Fortran 88 Arrays – Paper Clips and Rubber Bands[*]

Robert Bernecky

Snake Island Research, Inc.
18 Fifth St., Ward's Island, Toronto, Canada M5J 2B9
Tel: +1 416 203-0854
email: bernecky@acm.org

February 14, 2001

Abstract

Excellent application performance should not require tour de force programming efforts by users. Fortran 88, in an attempt to bring it from a scalar orientation into an array notation, has adopted some of the early concepts of APL, such as array operations. The introduction of these ideas is shown to be inadequate in meeting the algorithmic needs of programmers, in terms of expressiveness, consistency, and conciseness. Comparisons with APL show Fortran 88 to be a mongrel, neither scalar- nor array-oriented, unable to achieve the productivity, performance, reliability, and maintainability requirements of computer users in the 1990s.

1 Introduction

Recently, the X3J3 Fortran standards committee proposed an extended version of FORTRAN, denoted Fortran 88 (aka Fortran 8x aka Fortran 90 – since X3J3 can't decide what to call it, I'm sticking with Fortran 88, which is what [Cam89] calls it.). Fortran 88 is intended to "promote portability, reliability, maintainability, and efficient execution of Fortran programs..."[Cam89]. One of the major additions to Fortran 88 is the inclusion of elemental array operations, vector constants, structural and selection verbs, and a few reductions, largely following the design of early APL systems[Ive62, Int84].

As laudable as the goals of X3J3 may be, Fortran 88 remains a language which is far harder to use effectively than an applicative array language, such as APL[BB93, IBM94, IBM94, Int84] or J[HIMW90][1].

The problems with Fortran 88 may be roughly broken into two major categories:

- Limited expressiveness
- Inconsistency

The fundamental problem with Fortran 88 is that array extensions have been bolted onto an existing language, rather than integrated into it. This paper clip and rubber band approach to design has resulted in a language which is significantly more complicated than its predecessor, yet which doesn't offer the concomitant increases in productivity or reliability that could accompany a total re-design.

The following sections discuss these problems in more detail, and show how applicative, array-oriented languages solve them. APL is used for this exposition, because I am familiar with it, and because it implements the principles which I propound. Furthermore, because APL

[*]Originally appeared in *SIGPLAN Notices*, Volume 13, No. 4, February 1991

[1]J is a modern language derived from APL. Among the design principles of J which differ from other APL dialects are: operations across the first (major) axis of arrays, with operation across other axes achieved via operators such as the *rank adverb*; adherence to a simple and consistent functional syntax; and use of the ASCII character set. APL and J interpreters are available at www.jsoftware.com

is available in executable form, the utility of relevant concepts may be tested, rather than merely argued. APL is not presented as a panacea – APL has its own problems, which are mentioned in a later section.

Discussion of other aspects of Fortran 88, such as improved facilities for numeric computation, user-defined data types, and modular definitions, are beyond the scope of this paper.

2 Limited Expressiveness

Freedom of expression is the matrix... – Palko vs Connecticut

Expressiveness entails providing a vocabulary from which sentences can be built that express a thought – part of an algorithm. Fortran 88 fails to offer a rich vocabulary of words which can be combined to form sentences. Instead, it provides a collection of already completed sentences, in the form of *intrinsic procedures*,[2] such as MATMUL and ADJUSTL. This may seem to be a superior approach, but it in fact limits the creativity of the user of the language to a phrasebook approach to communication, and stifles the creation of algorithmic poetry.

This section presents a few specific examples of the limitations of Fortran 88, and highlights how languages which appear to do less, in fact do more for the application programmer, by offering improved semantics, more compact and powerful expression, better code reliability, and the possibility of improved performance.

2.1 Adverbs

Adverbs and adjectives increase our ability to express ourselves in a natural language, by allowing us to alter the meaning of nouns and verbs[Ber90c]. Knowing `x` verbs and `y` adverbs in a natural language gives you effective use of `x×y` *derived verbs*. For example, given the verb *run*, I can combine it with adverbs to create a large number of related or derived verbs, in a consistent manner: *run slowly, run quickly, run cautiously*. This provides a vastly increased vocabulary, *without* requiring the knowledge of all possible verbs. The synergy of verbs and adverbs offers a great expressive power – if you know `x` verbs, and learn just one more adverb, you have really gained the ability to use `x` new verbs. In Fortran 88, if you learn one more intrinsic procedure, you have learned exactly that – one more intrinsic procedure. The knowledge gained is not synergistic with previous knowledge.

[2]Fortran 88 defines *intrinsic procedures* to be the set of inquiry functions, elemental functions, transformation functions, and subroutines which are inherent to the Fortran processor.

2.2 Reduction

Reductions are operations which may be thought of as placing a specified operation between each subarray of an array, and evaluating the resulting expression. For example, the SUM reduction of `1 2 3` might be viewed as `1+2+3`, giving the result `6`.

Fortran 88 defines seven reductions: SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL. Fortran 88 views each reduction as unique, with no common principle or consistent naming convention tying them together, even though they are are closely related by virtue of being reductions.

An approach which offers more expressive power and is easier to teach and learn, is to consider operations such as + to be verbs, and to introduce an adverb which denotes reduction. We can then combine *any* verb with the reduction adverb to create a specific reduction in a simple, predictable, and consistent manner.

APL denotes reduction by a function `f` as `f/`, thereby forming SUM as: `+/`. Figure 1 shows a few elemental functions and their related reductions in Fortran 88, in APL, and in J. Compare the consistency and generality of APL with the haphazard, limited notation in Fortran 88.

Other reductions, such as `−/t` to compute the alternating sum, `≠/t` to compute the parity, as well as reductions using user-defined verbs, are not defined in Fortran 88, but are a natural part of APL. So are a variety of reductions which appear at first to be meaningless on arrays, but which are very convenient when applied to two-column arrays.

From this, it can be seen that one key to effective design is to provide what at first appears to be less, rather than more, functionality, and thereby give the user more freedom for expression through the power of composition of verbs and adverbs. Although Fortran 88 could presumably obtain this freedom of expression by introducing adverbs, it's easier to train users in a new, simpler, language which doesn't include the baggage of Fortran's history.

2.3 Recurrence Relations and the Scan Adverb

Although Fortran 88 provides a few reductions, it does not provide any *scans*. Scans are related to reductions in that they provide the partial results of the reduction. For example, the sum scan (+\) of `3 1 2 4` is `3 4 6 10`. Scans offer considerable expressive and computational power, particularly with Boolean arguments. In APL, quoted strings can be removed from a character vector by the *exclusive or* or *not equal* scan (≠\) of the Boolean vector which represents the location of the quotes:

```
  (b⍱≠\b←t='''')/t
```

Here is how it works on a text vector:

```
       q
a 'why' is not 'ok'.
       t←q=''''
       t
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0
       ≠\t
0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0
       t⍱≠\t
1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1
       (t⍱≠\t)/q
a is not .
```

Recurrence relations have many applications, of which one of the simplest is computing the remaining balance on a loan, given an initial balance, and possibly varying payments and interest rates. In Fortran, this might be written as:

```
    DO 4 I=2,N} \\
4   BAL(I)=BAL(I-1)*INT(I)+PAY(I)}\\
```

The indexed references to `BAL(I)` and `BAL(I-1)` make the code difficult to vectorize. Fortran 88's lack of scan makes it impossible to express recurrence relations using Fortran array notation. In APL, the recurrence relation may be expressed as:

```
  t×+\pay÷t←×\int
```

Here is how it works on a loan with principal of $100, payments of $30, and a 20% interest rate (numbers have been rounded to fit in one column):

```
        pay
100 ¯30 ¯30 ¯30 ¯30 ¯30 ¯30
        int
1.2 1.2 1.2 1.2 1.2 1.2 1.2
        ×\int
1.2 1.44 1.72 2.07 2.48 2.98 3.58
        pay÷×\int
83 ¯21 ¯17 ¯14 ¯12 ¯10 ¯8
        +\pay÷×\int
83 62 45 31 19 9 0.2
        (×\int)×+\pay÷×\int
100 90 78 64 46 26 0.7
```

Since the recurrence contains no explicit reference to adjacent items, the computation can be vectorized or parallelized with *no* changes to the source code. On large floating point arrays there is a theoretical possibility of precision loss, but this form has been used by actuaries for 20 years[3] to compute such things as mortality rates, on arrays of several thousand elements, without problems.

Partitioned reductions and scans are also popular in APL applications. Partitioning operations break arrays into smaller arrays of the same rank, along some specified axis, so that scans, reductions, and other operations may be applied to each of the smaller ones. A common example of partitioning operations is computing subtotals by city, state, and country in a sales report.

Although these capabilities can be performed in parallel using techniques similar to the recurrence relation above, their generality and precision can be improved by the use of partitioning and tiling adverbs such as the *cut* of SHARP APL or J, as will be shown in a later section.

2.4 The Rank Adverb

A number of Fortran 88 intrinsic procedures allow optional specification of values such as DIM or ORDER, used to control the axis of application of the procedure. However, the specification of these values, and their effect, is not consistent, nor does it permit operation along a specific axis or axes on all functions. For example, DIM can be used with reductions, but is not permitted to be

[3] In 1971, before scan was an integral part of APL systems, John Heckman created user-defined APL scan functions, which ran in two log n iterations. This algorithm was "discovered" much later by parallel computer researchers who were apparently ignorant of work done many years earlier in the APL community.

used to control the operation of elemental functions or user-defined functions.

The *rank adverb* [Ber87, Ive87, I.P87] in APL bears strong semantic resemblance to DIM, in that it is used to control the axes across which array operations are performed. However, the definition of the rank adverb is more expressive: *any* primitive, derived, or user-defined function can be applied consistently and independently upon sub-arrays chosen from the trailing axes of its argument(s).[4] The rank adverb provides the programmer with considerable power in a very general fashion. A vector may be added to each column of a matrix by the expression which selects vectors (rank-1 sub-arrays) from the matrix and scalars (rank-0 sub-arrays) from the vector:

```
v+⍤0 1 m
```

For example:

```
       v←10 20 30
       v
10 20 30
       m←3 2⍴⍳6
       m
0 1
2 3
4 5
      v+⍤0 1 m
10 11
22 23
34 35
```

Similarly, a matrix may be catenated to each plane of a tensor by selecting matrices(rank-2) from the left and from the right:

```
x,⍤2 2 y
```

Fortran 88 requires DO loops or use of SPREAD or RESHAPE to achieve this effect on elemental operations, and probably cannot handle non- elemental operations in general.

Traditional mathematical notation suffers from similar problems – Karmarkar [Kar85] wrote a matrix-vector multiply as an inner product, in which the vector was represented as a diagonal matrix. With the rank adverb, it could have been written as a vector-vector operation, which is conceptually a much simpler computation than inner product:

```
m×⍤1 1 v
```

[4]APL stores arrays in row-major order, unlike Fortran 88. This means that the subarrays specified by the rank adverb refer to adjacent (stride-1) items in an array.

2.5 Inner Product

Fortran 88 provides two methods of performing inner products: DOTPRODUCT and MATMUL. Both perform a SUM of PRODUCTs on arrays – DOTPRODUCT on numeric vectors; MATMUL on numeric matrices. They also perform ANY of ALL on logical arrays, which is valuable for graph computations such as transitive closure.

However, it so happens that inner product has the potential for much more than mere SUMs of PRODUCTs. If the `+` and `*` of the Fortran 77 DO loops for inner product are replaced by other functions, a whole family of interesting inner products appear, several of which are shown in Figure 2. Because Fortran 88 only offers one type of inner product, much generality and expressiveness is lost – the user is forced to return to the land of DO loops.

APL achieves greater expressiveness by treating inner product as an adverb, denoted by the dot "`.`". The adverb accepts two verbs which describe the particular type of inner product to be performed. For example, the classical sum of products is written as:

```
x+.×y
```

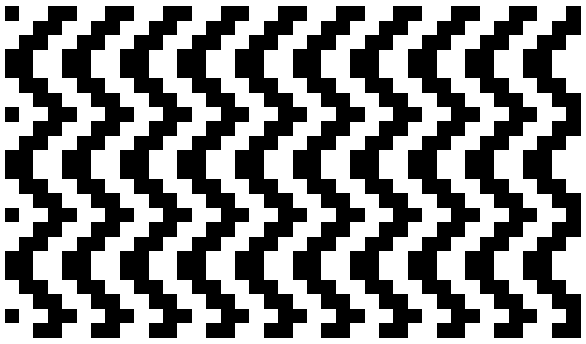
The symbols used by APL to denote this, by themselves, are of no greater utility than names such as DOTPRODUCT or MATMUL. The key difference lies in the fact that since inner product is written as an adverb, we can meaningfully alter its meaning in a *predictable* and *consistent* fashion by using it with other verbs in our vocabulary, increasing our expressiveness.

As an example of the value of expressing inner product as an adverb, consider a program which models a loom, by accepting arguments describing the treadling pattern, harness connections, and thread patterns to be used, producing a picture of the resulting fabric. Such a program, written in BASIC[Hei82], occupied two and one half pages. An equivalent expression in APL [Ber86] is:

```
' *'[(t∨.=th)[tr;]]
```

For example, if executed with these values:

```
t←4 2⍴0 3 2 3 1 2 0 1
th←1 40⍴0 1 2 3
tr← 46⍴0 1 2 3 3 2 1
```
the result could be plotted as:



The key to the compact expression of this algorithm lies in the use of a variant of inner product, `∨.=`, which is obvious to a user of APL, yet uncommon, and certainly not a variant which one would expect to find in a Fortran library.

2.6 Matrix Transpose

Fortran 88 defines matrix transpose only on arrays of rank 2. This needlessly restricts the expressiveness of matrix transpose. Higher dimensional transposes are of significant value in common applications. For example, a suitable rank-4 transpose, combined with a few reshapes, allows a rank-3 array of mailing labels `m`, each plane representing one label, to be printed `n`-up:
```
s←((×/2⍴⍴m)÷n),n×¯1↑⍴m
s⍴0 2 1 3⍉(((1⍴⍴m)÷n),n,1↓⍴m)⍴m
```
The first line determines the result shape. The second inserts an extra axis into `l`, performs the rank-4 transpose to bring the pieces into proper alignment, and then reshapes the result to its final form.

2.7 Array Notation

In a critique of Fortran 88, Curtis[Smi88] says:

> An important class of loops not expressible in array syntax is exemplified by the Hilbert matrix. (The basic problem is that) DO loop indices are used as part of the calculation as well as subscripts for arrays.

APL offers a number of ways to express a Hilbert matrix using array expressions. Here are two simple ones, one exploiting outer product, and the other using the generalized scalar extension concept of function rank:
```
1÷(⍳n)∘.+¯1+⍳n
1÷(⍳n)+⍤1 0,¯1+⍳n
```
Here is an example of a Hilbert generator in action:
```
      ⍳3
1 2 3
      (¯1+⍳3)
0 1 2
      (¯1+⍳3)∘.+⍳3
1 2 3
2 3 4
3 4 5
      1÷(¯1+⍳3)∘.+⍳3
1      0.5    0.3333
0.5    0.3333 0.25
0.3333 0.25   0.2
```
Since the expression is purely functional, the computation can be vectorized or parallelized across an arbitrary number of processors, without requiring *any* changes to the expression. As proof of the viability of this approach, Dr. Hans-Peter Meinzer recently noted at APL90[MMS90] that his *unaltered* APL tomography visualization programs were able to drive an IBM 3090 vector facility to several hours of use per month, whereas his colleagues, after having made substantial changes to their Fortran code, were still unable to achieve more than some seconds per month of vector use.

2.8 Vector-valued subcripts

Fortran 88 prohibits the use of vector-valued subscripts on the left side of assignment. This places a severe restriction on the utility of subscripting, limiting the expressiveness of the language. According to A. Marusak, the restriction arose from "confusion in many to one stores"[Smi88].

Interestingly enough, the ISO APL Standards Working Group went through the same angst some years ago. Virtually all extant APL implementations operated identically in the presence of duplicate subscripts: the value stored corresponded to the last occurence of the subscript. For example, the execution of:
```
x[1 1 1]←2 3 4
```

would result in a value of `4` in `x[1]`. A number of delegates wished to make this behavior an error, so that parallel machines might legally implement it in a non-deterministic form. Since the ISO APL Standard permits a conforming implementation to turn an error into some other behavior, both definitions would then be legal. In the end, the decision was based on the guiding principle of the Working Group – to enshrine existing industry practice – and the definition stood as in the example above.

2.9 ADJUSTL and ADJUSTR

The Fortran 88 intrinsic procedures ADJUSTL and ADJUSTR are defined on character strings, and have the effect of deleting leading or trailing blanks, respectively, from their arguments, padding the vacated spaces with blanks. There are a number of problems with the definitions of these procedures, all of which result from their being overspecified:

- They are defined on characters only. Shifting of logical or numeric arrays is not permitted.
- They are defined on rank-1 objects (lists) only. Shifting of matrices is not permitted.
- They presume the desired character to search for is non-blank. Searching for other characters, or sets of characters, is forbidden.

To understand why the characteristics of ADJUSTL restrict expressiveness, let's analyze a version of ADJUSTL written in APL. We observe that, since it shifts in as many blanks as it shifts out, one way to write it is to *rotate* the string up to the first non-blank:

```
adjl:(+/∧\ω=' ')⌽ω
```

Here is how `adjl` works on a text matrix:

```
      t
 l  add x,y
    st  x,z
    jmp l
      t=' '
0 1 1 0 0 0 1 0 0 0
1 1 1 0 0 1 1 0 0 0
1 1 0 0 0 1 0 1 1 1
      ∧\t=' '
0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
      +/∧\t=' '
0 3 2
      (+/∧\t=' ')⌽t
 l  add x,y
 st  x,z
 jmp l
```

A comparison of ADJUSTL and `adjl` reveals the following:

- ADJUSTL only works on strings. `adjl` works on arrays of *any* rank. For example, given a matrix of text as an argument, it will left-adjust each row of the argument to the first non-blank in each row, without requiring a loop.
- ADJUSTL is restricted to search only for the first non-blank. In `adjl`, the character to be searched for is *explicitly* specified in the expression, and could be changed if we wished to search for something else. The = makes it clear that we are searching for the presence of the blank, rather than its absence. We could, if we desired, instead search for the first non-blank. This is useful in applications such as an assembler. With `rtb` as a companion which left-adjusts to the next blank, we can analyze an entire source program in parallel:

  ```
  rtb:(+/∧\ω≠' ')⌽ω
  ```

  or

  ```
  rtb:(ω⍳⍤1 ' ')⌽ω
  ```

  The assembler would process all labels, then use `adjl rtb text` to rotate the labels out of the way, and rotate the op codes to the first column. Op codes, operands, and comments would be handled similarly. Thus, the flexibility of being able to specify the character being searched for, or not searched for, and the ability to operate on arrays, are both seen to have much more power than ADJUSTL.

- Explicit specification offers the ability to operate on non-character arguments. An expression to drop leading zeros in an argument is:

  ```
  dl1:(+/∧\ω=0)↓ω
  ```

  Note the similarity of `dl1` to `rtb`. A key design principle in APL is to provide generalized primitives which act in the same way on all data types, even if the result is primitives which at first appear to be less powerful in specific situations.

A few other capabilities which are trivially achieved with similar functions include:

- Shifting left/right to the first/last occurence of any of a set of interesting values.
- Shifting left/right to the first/last non-occurence of any of a set of interesting values.
- Shifting along different axes.
- Padding with values other than blank.

None of these are possible using ADJUSTL or ADJUSTR.

2.10 Boolean Poverty and Data type Abstraction

Fortran 88 treats logical (Boolean) data as a poor cousin, permitted only on the periphery of computation.

Booleans may be used to control computations, in the form of the optional argument MASK=, and are permissible in a few intrinsic procedures (such as COUNT, ANY, and ALL) but they may not be used as numerics in computation. This leads to needless complication, restricts our ability to express algorithms in a clear and concise manner, and has a negative impact on application performance and space requirements.

The inability to mix Booleans and numerics in computation leads to needless complexity in application code. The ISING spin model in Appendix C of the Fortran 88 Draft Standard is complicated and slowed by the need to convert a Boolean array to integers on each iteration of the inner loop, merely because of the lack of capability to perform arithmetic and shifts on Booleans. APL permits direct computation on Booleans as numerics, allowing the 12 lines of the Fortran 88 inner loop of the ISING spin model to be written as:

```
   c←(1⊖⍤1 i)+(1⊖⍤2 i)+1⊖⍤3 i
   c←(¯1⊖⍤1 i)+(¯1⊖⍤2 i)+(¯1⊖⍤3 i)+c
   c←c+(~i)×6-c
   i←i≠p[c]≥rand i
```

In the above, the first three elements of `p` contain 1, and `rand` generates an array of random probabilities of the same shape as its argument. The Boolean ISING array `i` is used directly in rotation, addition, negation, and exclusive or. The computation more closely reflects the problem, and the nugatory Fortran 88 computations involving the array ONES do not appear.

The first two lines of the above computation of `c` could also have been written in APL with a *cut* adverb[Ive87], albeit using a truncated, rather than cylindrical universe, as:

```
    (2 3⍴1 1 1 3 3 3) 3⍤(+/⍤,⍤∧¨s) i
```

The cut adverb expression `(2 3⍴1 1 1 3 3 3)⍤` performs an overlapping tesselation of the matrix argument into tiles of shape `3 3 3`, with beginning points for each tile which are `1 1 1` apart. The function `+/⍤,⍤∧¨s` is applied to each of the tiles. It computes the sum (`+/`) of the ravel (`,`) of each tile after anding (`∧`) it with a Boolean stencil array, `s`, of shape `3 3 3` which isolates the six neighbors of interest. The compositions `⍤` are used to pipeline the results of each independent tile computation though the three functions sum, ravel, and. A similar expression can be used to perform convolutions.

APL's expressive power on logical operations stems from treating Boolean .FALSE. and .TRUE. as the numbers `0` and `1`, typically represented as one bit per element. Any verb which may accept numbers accepts Booleans. Relational verbs produce Boolean arrays as results. This permits Booleans to be first-class citizens in the computational world. In APL, instead of IF/THEN/ELSE and WHERE constructs, one often sees expressions such as: `a+5×a>10`, which adds `5` to the elements of `a` which are greater than `10`. This simplifies compilers and interpreters, removes pipeline bottlenecks, and reflects a SIMD world view.

Fortran 88 forbids the application of relational intrinsic operations, such as *greater than*, on logical data. This severely limits the expressiveness of computations upon Booleans, which are powerful ways to control array operations. APL treats relational operations on Booleans exactly the same as relational operations on other numeric types, increasing our expressive power, and making certain set-related computations highly efficient. Support for relationals on Booleans completes the set of 16 dyadic Boolean functions.

2.11 String Operations

The Fortran 88 SCAN intrinsic procedure is defined to "scan a string for a character in a set of characters". SCAN('FORTRAN','TR') returns 3, just as the APL expression `⌊/'fortran'⍳'tr'` would do. However, SCAN is overspecified – it does *too much* work: In performing the minimum reduction (`⌊/`) on 3 4 (the result of `'fortran'⍳'tr'`), SCAN discards potentially useful information. The power of SCAN would be enhanced if it did *less*, and let the user invoke MINVAL when it was desired. As with INDEX, the user will often end up writing code to mimic 90% of SCAN, when a problem doesn't exactly fit the Procrustean bed of SCAN's definition. SCAN also shares the same limitations as INDEX – not defined on numerics, nor on arrays.

APL's *find* verb (`⍷`)is defined on all array types, of any rank. Find may be viewed as a string search primitive on character vectors, or as a stenciling operation on matrices, for image analysis. Find returns a Boolean array of the same shape as one argument, with 1's in any position where the upper-left corner of that subarray matches the other argument. In Fortran 88, it appears that if you are interested in finding subarrays which are not both vectors and characters, you'll have to write your own code to do it.

VERIFY has similar problems. VERIFY is defined to "verify that a set of characters contains all the characters in a string". It returns 0 if the string characters all belong to the set. Otherwise, it returns the index into the string of the first character not in the set. As with the above procedures, it is neither defined on arrays nor on numerics, needlessly limiting the utility of the procedure, and making life harder for application writers who could otherwise benefit from its use.

APL's cognate of VERIFY is set membership: `x∊s` returns a Boolean of the same shape as `x`, containing a 1 at each location which corresponds to an element of `x` which occurs in `s`. A number of useful variants are obtained with a few more verbs. Removal of a subset of elements from a list (such as removal of blanks) may be done with: `(~x∊s)/x`. Assertion that all elements are contained in the set is: `∧/,x∊s`. The indices of the elements which are not in the set is found by: `(~x∊s)/⍳⍴x`. An approximate equivalent to VERIFY is: `(~x∊s)⍳1`. Finally, removing multiple blanks from a string might be performed with one of the following expressions:

```
    (t⍲¯1↓0,t←s∊' ')/s
    (~'  '⍷s)/s
```

INDEX returns the starting position of a substring within a string. As with SCAN, the APL *find* verb performs the same function, yet offers significantly more generality, operating on any data type, and on arrays of any rank. Hence, it is natural for applications such as computer-aided vision, image processing, text editing, and so forth.

3 Language Inconsistencies

With consistency a great soul has simply nothing to do. – Emerson

Consistency in a notation is critical to its clarity.

Consistency implies fewer rules than inconsistency. Occam's razor should be sufficient reason to keep language designers as far away as possible from inconsistent rules. It often works. However, when a system already has an inconsistency of one sort, often initially introduced for reasons of "user convenience", it's thought reasonable to add one more inconsistency of the same sort – for reasons of consistency, of course! This is a big mistake – although computers can deal with any set of rules, *people* are simply not built to deal with the large sets of rules which accompany inconsistencies.

Consistency produces several other desirable effects:

- The resulting language is easy to teach and easy to learn.

- The notation can enhance and encourage creativity.

- The user is not hobbled by the compiler writer's ideas of what might constitute a "useful" intrinsic.
- Program faults and bugs are reduced, because the opportunity for incorrect comprehension is reduced.
- Performance can be improved – new optimization or parallelization techniques which can be applied to one construct often then apply in a uniform manner to all constructs.
- Compilers and interpreters are easier to construct.

3.1 Intrinsic Procedure Names

Fortran 88 offers the programmer no assistance in predicting the names, behavior, or existence of many facilities. For example, SUM performs a sum reduction of a numeric array. COUNT performs a sum reduction of a logical array, if .FALSE. and .TRUE. are viewed as the values 0 and 1, respectively. In spite of their similarity, there is no way for someone who knows SUM to *predict* the name of the very similar reduction COUNT. This lack of consistency makes the language hard to teach, and hard to use.

Figure 1 shows a comparison table, showing some simple expressions in Fortran 88, APL, and J for elemental operations, and for their corresponding reductions on vectors. Note the lack of correspondence between the Fortran 88 elemental function name, and its associated reduction, and compare it to the rigorous consistency and simplicity of APL. Note also the lack of consistency in the Fortran elemental function syntax, which appears to depend not on the function type, but on some other, unstated, criterion.

In Fortran 88, a user seeking to perform an `f` reduction must either memorize all the Fortran 88 intrinsic procedures, or must search the entire reference manual to see if a suitable intrinsic exists. Knowledge of `f` is of no help in reducing the scope of the search, because the name of the intrinsic is not directly related to `f`.

In APL, by contrast, all reductions are written as `f/`, regardless of the verb `f` being used. In a sense, the user creates the required intrinsic procedure on the spot, as needed. An alternate view is that the intrinsic procedures follow a rigorously consistent naming convention, and the user merely composes the name of the desired intrinsic. Of course, this only works if the intrinsics also have rigorously consistent semantics, as they are in APL.

Many operating systems have similar problems – large command sets and libraries, chock full of wonderful services – *if* you can find the one you need. It would be interesting to look at operating systems, and see if we can bring their definitions and behavior into the world of consistency.

3.2 Consistent Semantics

Fortran 88 has overly complicated rules for array operations. Consider inner product: the intrinsic procedure which must be used depends on the rank of the arguments. Figures 3 and 4 graphically display the kinds of inconsistency which is forced upon the Fortran 88 user. What gain is made by making the user remember two distinct library routines, when one would suffice if it were designed properly?

Fortran 88's inability to handle arrays of rank greater than two is inexcusable. As a trivial example of the value of handling arrays of rank greater than two, consider a 2-d graphics application in which a tensor of linear transform matrices is to be multiplied by another transform. This requires a DO loop around a MATMUL call because of the restriction to arrays of rank two or less. In APL, it is merely `t+.×m`, because APL extends systematically to arrays of arbitrary rank.

Another area where Fortran 88 is inconsistent is in the specification of controlling parameters. For instance, DIM has to be an integer lying within the rank of the argument – except when it doesn't – as in SPREAD. This inconsistency leads to bugs and slower development cycles, and is harder to learn than a consistent notation. Furthermore, these controlling parameters are only permitted in a specified list of intrinsic procedures, and are not defined on elemental functions, nor on user-defined functions. By contrast, the APL rank adverb provides identical capabilities in a completely general and consistent fashion.

3.3 Catenation

Fortran 88 treats catenation in an inconsistent manner: two character strings S1 and S2 may be catenated by the expression S1//S2, but there is no provision for catenating arrays or non-character data types, forcing users to write different code for the same operation, because the argument types differ or because they are not vectors. By contrast, APL allows any two arrays, regardless of type or rank, to be catenated along a specific axis using expressions such as:

First	Last	Axis k from end
`s1⍪s2`	`s1,s2`	`s1⍪⍤k s2`

APL's approach is simple, consistent, and more general. APL provides a far richer set of capabilities, in a completely consistent manner, making it easy to learn and easy to compile and execute efficiently.

3.4 Inconsistent Treatment of Character Data

Fortran 88's lack of support for variable size character strings or arrays is puzzling. The facilities which are provided seem stunted and non-intuitive, but this may arise from requirements for compatibility with older dialects of the language. In any event, it makes the language difficult to use for text applications such as editors, compilers, and so on. In APL, character arrays are treated on a par with numeric arrays – all structural and selection verbs operate in a semantically identical manner on all array types.

3.5 Performance and Parallelism

Although at first glance, the inconsistencies above would seem to simplify the task of producing efficient code, the converse is true:

Fortran 88 may produce efficient code for arrays of rank 0, 1, and 2, but that generated for arrays of rank 3 or higher suffers from the same problems as those of FORTRAN 77 – DO Loops are still required.

Of course, one could embed a MATMUL call in the inner loop, but that would remain sub-optimal:

- Only the newest compilers vectorize or optimize over function calls.
- The semantic content of the algorithm is obscured, making the program harder to understand.
- Code complexity and volume grows, increasing the probability of introducing an error, and making maintenance or enhancement more difficult.
- Parallelization which is trivially exploited in the APL expression `b+.×c` is only applicable in a granular fashion in most FORTRAN compilers – on a DO basis, or within the MATMUL. APL can trivially distribute the entire computation over as many processors as are available, because it possesses more knowledge about the total computation than FORTRAN does.

In SHARP APL[Ber81] we took advantage of APL's consistent notation to implement the CDC Star Algorithm[Gri73] on inner products of many types, on arrays of any rank. The Star Algorithm reduces storage traffic, by re-ordering the problem so that each left argument element is only fetched once, and applied to the right argument and result in stride-1 fashion (along the ravel, in APL terms), scalar-vector. Similar knowledge about data types and functions allowed us to design a version of Boolean inner product which exploited the minimal, 32-bit parallelism available in the System/360, achieving a speedup factor of 1000 over the old algorithm. It is doubtful if most application programmers would, or should, on a routine basis, subject themselves to the level of coding complexity this required, but it came for free to all users of that APL system.

Furthermore, we introduced code which made loop ordering, prefetch, and other run-time decisions based on the actual shapes and types of the arrays involved, and the working storage available at the time, which meant that even simple matrix products ran about 2.5-3 times faster than VS FORTRAN.

3.6 Storage Management

Fortran 88 introduced ALLOCATE, a primitive form of explicit storage management. FORTRAN 77 doesn't support dynamic storage management, providing static storage only. The semantics of ALLOCATE associate a specified name with the newly-allocated storage for an array of specified shape. For example, ALLOCATE (X(N)) allocates an N-element list, giving it the name X.

The problems with names and ALLOCATE are the following:

- Inability to create recursive data structures, lists, and trees by allocating successive nodes of the structures.

- Inability to easily alter the size of an array.

- Bolting the name into the semantics of ALLOCATE violates a major principle of functional programming.

Consider an array whose elements are formed by successive catenations, such as a queue. FORTRAN 77 would require static allocation of an array large enough to hold the "largest possible" result, a scalar indicating the current number of entries in the array, and, for the faint-hearted, explicit code to perform array bounds checking in case of array overflow.

In Fortran 88, one obvious approach would be to allocate an empty array, then catenate to it. Catenation makes the array larger, and therefore, a new array must be allocated. But Fortran 88 prohibits allocation of a new X when X is already allocated, so typical code for catenation might have to include a temporary array, and look like this:

```
ALLOCATE (TEMP((SIZE(N)+SIZE(X)))
TEMP = X // N
DEALLOCATE (X)
ALLOCATE (X(SIZE(TEMP)))
X = TEMP
DEALLOCATE (TEMP)
```

In APL, the same expression would be written as:

```
x←x,n
```

Under the covers, APL is probably implicitly performing the same operations as the above Fortran 88 code is doing explicitly, but:

- The work *is* happening under the covers. Hence, the programmer need not be concerned with the details of how it is implemented. Furthermore, if a compiler writer is able to improve the algorithm used, the code will run better, with no changes at the source level.

- APL is fewer characters to type, therefore less work to write. In the above example, there are more lines of Fortran 88 code than there are characters of APL! The conciseness of APL expresses the algorithm in a "chunk" of such size that our brains can treat it as a single unit.

- Assuming that error rates are proportional to code volume, the APL expression is more likely to be correct than Fortran 88. The conciseness gives the expression a clarity which is masked in Fortran 88. The APL expression can be seen at a glance to be correct, whereas the Fortran 88 code is opaque, and can mask errors in the code. For example, the first line of the example has unbalanced parentheses.

- Increased semantic content makes APL easier to optimize than Fortran 88. The APL expression is a single verb – catenation, expressed by the comma – and a single assignment whose target is the same as one argument to catenate.

4 Portability

Fortran 88 may be viewed as a high-level assembler language. As such, it tempts programmers to make architecture-specific optimizations to code in order to improve performance on a specific hardware platform. Worse yet, this approach is even encouraged by the suppliers of hardware and software, who suggest that:

> This activity of rewriting a program to enable vector computation is an important and productive activity[DSK85].
>
> In many cases, a loop... should be replaced with a call to an optimized routine in the $SCILIB library[CRA86].
>
> In order to obtain high performance from the IBM 3090 Vector Facility, we must modify the way in which we construct our numerical method programs[Sam88].

These people ignore the problems engendered by tinkering code in order to make it run better on one specific machine:

- Reduced portability
- Reduced maintainability.
- Reduced algorithmic clarity.

If they were using a language which was suitable to the problem, the performance issues could, by and large, be left to the compiler writers to solve, as they should be, rather than placing a *future* burden on the application writer who may late have to move the application to yet another computing platform.

Moving *within* an architecture (scalar to vector on the same machine) is bad, but moving *between* architectures can be a nightmare:

A researcher at a U.S national laboratory has a very large model of the earth's climate, originally written in FORTRAN for a Cray supercomputer. The model will require several compute-years using the fastest currently available machines. This restricts the researcher to running the model in segments on whatever machine is available in any available time slots. Recently, a hypercube-architected machine became available for use as well as the Cray. The Cray has a few very high speed processors. By contrast, the hypercube machine is a moderately parallel machine, with 1024 medium speed processors. Fortran 88 coding techniques to achieve maximum efficiency on the two architectures are radically different, and at odds with each other. This leaves the researcher in a quandary: Either suffer poor performance on one of the two machines involved, or write and maintain two distinct models, coded specifically for each architecture.

The first approach negates the effectiveness of using a supercomputer; the second requires significantly more human effort to be placed on program maintenance and development. Furthermore, the second technique is much more liable to result in the introduction of program faults. How can the two models be proven to be identical?

Software vendors who wish to have their products run efficiently on a wide range of architectures have this problem in spades.

Writing at a high semantic level hides these dependencies, and yet offers a good degree of performance, by giving the compiler more information to work with regarding the actual computation to be performed.

5 Conciseness

Brevity facilitates reasoning[Ive79].

The ability to express an algorithm in a concise, straightforward manner is key to effective use of computers – coding is faster, comprehension is easier, and the probability of program faults is reduced. Conciseness is beneficial to compilers, in terms of the amount of information available for making decisions about code generation, optimization, and scheduling on parallel systems. Fortran 88's verbosity makes it a weak contender for an effective programming tool in the 1990's.

Verbose programming languages create problems for compilers. Basic blocks – the straight-line pieces of code which are the basis for optimizers, tend to be small, limiting the efficacy of optimization. Concise languages, by their very nature, create larger basic blocks, and, by hiding the details of the computation, give the compiler a leg up.

Verbose languages are also a maintenance nightmare. Compare the loom model previously shown to its equivalent in BASIC. Think about how hard it would be to extend or change it, or about what changes might be made to improve the algorithm. Conciseness improves our ability to understand what a program does – the key to correct maintenance.

6 Summary

The utility of Fortran 88 is limited by the imagination of the language designer, rather than by the imagination of the user. Consistently defined, array-oriented, concise adverbial languages offer significantly greater flexibility, ease of maintenance, and power of expression.[5]

These languages also allow effective exploitation of SIMD, MIMD, and vector hardware. They enhance our thought processes, by making it easier to think about algorithms, without having to think about computers. Fortran 88 is a poor tool of thought, and remains a language which cannot fully exploit array thinking, nor the multicomputers which are now appearing on the market.

7 If You're So Smart...

If APL is so wonderful, why hasn't it become a more popular computing language? Looking back, there are a number of reasons for this:

[5]My paper *Ergonomics and Language Design*, in press, goes into more detail on these issues, but largely ignores Fortran 88.

- Character set
- Poor performance
- Lack of iterative control structures
- Isolation of environment
- Storage requirements.

The APL character set represented a major psychological and technological barrier to the spread of APL to new computing platforms in the last two decades. Recently, the availability of bit-mapped displays, allowing the display of meaningful symbols such as garbage cans, has removed much of this barrier. However, many text editors and other utilities still limit the user to the 128 element ASCII character set, so the barrier still exists. For this reason, new dialects of APL[6] have eliminated the requirement for special character sets.

A second barrier was performance, caused partially by the interpretive overhead of APL systems. Compilers [Ber90b, BBJM90, Bud83, CNS89, Wie85] for APL dialects are starting to appear, and changes in the APL language are occuring – changes such as adoption of lexical scoping rules, which dramatically simplify compilation and improve execution speed, but which make little difference in the writing of practical applications. Performance is very promising, especially considering the relative amount of work which has gone into these compilers compared to Fortran compilers.

Although the adverbs of APL perform much of the work of traditional control structures such as DO, APL is missing a construct corresponding to WHILE, which is desirable for applications which are inherently iterative. Such constructs also simplify the job of optimizing code, and make program maintenance easier. New dialects of APL are beginning to deal with these issues. IF/THEN/ELSE, and CASE statements can be dealt with directly via *function arrays*[Ber84].

The isolation of APL made it difficult to combine programs written in APL with those written in other languages. Accessing data located outside the workspace was difficult. The workspace concept, although now common in other environments, was too effective in isolating people from the machine – they could not use the tools they were familiar with, such as text editors with APL. One goal of J and compiler efforts is to integrate APL into other environments as just another computing tool, rather than a poor cousin or arrogant uncle.

[6]See Appendix A: Examples in APL and J.

Finally, the storage requirements for an APL interpreter were rather large on the computers of the 1970s and 1980s – running the smallest application required several hundred kilobytes of storage. This is obviously no longer a serious problem, given today's storage costs and densities.

APL language designers and implementors have realized that APL cannot be successful in a vacuum, and that a language must be totally integrated into the computing environment. APL language, interpreter, and compiler design work is being directed at the above problems, and new dialects are expected to compete favorably with other languages, particularly in the realm of massively parallel computers, where the expressive power of APL far exceeds that of other languages.

8 Acknowledgements

I wish to thank Elena Anzalone, Jim Lucas, and Ray Polivka for their helpful criticisms of this work. Their comments led to significant revisions, which substantially improved the paper.

References

[BB93] Robert Bernecky and Paul Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 33 Major St., Toronto, Canada, 2nd edition, 1993.

[BBJM90] Robert Bernecky, Charles Brenner, Stephen B. Jaffe, and George P. Moeckel. ACORN: APL to C on real numbers. *ACM SIGAPL Quote Quad*, 20(4):40–49, July 1990.

[Ber81] Paul C. Berry. Enhancements in new release of SHARP APL. *I.P. Sharp Newsletter*, 9(4), July 1981.

[Ber84] Robert Bernecky. Function arrays. *ACM SIGAPL Quote Quad*, 14(4):53–56, June 1984.

[Ber86] Robert Bernecky. APL: A prototyping language. *ACM SIGAPL Quote Quad*, 16(4):221–228, July 1986.

[Ber87] Robert Bernecky. An introduction to function rank. *ACM SIGAPL Quote Quad*, 18(2):39–43, December 1987.

[Ber90a] Robert Bernecky. Compiling APL. In Lenore M.R. Mullin, Michael Jenkins, Gaétan Hains, Robert Bernecky, and Guang Gao, editors, *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1990.

[Ber90b] Robert Bernecky. Portability and performance. In *Cray User Group Meeting*, April 1990.

[Ber90c] Michael J.A. Berry. Adverbial programming. *ACM SIGPLAN Notices*, 25(8), August 1990.

[Bud83] Timothy A. Budd. An APL compiler for the UNIX timesharing system. *ACM SIGAPL Quote Quad*, 13(3), March 1983.

[Cam89] Lloyd W. Campbell, editor. *Fortran 88: A Proposed Revision of FORTRAN 77*. ISO/IEC JTC1/SC22/WG5-N357, March 1989.

[Chi86] Wai-Mee Ching. An APL/370 compiler and some performance comparisons with APL interpreter and FORTRAN. *ACM SIGAPL Quote Quad*, 16(4):143–147, July 1986.

[CNS89] Wai-Mee Ching, Rick Nelson, and Nungjane Shi. An empirical study of the performance of the APL370 compiler. *ACM SIGAPL Quote Quad*, 19(4):87–93, August 1989.

[CRA86] CRAY Research, Inc. *FORTRAN (CFT) Reference Manual*, 1986.

[CX87] Wai-Mee Ching and Andrew Xu. A vector code back end of the APL370 compiler on IBM 3090 and some performance comparisons. *ACM SIGAPL Quote Quad*, 18(2):69–76, December 1987.

[DSK85] A.A. Dubrille, R.G. Scarborough, and H.G. Kolsky. How to write good vectorizable fortran. Technical Report Palo Alto Scientific Center Report No. G320-3478, IBM Corporation, 1985.

[Gri73] Mike Grimm, 1973. Private communication regarding CDC STAR100 APL Project.

[Hei82] Paul Heiser. A weaving simulator. *BYTE Magazine*, September 1982.

[HIMW90] Roger K.W. Hui, Kenneth E. Iverson, E.E. McDonnell, and Arthur T. Whitney. APL\? *ACM SIGAPL Quote Quad*, 20(4):192–200, August 1990.

[IBM94] IBM. *APL2 Programming: Language Reference*. IBM Corporation, second edition, February 1994. SH20-9227.

[Int84] International Standards Organization. *International Standard for Programming Language APL*, ISO N8485 edition, 1984.

[I.P87] I.P. Sharp Associates Limited. *SAX: SHARP APL/UX User Guide*, 1987.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[Ive79] Kenneth E. Iverson. Notation as a tool of thought. *Communications of the ACM*, 23(8), August 1979.

[Ive87] Kenneth E. Iverson. A dictionary of APL. *ACM SIGAPL Quote Quad*, 18(1), September 1987.

[Kar85] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 1985.

[McI80] Donald B. McIntyre. APL in a liberal arts college. In *APL Users Meeting 1980*. I.P. Sharp Associates Limited, Toronto, 1980.

[MMS90] Hans-Peter Meinzer, K. Meetz, and D. Scheppelmann. Visualization of medical tomography images series. *ACM SIGAPL Quote Quad*, 20(4), August 1990. Paper did not appear in Conference Proceedings.

[Sam88] H. Samukawa. Programming style on the IBM 3090 vector facility considering both performance and flexibility. *IBM Systems Journal*, 27(4), 1988.

[Smi88] B.T. Smith. A review and analysis of Fortran 8x. *SIGNUM*, 23(2), April 1988.

[Wie79] Clark Wiedmann. Steps toward an APL compiler. *ACM SIGAPL Quote Quad*, 9(4–Part 1):321–328, June 1979.

[Wie85] Clark Wiedmann. Efficiency in the APL environment, a full arsenal for attacking CPU hogs. *ACM SIGAPL Quote Quad*, 15(4):77–85, May 1985.

Fortran 88		APL		J	
Elemental	Reduction	Elemental	Reduction	Elemental	Reduction
B+C	SUM(C)	`b+c`	`+/c`	`b+c`	`+/c`
B-C	DO loops	`b−c`	`−/c`	`b-c`	`-/c`
B*C	PRODUCT(C)	`b×c`	`×/c`	`b*c`	`*/c`
MAX(B,C)	MAXVAL(C)	`b⌈c`	`⌈/c`	`b>.c`	`>./c`
B .AND. C	ALL(C)	`b∧c`	`∧/c`	`b*.c`	`*./c`
IF stmts	COUNT(C)	`b+c`	`+/c`	`b+c`	`+/c`

Figure 1: Elemental functions and their associated reductions.

Application	APL	J
Associative search	`x∧.=y`	`x *./..=y`
Inverted associative search	`x∨.≠y`	`x+./..~:y`
Minima of residues for primes	`x⌊.|y`	`x<./..|y`
Transitive closure step on Booleans	`y∨.∧⍉y`	`y+./..*.y`
Minima of maxima	`x⌊.⌈y`	`x<./..>.y`

Figure 2: APL and J variants on inner product.

Fortran 88		APL		J	
Boolean	Integer	Boolean	Integer	Boolean	Integer
ALL(B)	PRODUCT(b)	`∧/b`	`∧/b`	`*./b`	`*./b`
COUNT(B)	SUM(B)	`+/b`	`+/b`	`+/b`	`+/b`
ANY(B)	MAXVAL(B)	`⌈/b` or `∨/b`	`⌈/b`	`>./b or +./b`	`>./b`

Figure 3: Inconsistencies in intrinsic procedures on Boolean-valued data.

Left Rank	Right Rank	Fortran 88	APL	J
0	0	A*B	a×b or a+.×b	a*b or a+/..*b
0	1	DOTPRODUCT((/A/),B)	a+.×b	a+/..*b
1	1	DOTPRODUCT(A,B)	a+.×b	a+/..*b
1	2	MATMUL(A,B)	a+.×b	a+/..*b
2	1	MATMUL(A,B)	a+.×b	a+/..*b
2	2	MATMUL(A,B)	a+.×b	a+/..*b
n>2	n>2	n+1 DO LOOPS	a+.×b	a+/..*b

Figure 4: Inner products for various argument ranks.

A Examples in APL and J

Quoted string removal	`(b⍱≠\b←t='''')/t`	`(b+:~:/\b=.t='''')#t`
Recurrence Relation	`t×+\pay÷t←×\int`	`t*+/\pay%t=.*/\int`
Vector+matrix row	`x+⍤1 1 y`	`x+"1 1 y`
Vector+matrix column	`x+⍤0 1 y`	`x+"0 1 y`
Matrix times tensor plane	`m×⍤2 1 v`	`m*"2 1 v`
Inner product	`x+.×y`	`x+/..*y`
Loom model	`(' *'[t∨.=th])[tr;]`	`tr{|:(t+./..="2 0 th){' *'`

Mailing labels n-up	`s←((×/2⍴⍴m)÷n),n×¯1↑⍴m`
	`t←(((1⍴⍴m)÷n),n,1↓⍴m)⍴m`
	`s⍴0 2 1 3⍉t`
		`s=.((*/2$$m)%n),n*_1{.$m`
		`t=.(((1$$m)%n),n,1}.$m)$m`
		`s$0 2 1 3|:t`

Hilbert Matrix 1	`1÷(⍳n)∘.+¯1+⍳n`	`1%(i.n)+/_1+i.n`
Hilbert Matrix 2	`1÷(⍳n)+⍤1 0,¯1+⍳n`	`1%(i.n)+"1 0 ,_1+i.n`
ADJUSTL	`adjl:(+/∧\ω=' ')⌽ω`	`adjl:(+/*./\y.=' ')|.  y.`
Skip to next blank 1	`rtb:(+/∧\ω≠' ')⌽ω`	`rtb:(+/*./\y.~:'  ')|."1 y.`
Skip to next blank 2	`rtb:(ω⍳⍤1 ' ')⌽ω`	`rtb:(y.i."1 ' ')|."1 y.`
Delete leading zeros	`dl1:(+/∧\ω=0)↓ω`	`dl1:(+/*./\y.=0)}.y.`
Remove multiple blanks 1	`(t⍲¯1↓0,t←s∊' ')/s`	`(t*:_1}.0,t=.s e.'  ')#s`
Remove multiple blanks 2	`(~'  '⍷s)/s`	`(-.'  'E.s)#s`
Catenate first axis	`x⍪y`	`x,y`
Catenate last axis	`x,y`	`x,"1 y`
Catenate axis k from end	`x⍪⍤k y`	`x,"k y`