OR department. STSC's nested arrays were introduced in the middle of that year, and we have been using them heavily ever since. Most of our APL programmers have never experienced any other implementation of APL, and would be distinctly uneasy if they found themselves working in an implementation without nested arrays.

Although nested arrays represent a considerable advance over standard APL, we are not always entirely happy with them. This paper discusses how we use nested arrays, and looks at alternative ways of achieving similar results.

The viewpoint taken is not a theoretical one, but a pragmatic one. APL as a tool with which we have to produce results. It is easier use of .the tool that interests us, rather than the precise semantics of the tool itself.

BA BACKGROUND

We are the world's largest international airline in terms of passengers carried. We also operate domestic services within the UK - about 20% of our passengers are carried on these services. Our network is diverse and sometimes complicated and covers a range of products from supersonic Concorde to "commuter" services in the Scottish Highlands and Islands. We serve North and South America, Europe, Africa, the Middle East, the Indian sub-continent, the Far East and Australia-New Zealand.

Like all major airlines, we are heavily dependent on computers for our day-to-day operation. We have over 600 program development staff. The systems we develop are usually large, and whatever the technology used, we always seem to get to the edges of it very quickly - whether it is size of data, complexity, number of users, or their geographic spread.

NARS and NAPS

There is considerable confusion about what is included in STSC's nested array system. Originally, there was NARS, the Nested Array Research System [Cheney81]. This included many extensions to the language, but was never marketed. Instead a core of the extensions was recoded in assembler, and added to the 1140 in-house APL*Plus extensions product.

The extensions in this product are:

The pre-1982 APL*Plus extensions:

- Replicate and statement separator
- Error handling
- Shared file system
- Commercial formatting (⎕FMT)
- Various extra system functions and variables

to which were added nested arrays and language extensions to deal with them:

- Strand notation and strand notation assignment
- Enclose and disclose (⊂ω ⊃ω)
- Partitioned enclose and pick (α⊂ω α⊃ω)
- Mix and split (↑ω ↓ω)
- Match and depth (α≡ω ≡ω)

Redefinition of dyadic ⍳ and ∊ to use match
Type (∊⍵)
Each operator (f¨)
Extension to the domain of operators to include user-defined functions
Scatter point indexing and scatter point indexed assignment
Scalar functions operate pervasively - i.e at all levels of nested arguments

These extensions allowed the generalisation of arrays to be arrays of arrays, with heterogeneous (mixed character and numeric) simple arrays as a limiting case.

More recently, ambivalent functions (dyadic functions which can be called monadically) and support for compiled functions have been added.

At BA, we have christened this product NAPS (the Nested Array Production System) to try to avoid confusion with the original research system NARS.

The NAPS extensions are described in the APL*Plus Extensions Manual [APL*Plus 1985]

RELATED IMPLEMENTATIONS

APL2 from IBM is derived from the same theory (see for example [More82]), and is very similar. However, some primitives have different meanings, and some additional extensions have been implemented, such as extensions to the functions in the domain of axis and user-defined operators. See the APL2 Language Reference Manual [APL2 1985] for a full description.

Dyalog APL is based on the nested arrays research system NARS, but has included rather more of the original than STSC's NAPS. Recently, user-defined operators and function assignment have been added. Error handling facilities are based on the Sharp implementation. The product is described in the Dyalog APL User Guide [Dyalog85].

OTHER IMPLEMENTATIONS

IPSA have also extended their APL data structures to allow generalised arrays, but have done so from a different theoretical viewpoint. Differences start with the result of enclosing a scalar.

A Dictionary of APL [Iverson86] has been used as the guide to the facilities here, although many of the features described are not yet implemented.

THE TWO DIRECTIONS

Although much time at APL conferences is devoted to proponents of the two basic directions arguing with each other, not a lot of listening seems to go on. There is little sign of any cross-fertilisation, although in the papers at APL86 there were a few APL2 papers prepared to show how the IPSA rank operator could be implemented in APL2 (e.g. [Graham 86]), and Rob Hodgkinson's paper on SHARP APL/HP drew on some ideas from APL2 [Hodgkinson 86].

Not only is it difficult to find papers drawing on ideas from both directions, it is difficult to find people with experience of using both systems seriously. The dialects are starting to develop into languages in their own right, requiring their own patterns of thought. Those who are fluent in one dialect can experience difficulty with the other. The author's experience is almost exclusively with NAPS. He would be pleased to learn through his mistakes with the IPSA notation.

It is interesting to observe a fundamental difference between the Sharp and APL2/STSC/Dyalog approach. When the respective systems are being described, presentations on the APL2 approach always seem to start with a discussion of the data representations; those from IPSA always seem to start with a discussion of notation.

The job of the professional programmer is largely to produce systems to manipulate data - if you are allowed to choose a convenient representation, you can make the job very easy for yourself.

The more flexibility you have in the ways you can represent and manipulate your data, the more chance you have of choosing a good representation. While notation and the consistency of the language are important, flexibility in data structures is important in getting things done. Perhaps this accounts for the continuing popularity of many other languages.

USE OF EXTENSIONS AT BA

By far the most useful of the extensions is the shared file system. This is used for all APL data, and has enabled us to do many things which would not have been possible using native IBM files under CMS.

Our heavy use of this file system precludes any migration to APL2 (unless we bought on of the products available to simulate the file system under APL2). But this is not a contentious issue - nearly all suppliers of APL (excluding IBM) offer such a file system - so I will not dwell on it here.

The next most important extension in our use of APL is the relaxation in the data structures available. This has enabled us to develop a much better programming style than would have been possible - we have data structures to represent screens and files which do away with the necessity to use globals. This enables a much more modular approach to be taken to program design. The language extensions then allow these new data structures to be manipulated easily. The each operator and its ability to take user-defined functions as arguments is particularly important here. Together these extensions often allow very fast development of solutions to tricky problems as viewed from standard APL. This will be discussed later, taking some examples drawn from VECTOR competitions.

Error handling comes next in importance, mainly allowing us to provide more user-friendly interfaces. In full screen applications, we can signal input errors though ⎕ERROR and have them trapped and presented to the user in friendly form.

The numerous other enhancements, from speedups to primitives through replicate and diamond to extra system functions also play a part in improving the programming environment, but I mention them mainly for completeness.

Finally the recent enhancements provided with compiler support have enabled us to improve the run-time and response-time characteristics of many of our APL systems. Further development here should allow us to extend the domain of APL.

Index origin

A house standard is that ⎕IO is 1. In any examples where ⎕IO is not defined, the reader may assume a value of 1.

DATA REPRESENTATION

Apart from allowing more flexibility in how data can be represented and manipulated, extended data structures allow the bundling of associated data for passing to procedures and files. One important use is in file design - data of different representations can be stored within the same component. This greatly simplifies the file design process if all data which normally changes together is held in the same component - the programmer does not have to worry about system crashes during updates.

Quite complex single variables can be used to hold all the properties of full-screen panels

```
 e.g.Fields on screen, their positions and
      attributes
     Associated APL variables
     Functions to translate contents of APL
      variables to text on screen
     Functions to translate and validate input
      from screen
     Position of screen window(s) if scrolling in
      operation
(panel definition variables)
```

or pointers and other information about files

```
 e.g.Fields on file
     Corresponding components
     How data is stored (sparse or replicated)
     Is field keyed for look-ups and where key is
     Properties of field for output
(file control blocks).
```

These can easily be passed as arguments to functions, and a modular approach to programming without globals becomes possible. This makes recursion in handling screens and simultaneous handling of several files much neater than otherwise is possible.

STRAND NOTATION

Using standard APL, one is limited to two explicit arguments to user-defined functions. Although as time goes on, it becomes easier to define functions so that they only have two arguments, it is not always possible.

Our main use of strand notation is to give functions more than 2 arguments, and let them return more than one result. (Sometimes the overall effect is FOO-dual-link)

This is really a bit of a fudge. In the NAPS implementation, it can also mean a heavy overhead in workspace storage, as in the case

```
     FOO A B C
```

the interpreter builds a temporary object A B C (taking full copies - i.e. not using pointers) before passing it to FOO. If any of A B or C are large, the probability of WS FULL in FOO is greatly enhanced.

With this scheme, strand notation assignment is also desirable:

```
     ∇ FOO A∆B∆C
 [1]   A B C←A∆B∆C
```

is much more convenient than the alternatives.

Falkoff's semi-colon notation for this, [Falkoff82], would be a significant improvement. One would then have a more controllable mechanism, and prevent the hidden overheads.

In simple strands, strand notation is simple and much used. Any degree of complexity in the strand (e.g. indexing) means difficulty - one usually has to use the interpreter to find out what it will do. This is an irritation, and IPSA's link function has many attractions in this area, but strand notation assignment is a considerable benefit, and I would be loath to give it up.

The problem with strands is that space has been given an implicit meaning, and that it is acting as some sort of function. Note that the problem arises from vector notation itself, where the space acts as a high priority catenate, and

```
       1
```

has different properties to

```
       1 2 3
```

(In the SMIPSA function below, try writing 0,C instead of C,0 - after all, it is only ensuring there is at least one zero in each row.)

LANGUAGE EXTENSIONS

It is difficult to find problems that have been solved in different versions of APL extensions. Very few (if any) APL users are fluent in more than one dialect. The competition section of VECTOR (the Journal of the British APL

Association) encourages entries in the various forms of APL, even if they are not eligible for prizes, and often the alternatives get published. One can therefore compare the approaches taken by experts in their own dialects.

The value of extensions can also be seen from these competitions. It appears difficult to pose problems that are sufficiently challenging in standard APL to be worthy of a competition (without making it so difficult that no-one enters), but yet non-trivial using extensions.

Competition 1

The first competition (VECTOR 1.1; discussions in VECTOR 1.3) related to the game of life, represented in a sparse-data form. The competition involved finding two functions to convert between a simple representation

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

and to a representation recording numbers of successive runs of 0 and 1

```
2 2 4 5 2 2 3
```

(the first element of the code vector always representing number of leading zero's)

Although no nested array entries were discussed when the results were given, it was pointed out that the run-code to boolean was trivial with replicate:

```
    ∇ R←RTB V
 [1]  R←V/(ρV)ρ0 1
    ∇
```

In several other competitions, replicate would have been a valuable extension to standard APL. Perhaps we should have a rule that extensions only become part of the standard language if they are as simple and as natural as replicate.

Note also that the 'best' solution to boolean to run-time

```
    ∇ R←BTR B
 [1]  R←R-⎕IO,¯1↓R←R/⍳ρR←(0,B)≠B,2
    ∇
```

is essentially successive applications of N-wise reduce:

```
    ∇ R←BTR B
 [1]  R←¯2-/0,(2≠/B,2)/⍳ρB←0,B
    ∇
```

which, although a neater concept, is not a lot simpler in practice.

Competition 2

The competition in Vector 2.2 was to validate and translate a character matrix, each row of which represents a single numeric.

Again, the code had to be ISO-standard conforming, so it couldn't use system functions such as ⎕FI and ⎕VI(APL*Plus) or ⎕VFI(Dyalog APL). Although the basic part of the problem is trivial with these system functions and the ability to use split (APL2 enclose-with-axis), and each

```
⎕VFI¨↓CHARMAT ⍝ Dyalog APL
```

(or simply the rank operator in IPSA), the task of doing this in ISO-conforming APL was too daunting, and no solutions were submitted.

So we shouldn't forget system functions when discussing extensions to APL. Although they may not look pretty, they are very effective in avoiding nasty and often inefficient APL code. They are even more effective when they fall in the domain of operators.

Competition 3

This competition was set in Vector 1.3 and discussed in 2.1 and 2.3.

Here, the skills of a group of staff were to be matched with the requirements of jobs.

The staff skills were to be represented by rows of a matrix, each non-zero entry being the index into a table of skill descriptions.

e.g.
```
     CONS
 1 2 3 7 0 0
 1 3 7 9 2 6
 7 9 0 0 0 0
 1 6 5 3 9 0
```

The skills required for jobs were similarly recorded

e.g.
```
     JOBS
 1 2 3
 7 9 0
 1 4 0
 1 3 0
 1 6 3
```

The competition involved writing a dyadic function to match job requirements to skills available:

```
     JOBS SKILLMATCH CONS
 1 1 0 0
 0 1 1 0
 0 0 0 0
 1 1 0 1
 0 1 0 1
```

This produced a range of solutions in standard APL, but also drew alternative entries in Sharp APL

```
     ∇ R←J SMIPSA C
 [1]    R←⍉∧/J∊⍤2 1 C,0
     ∇
```

and APL2

```
     ∇ R←J SMAPL2 C
 [1]   R←∧/¨((⊂[2]J)~¨0)∘.∊⊂[2] C
     ∇
```

while at least two people at British Airways immediately jotted down something very similar, but in STSC's nested arrays:

```
     ∇ R←J SMNAPS C ⍝ will also run in Dyalog APL
 [1]   R←∧/↑(↓J)∘.∊↓C,0
     ∇
```

One point to note here is that the IPSA and NAPS solutions both performed better than many of the standard APL solutions, even though some of these had been worked on a lot for efficiency. Certainly the NAPS solution involved a lot less programming effort. (No timing comparisons were noted for the APL2 solution, although rewriting as

```
     ∇ R←J SMAPL2B C
 [1]   R←∧/¨(⊂[2]J)∘.∊⊂[2] C,0
     ∇
```

would probably help.)

Note that in the NAPS solution, split and mix are being used like IPSA's dual operator. Both arguments are split into vectors of rows, and the result reassembled afterwards.

EVENT HANDLING

Here there are several approaches. IPSA and Dyalog have the very powerful ⎕TRAP facility, while STSC provide ⎕ELX and IBM have ⎕EA. Other vendors have other implementations.

Event handling systems allow all sorts of wierd and wonderful opportunities to the wily programmer. We have attempted, and been largely successful in limiting implementation of error handling through two basic utilities:

```
     ∇ ELX←PASSBACKERRORS
 [1]  ⍝∇ Returns ⎕ELX setting to pass errors  back
         to calling environment
 [2]  ⍝ Values of ⎕DM  ⎕SI  at  the  lowest  level
         will be in globals ΔDM ΔSI.
 [3]   ΔDM←'' ⋄ ΔSI←''
 [4]   ELX←'ΔDM←ΔDM,(0∊⍴ΔDM)/⎕DM⋄
        ΔSI←(⎕IO+0∊⍴ΔSI)⊃(ΔSI ⎕SI)⋄
       ⎕ERROR((∧\⎕DM≠⎕TCNL)/⎕DM),⎕TCNL,''(ΔDM ΔSI
        set)'''
     ∇
```

and

```
     ∇ ELX←ERRORTRAP ELX;⎕ELX
 [1]  ⍝∇ ERROR:Modifies ⎕ELX  setting  ELX(TV)  to
        cutback stack to this level
        before executing input ELX
 [2]  ⍝ It assumes ⎕ELX is  localised  in  calling
        function.
 [3]   ⎕ELX←'⎕DM'⍝ Avoid ⎕ELX IMPLICIT ERROR
 [4]   ELX←⍕ELX ⍝ Force error if ELX unacceptable
 [5]   ΔSI←ΔDM←''
 [6]   ELX←'ΔDM←ΔDM,(0∊⍴ΔDM)/⎕DM⋄
        ΔSI←(⎕IO+0∊⍴ΔSI)⊃(ΔSI ⎕SI)⋄
        ⎕ERROR(2≠1↑,⎕IDLOC''⎕ELX'')/((∧\⎕TCNL≠⎕DM)
        /⎕DM),⎕TCNL,''(ΔDM ΔSI set)''⋄',ELX
     ∇
```

A simple setting of PASSBACKERRORS in a function makes that function behave as an APL primitive, signalling the error at the call of the function in which the error is found.

ERRORTRAP is more complex, cutting back the stack to the function from which it was called before executing the expression passed as its argument. It depends on ⎕ELX being properly localised. It is often used in expressions such as

```
      ⎕ELX←ERRORTRAP '→ERR'
       .
       .
       .
     ERR:
```

and other occasions where a simple

```
      ⎕ELX←'→ERR'
```

could have disastrous consequences because it could be executed in a lower level function.

Two functions, STOPTRAPS and RESTORETRAPS are provided to turn off error handling implemented through these utilities and to restore it, so that real causes of problems can be investigated. As an aid in this, the original error message and the state of the stack when it was encountered are stored in globals.

Use of these utilities enables us to do as much in the way of event handling as we would want. They can be simulated in the Sharp or Dyalog environments, using ⎕TRAP. But they let you do a lot more than is possible with ⎕EA. In particular, the error recovery procedure can be varied during a function in a way that seems difficult with ⎕EA.

```
      ELX←⎕ELX←PASSBACKERRORS
      ⎕FHOLD T←FSTIE 'DATAFILE1'⍝ Share  tie  file
          ⍝ and put hold on it
      ⎕ELX←'⎕FUNTIE T⋄',ELX⍝ Untie  it  on  error,
          ⍝ then error handling as before
      A few lines of processing
       .
       .
      ⎕FUNTIE T⍝ Untie the file again
      ⎕ELX←ELX ⍝ Restore the original setting
      A few more lines of processing
       .
       .
       .
```

With ⎕EA, one seems to be forced to spawn subfunctions whose boundaries are determined by the desired error handling, rather than the overall logic of the process being programmed. No doubt it can be done, but it will look and be artificial in many cases, and the beaking up of the code into too many little pieces will detract from readability and comprehensibility. Too many small functions can be as great a menace as too many large ones.

An additional problem is that ⎕EA is essentially an extension to execute, which creates problems with programs to analyse APL code, and detracts from readability, at least of the main path of the program.

EFFICIENCY AND COMPILED FUNCTION SUPPORT

To allow the use of STSC's APL compiler, NAPS now allows the use of compiled functions. In practice, this means functions written in assembler, as this this the end-product of the compiler process. A workspace of such functions (FASTFNS) is supplied with the in-house product. It is interesting to note that few of these functions are the product of compiling APL - they have generally been written in assembler or TABL using algorithms very different to those normally employed in APL. The side-effects of this approach are highlighted in the behaviour of the resulting functions with empty array arguments.

Also provided is a facility to measure the efficiency of APL code - ⎕MF, so that one's efforts with the compiler can be directed. Using this facility, and the supplied FASTFNS, we have been able to dramatically improve the performance of many systems. Many of the speedups are similar in magnitude to those claimed for the compiler itself. Where suitable FASTFNS exist, we can now contemplate tackling problems where performance in APL would have been a considerable problem.

IRRITATIONS WITH NESTED ARRAYS

Life with nested arrays is not always as simple and straightforward as one would like. A few of the major irritations with nested array APL are listed here. Some arise from the nature of APL itself, others from STSC's implementation.

DATA REPRESENTATION

Although in theory, we have almost complete flexibility in how we can represent data, there are considerable restrictions in practice. Just as performance with standard APLs suffers when working with scalars, so performance with nested arrays suffers when the data is fragmented into small nested items. Workspace requirements can also grow alarmingly. It is seldom a good idea to use non-nested heterogeneous arrays for data to be manipulated.

One fairly reliable sign that nested array representations are going to take forever to run is a generous sprinkling of the each operator - Alan Graham's "pepper". This is usually indicative of a looping approach to the problem - each ¨ represents a loop. Such code is not without its value - it can be written very quickly, and can be used as an executable specification for more efficient code to be checked against. But if left in place in a production system, it can cost many hours of development time and end-user time through bad response.

We have found that it is much better to avoid partitioned enclose, but to use partitioned data techniques as developed by Bob Smith eg[Smith 79]. The problem does not lie with the partitioned enclose, although it has only recently been implemented in assembler (and with a boolean right argument can still be beaten by an APL loop) but with the application of functions to each part of the resulting nested structure. The classic

```
    ∇ Z←P ΔORREDUCE V;C
[1]   ⍝∇ PARTITIONS: Simulate V/¨P⊂V without
      ⍝  using Partition enclose.
[2]   ⍝ P and V must be logical vectors.
[3]    Z←(C/1⌽C←(P∨V)/P)≤P/V
    ∇
```

can beat

```
    V/¨P⊂V
```

by a factor of 100.

NONCE ERRORs

The major source of irritation is the NONCE ERROR generated by

```
       FOO¨⍳0
```

At BA, comments

```
       ⍝ to avoid NONCEnse ERROR
```

have started appearing. Many and wonderful are the ways adopted of avoiding nonce error, from testing for empty arguments before doing anything, to overtakes and providing fill elements.

Most of these could be avoided if the simple behaviour of Dyalog APL [Dyalog85] were adopted:

R←f¨⍵	If ⍵ is empty, the derived function is applied once to the prototype of ⍵, and the shape of R is the shape of ⍵.
R←⍺f¨⍵	If ⍺ or ⍵ is empty and scalar conformable, the derived function is applied once to the prototypes of ⍺ and ⍵, and the shape of R is determined by the rules for scalar conformability.

This still allows special cases where f applied to the prototype produces a DOMAIN ERROR to be dealt with as at present.

The work involved in protecting against this class of NONCE ERROR would make any migration of code written in Dyalog to APL*Plus nested array systems extremely tedious. If STSC were to introduce user-defined operators, an EACH which avoided NONCE ERROR would be the first written, and probably the most used.

THE MIX PRIMITIVE - ↑

Under APL*Plus,

```
        ↑A
```

generates a LENGTH ERROR if the elements of A are not all the same length. The first nested array utility we wrote, and still the most heavily used is:

```
     ∇ R←MIX A;⎕ELX;⎕DM
 [1] ⍝∇ Does MIX on array of nested arrays of same
rank but not shape, using overtake to make shapes
conform
 [2]    ⎕ELX←'→(''LENGTH ERROR''≡12↑⎕DM)/ERR⋄',
   PASSBACKERRORS
 [3]    R←↑A ⋄ →0
 [4]   ERR:⎕ELX←PASSBACKERRORS
 [5]    R←↑(⊂⌈⌿↑,⍴¨A)↑¨A
    ∇
```

which emulates the behaviour of the analogous APL2 disclose-with-axis, or the Dyalog primitive mix.

Note that line 3 avoids the need to protect line 5 against NONCE ERROR when A is empty.

EVENT HANDLING IRRITATIONS

When one wants to trap a specific error (e.g. NONCE ERROR or WS FULL as in the SPLITARRAY example below) in order to try a different algorithm, it is necessary to compare the error message produced against a text string. This is cumbersome compared to the alternative scheme of using numbers associated with different events.

Note also that the heavily used MIX cover function can upset the environment (⎕DM cannot be localised despite the effort to do so), resulting in many other errors being reported as LENGTH ERROR when attempts to log errors include calls of MIX.

Error handling, as it stands, offers too much power to the cunning programmer. I feel it should be restricted to only execute expressions within the function in which the trap is set, or to return control to the level above, with a suitable error message. One needs a mechanism to enable one to distinguish different events simply and clearly, and the ability to have several statements in the domain of the same trap.

FACILITIES AVAILABLE ELSEWHERE

In working with nested arrays, one often comes across problems where facilities provided in other implementations look attractive. I mention some of them here; it is not intended to be an exhaustive list.

COMPOSITION

This operator appeared in the research NARS, but did not find its way into the production NAPS version, perhaps because of the symbol chosen for it (jot). It did find its way into Dyalog APL. Composition allows functions to be 'glued' together to build more complex functions, or arguments to be 'glued' to functions. (Note that it is not the same as the IPSA composition-it covers aspects of ¨ ∘̈ and Ö: u¨n;m¨v;cases of u∘̈v and uÖv)

A very common construct in our code is

```
        (⊂⍺)FOO¨⍵
```

(Apply ⍺ FOO to each element of ⍵). A much more natural way of writing this would be

```
        ⍺∘FOO¨⍵
```

Here the left argument is a modifier to the function verb - an adverb. Composition enables the glueing of these together to produce a compound verb (as in German).

Another common construct is

```
        FOO¨GOO¨⍵
```

Here GOO is applied to each element of ⍵, then FOO is applied to each element of the result. The workspace requirements can be daunting, for instance

```
        ⍴¨⎕FREAD¨⍵
```

Without composition, (or direct definition), we have to define trivial functions elsewhere, leading to a loss of continuity in the code:

```
      ∇ Z←HOO R
 [1]    Z←FOO GOO R
      ∇
```

and then use

```
        HOO¨R
```

(Apart from WS considerations, this nearly always executes faster, presumably because there are fewer space manipulation overheads.)

With composition we could write

```
        FOO∘GOO¨R
```

and achieve the same result.

Such an operator seems a sensible step on the way to arrays of functions, a concept much easier to handle if all functions in the array are monadic.

In our screen software, all input can be forced through a validation routine. This can be thought of an array of functions with different functions applied to different fields on the screen. The validation functions can be monadic, or dyadic with left argument provided. It would be much more natural and simpler if we could use composition in defining this concept, so that all these functions are monadic.

The introduction of function assignment to Dyalog APL, which can be thought of as a simple direct definition or as a first step towards arrays of functions, depends rather heavily on composition for its usefulness.

INDEXING ALTERNATIVES

I have one major application where I don't know the rank of the data that it will be processing. This has been a major headache throughout, especially where indexing is needed. From ({) and merge (}) would be helpful here, but not sufficient on their own.

Often the approach has been to split the array into a vector of sub-arrays, to perform the operation, and put the data together again.

Here the implementation of split (Enclose-with-axis) is not ideal. Like so much else in APL there are few problems with matrices, but when the data has rank greater than 2, things get much more complicated. For this case one invariably wants the result of the split to be a vector of subarrays, (occasionally a matrix of subarrays), but the primitive gives an array of vectors. This has resulted in the SPLITARRAY utility. Its importance to some applications can be judged by the work that has obviously be expended in trying to make it efficient in workspace and execution:

```
      ∇ AA←I SPLITARRAY A;J;K;R;S;T;⎕ELX;ELX
 [1]    ⍝∇ NAPS: Splits array A into array of
         subarrays along axes I.
 [2]   ⍝ For vector I, splits along each axis, so
         that(⍴Z)←→(⍴A)[,I]
 [3]    ELX←⎕ELX←PASSBACKERRORS
 [4]    S←⍴A ⋄ K←⍋I,T←(~(⍳⍴S)∊I)/⍳⍴S
 [5]    →(K≡⍳⍴K)/L0 ⋄ A←K⍉A
 [6]   L0:R←⍴AA←↓((×/S[I]),×/S[T])⍴A
 [7]    ⎕ELX←'→((''WS FULL'')(''NONCE E'')∊⊂7↑⎕DM)
        /L1 L4⋄',ELX
 [8]    AA←S[I]⍴(⊂S[T])⍴¨AA ⋄ →0
 [9]   L1:⎕ELX←ELX ⋄ J←0 ⍝ If WS FULL
 [10]  L2:→((J←J+1)>R)↑L3 ⋄ AA[J]←⊂S[T]⍴J⊃AA ⋄ →L2
 [11]  L3:AA←S[I]⍴AA ⋄ →0
 [12]  L4:AA←S[I]⍴⊂S[T]⍴⊃AA
      ∇
```

```
      4 3 SPLITARRAY 1 1 3 4 2 2⍴⍳48
┌────┐ ┌─────┐┌─────┐
│1 2 │ │17 18││33 34│
│3 4 │ │19 20││35 36│
└∩───┘ └∩────┘└∩────┘
┌────┐ ┌─────┐┌─────┐
│5 6 │ │21 22││37 38│
│7 8 │ │23 24││39 40│
└∩───┘ └∩────┘└∩────┘
┌─────┐┌─────┐┌─────┐
│ 9 10││25 26││41 42│
│11 12││27 28││43 44│
└∩────┘└∩────┘└∩────┘
┌─────┐┌─────┐┌─────┐
│13 14││29 30││45 46│
│15 16││31 32││47 48│
└∩────┘└∩────┘└∩────┘
```

SPLITARRAY is used in effect to simulate an ALONG operator. It is used, with its inverse MIXARRAYS, to allow simple functions to be applied along axes of the data - for example along the MONTH axis (or the MONTH and YEAR axes) of data whose dimensions are class of travel, route, where sold, year, month. The data is processed in subarrays by a function coded as though it deals with a simple vector (or matrix).

The along operator would have much in common with bracket-axis notation (compare +/[1]), and the concept behind it has much in common with the rank operator. But implementing it through rank would also need the prefer/defer ⍥ operator.

EACH vs DUAL

The great joy of the each operator in NAPS, APL2 and Dyalog APL is the fact that it is possible to use it with user-defined functions. Although formally defined in the same way as

```
        FOO¨< ⍝ FOO with disclose
```

it is not thought of as such by our programmers - it is simply seen as a mechanism that applies FOO to each of the elements of the data, a very much simpler concept for them to grasp. Once someone thinks of it in this way, as a mechanism whereby a function is applied to each element of its arguments, an explanation of with-disclose seems highly recursive. (To apply a function to each of the elements of ⍵, first apply the disclose function to each element, then apply the function you first thought of to each resulting element, then apply the enclose function to each of the results.)

In some situations, each is used as dual. An operation is done to the structure of the data, a function is applied to the result, and the original structure restored. But the first operation is rarely a primitive APL function, and the inverse needs explicit application.

An example is the provision of a fill element in case an array is empty. A fill element to ensure the correct behaviour is appended to the front of the array, FOO¨ called, and the fill element stripped off again afterwards.

Although it is possible to provide ways of tackling these problems using further language extensions - a "fill" operator (eg [Pesch 1984]) could be used in the above case, there seem to be too many situations needing their own specific extensions, and I prefer in general the explicit coding of prior and post operations.

In many other situations the dual concept is used without any mention of each. A popular technique is to sort some data, perform an operation, and apply the inverse of the sort to the result, so that it lines up with the original input.

The use of SPLITARRAY also falls into the domain of the dual concept. After the data has been split into its subarrays, a simple function is called to process the subarrays. This is sometimes called using the each operator, but more usually not. Afterwards, a function MIXARRAYS, the inverse of SPLITARRAY, is called to restore the data to its original format.

A favorite analogy used in the each versus with-disclose argument is the notation for inner product. The power of the inner product notation is that it makes explicit the functions being applied. This has led to considerable exploitation of inner product - to the extent that +.× is now a minority use [Kanner82]. Will the same ever be true of with-disclose? Rather than making operations explicit, the dual operator makes implicit the application of the inverse.

User-defined functions have been within the domain of the each operator from the start in all implementations, and account for much of the use of the operator. How much has the need to provide inverse functions complicated the implementation of user-defined functions within the domain of operators in IPSA's APL?

EFFICIENCY

Efficiency is important on our APL systems. Much of the use of them is interactive. Poor response is very irritating for interactive work.

Buying extra computer power is not always a feasible answer. If the early growth in computer power needed to run APL at BA had been allowed to go unchecked, the annual cost of providing new power would soon be rivalling the cost of investing in new aircraft. As an airline, investment in new aircraft and operational equipment gets priority - quite rightly. Investment in computer systems is already large and highly visible. Of the money available for investment in computer systems, it is easier to justify expenditure on the real time systems which support our operation - the reservations system, departure control system, operations control - where improvements can improve the product we offer and our competitive standing.

Although many systems are now more efficient than they were a year ago, we are limited in the use we can make of the APL compiler. This is largely due to it not being able to deal with nested arrays. In fact, one sometimes wonders how much the implementers of the compiler are aware of nested arrays and the facilities offered. One supplied compiled function implements scatter point indexing - less efficiently than the same nested array feature in the interpreter. An example published in the literature promoting the compiler compiles a function which tries to implement the split primitive using loops and globals in standard APL.

Another point is that many of the fast functions which we have used effectively are not the result of compiling APL. Instead, they have been written in assembler (or the TABL language developed for the compiler) and use algorithms different from those an APL programmer would naturally use. This non-APL approach is highlighted when the functions are presented with empty arguments.

Our use of the ⎕MF monitoring facilities has shown up that in well-written APL it is not the interpretive overhead that costs, but being forced into unnecessary processing. Perhaps this is best illustrated by the optimisation that gets done within interpreters - special casing operators so that ∧.= does not actually perform all the comparisons, but stops as soon as a mismatch has been detected, or ∨/ stops as soon as a 1 is found. Many user-defined operators deal with this area - associative scan, and many of the operators in Jim Brown's operators for logic programming [Brown86].

To use alternative algorithms, or to break down the modularity of APL systems to prevent unnecessary processing, the programmer will usually need to resort to scalar programming. APL does not support this well - the code starts to consist mainly of loop and conditional control. Instead of using an APL compiler with this mess, I prefer the idea of writing it in a language which is designed for this sort of task (or rather getting someone else to write it) and then being able to call the resulting compiled code from APL.

Of the APLs I have experienced, Dyalog APL seems the best equipped here, with its ability to access the Unix shell, and hence functions written in C. It provides tools to interface APL data structures, including nested arrays to C.

The importance of this is not just in making APL systems more efficient - many skilled APL programmer-days have been expended in trying to achieve reasonable response times. Many of the hierarchical data structures pervasive to our systems, whether operational financial or reporting, are based on the sort of confused logic that is easy to implement in scalar languages, but very painful in APL. Perhaps this is a problem specific to BA, but I suspect it also occurs in other installations where APL is a relative newcomer. The effort expended in dealing with these structures could have been avoided if easy access to scalar languages had been possible.

CONCLUSION

Extensions to standard APL have made a beneficial difference to our work. They have not solved all our problems. They have extended the boundaries so that problems are encountered further on, if at all.

Neither direction (APL2/IPSA) have a monopoly of right or wrong. There are useful features and ideas in each system. From the viewpoint of programming users of APL, the more central role of data structures in the APL2 style is preferable.

Until the advent of extended data structures, and the split in philosophy, one of the major assets of the two large time-sharing bureaux was their pragmatism, with such developments as file systems, formatting, event handling and packages. In the arguments over nested arrays, this has been lost.

The pragmatic approach of the implementers of Dyalog APL has much to commend it, particularly the access to code written in other languages.

Acknowledgements

I would like to thank Dave Ziemann for his work in organising the competition section of Vector, and the competitors whose solutions I have used.

References

A.D. Falkoff (1982) Semicolon-Bracket Notation - A Hidden Resource in APL (APL82 Conference Proceedings p113)

C.M. Cheney (1981) Nested Arrays Reference Manual STSC Inc.

APL*Plus Enhancements (1985) STSC Inc.

Dyalog APL User Guide (1985) Dyadic Systems Limited.

APL2 Programming: Language Reference Manual SH20-9227-0

R Hodgkinson (1986) APL Procedures (APL86 Conference Proceedings p179)

J.A. Brown (1986) Logic Programming in APL2 (APL86 Conference Proceedings p282)

T. More (1982) Rectangularly Arranged Collections of Collections (APL82 Conference Proceedings p219)

K.E. Iverson (1986) A Dictionary of APL. IPSA publication.

A. Graham (1986) Idioms and problem solving techniques in APL2 (APL86 Conference Proceedings p172)

R. Smith (1979) A Programming Technique for Non-Rectangular Data (APL79 Conference Proceedings p362)

R. Kanner (1982) The use and disuse of APL: an empirical study (APL82 Conference Proceedings p154)

R. Pesch (1984) On the Question of Fill (APL Quote-Quad 15.1 p9)

Vector(The Journal of the British APL Association) Competition sections in volumes 1.1 thro 2.3