Idioms and Problem Solving Techniques in APL2

March 25th, 1988

Alan Graham

IBM APL Development
Dept M30/Bldg B231
555 Bailey Avenue,
San Jose, CA 95141
GRAHAM at STLVM20
8-543-3679, 408-463-3679

Introduction

Idioms form an intermediate language, higher level than individual primitives but lower level than subfunctions. When a common task arises (such as sorting a vector of strings), and the programmer knows the idiom that performs that task, (VS[⎕AV⍋⊃VS]), the task is immediately solved using the idiom. An order of magnitude gain in programmer productivity is not uncommon. The use of idioms increases readability, while allowing APL implementers to optimize through idiom recognition.

Since APL is written symbolically, a pattern recognition mode of reading evolves. Like a theme in music, an idiom may contain variations. Idioms are generally written in line, rather than appearing as subfunctions. A single idiom may vary slightly according to context. When a programmer recognizes an idiom, it is inspected for variations. Sometimes the idiom is extended to handle the scalar case, or the empty case, or the case of arbitrary rank. Other times it is specialized.

The more you know about your data the simpler the idiom becomes. For example, starting with the most general form of a phrase and simplifying, the phrase ⊂[(0≠⍴⍴A)/⌈/⍳⍴⍴A]A will take an array of any rank and make an array of vectors, putting items along the last dimension. This idiom works in any origin and for arrays of any rank including scalars. If the scalar case can be eliminated, the phrase becomes ⊂[⌈/⍳⍴⍴A]A . Furthermore, if origin 1 is assumed, ⊂[⍴⍴A]A can be written. Finally, if the array is always a matrix (rank 2) the idiom collapses to ⊂[2]A . The particular form depends on context.

APL2 is a relatively new programming language. New idioms are constantly being discovered and old ones are being refined as progress is made up the learning curve. It is expected that some of these initial idioms will seem naive in light of future progress.

These idioms have been collected together because of their benefits in programmer productivity. Although, machine efficiency was ignored while inventing these idioms, the author has not found any of them to be particularly wasteful in either space or time.

Mini-Applications

Several small problems are posed and solved using the idioms found in the appendix.

Problem 1 - Manipulating Name Lists

When a utility package processes all objects in a workspace, it is desirable to remove its own names from the name list of workspace objects before proceeding.

If a name list is manipulated as a simple character matrix, care must be taken to get the number of columns right. The names that are to be removed from the name list must be reshaped into a matrix where the shorter names are padded to the length of the longest name. This is sometimes performed by a utility function, but then the name of the utility function must also be included in the name list and the function carried around with the overall package. More often, the names to be removed are reshaped in line where the programmer carefully counts spaces. When comparing the two matrices the number of columns of each must be forced to the same size by overtaking. In APL1 you could code

```
⍝ All object names as a matrix
NL←⎕NL 2 3 4
⍝ Matrix of names to be removed
RL←4 8⍴'FUNCTIONDATA    F1      F2      '
⍝ Width of largest matrix
N←(1↓⍴NL)⌈1↓⍴RL
⍝ Force to the same width
NL←((1↑⍴NL),N)↑NL
RL←((1↑⍴RL),N)↑RL
⍝ Compare and Remove
NL←(NL∨.≠⍉RL)⌿NL
```

There is considerable simplification if the name lists are manipulated as vectors of strings. If each vector is forced to contain strings without blanks, then direct comparisons can be performed without worrying about lengths. For example:

```
⍝ All object names as a matrix
NL←⎕NL 2 3 4
⍝ Vector of strings with no blanks
NL←(⊂[1+⎕IO]NL)~¨' '
⍝ Remove private names from list
NL←NL~'FUNCTION' 'DATA' 'F1' 'F2'
```

Not only is there less code, but it is easier to understand. The vector of vectors form of the name list is retained in preference to the simple character matrix form. The idiom *NL*←⊃*NL* can be used to force the vector of vectors back into a matrix.

Problem 2 - Iteration with Each

When a program is written that solves a single case of a problem, it is often necessary to iterate with the program over a range of cases. Consider the problem of timing a vector algorithm over a range of random vectors of various lengths.

The skeleton of a timing function that times a single vector case of length *N* could be as follows.

```
[0]    T←TIME1 N;V;T1;T2
[1]   ⍝ Time operation on N-Item vector
[2]    V←?Nρ1000              ⍝ Random vector
[3]    T1←↑1↓⎕AI              ⍝ Start Time
[4]   ( statement that uses vector V )
[5]    T2←↑1↓⎕AI              ⍝ End Time
[6]    T←T2-T1                ⍝ Elapsed Time
```

Therefore, to time line [4] for a single 100 item vector you would execute `TIME1 100` which would return the CPU time in milliseconds. There is a slight overhead used in the timing mechanism itself. It is assumed that the statement being timed is non-trivial so that the timing overhead is small in comparison.

You would want to time the statement over various values of *N* and repeatedly for the same value of *N*, to observe the effect of CPU load and different random vectors. A natural data structure is a table where one dimension varies and the other dimension does not. An example of such a table is,

```
      N←(1000×⍳7)∘.+4ρ0
      N
1000 1000 1000 1000
2000 2000 2000 2000
3000 3000 3000 3000
4000 4000 4000 4000
5000 5000 5000 5000
6000 6000 6000 6000
7000 7000 7000 7000
```

This matrix could not be given to the timing function directly because `TIME1` expects a single scalar *N* to generatean N-item random vector. The Each (¨) operator can be used to automatically apply `TIME1` to the input array. `TIME1` will be called a total of `×/ρN` times.

```
      T←TIME1¨ N
      T
 37  35  36  36
 76  75  76  75
117 118 118 119
162 162 162 160
207 205 206 206
260 258 261 259
311 313 307 302

      ⌊/T         ⍝ Fastest times?
35 75 117 160 205 258 302

      (+/T)÷4     ⍝ Average times?
36 75.5 118 161.5 206 259.5 308.25
```

All of the details of iteration are subsumed by the Each operator. The program is unaware that it is being called multiple times. The structure of the input array controls the function application process and defines the shape of the result. The emphasis is shifted away from explicit program control through loops and toward creating the appropriate data structure to control application.

Problem 3 - Generalizing for Rank

Often it is possible to solve a problem for a given rank array, but difficult to generalize the code for a higher rank array. Problems naturally fall into working with arrays of a set rank. Iverson (*IVERSON*[2]) has assigned primitive functions a "function rank" and suggested a rank operator whereby a specific function rank can be assigned to any function. In "Principles of APL2" (*BROWN*[1]), Brown defines a rank operator implemented as a defined APL2 operator.

Consider the problem of replacing all occurrences of one string with a new string. This is naturally a vector problem because the lengths of the new and old strings may be of different length requiring stretching or shrinking of the target vector. Given an APL2 function called *REPLACEV* that takes a left argument of a two-item vector of old and new strings and a right argument of the vector in which to do the replacement. The result is a vector with the replacements made. How can this be generalized to replace strings in the rows of a matrix or higher rank array? The function *REPLACE* below is one solution.

```
[0]    Z←OLD_NEW REPLACE A;LAST
[1]   ⍝ Replace OLD with NEW along last axis
[2]    LAST←(0≠⍴⍴A)/⌈/⍳⍴⍴A
[3]    Z←⊃(⊂OLD_NEW) REPLACEV¨ ⊂[LAST]A
```

It is possible that the lengths of the items resulting from *REPLACEV*¨ will vary. The Disclose (⊃) function in *REPLACE*[3] handles this by padding the shorter vectors to the length of the longest. *REPLACE* can be rewritten to handle both the vector case and the higher-rank case as a single function as follows:

```
[0]    Z←OLD_NEW REPLACE A;OLD;NEW;B;I
[1]   ⍝ Replace OLD with NEW along last
[2]    →(1<⍴⍴A)/HIRANK
[3]   ⍝ Vector (LOrank) case
[4]    (OLD NEW)←,¨OLD_NEW
[5]    B←OLD⍷A      ⍝ Find the old string
       (...and so on...)
[10]  HIRANK:
[11]   LAST←(0≠⍴⍴A)/⌈/⍳⍴⍴A
[12]   Z←⊃(⊂OLD_NEW) REPLACE¨ ⊂[LAST]A
```

The technique used in *REPLACE* can be made into a general-purpose defined operator that will apply any vector function to vectors along the last dimension. This operator is a special case of Iverson's proposed Rank operator (*IVERSON*[2]).

```
[0]    Z←L (F last) R;V
[1]   ⍝ Apply FN along last dimension
[2]   ⍝ Build enclose last FN (V)
[3]    Z←'Z←V A' 'Z←⊂[(0≠⍴⍴A)/⌈/⍳⍴⍴A]A'
[4]    ⎕ES(0=↑0⍴⎕FX Z)/1 2    ⍝ FIX OK?
[5]   ⍝ Z←F¨(V R) or  Z←(V L)F¨(V R)
[6]    Z←'Z←⊃',(0≠⎕NC 'L')/'(V L)'
[7]    Z←Z,'F¨(V R)'
[8]   ⍝ (Do-or-Die idiom)
[9]    '⎕ES ⎕ET' ⎕EA Z
```

(Note: My naming convention is to name defined operators using only lower case letters.)

Problem 4 - Generalizing for Depth

Nesting is a natural way of making collections of arrays, especially when the arrays may vary in shape. Some operations may apply to the items (cells) and not to the collections (frames). Consider the problem of deleting leading blanks on a string. How can we generalize this to collections of strings?

The solution for a single string is simply

```
[0]    Z←DLB V
[1]   ⍝ Delete Leading Blanks
[2]    Z←(+/∧\' '=V)↓V.
```

As we have seen, the Each operator can be used to apply *DLB* to all items but it is tedious for the user always to be aware of whether the array is the atomic item of interest or a collection of such items. The problem gets worse when considering collections of collections, collections of those, or when some items are atomic and some are collections. The general solution emerges as follows: When can we use the above *DLB* solution?. When we have a simple string, that is, its depth is less than or equal to one (1≥≡A). If not, the function is recursively applied with the Each operator until a simple string occurs. This general form of *DLB* (called *DLBT*) is

```
[0]    Z←DLBT A
[1]   ⍝ Delete Leading Blanks throughout
[2]    →(1<≡A)/RECUR
[3]    Z←(+/∧\' '=A)↓A←,A
[4]    →0
[5]   RECUR:
[6]    Z←DLBT¨ A
```

This is such a common structure that it can be generalized with a defined operator "*depth*."

```
[0]    Z←(F depth N) A
[1]   ⍝ Apply monadic function at Depth N
[2]    →(N<≡A)/RECUR
[3]    Z←F A
[4]    →0
[5]   RECUR:
[6]    Z←(F depth N)¨ A
```

With the *depth* operator the original *DLB* function can be applied to arrays of any depth by writing (*DLB depth* 1) *ARRAY*. (Note: Redundant parentheses are used to help clarify what is the derived function produced by depth.)

Problem 5 - Producing a Derived Function

Operators in APL2 apply universally to any function; primitive, system, derived, or defined. Quite often, an operator must be applied to a phrase composed of more than one function. The phrase can be put into a defined function and then the operator applied to it, but this interruption distracts the programmer from the problem at hand.

A defined operator can be made that will take a character string expression and produce a derived function representing that expression. The representation that is used here is a simple expression form of direct definition (*IVERSON*[2]). Implementation of the Else-If-Then forms and other elaborations are left as an exercise to the reader. Alpha and omega characters represent the left and right arguments respectively. The importance of the derived function is that it can be used as an operand to another operator. For example, given an array of numeric vectors, each vector can be sorted by:

```
'ω[⍋ω]' dd¨ AV
```

Where 'dd' is a direct definition operator. The derived function is 'ω[⍋ω]' dd , which is applied to each item of *AV*. The direct definition operator is surprisingly easy to code.

```
[0]    Z←L(F dd)R
[1]   ⍝ Apply Direct Definition to args
[2]    (('ω'=F)/F)←⊂' R '  ⍝ 'ω' with ' R '
[3]    F←∊F                ⍝ Simple string
[4]    (('α'=F)/F)←⊂' L '  ⍝ 'α' WITH ' L '
[5]    F←∊F                ⍝ Simple String
[6]    F←'Z←',F            ⍝ Ready to ⍎
[7]    '⎕ES ⎕ET' ⎕EA F     ⍝ Execute
```

The general idea is to replace all occurrences of the single characters 'α' and 'ω' with the names of the actual arguments *L* and *R*, and execute the resulting string. Note the technique of replacing single characters with strings by using Enclose (⊂) to create a scalar. Square pegs can fit into round holes. Enclose is the "universal rounder."

Appendix - APL2 Idiom List

The following is not meant to be a complete APL2 idiom list, but instead a sampling of the more popular phrases. An effort has been made to keep the expressions simple and correct for all arrays within the domain of the expression. Sometimes the scalar (*rank* = 0) case or the empty (*shape* ∊ 0) case doesn't work correctly. Notes are included to indicate the special restrictions. Since it is expected that most of these idioms will be used in-line, as opposed to being made into utility subfunctions, the simpler version of the expression is favored. Occasionally, the more sophisticated or general versions of the idioms are also shown.

An informal naming convention is used; *V* for vector, *A* for array, *M* for matrix, and *A V* for Array of Vectors. I prefer that the expressions are natural, rather than brutally following a rule cast in concrete. Some idioms are named by single words or short phrases to make them easy to talk about them. Some expressions are duplicated and used within other idioms. This was done intentionally to show their different uses.

1. First. The first item of a simple array as a scalar.

   ```
   ↑A
   ```

2. Last. The last item of an array.

   ```
   ↑⌽,A
   ```

3. First Dimension. Size of the first dimension as a simple scalar.

   ```
   ↑⍴A
   ```

4. Last Dimension. Size of the last dimension as a simple scalar.

   ```
   ↑⌽⍴A
   ```

5. Simple? Test for simple array.

   ```
   1≥≡A
   ```

6. Nested? Test for nested array.

   ```
   1<≡A
   ```

7. Item List. Make a simple array into a single item.

   ```
   ⍎(1=≡A)/'A←⊂A'
   ```

8. Item Equal. Non-pervasive equal, sometimes called a string equal for its most common use. Compare items for equality.

   ```
   A≡¨B
   ```

   For example to search for 'NAME' in a vector of vectors:

   ```
   VV≡¨⊂'NAME'
   ```

9. Type. Replace numbers with 0, characters with ' ' (blank). Preserves the shape, rank, and structure of the array. (This was the primitive function Type in the APL2 IUP).

   ```
   ↑0⍴⊂A
   ```

10. Array Type. Type of array: character, numeric, or mixed.

    ```
    I←2⊥0 ' '∊↑0⍴⊂∊A
    ↑I↓'?' 'CHARACTER' 'NUMERIC' 'MIXED'
    ```

    Note: Fails to detect arrays of empty arrays of mixed type as MIXED, such as the two-item vector '' (⍳0) .

11. Array Type. Type of array: character, numeric, or mixed.

    ```
    3 11 ⎕NA 'PFA'
    I←2⊥'AIBJE' 'C'∨.∊¨⊂PFA A
    ↑I↓'?' 'CHARACTER' 'NUMERIC' 'MIXED'
    ```

    Note: Works for all cases.

12. Convert a vector into a one-column matrix.

```
,[⍳0]V
```

13. Convert a matrix into a vector of row vectors.

```
⊂[1+⎕IO]M
```

14. Columns. Convert a matrix into a vector of column vectors.

```
⊂[⎕IO]M
```

15. Split. Convert a rank N array into a rank N-1 array of vectors, taking vectors along the last dimension.

```
⊂[(0≠⍴⍴A)/⌈/⍳⍴⍴A]A
```

or

```
⊂[⌈/⍳⍴⍴A]A    ⍝ 0<⍴⍴A
```

16. Mix. Convert a rank N array of vectors into a rank N + 1 array, putting the vectors along the last dimension.

```
⊃AV
```

17. Row Table. Assemble vectors into rows of a matrix.

```
⊃V1 V2...Vn
```

18. Column Table. Assemble vectors into columns of a matrix.

```
⍉⊃V1 V2...Vn
```

19. General Laminate. Assemble arrays of equal rank into an array of rank one greater, putting their dimensions last.

```
⊃A1 A2...An
```

20. Glue Along. Catenate arrays of equal rank into an array of the same rank, along a single dimension I. All dimensions of each array must match except dimension I. The result is an array where the Ith dimension is the sum of the Ith dimensions of the argument arrays.

```
⊃,[I]/A1 A2...An
```

For example, to make a single N-column matrix out of a list of three N-column matrices.

```
⊃,[⎕IO]/M1 M2 M3
```

21. Quick List. Vertical list of all visible defined functions and operators.

```
,[⍳0]⎕CR¨⊂[1+⎕IO]⎕NL 3 4
```

22. Quick Input. Input N strings into an N-item vector.

```
VS←⍎¨Nρ'⍞'
```

23. "Chipmunk." Select sub-arrays given a set of paths P1, P2, ... Pn.

```
P1 P2 ... Pn⊃¨⊂A
```

24. "Chipmunk with glasses and a toothache." Select multiple sub-arrays given a set of sets of paths.

```
(P11...P1n)...(Pk1...Pkn)⊃¨¨⊂⊂A
```

25. Pair-wise. Apply a dyadic function between corresponding pairs of arrays in two lists.

```
AL1 ... ALn f¨ AR1 ... ARn
```

26. All Right. Apply a dyadic function to a list of arrays as a left argument and a single array right argument.

```
A1 A2 ... An f¨ ⊂A
```

27. All Left. Apply a dyadic function between a list of arrays as a right argument and a single array left argument.

```
(⊂A) f¨ A1 A2 ... An
```

28. Apply Last. Apply a function to vectors along the last dimension and assemble equal rank results at the end.

```
⊃f¨⊂[⌈/⍳⍴⍴A]A
```

or

```
⊃f¨⊂[(0≠⍴⍴A)/⌈/⍳⍴⍴A]A    ⍝ Scalars too
```

29. Word Proof. Given a vector of words and a dictionary vector or words, select the words that are NOT in the dictionary (misspelled words).

```
WL~DICTIONARY
```

30. Cartesian Product. Generate all combinations of pairs from the left and right arguments.

```
(⊂¨AL)∘.,(⊂¨AR)
```

31. Count Occurrences. Count the number of occurrences of each item in a vector V1 found in a vector V2.

```
+/V1∘.≡V2
```

32. Sum By Bucket. Given a vector of categories with possible duplicates, a corresponding vector of counts, and a category vector without duplicates; give a corresponding vector of total counts.

```
COUNTS+.×CATS∘.≡UCATS
```

33. De-Duplicate Row Table. Delete duplicate rows of a matrix. The result is a matrix with possibly fewer rows than the original where each row appears exactly once in the order of the original.

```
V←⊂[1+⎕IO]M
((V⍳V)=⍳⍴V)⌿M
```

34. De-Blank. Delete all blanks in string.

```
V~' '
```

35. Standard Name List. Force string, vector-of-strings, or matrix into a vector-of-strings with no blanks.

```
(,⊂[⎕IO]⍉⊃1/NL)~¨' '
```

This is useful in producing a name list. For example:

```
(,⊂[⎕IO]⍉⊃1/⎕NL 3 4)~¨' '
```

36. Word List. Convert a delimited simple character vector of words into a vector of vectors. (Release 2 only).

```
3 11 ⎕NA 'DAN'        ⍝ only do once
VV←DELIMITERS DAN V
```

37. Convert a vector of strings into a simple vector with one blank between each string.

```
   1↓∊' ',¨VV
or
   1↓¯1↓⍕VV
```

38. Quote. Wrap string in quotes and double-quote internal quotes.

```
 '''',((1+''''=V)/V),''''
```

39. Scalar Extend. Extend all scalar array items to the shape of the non-scalar items assuming scalar compatibility.

```
(⍴¨≡¨/,A)⍴¨A
```

40. Vector Pick. Pick a single item from a vector using an origin zero index. This works the same in either origin.

```
↑I↓V
```

41. If Then Else.

```
   ⍎↑CONDITION↓⌽'then part' 'else part'
or
   ⍎↑CONDITION↓'else part' 'then part'
```

42. Case. Execute the Ith expression.

```
⍎↑I↓0 'case1' ... 'casen'
```

43. Simple Image. Convert a nested array to a simple character array image.

```
⍕A
```

44. Matrix Image. Format any array and yield a simple character matrix result.

```
0 1↓0 ¯1↓⍕⊂(1 1,⍴A)⍴A
```

45. Format and right-justify all columns of a report.

```
1↓[⎕IO]⍕0,[⎕IO]A
```

or

```
1↓[⎕IO]N⍕0,[⎕IO]A     ⍝ NUMERIC N
```

46. Alphabetize Matrix. Sort a simple character matrix alphabetically by rows.

```
M[⎕AV⍋M;]
```

47. Alphabetize List. Sort a vector of strings alphabetically.

```
VV[⎕AV⍋⊃VV]
```

48. Sort Number Lists Sort each simple numeric vector item.

```
(⍋¨VV)⊃¨¨⊂¨VV
```

49. Poor Man's Rhyming Dictionary. Sort a vector of strings in approximate rhyming order.

```
VV[⎕AV⍋⊃⌽¨VV]
```

50. Matrix Number. Attach row numbers to a matrix.

```
(⍳↑⍴M),M
```

51. Execute Along Last. Convert a simple character array representation of numbers into a simple numeric array of non-scalar rank.

```
⊃⍎¨⊂[⌈/⍳⍴⍴A]A←'.',A
```

Note: This will always produce a non-scalar.

52. Event Type for Expression. Get event type of a specific error dynamically. This is meant to be used in immediate execution mode. Although it will work properly in any context, it is unreliable because it is dependent on future extensions. Note: If the execute of the right argument unintentionally produces a *WS FULL* then the result will be 1 3 instead of the intended event type.

```
[..] '⎕ET' ⎕EA '..ERROR..'
```

53. Do Or Die. Report error at defined operation call if expression fails. This is especially useful with a defined operator where the function is blindly applied to its argument(s). The operator has no way of determining if the arguments are appropriate to its function operand, except for trying it.

```
'⎕ES ⎕ET' ⎕EA '..expression to protect..'
```

54. Rank Each. Simple integer array of the rank of each item.

```
↑¨⍴¨⍴¨A
```

55. Raise. Strip off one level of nesting, for example vector of vectors of matrices becomes a longer vector of matrices.

```
↑,/,¨,A
```

56. Structure. Make an array of the same structure as 'A' but consisting of all zeros.

```
A≠A
```

57. Scalar Extend. Assuming all items of the same rank, force all items to the same shape by overtaking.

```
(⌈/ρ¨A)↑¨A
```

58. Replace All Items. Replace all items in A with NEW. The shape of A is unchanged.

```
(,A)←⊂NEW
```

59. Replace Selected Items. Replace selected items corresponding to the 1s in left argument with the corresponding items from NEW. The shape of A is unchanged.

```
((,COMPARE A)/,A)←,NEW
```

Note: For conformability
`+/,COMPARE A ↔ ρ,NEW or 1 ↔ ×/ρNEW`

60. Combine real arrays representing the real and imaginary parts, into a single complex array.

```
R+0J1×I
```

61. Split complex array into real & imaginary. Shape is `2,ρC` .

```
9 11∘.○C
```

62. Join real & imaginary into complex array along the first axis.

```
1 0J1+.×RI
```

63. Split complex array into two-item real & imaginary array. Shape is 2 .

```
9 11○⊂C
```

64. Join real & imaginary two-item vector into single complex array.

```
↑1 0J1+.×RI
```

65. Real from A, imaginary from B.

```
(9○A)+0J1×11○B
```

66. Swap real & imaginary.

```
0J1×+C
```

67. Heterogeneous output can be produced by NOT including the semicolons and making sure all sub-expressions that include functions are parenthesized.

For example:

```
'PRODUCED' N 'ITEMS, TOTALING' (N×K)
```

Note: This form is actually more flexible than the old form of heterogeneous output using semicolons because any item can be an array of any rank, shape, type, or structure.

References

1. Brown, J. A., "The Principles of APL2," IBM Santa Teresa Laboratory Technical Report, TR 03.247, March 1984, pp 86.

2. Iverson, K. E., "A Dictionary of the APL Language," I.P. Sharp Associates, September 1985.

3. Benkard, J. P., "Rank vs Depth," APL83.

Acknowledgements

The following people have contributed to this document: Phil Benkard, Jim Brown, Bill Buscher, Harlan Crowder, Dick Dunbar, Ed Eusebi, John Gerth, Michael Golding, Alan Graham, Lou Groner, George Henry, Gary Logan, Blair Martin, Jon McGrew, Ray Polivka, Dave Rabenhorst, Howard Smith, Karl Soop, Ray Trimble, and probably others. My thanks to all of them.