THE APL IDIOM LIST

Alan J. Perlis

Spencer Rugaber

Research Report #87

April 1977

Publication of this document was supported by a grant from the Mobil Foundation.

TABLE OF CONTENTS

1.0 INTRODUCTION

APL programs have been criticized for being hard to read. On the other hand, the conciseness of the language is a necessary ingredient of its power. In an attempt to understand the processes of reading and creating APL programs, many examples have been studied. It has been found that certain expressions occur repeatedly in varying contexts. For example, the APL expression `W←((V⍳V)=⍳⍴V)/V` is often used to remove duplicate elements from a vector. Such expressions are called "idioms," and this report is a collection of them culled from the literature and from personal experience.

Idioms are characterized by their frequency of occurrence, application-independence, power, and "chunkability" (their ability to act as a constituent in higher order idioms). The key point to be made about them is that, once they are learned, they are thereafter recognized as a unit. No further analysis need be done. This suggests that idioms should be taught early in the APL experience.

To understand programs, one must enter into them. Understanding is discontinuous. Examination is initially puzzlement. Then a flash of insight and a model takes form. The program is studied to verify the model. The insight is more easily attained if the intersection of idioms between writer and reader is large.

No language, APL included, provides the perfect set of representations and functions for every algorithm. A program is a mapping of the perfect set into the available set. Programming is the art of organizing these mappings. Generally we seek to define mappings that are of utility beyond the particular algorithm being studied.

In the case of APL, representations and processing which are simple and require uniform scanning should be added to the language. Note ⍋ and SCAN. Consider the representation of a graph as a 2 by N array of edges. The computation of the set of vertices connected to a subset S of the nodes (transitive closure) should perhaps be a primitive. In this way, the detection of idioms may be useful in guiding future extensions to the language.

Idioms are not restricted to APL, but APL′s conciseness and composability make idioms much more common in that language. FORTRAN, for example, contains idioms such as `I-(I/2)*2` to tell whether I is odd or even, `TEMP=A, A=B, B=TEMP,` for exchanging two values, and `A(I,J)=(I/J)*(J/I),` inside of a loop, for making A an identity matrix. LISP, with its highly regular syntax, is typically taught by the presentation of idiomatic examples. In Irons′ extensible systems programming language, IMP, the expression (S←S-1)=Ø => {actions}, is the most common way of programming V, Dijkstra′s semaphore synchronization operation.

2.Ø <u>EXTENSIBILITY</u>

This list of APL idioms was compiled by the authors in the hope that it might act as the first step in the collection of a large body of APL programming knowledge. As such, suggestions, corrections, and additions are solicited.

3.Ø SOURCES

Material for this report has come from the authors and from the following additional sources.

1. APL ⍞.

2. Proceedings of the APL annual conferences (1-7).

3. On-line library of VS-APL on the resident IBM/37Ø system.

4. Raymond P. Polivka and Sandra Pakin, APL: The Language and Its Usage, Prentice-Hall, 1975.

5. Ned Irons, Michael Condry, Terry Miller, and many other people in the Yale University, Department of Computer Science.

4.Ø CATEGORIES

A mild attempt has been made to organize the following material into categories. Sections include PREDICATES, MINI-OPERATIONS, IDENTITIES, SCAN, TEXT PROCESSING, etc. In some cases, a short discussion is also given.

5.Ø ANNOTATED EXAMPLES

1. REMDUP or Remove duplicates: `W←((V⍳V)=⍳⍴V)/V`.

   W is to be set to a copy of *V* with duplicates removed. To demonstrate this, consider the *K*th element of *V*. There are two possibilities. If this is the first occurrence of the *K*th element, that is, if there is no *J* less than *K* such that `V[J]=V[K]`, then `V⍳V[K]` will be *K*. `(⍳⍴V)[K]` will also be *K*. Thus the *K*th element of the compressor will be 1 indicating that `V[K]` should be included in *W*. If, on the other hand, there exists a *J* less than *K* such that `V[J]=V[K]` then `V⍳V[K]` will be *J*. `(⍳⍴V)[K]` will still be *K*. Because *K* does not equal *J*, the *K*th element of the compressor will be 0 and `V[K]` will not be included in *W*. Hence, for each *K*, `1≤K≤⍴V`, `V[K]` is included in *W* iff it is the first (leftmost) occurrence of its value in *V*. *W* will contain each of the different elements of *V* and it will contain no duplicates. Therefore, *W* is *V* with duplicates removed.

   Another variant of this idiom is: `W←(1 1⍉<\V∘.=V)/V`. The `<\` leaves only the first (leftmost) 1 in any row. The `1 1⍉` looks at the main diagonal. A 1 there indicates that this is the first occurrence of the corresponding value thus far in *V*. This idiom directly generalizes to higher dimensions such as the rank-three case:

```
W←(1 1⍉<\∧/1 3 3 2⍉A∧.=⍉A)⌿A.
```

In situations where the length of *V* is large compared to its span (the difference between its largest and smallest elements), the following idiom can be used: `(R∊V)/R←(⌊/V) TO ⌈/V` (where *TO* is described in idiom #4 below).

2. EXPAND:

```
D←(1↓ρA)⌈ρB
A←(((1↑ρA),D)↑A),[1]D↑B.
```

A is an "old" array. *B* is a new entry to be catenated along the first axis of *A*. *B* must be of rank one less than *A*. If either *A* or *B* is larger than the other along any axis, then the length of that axis must be expanded to make them match. TAKE is used to accomplish this.

3. Bar Graph: `W←V∘.≥⍳⌈/V.`

V is a vector of non-negative integers. A boolean matrix is to be created such that the *I*th row contains `V[I]` consecutive 1s, starting in the leftmost position. The above idiom accomplishes this.

As another application, suppose *V*[*I*] is the length of the *I*th word of a string. It is desired to create a table (matrix) with one word from the string per line. Then (,*W*) is an expansion mask that, when applied to the string (with blanks removed) and reshaped to have ⍴*V* rows, will be the desired table. This entire idiom is called *MAKEARRAY*:

```
∇ Z←MAKEARRAY S;A;L;Y;H
  A←(' '=S,' ')/⍳1+⍴S
  L←¯1+A-0,¯1↓A
  Y←(S≠' ')/S
  H←L∘.≥⍳⌈/L
  Z←(⍴H)⍴(,H)\Y ∇
```

A contains the positions of blanks in *S*. It is used to compute the lengths of the words. These lengths are saved in *L*. *Y* is *S* with blanks removed. *H* is the bar graph idiom described above. It is ravelled and used to expand *Y*. The result is then reshaped into the desired word list.

To actually create a bar graph, the following idiom may be used: `G←' ⎕'[1+(⌽⍳⌈/V)∘.≤V]`. Here the boolean array is used to select either a blank or a ⎕ to create the graph.

4. *TO*: `V←A,A+(×B-A)×⍳|B-A`.

TO makes *V* a vector of integers from *A* to *B*. It works for any integers *A* and *B*.

5. Blank removal:

   a. Eliminate leading blanks: `W←(∨\S≠' ')/S`.
      The `∨\` of a boolean vector turns on all bits that are to the right of the first 1. Therefore, only those consecutive blanks at the start of the string are eliminated.

   b. Eliminate trailing blanks: `W←(⌽∨\⌽S≠' ')/S`. This idiom is the same as (a) applied to the reversal of the string. Another way of accomplishing this uses DECODE to, in effect, `∨\` from the right end:
      `W←(¯1+(S=' ')⊥1)↓S`.

   c. Multiple blanks reduced to a single blank:
      `W←(A∨1↓1⌽1,A←S≠' ')/S`. If either `S[I]` or `S[I+1]` are non-blank, then the compressor will have a 1 in the *I*th position. Only if both are blank, will a 0 appear in the compressor indicating that a blank can be removed.

6. Character-number conversions:

   a. Compute the number of digits needed to express a number: `D←⌊1+10⍟N` for positive integers or
      `D←⌊1+10⍟|N+N=0` for all integers.

b. Convert a number into a vector: V←(Dρ10)⊤N (where D is computed in (a)).

c. Convert an integer into a character string: S←⎕D[1+V] (where V is computed in (b)).

d. Convert a string of digits, represented as characters, into a vector: V←¯1+⎕D⍳S.

e. Encode a vector of positive integers as an integer: I←(10*⌊1+10⍟⌈/V)⊥V.

f. Make an array in which each row contains the character representation of one element of the input vector V: M←⎕D[1+⍉((⌊1+10⍟⌈/V)ρ10)⊤V].

7. AMTOAL and ALTOAM:

Let A be a multi-dimensioned array, stored in linear memory in row-major order. If the memory addresses of some entries are given in M, and if a list of subscript positions, S, for those same elements is to be computed, then the following idiom can be used: S←1+(ρA)⊤M-1. Going the other way converts subscripts into ravel order positions. This idiom is: M←1+(ρA)⊥S-1.
Viewed from another perspective, this is similar to the problem of converting a directed graph, AM, represented in adjacency matrix form, into a list of its arcs (AMTOAL): AL←1+(ρAM)⊤¯1+(,AM)/⍳ρ,AM.

Going in the other direction is nearly as easy

```
(ALTOAM):     AM←(A,A)⍴(⍳A×A)∊1+(A,A←⌈/,AL)⊥AL-1.
```

6.0 PREDICATES

The APL language can be used to make assertions about APL programs. The inclusion of logical vectors and operators in the language, in effect, provides the expressibility of the first order predicate calculus. In particular, "for all" is expressed by ∧/ and "there exists" with ∨/. This section gives further evidence of this power.

1. Example of input specifications given in APL. This problem comes from the article "An illustration of current ideas on the derivation of correctness proofs and correct programs," D. Gries, IEEE Transactions on Software Engineering, Volume SE-2, Number 4, December, 1976. The problem is to justify a line of text given the following input parameters.

   Z is the current line number.

   N is the number of words on the line, each word separated by exactly one blank.

   S is the number of extra blanks at the end of the line.

   B on input is a vector of starting positions of words on the line.

   B on output is a vector of starting positions of words on the justified line.

Even numbered lines are to have extra blanks separating words toward the left end of the line, odd numbered lines toward the right. The input specifications for this problem can be stated by the following APL expression: `∧/(T>0),T=⌊T←B,Z,¯1+N,S` where *T* is a vector containing column positions, line number, number of words on the line minus one, and number of blanks terminating the line minus one. The specifications state that all of these numbers must be integers greater than zero.

A solution to this problem is given by the following APL function, annotated below:

```
∇ JUSTIFY ;A;M;F;G
  →0×⍳0≥M←N-1
  A←S+M
  C←⌊A÷M
  F←M↑((A-C×M)⍴1),M⍴0
  G←(M⍴C)+(F×~2|Z)+(2|Z)×⌽F
  B←(1+B-⍳N)++\0,G ∇
```

The function is executed only if *N* is at least one, otherwise the input line is already justified. *A* is the total number of blanks on the line. *C* is the minimum number of blanks that will separate any adjacent pair of words. *F* indicates how the unallocated blanks should be distributed among the other words, depending on the parity of *Z*. Together they form *G*, the actual number of blanks separating adjacent words. Finally, *B* is computed by taking the old value of *B*, subtracting the number of blanks to the left of each word in the input string, and adding the number to the left in the output string. SCAN is used to compute this running count.

2. Example of APL used to express a proof. To prove: the expression `R←(2=+/0=(⍳N)∘.|⍳N)/⍳N` sets *R* to be a vector whose components are all of the primes less than or equal to *N*. Proof:

   a. Let `C←2=+/0=(⍳N)∘.|⍳N`.

   b. *C* is a boolean vector of length *N*. It compresses ⍳*N*. Hence, *R* contains only elements from ⍳*N*.

   c. For some *J*, 1≤*J*≤*N*, consider *C*[*J*]. Then

```
C[J] is (2=+/0=(⍳N)∘.|⍳N)[J]
     is 2=+/0=(⍳N)∘.|J
     is 2=+/0=(⍳N)|J
     is 2=+/0=(⍳J)|J
```

   d. *C*[*J*]=1 iff *J* has precisely two exact divisors from ⍳*J*, that is, if *J* is prime.

   e. *C*[*J*]=1 selects *J* in *C*/⍳*N*. Hence, *R* contains primes ≤*N*.

   f. Conversely, consider any prime *P*, 2≤*P*≤*N*. Then *P*∊⍳*N* and 2=+/0=(⍳*N*)∘.|*P* is 1.

   g. Thus, *C*[*P*]=1 and *P*∊*R*.

3. Is *X* an integer? *X*=⌊*X*.

4. Is *A* numeric, as opposed to character? 0=0\0⍴,*A*.

5. Is *L* made up entirely of elements of *V*? ∧/*L*∊*V*.

6. Is *M* a symmetric matrix? ∧/,*M*=⍉*M*.

7. Are all of the elements of *V* unique? ∧/(*V*⍳*V*)=⍳⍴*V*.

8. Is *V* nondecreasing? ∧/(⍋*V*)=⍳⍴*V*.

9. Are two equal-length vectors equal? ∧/*V*=*W* or *V*∧.=*W*.

10. Does the vector *V* contain any elements from *L*? ∨/*L*∊*V*.

11. Are any rows of the matrix *M* duplicated?

    ∧/1=+⌿*M*∧.=⍉*M*.

12. Is the string *S* free from repeated spaces?

    ∧/*B*∨1↓1⌽1,*B*←1,*S*≠' '.

13. Is the vector *X* a permutation of the vector *Y*?

    X[⍋*X*]∧.=*Y*[⍋*Y*].

14. Is the Boolean vector, *V*, either all 1's or all 0's?

```
0=(⍴V)|+/V or
∧/∨/0 1∘.=V or
∧⌿V=1↑V or
(∧/V)∨~∨/V or
∧/V=1⌽V or
(⌈/V)=⌊/V or
∧/1=V⍳V or
∧/V÷∨/V or
~∧/V∊~V or
≠/1 0∊V.
```

15. Are two entities (of any sort) identical?

```
∇ Z←X EQUALS Y
  Z←0
  →0×⍳(⍴⍴X)≠⍴⍴Y
  →0×⍳∨/(⍴X)≠⍴Y
  Z←∧/,X=Y ∇
```

7.Ø SCAN

The SCAN operator is a powerful device in APL for avoiding loops. Some of its uses are described below.

1. +\ : Running sum. (See, for example, PREDICATES #1 - *JUSTIFY*.)

2. ⌈\ : Progressive maxima.

3. ⌊\ : Progressive minima.

4. -\ : Generate the sequence 1 1 2 2 ... *N N* with |-\⍳2×*N*.

SCANs of Logical Arrays

5. ∨\ : Turn on all 0s after (to the right of) the first 1.

6. <\ : Leave only the first (leftmost) 1 turned on. (See ANNOTATED EXAMPLES #1 - other variants.)

7. ≤\ : Leave only the first 0 turned off.

8. ≠\B : Create a vector of running even parity on *B*.
Also, convert reflected Gray code to binary.

9. ∧\\ : Turn off all elements after the first 0.

8.0 TEXT PROCESSING IDIOMS

Because APL does not contain a complete complement of text processing operators, the frequency of occurrence of this application has necessitated the development of a library of text processing idioms. *MAKEARRAY*, *EXPAND*, and the blank removal idioms have already been described in the section ANNOTATED EXAMPLES. Others are given below.

1. Pattern matching: Return the starting positions in the string *B* of all occurrences of the substring *A*:

   ```
   Z←(¯1↓∧⌿(¯1+⍳⍴A)⌽(A∘.=B),0)/⍳⍴B.
   ```

2. Left justify a word list: `(+/∧\L=' ')⌽L.`

3. Right justify a word list: `(1-(L=' ')⊥1)⌽L.`

4. Replace '*'s by ' 's:

   ```
   A←B≠'*'
   B←A\A/B.
   ```

5. Filter out blank rows from a word list:

   ```
   M←(L∧.≠' ')⌿L.
   ```

6. Alphabetically sort a word list: `M←L[⍋27⊥⍉¯1+⎕A⍳L;].`

7. Sort a word list by length of word: `M←L[⍋L+.≠' ';].`

8. Remove duplicate words from a word list:

```
M←((Q⍳Q)=⍳⍴Q←2⊥L∧.=⍉L)⌿L.
```

9. Find the number of occurrences of word *W* in list *L*:

```
N←+/L∧.=W.
```

10. Create a text array from user input:

```
      ∇ T←ENTERTEXT ;LINE
        T←0 0⍴''
LOOP:   →0×⍳0=⍴LINE←,⍞
        T←T EXPAND LINE
        → LOOP ∇
```

9.Ø PHRASES

A few APL idioms are so commonly used that, when reading an APL program aloud, they are typically replaced by an English word or phrase. Examples of these idioms are given below.

1. If -- conditionally branch to line *L* if condition *C* is true: →*L*×⍳*C* or →*C*/*L*.

2. Positions -- of elements of *V* in *W*: (*W*∊*V*)/⍳⍴*W*.

3. Round -- to the nearest integer (for positive real numbers, *R*): ⌊.5+*R*.

4. Sort -- up or down: *V*[⍋*V*] or *V*[⍒*V*].

5. Indices -- into a vector: ⍳⍴*V*.

6. Last -- element of a vector: *V*[⍴*V*]
   -- row of a matrix: *M*[(⍴*M*)[1];]
   -- column of a matrix: *M*[;(⍴*M*)[2]].

7. All -- (see PREDICATES): ∧/.

8. **There exists -- (see PREDICATES):** ∨/.

9. Ordinality -- of a vector: ⍋⍋*V*.

10. Average -- of a vector: (+/*V*)÷1⌈⍴*V*.

10.0 TEMPLATES

A template is a pattern that can be used in various situations for creating useful constructs. Some examples are given below, where α stands for a class of appropriate operators or idioms.

1. (ρ*A*)α,*A*: Ravel *A*, operate on it, then put it back together again.

2. (*B*×*C*)+*D*×~*C*: Avoid conditional branching of control by constructing a vector with two types of components, *B* if the condition is true, and *D* otherwise. (See, for example, PREDICATES #1 - *JUSTIFY*, line 5.)

3. *A*⍉*B*∘.α*C*: Use outer product to compute a multi-dimensioned array of values, and then use diagonal transpose to get rid of unneeded elements.

11.0 IDENTITIES

Because APL is closer to a mathematical formalism than most other programming languages, mathematical properties of APL programs are more apparent. As an example of this phenomenon, several mathematical identities expressed in APL are given here.

1. *-*A* is ÷**A*.

2. **A*+*B* is (**A*)××*B*.

3. -⌈*A* is ⌊-*A*.

4. ~*A*∧*B* is (~*A*)∨~*B*.

5. -÷*A* is ÷-*A*.

6. -*A*+*B* is (-*A*)+-*B*.

7. *N*×*A*+*B* is (*N*×*A*)+*N*×*B*.

8. ×/⍴*A* is ⍴,*A*.

9. `V[A]B` is `(⍴A)⍴V[,A]`.

10. `A⍉B⍉C` is `A[B]⍉C`.

12.0 EXTENSIONS

Some APL vector operators do not extend to higher dimensioned arrays. Typical ways of achieving the same effect using idioms are given below.

1. Epsilon: If W is a word and L is a compatible word list then ∨/L∧.=⍉W is a simple way to get the effect of epsilon on a rank two array. W could as well be rank two, itself.

2. Compression: Given a boolean array A and a compatible array B, it is desired to create a resultant array C, with the same number of rows, such that C[I;] is the same as A[I;]/B[I;], possibly padded with blanks or zeros. This effect can be accomplished with the following idiom:

```
H←⌈/+/A
C←((1↑ρB),H)ρ(,(+/A)∘.≥⍳H)\(,A)/,B.
```

3. Primitive scalar operations: For example, to multiply each row of the matrix M by the compatible vector V, two approaches are possible. First, C←M×(ρM)ρV. Alternatively, C←1 2 1⍉M∘.×V.

4. Dyadic iota: To look up an entry, E, in a table, T, the

following idiom can be used:

```
A←(ρE)⌈1↓ρT
Z←((((1↑ρT),A)↑T)∧.=A↑E)⍳1.
```

To achieve the effect of dyadic iota working on two compatible tables, B⍳A, the following approach is useful: `C←(<\(A∧.=⍉B),1)⌈.×⍳1+1↑ρB`

13.0 CODING TRICKS

Several miscellaneous APL programming tricks are given below.

1. To accomplish the effect of (A×C)÷B×D, ÷/ can be used: ÷/A,B,C,D.

2. To sort a vector in either ascending or descending order, depending on whether B is +1 or ¯1: W←V[⍋V×B].

3. To achieve the effect of A[I;J;J]←B[I;J] for all I and J (as long as ⍴A <u>is</u> ⍴1 2 2⍉A):

```
C[,1 2 2⍉(⍴A)⍴⍳C←,A]←,B
A←(⍴A)⍴C.
```

4. To achieve the effect of A[I;J;K]←B[J;I]+C[I;J]×D[K;J;I] for all I and J: A←2 1 1 2 3 2 1⍉B∘.+C∘.×D.

14.0 MINI-OPERATIONS

Some short APL idioms act as additions to the set of primitive operators. They suggest potential extensions to the language.

1. Find the position of the left-most 0 of a boolean vector: P←1++/∧\V.

2. Remove elements satisfying a given condition, such as being non-zero: G←(G≠0)/G.

3. Convert a set of positive integers into a mask: M←(⍳⌈/V)∊V.

4. Count how many occurrences there are of each of the different elements of a vector:
```
Y←REMDUP V
N←+/Y∘.=V.
```

5. Find where the 1s occur in a boolean vector: P←(+/B)↑⍒B or P←B/⍳⍴B.

6. Locate the maximum and minimum elements of a vector: V⍳⌈/V and V⍳⌊/V.

7. Find the position of the first occurrence of any element of *W* in *V*: `P←⌊/V⍳W`.

8. Extend a transitive binary relation: `B←B∨.∧B`.

9. Isolate the fractional part of a real number: `F←1||N`.

10. Separate a real number into integer and fractional parts: `V←0 1⊤N`.

11. Create an arithmetic progression (*N* elements with difference *D* and first element *S*+*D*): `V←S+D×⍳N`.

12. Find the number of occurrences of scalar *S* in vector *V*: `N←+/S=V` or `S+.=V`.

13. Compute the original of a sum scanned vector: `W←V-0,¯1↓V`.

14. Simulate an integer CASE statement branch: `→I⌽L1,L2,L3`.

15. Simulate a multi-way conditional branch: `→(C1,C2,C3)/L1,L2,L3`.

16. Encode a boolean vector into an integer: `I←2⊥V`.

17. Create an identity matrix: `M←(⍳N)∘.=⍳N`.

18. Create an upper-triangular matrix: `M←(⍳N)∘.≤⍳N`.

19. Create a lower-triangular matrix: `M←(⍳N)∘.≥⍳N`.

20. Align the diagonals of a matrix into columns (with wrap-around): `N←(¯1+⍳⍴M)⌽M`.

21. Invert a permutation: `V←⍋P`.

22. Form the transitive closure of a relation:

```
∇ T←TC G
  →∨/,(G←G∨G∨.∧G)≠T←G ∇.
```

23. Change all 0′s to N′s in the vector V: `V←V+N×V=0`.

24. Evaluate a polynomial with coefficients C at point X: `A←(X*⌽¯1+⍳⍴C)+.×C`. Better yet: `A←X⊥C`.

15.Ø OTHER OPERATIONS

Other, more extensive, operation-like idioms are given below.

1. ROWSORT -- Sort the rows of a matrix independently:

```
C←⍋,A
Z←(ρA)ρ((,A)[C])[⍋(,⍉(⌽ρA)ρι1↑ρA)[C]].
```

   Or, alternatively:

```
R←ρA
X←(⍋,A)-⎕IO
Z←Rρ(,A)[⎕IO+X[⍋⌊X÷¯1↑R]]
```

2. Make *A* the same rank as *C* without changing its contents: `A←(((ρρC)-ρρA)ρ1)ρA)ρA.`

3. What are the "upper-left-hand-corners" in array *C* of places where *A* can fit, where *C* is of arbitrary rank and *A* is the same rank as *C* (AMTOAL is defined in ANNOTATED EXAMPLES - #7): `B←(ρC)ρ(1+(ρC)-ρA)∧.≥ AMTOAL (ρC)ρ1.`

4. Map a vector, *V*, uniformly into *N* buckets (*R* is a vector of bucket numbers for the corresponding elements of *V*): `R←+/(A×N÷⌈/A←V-⌊V)∘.≥¯1+ιN.`

5. Merge S1 and S2 under control of boolean vector B:

```
S1←B\S1
S1[(~B)/⍳ρB]←S2.
```

Or, alternatively:

```
(S1,S2)[⍒B]←S1,S2.
```

6. Put the string (or vector), B, into the string, A, at position N: `S←(A,B)[⍋(⍳ρA),(ρB)ρN].`

7. Depth of parenthesization: `D←+\(S='(')-0,¯1↓S=')'.`

8. Create a truth table of order N: `T←⍉(Nρ2)⊤¯1+⍳2*N.`

9. Column indices of first occurrences of E's in rows of M: `P←1++/∧\M≠E.`

10. Move all of a set of points into the first quadrant: `D←1 2 1 2⍉D∘.-⌊/D.`

11. Find the number of elements common to two vectors. Do not count duplications: `N←+/A∊B.`

12. Gap Opener: X is a vector. P is a vector of indices into X. G is a vector the same length as X, containing

non-negative integers. The object is to open up gaps at X[P[I]] of size G[I]:

```
X←((⍳(⍴X)++/G)∊(⍳⍴X)+(+\0,G)[1++/(⍳⍴X)∘.>,P])\X.
```

16.Ø APPLICATIONS

Some short expressions, peculiar to specific applications, are given below.

1. Round the number *N* to *P* places: *R*←(10⋆-*P*)×⌊.5+*N*×10⋆*P*.

2. Convert degrees to radians: *R*←*D*×○÷180.

3. Convert radians to degrees: *D*←*R*×180÷○1.

4. Compute the value of principal *P*, computed at interest rate *R*, for *N* periods: *V*←*P*×(1+*R*)⋆*N*.

5. Compute the effective rate of interest, given the nominal rate *NR*: *ER*←(1+*NR*÷*N*)⋆*N*.

6. Compute the limit of the nominal rate *NR* when continuously compounded: *ER*←⋆*NR*.

7. Compute the binomial coefficients of (*X*+*Y*)⋆*N*: *V*←*N*!0,⍳*N*.

8. Compute the distances among a set of points in two-space where the points are represented by a vector of

```
coordinates:    R←(+/(1 3 2 3⍉P∘.-P)*2)*.5.
```

17.Ø SUBROUTINES, MACROS, AND IDIOMS

Idioms are neither macros nor subroutines. Because of their flexability, they would require too many arguments to be implemented by macros. Using subroutines would require a higher degree of generality than the problem might warrant. These points are illustrated by the following four functions which make use of the same idiom, but in different ways.

1. Compute the third-order determinant of a matrix:

```
∇ Z←DET A
  Z←-/+/×/[2](2 3ρ0 1 2 0 2 1)⌽A,[.5]A ∇
```

2. Determine whether a pair of line segments intersect. The array *A* contains four rows and two columns. Each row gives the coordinates of a point. Each pair of rows determines a line.

```
∇ Z←INTRSCT A;C;D
  C←0 ¯1 0↓(⍉3 4ρ0,⍳3)⌽[2]4 4 3ρ1,A
  D←∧(2 1 3⍉4 2 3ρ0 1 2 0 2 1)⌽C,[.5]C
  Z←∧/0≥×/2 2ρ1⌽-⌿+/×/[3]D ∇
```

3. Determine whether a polygon is convex. *A* is an array of vertices, no three of which are co-linear.

```
∇ Z←CVX A;N;Q;P
  N←1↑ρA
  Q←N|(ιN)∘.+0,ι2
  P←(1,A)[Q+N×0=Q;]
  Z←0=N|+/0≤-⌿+/×/[3](2 1 3⍉(N,2 3)ρ0 1 2 0 2 1)⌽P,[.5]P
∇
```

4. Determine whether a point is inside of a convex polygon. *H* is the point in question and *A* has the same form as in #3.

```
∇ Z←H INSIDE A;N;Q;P;S;D
  N←1↑ρA
  Q←N|(ιN)∘.+0,ι2
  P←(1,A)[Q+N×0=Q;]
  S←P
  S[;3;]←(N,3)ρ1,H
  D←1 3 2 4⍉(2,N,2 3)ρ0 1 2 0 2 1
  Z←∧/0≤×⌿-/[2]+/×/[4]D⌽(S,[.5]S),[.5]P,[.5]P ∇
```

18.Ø SUBSCRIPTION

APL's subscription operator is more general than that of any other programming language. It is possible, for example, to subscript a vector by an appropriate array of arbitrary rank. The power of this operator is demonstrated by the following examples.

1. Print out the string *S* with large characters (where *C* is a boolean, rank three array, having one (7×5) character per page):

```
A←' ',⎕A
I←,A⍳S
Q←2 1 3 2⍉CHARS[I;;]∘.×I
T←(A,'?')[Q+Q=0]
Z←(7,×/1↓⍴T)⍴T
```

2. Create a checkerboard:

```
X←2 3 5⍴⍉15 2⍴'\ '
Y←0 ¯1↓8 9⍴2 1
Z←((⍴Y)×1↓⍴X)⍴1 3 2 4⍉X[Y;;]
```

3. *WALLPAPER*:

```
∇ Z←H WALL W;U;X;Y
  U←W[H[1]?⍴W]
  X←2 3 1⍉(H[2],H[2],H[1])⍴U
  Y←?(¯2↑H)⍴H[1]
  Z←((⍴Y)×1↓⍴X)⍴1 3 2 4⍉X[Y;;]
```

4. Instead of using *Q*, as in the idioms #3 and #4 of the last section, an array *B* can be generated from *A* (*B* is

made of *N K* by *K* arrays of all *K* consecutive rows of *A*.
⍴*A* is *N* by *K*.)   *B*←*A*[(¯1+⍳*K*)⊖⍉(*K*,*N*)⍴⍳*N*;].

19.Ø PROBLEMS

The useful application of idioms to the teaching of APL can be seen in the solutions to the classroom problems given below. All of the problems require that explicit looping be avoided.

1. The Symbol Table Update Problem: *A* is a vector of integers (the symbol table), *X* is an integer (the key), and *B* is a vector of integers such that ⍴*B* is ⍴*A* (the usage count list). If *X* is in *A* at position *I*, then *B*[*I*] should be incremented. Otherwise, *A* should be extended with *X*, and *B* should be extended with 1.

SOLUTION

```
A←A,(Y←~X∊A)/X
B←B,(Y/0)+X=A
```

DISCUSSION

Y indicates whether or not *A* and *B* should be extended. If *Y* is 1, then the 1/'s yield *X* and 0, respectively. Otherwise, the compressions yield the empty vector. *X*=*A* is a logical vector indicating that 0 should be added to all of the elements of *B* except the one corresponding to *X*, which will be incremented by 1.

2. Given two input vectors, X and Y, such that ⍴X is ⍴Y, let X be a set of elements and Y be a vector of positive integers. It is desired to produce a vector of Y[1] copies of X[1], Y[2] copies of X[2], and so on.

SOLUTION

V←X[+\(⍳+/Y)∊¯1↓1++\0,Y]

DISCUSSION

Indexing is a powerful operation in APL. Instead of creating the desired number of copies of each element, perhaps using outer product, their indices into X are produced. The output vector is then generated with subscripting.

Incidently, if X is ⍳⍴Y, then another, shorter solution is given by: V←1++/(⍳+/Y)∘.>+\Y. To compute the inverse of this function, that is, to find how many times each element occurs, the following expression can be used: Y←+/(⍳⌈/V)∘.=V.

3. *KWIC*: Given a list of titles (of papers, books, etc.), it is desired to create a "Key Word In Context" (KWIC) index. Each title will occur in the index as many times as it contains key words. The titles are to appear in

alphabetical order by the key word being emphasized on that line. On input, *L* is a list of non-key words to be ignored, and *A* is the list of titles.

SOLUTION

```
∇ Q←L KWIC A;V;G;F;FF;H;K;C;U;Z;R;B;D;S;P;M;J;N
  U←~(' ',A)∊',;.:?! '
  G←U>((1↑⍴A)⍴¯1)⌽U
  F←+/G
  FF←+/F
  H←¯2+(,G)/,(⍴U)⍴⍳1↓⍴U
  K←1++/(⍳FF)∘.>+\F
  V←,U,0
  C←V≠¯1⌽V
  Z←|-/(((+/C)÷2),2)⍴C/⍳⍴V
  R←⌈/Z
  B←(FF,R)⍴(,Z∘.≥⍳R)\(,0 1↓U)/,A
  D←1↓(⍴B)⌈⍴L
  S←∧/((FF,D)↑B)∨.≠⍉((1↑⍴L),D)↑L
  M←S⌿B
  P←⍋((1↓⍴M)⍴27)⊥⍉¯1+⎕A⍳M
  J←(S/H)[P]
  N←J⌽A[(S/K)[P];],((⍴J),5)⍴' '
  Q←(-⌊(1↓⍴A)÷2)⌽(((⍴J),3)⍴' '),N ∇
```

DISCUSSION

U is a logical array that acts as a mask on *A*, indicating punctuation marks. *F* is the number of words on each line, and *FF* is the total number of words in *A*. *H* is a rotator, which, for each word, indicates how far it would have to be rotated to bring it to the start of its title. *K* gives, for each word in *A*, the number of the title in which it appears. *Z* gives the length of each word. These variables are then used in the fashion of *MAKEARRAY* to create a list with one word per line. This is stored in *B*. *S* is a mask for throwing out non-key words. *P* does the sorting by key word. It is used as an index into *H* (with non-key words eliminated) to generate *J*. The rotation of the words is done on the last line.

4. FIND: Given an array *B* of arbitrary rank and an array *A* that will "fit" into *B* (that is, ((⍴⍴B)≥⍴⍴A) and ((-⍴A)↑⍴B)≥⍴A), it is desired to find all of the places in *B* where *A* matches. A match is indicated by giving the coordinates of its "upper left hand corner." This could be called the multi-dimension pattern matching problem.

SOLUTION

```
∇ M←A FIND B;K;U;V;H;C;D
  D←((((⍴⍴B)-⍴⍴A)⍴1),⍴A)⍴A
  K←1+(⍴B)-⍴D
  U←1+K⊤¯1+⍳×/K
  V←U∘.+(⍴D)⊤¯1+⍳⍴,D
  H←1+(⍴B)⊥¯1+1 2 1 3⍉V
  C←(,B)[H]
  M←(C∧.=A)/U ∇
```

DISCUSSION

D is the idiom (OTHER OPERATIONS - #2) for making *A* have the same rank as *B*. *K* is the number of positions on each axis that must be checked during the comparison procedure. *U* converts these positions from linear form to subscript form as in idiom #7 of ANNOTATED EXAMPLES. These are the "upper left hand corner" positions. *V* gives the subscript positions of all of the places that must be compared. These are converted back into linear form in *H*. Each row of *C* is a possible match for *A* in *B*. The actual checking is done on the last line. The subscript positions are returned.

5. Let *S* be a vector of elements. Let *V* be another vector such that ⍴*S* is ⍴*V* and *V*[*I*] tells which set *S*[*I*] belongs to. That is, *V* contains all of the positive integers in the set ⍳⌈/*V*, possibly including duplications. It is desired to return a vector *W* containing the maximum element of each set.

SOLUTION

```
W←S⌈.×V∘.=⍳⌈/V
```

DISCUSSION

V∘.=⍳⌈/*V* is a way of creating masks for the elements of each of the sets. ⌈.× is an idiom useful when the right operand is a logical array. It selects the largest element of each of the designated sets.

6. Find the integers ≤*N* whose squares have decimal expansions that are palindromes (numbers that read the same backward as forward).

SOLUTION

```
Y←⍉((1+⌊10⍟N*2)⍴10)⊤(⍳N)*2
T←Y=(-+/∧\Y=0)⌽⌽Y
Z←(∧/T)/⍳N
```

DISCUSSION

Y computes the decimal expansion on ⍳*N* as given by idiom #6 in ANNOTATED EXAMPLES. *T* compares a number with its reversal. *Z* is the set of answers.

7. Create Ulam's spiral of primes. (See Scientific American, Volume 210, Number 3, March 1964, pages 120-128.) Begin by creating a rectangular array, *Y*, containing the integers ⍳⍴,*Y* in the following configuration:

```
9 2 3
8 1 4
7 6 5
```

Notice how the integers increase as they spiral out from the center. Now, replace all the prime numbers by *'s and all of the composite numbers by blanks. The results is a dramatic presentation of the distribution of prime numbers.

SOLUTION

```
  Y←SPIRAL N
  X←PRIMES Y
  Z←' *'[1+X]

∇ Z←SPIRAL N;A;C;G;D;E
  A←⍳N×2
  C←4| A COPIES |-\A
  G←⌊0.5+N÷2
  D←C+4×0=C
  E←L[;(¯1+N×N)↑D]
  Z←(N,N)⍴⍋N LINEAR 1 1 2⍉(G,G)∘.++\0,E ∇

∇Z←N LINEAR C
  Z←1+(N,N)⊥C-1

  L←2 4⍴¯1 0 1 0 0 1 0 ¯1
```

COPIES is idiom #2 of this section and *PRIMES* is derived from PREDICATES - #2.

DISCUSSION

The full power of using idioms is demonstrated by this example which makes use of no less than seven other idioms from this report.