APL

ADVANCED TECHNIQUES AND UTILITIES

Gary A. Bergquist

APL
Advanced Techniques and Utilities

by Gary A. Bergquist

Zark Incorporated
53 Shenipsit Street
Vernon, Connecticut 06066

APL Advanced Techniques and Utilities
Gary A. Bergquist

Printed in the United States of America

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

INTRODUCTION

It is a mystery that APL is more than 20 years old and there are no APL textbooks which treat the reader as if he or she has some understanding of the language. The introductory APL textbooks available are excellent at accomplishing their objectives. However, they leave the novice APL programmer stranded in the real world. The novice APLer has the tools but not the techniques, the knowledge but not the experience.

This book picks up where introductory APL textbooks leave off. Its goal is to build your experience quickly by exposing you to applications of APL. This is accomplished by presenting real world problems and their APL solutions.

Most sections of the book begin with the presentation of a problem. You should read the problem and formulate a solution to it, given your knowledge of APL. Then read on. The problem is followed by a "good" APL solution. Compare it to yours. If different, learn from the differences.

Each chapter is followed by a set of problems. The purpose of the problems is to confirm your understanding of the material presented in the chapter. You will reinforce that understanding by working on the problems. The solutions to the problems are in the back of the book.

Some of the most valuable material in the book is presented as utility function solutions to problems. Therefore, you should at least scan the problems and solutions after reading each chapter, even if you feel you need no reinforcement.

The book assumes you understand the APL primitive functions. If you encounter a primitive function with which you are unfamiliar, look it up in an introductory APL textbook. Though the book does not assume you are using any particular implementation of APL, it does make specific references to three versions: APL2, APL⋆PLUS, SHARP APL. APL2 is a product of IBM; APL⋆PLUS is the trademark of a product of STSC, Inc.; SHARP APL is the trademark of a product of I. P. Sharp Associates, Ltd.

If you are using a different implementation of APL, the material in the book will still be pertinent. Only one primitive function is assumed which you may not have: replicate (/). If your version of APL supports compression but not replicate, i.e. generates a DOMAIN ERROR on 3/4, you will need to substitute your own replicate function, say REPL, whenever replicate is used. The listing of one such REPL function is included at the end of this Introduction.

Most experienced APL programmers collect a set of their favorite utility functions. These utility functions are used to increase programmer productivity by solving the same problem over and over again. This book contains and describes more than 150 commented utility functions. These functions are available on a floppy disk. See the Postscript at the end of the book.

Notice the workspace ID (WSID:) displayed above the header of the REPL function below. The WSID refers to the name of the workspace in which the REPL function may be found. This convention is used throughout the book. Every function for which a WSID is provided is available on floppy disk.

In several sections of the book, you are asked to imagine extensions to the APL language. Each extension is then implemented via a utility function. These imaginary extensions to APL are intended as instructive and mnemonic devices to help you to quickly understand and remember the definition of the utility function. Please do not misinterpret my intent. I do not seek to have these extensions implemented in the current versions of APL. In some cases, the extensions are half-baked or are inconsistent with existing APL conventions. No matter. Use them as they are intended. Allow yourself to imagine the extension and then view the utility function as the implementation of that extension.

There is much emphasis in the book on efficiency considerations. A chapter is devoted to the topic. In addition, relative efficiencies of alternative algorithms are considered throughout the book. Emphasis is placed upon efficiency because of its importance. At an introductory level of APL, you can concentrate on the conciseness of APL and on its elegance. But in the real world, the practical APL programmer must take a blue collar approach.

Sometimes a concise and elegant algorithm requires much more processing time than a somewhat more complicated algorithm. The difference may be significant enough to make an application feasible or infeasible depending upon the algorithm chosen. However, efficiency does not need to come at the cost of clarity. If comments are used generously and subfunctions used judiciously, you can have the best of both worlds: fast, readable functions.

I solicit comments and suggestions about the topics, presentation and utility functions contained herein. In fact, if you make a suggestion that is incorporated into the text or utility functions of the next edition of the book, you will receive a free copy of the current version of the utility functions.

I wish to acknowledge the efforts of everyone who has contributed to the creation of this book, especially: Bob Richmond, Christine Bell, Daryl Burbank-Schmitt, George Dobbs, Joe Hatfield, Bruce Hitchcock, Roger Hui, Don Lagosz-Sinclair, Lori McNichols, Jack Reynolds, David Routhier, Tapan Roy, Jerry Turner and Andi.

Gary A. Bergquist
Zark Incorporated
53 Shenipsit Street
Vernon, CT 06066

March 1987

```
                                          [WSID: UTILITY]
      ∇ R←B REPL V;I;N;P;T;⎕IO
[1]    ⍝ Emulates the replicate function (R←B/V), where B
[2]    ⍝ may be non-Boolean.  Works for scalar/vector right
[3]    ⍝ argument only.
[4]    ⍝ Branch if right argument not a singleton:
[5]     →(1≠N←×/⍴V)⍴L1
[6]     R←(+/B)⍴V
[7]     →0
[8]    ⍝ Branch if left argument not a singleton:
[9]    L1:→(1≠×/⍴B)⍴L2
[10]    R←,⍉(B,N)⍴V
[11]    →0
[12]   ⍝ Origin 0 logic is simpler:
[13]   L2:⎕IO←0
[14]   ⍝ Flag nonzero replication factors:
[15]    P←×B
[16]   ⍝ Indices into V of values to replicate:
[17]    N←⍴I←P/⍳N
[18]   ⍝ Indices into result of starts of runs:
[19]    T←+\P/B
[20]   ⍝ All-zero vector with length of result:
[21]    R←(¯1↑T)⍴0
[22]   ⍝ Insert 1st differences for subsequent +\:
[23]    R[N⍴0,T]←I-N⍴0,I
[24]   ⍝ Replicate selected elements of V:
[25]    R←V[+\R]
      ∇
```

Chapter 1

LIMBERING UP

The purpose of this chapter is to give you an opportunity to crack your knuckles and stretch you muscles on some APL problems. If you have not used APL for awhile, you will want to spend some time solving these problems. The effort will put your mind in the proper APL orientation to get the most out of the book. If the solutions are different from your own, spend some time studying them. Review any primitive functions with which you are unfamiliar.

If you use APL daily and the problems seem simple to you, skip this chapter altogether. (Solutions on pages 320 to 323).

1. What expression will change the value 645 in the vector AMOUNT to 845?

2. What expression will return the scalar 1 if all elements of the numeric vector PREMS are between 100 and 500, and will return the scalar 0 otherwise?

3. What expression will return the number of elements in the numeric vector WEIGHT which are approximately equal to 24? "Approximately" means the numbers are rounded to the nearest integer before comparing to 24.

4. What expression will return the number of elements in the matrix MAT?

5. Given a variable ANS which represents a numeric scalar (say 56.5), what expression will return the character vector 'ANSWER IS 56.5 YEARS'?

6. What expression will cause the character vector NAME to be catenated as a new row of the character matrix NAMES, assuming the number of elements in NAME is less than or equal to the number of columns in NAMES?

7. What is the effect of the expression, →ρ12 ?

   A. Proceed to the next statement

   B. Proceed to line 12

   C. Proceed to line 1

   D. Exit the function

   E. RANK ERROR

8. Given an integer vector V of length 2×N, construct an N-element vector R by adding the odd elements of V (1, 3, 5, ...) to the even elements (2, 4, 6, ...) times 256.

9. What is the result of ⌊/⍳0 and why?

10. What is the meaning of -\VECTOR?

11. How do you resume execution of a function after an error has occurred and been corrected?

12. What happens when closing function definition mode after editing the header of a suspended function?

13. What is the meaning of ⍋⍋VECTOR?

14.a. What expression describes the shape of the result of A+.×B?

b. If either argument is a scalar?

15.a. What expression describes the shape of the result of A∘.>B?

b. If either argument is a scalar?

16. What system function can be used to determine the amount of CPU time consumed by an APL expression?

17. What expressions may be used to display the character vector PROMPT and to allow the user to enter a response (R) on the same line as, and following, the display of PROMPT?

18.a. What expression will construct a character matrix which will generate N blank lines when displayed?

b. What expression will construct a character vector which will produce the same effect?

19. What expression will cause all variables in the workspace to be erased?

20.a. What would you type before running the function MODEL to cause the computer to stop before executing each of the lines 12 and 14?

b. The following is a partial display of the function MODEL:

```
[11]  T←A÷2
[12]  Q←INTERPOLATE T
[13]  T←T,Q
```

By executing MODEL with the stops specified above, the value of T after the first stop is 6 11 13 and after the second stop is 6 11 10. What corrective action would you take?

21. What expression will return the number of lines in the function CALC?

22. Given two vectors, V1 and V2, an INDEX ERROR is signalled by the last of the following expressions:

```
IND←V1⍳V2
GOOD←IND≤⍴V1
K←(V1×⍳⍴V1)[GOOD/IND]
```

Why?

Chapter 2

BRANCHING AND LOOPING

Branching in APL is a paradox. The definition of the branch function (→) is simple but its application is not. In this chapter we discuss applications of the branching function for: conditional branching, multi-target branching and looping. Finally, the efficiency considerations of looping in APL are discussed.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Branch to the line labeled CALC if the value of the variable X is greater than 5.

TOPIC: Conditional Branching

This problem requires a conditional branch statement. If the condition (X>5) is true, you want the program's flow of execution to proceed to the line labeled CALC. If untrue, you want to continue at the next statement. The conditional branch statement can be expressed in many ways, the following being some of the more typical:

```
1. →(X>5)/CALC       4. →CALC×ιX>5        7. →(X≤5)↓CALC
2. →(5>5)ρCALC       5. →CALC⌈ιX>5        8. →CALC UNLESS X≤5
3. →(X>5)↑CALC       6. →CALC IF X>5
```

While all of these expressions appear to be adequate, there exist subtle differences between them. Algorithm 1 (/) is the most commonly used conditional branching algorithm. Algorithm 2 (ρ) is the fastest. Algorithm 3 (↑) is a graphic complement to algorithm 7 (↓). Algorithms 4 (×ι) and 5 (⌈ι) are more readable than algorithms 1, 2 and 3 since the word "IF" may be read in place of the ×ι or ⌈ι. However, ×ι does not work in index origin 0 and ⌈ι does not allow branching to line 0 (i.e. exiting the function) in origin 1.
~~~~

Algorithm 6 (IF) is the most readable algorithm but requires the existence of the subfunction IF (which is a problem if your function must be self-contained) and is slightly slower than the other algorithms.

Algorithms 7 (↓) and 8 (UNLESS) require the logical negation of the condition and so may be read as "unless". They may be used when the condition is expressed in such a way that the opposite condition requires the branch (e.g. →(M∊MVEC)↓APPEND instead of →(~M∊MVEC)/APPEND). Algorithm 8 (UNLESS) has the same slight disadvantages of algorithm 6 (IF).

Given such a variety of conditional branching algorithms, which should you use? I prefer to use IF and UNLESS when extreme efficiency and self-containment are unnecessary (most of the time). Otherwise, I use ρ and ↓. Whichever algorithm you use, be consistent. APL code is easier to read when conventions are used consistently.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Branch to the line labeled CALC if the value of the variable X is 4, to ENTER if X is 7, to STOP if X is 9 and to LOOP if X is 10.

TOPIC: Multi-target Branching

This problem requires a multi-target branch statement. You want the program's flow of execution to jump to one of four different locations within the program depending upon the value of the variable X. The branch statement can be constructed in many ways, the following being two of the more typical:

```
1. →(X= 4 7 9 10)/CALC,ENTER,STOP,LOOP
2. →(CALC,ENTER,STOP,LOOP)[4 7 9 10 ⍳X]
```

In general, the first algorithm (/) is used unless the branch variable (X) is an index value (1, 2, 3,...) which corresponds to the index of the desired label in the list of labels. Then indexing ([]) is used. For example, if the problem is restated such that X will have the value 1, 2, 3 or 4, then use the expression:

```
→(CALC,ENTER,STOP,LOOP)[X]
```

Notice that the first algorithm (/) actually causes a branch to one of five locations, not four. The fifth location is the next
~~~~

statement and is reached when the condition is untrue for all values supplied (e.g. if X is 6). This bonus branch location is not available when using the second algorithm ([]) since an invalid branch value (e.g. if X is 6) causes an INDEX ERROR. You may avoid the INDEX ERROR by including an additional label to which the branch should take place if there is no match:

```
→(CALC,ENTER,STOP,LOOP,OTHER)[4 7 9 10 ⍳X]
```

The additional label not only prevents an INDEX ERROR but has a possible advantage over the first algorithm (/) in that you are not forced to drop through to the following statement when there is no match.

Both algorithms cause a branch to the label corresponding to the first true condition. In the above example, the conditional expression (X= 4 7 9 10) may have at most one true condition. However, if the expression is rewritten (e.g. X≤4 7 9 10), there may be more than one true condition (e.g. if X←8), in which case only the first true condition will be honored.

When using multi-target branching algorithms, it is easy to overlook the fact that the expression

```
→(L1,L2,L3,L4,L5,L6,L7)[TYPE]
```

requires 8 primitive functions (→,,,,,,[]), not 2 (→[]). The catenation commas are readily dismissed as mere aesthetic punctuation. When extreme efficiency is important, the labels should be catenated once, outside of any loops in which the multi-target branching is being employed:

```
 LABS←L1,L2,L3,L4,L5,L6,L7
   :
   :
LOOP:
   :
   :
 →LABS[TYPE]
   :
   :
 →LOOP
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Construct looping logic which will allow the function PROCESS to be executed N times. The right argument of PROCESS is I, where I is the index number of the iteration (1, 2, 3,...,N).

TOPIC: Looping

The simplest looping logic is:

```
 I←1
LOOP:PROCESS I
 I←I+1
 →LOOP IF I≤N
```

However, this logic breaks down when N=0. The check for completion is not made until after PROCESS has been executed at least once. A safer, but less simple, set of logic is:

```
 I←1
LOOP:→ENDLOOP IF I>N
 PROCESS I
 I←I+1
 →LOOP
ENDLOOP:
```

Naturally, the conditional branch in both sets of logic above may be replaced by any valid form of conditional branching (discussed above).

Notice the looping overhead which takes place within each iteration. In particular, the counter (I) is incremented, a comparison (>) is made and a conditional branch (→ENDLOOP IF ...) is performed. When extreme efficiency is important, some of this overhead can be removed from the looping logic by precalculating the branch labels:

```
 I←1
 →LAB←(NρLOOP),ENDLOOP
LOOP:PROCESS I
 I←I+1
 →LAB[I]
ENDLOOP:
```

Notice that this logic works correctly for the N=0 case. This looping logic is the most efficient possible. However, you should be careful when using it. The shortcoming of this approach is that you must have available workspace for the entire label vector. For example, if you plan to iterate 5000 times, you must have room for a 5001 element integer label vector.

There are two other rather unconventional algorithms for looping which "loop" without branching back. One involves the use of execute (⍎):

```
          ⍎(N≥I←1)/LOOP←'PROCESS I ◇ I←I+1 ◇ ⍎(I≤N)/LOOP'
```

(Note: ◇ is an APL statement separator and is not available in all APL installations.) The other involves the use of a recursive function:

```
          LOOP N
```

where the function LOOP is defined as:

```
              ∇ LOOP I
          [1]   →I↓0
          [2]   PROCESS 1+N-I
          [3]   LOOP I-1
              ∇
```

These two looping algorithms are confusing, inefficient and may cause unexpected complications (e.g. STACK FULL or WS FULL).

Some extended APL systems which support nested arrays have a primitive "iterating" operator named "each" (¨). The problem stated above can be solved via:

```
          PROCESS¨⍳N
```

This expression is significantly simpler and more efficient than the sets of looping logic above. However, it has two drawbacks. The first is the same drawback which the precalculated label vector logic has, namely that you must have available workspace for the entire vector of counter (I) values.

The second drawback is that this expression will not work (as is) if N=0. A DOMAIN ERROR or NONCE ERROR will result because the nested array system does not know what fill value (prototype) to associate with the empty result of PROCESS, should it have a result. For example, if the normal result of PROCESS is a character scalar for a numeric scalar argument, you would expect the result of PROCESS¨⍳0 to be an empty character vector, not an empty numeric vector. The APL system has no way of knowing the nature of the result of PROCESS without executing it at least once.

Viewing the "each" operator as an "iterator" rather than as a parallel processor can quickly lead to expressions which over-kill a problem. For example, consider this file-summarizing logic:

```
 N←1000
 SUM←0
 I←1
LOOP:SUM←SUM++/READ I
 I←I+1
 →LOOP IF I≤N
```

This logic loops through 1000 components of an APL file, reading and summing the 5000-element numeric vectors found in each component. The equivalent nested arrays expression is:

```
 SUM←+/+/¨READ¨⍳N
```

The expression is certainly concise. However, at one point during the execution of the expression, the contents of the entire 5,000,000 element file exist in the workspace as a temporary nested variable. This problem can be circumvented by writing a function SUMREAD which returns the sum of the elements of a specified component:

```
    ∇ R←SUMREAD I
[1]   R←+/READ I
    ∇
```

Then, the expression can be rewritten as:

```
 SUM←+/SUMREAD¨⍳N
```

Now we have only the temporary 1000-element vector result of SUMREAD¨⍳N as extra baggage from the nested arrays approach.

One of the recurring criticisms of APL is its lack of primitive looping constructs. Because of APL's array-handling capabilities, looping is not required as often as in other programming languages. However, despite nested array extensions to APL, the need to loop still exists.

Imagine a looping primitive (⍏) which solves our PROCESS problem as follows:

```
 I⍏ENDLOOP,N
 PROCESS I
 ⍏I
ENDLOOP:
```

The left argument of dyadic loop (⍏) or the right argument of monadic loop is the counter variable. The right argument of dyadic loop may contain from 1 to 4 elements:

[1] the line number (exit line) to which execution will proceed at the completion of the loop;

[2] the number of iterations (infinite if omitted);

[3] the value of the counter variable during the first iteration (⎕IO if omitted);

[4] the amount by which the counter variable is to be incremented or decremented after each iteration (1 if omitted).

The monadic loop function increments the counter variable and branches back to the line immediately following the line containing the dyadic loop function if there are more iterations to perform. Otherwise, the counter variable is erased and the flow of execution proceeds to the exit line.

APL utility functions can be written to approximate this behavior:

```
     →LOOPI ENDLOOP,N
     PROCESS I
     →NEXTI
    ENDLOOP:
```

The definitions of the LOOPI and NEXTI functions follow:

```
                                    [WSID: LOOPI]
     ∇ R←LOOPI LAB
[1]    ⍝ Initializes globals (I,loopi) for looping.
[2]    ⍝ LAB: line to branch to when loop complete,
[3]    ⍝ no. iterations, starting I, increment.
[4]    ⍝ Used in conjunction with NEXTI as:
[5]    ⍝
[6]    ⍝    →LOOPI END,100
[7]    ⍝    PROCESS I
[8]    ⍝    →NEXTI
[9]    ⍝   END:
[10]   ⍝
[11]   ⍝ Default values of 1↓right arg if omitted:
[12]   ⍝  +infinity, ⎕IO, 1
[13]    R←LAB,⌊(⍴,LAB)↓0,(⌊/⍳0),⎕IO,1
[14]   ⍝ Exit if no iterations at all:
[15]    →(R[1+⎕IO]<1)⍴0
[16]    I←R[2+⎕IO]
[17]   ⍝ Top line, exit line, number of iterations,
[18]   ⍝ current I, increment, current counter:
[19]    R←loopi←(1+⎕LC[1+⎕IO]),R,1
     ∇
```

[WSID: LOOP]

```
    ∇ R←NEXTI;⎕IO
[1]  ⍝ Used in conjunction with LOOPI.  Returns line
[2]  ⍝ number for next iteration of loop.  Requires
[3]  ⍝ (may erase) globals: I;loopi.
[4]   ⎕IO←1
[5]  ⍝ Increment I:
[6]   I←loopi[4]←loopi[4]+loopi[5]
[7]   R←loopi[1]
[8]  ⍝ Increment current counter; exit if not done:
[9]   →(loopi[3]≥loopi[6]←loopi[6]+1)⍴0
[10] ⍝ Else return exit line; erase I and loopi:
[11]  R←loopi[2]
[12]  R←R,0⍴⎕EX 'I loopi'
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you want to compute the running balance of your savings account for the last 24 months. You have made a single deposit at the end of each month. DEPOSIT is a 25 element vector of the opening balance and the 24 monthly deposits. RATE is a 24 element vector of the monthly interest rates during this time, expressed as fractions (e.g. .0075 .0081 .0078 ...). You may compute the balances iteratively with the following formula:

BALANCE[T+1] = DEPOSIT[T+1] + (BALANCE[T] × (RATE[T]+1))

where T goes from 2 to 25 and where BALANCE[1]=DEPOSIT[1]. What APL algorithm may be used to compute this stream of cash balances without looping?

TOPIC: When to Loop in APL

Because APL code is interpreted and not compiled, a looping algorithm is generally less efficient than a non-looping algorithm. For example, the expression SUM←+/VECT will be significantly faster than the looping algorithm:
~~~~

```
 I←SUM←0
LOOP:→ENDLOOP IF I≥⍴VECT
 I←I+1
 SUM←SUM+VECT[I]
 →LOOP
ENDLOOP:
```

During each iteration of the loop, every symbol of code is reinterpreted. The addition function is actually a small portion of the processing being performed during the loop.

This situation leads to two conclusions:

1. Avoid looping in APL when possible;
2. When looping is necessary, remove as much code as possible from within the loop.

To illustrate, let us solve the above problem in a casual, looping fashion:

```
                                    [WSID: CASHBAL]
     ∇ BALANCE←RATE CASH1 DEPOSIT;I;N
[1]   ⍝ Returns stream of cash balances for deposits
[2]   ⍝ DEPOSIT and corresponding rates RATE.
[3]    N←⍴DEPOSIT
[4]    BALANCE←(⍴DEPOSIT)⍴0
[5]    BALANCE[1]←DEPOSIT[1]
[6]    I←1
[7]   LOOP:→END IF I≥N
[8]    I←I+1
[9]    BALANCE[I]←DEPOSIT[I]+BALANCE[I-1]×RATE[I-1]+1
[10]   →LOOP
[11]  END:
     ∇
```

Now let's squeeze everything possible from the loop:

```
                                        [WSID: CASHBAL]
     ∇ BALANCE←RATE CASH2 DEPOSIT;B;I;LAB;N
[1]   ⍝ Returns stream of cash balances for deposits
[2]   ⍝ DEPOSIT and corresponding rates RATE.
[3]    N←⍴DEPOSIT
[4]    BALANCE←N⍴0
[5]    BALANCE[1]←B←DEPOSIT[1]
[6]    RATE←RATE+1
[7]    LAB←(N⍴LOOP),END
[8]    →LAB[I←2]
[9]   LOOP:B←BALANCE[I]←DEPOSIT[I]+B×RATE[I-1]
[10]   →LAB[I←I+1]
[11]  END:
     ∇
```

Given these modest modifications, we can expect the function CASH2 to take perhaps 60% to 70% as long to run as CASH1.

Now let's look for a non-looping solution. Let's refer to the elements of 1+RATE as R1, R2, R3,... Let's refer to the elements of DEPOSIT as D1, D2, D3,... Then the elements of BALANCE which we seek may be computed by the following expressions:

```
D1  ,  R1×D1  ,  R1×R2×D1  ,  R1×R2×R3×D1  , ...
        +D2       +R2×D2       +R2×R3×D2
                     +D3          +R3×D3
                                     +D4
```

Our objective is to find some APL expression which will generate this vector. We will accomplish this by performing a series of transformations to these elements until the resulting elements can be easily produced with an APL expression. We will then apply APL expressions which will reverse the transformations.

Let's begin by defining the vector RSCAN:

```
1  ,  R1  ,  R1×R2  ,  R1×R2×R3  , ...
```

We will divide our desired result by RSCAN, giving:

```
             D2           D2  D3             D2  D3        D4
D1  ,  D1+--  ,  D1+--+-----  ,  D1+--+-----+--------  ,...
             R1           R1 R1×R2           R1 R1×R2 R1×R2×R3
```

Take the first difference (V[I+1]-V[I]) of these elements, giving:

```
       D2       D3          D4
D1  ,  --  ,  -----  ,  --------  , ...
       R1     R1×R2     R1×R2×R3
```

Multiply the result by RSCAN, giving DEPOSIT:

```
        D1  ,  D2  ,  D3  ,  D4  , ...
```

Now, undo each transformation in reverse order. Undo the multiplication by RSCAN:

```
              DEPOSIT÷RSCAN
```

To undo the first difference, you must realize that the cumulative sum (+\V) is the inverse of the first difference (V-¯1↓0,V):

```
              +\DEPOSIT÷RSCAN
```

Undo the division by RSCAN:

```
              RSCAN×+\DEPOSIT÷RSCAN
```

There it is. Expressed as a function:

```
                                              [WSID: CASHBAL]
              ∇ BALANCE←RATE CASH3 DEPOSIT;RSCAN
          [1]   ⍝ Returns stream of cash balances for deposits
          [2]   ⍝ DEPOSIT and corresponding rates RATE.
          [3]   ⍝ Performs:
          [4]   ⍝  BALANCE[I]←DEPOSIT[I]+BALANCE[I-1]×RATE[I-1]+1
          [5]    RSCAN←(⍴DEPOSIT)⍴1,×\RATE+1
          [6]    BALANCE←RSCAN×+\DEPOSIT÷RSCAN
              ∇
```

We can expect the function CASH3 to take perhaps 2% to 5% as long to run as CASH1! This significant improvement in speed does come at the cost of clarity. The algorithm in CASH3 screams out for comments, the least of which should be:

```
   ⍝ Performs:  BALANCE[I]←DEPOSIT[I]+BALANCE[I-1]×RATE[I-1]+1
```

After seeing an elegant application of the APL scan functions to perform an inherently iterative function, it is easy to become obsessed with the pursuit of non-looping algorithms. Beware! You may invest a greater value of human time than is saved in machine time. As a further irony, you may find that your elegant and sophisticated non-looping algorithm is slower than a compact looping algorithm.

As a guideline, do not spend your time looking for a non-looping algorithm unless all of the following are true:

1. You suspect one exists;
2. The function is used frequently and the looping algorithm is a "bottleneck" in the function;

3. The loop involves at least 20 iterations;
4. Transplanting all possible logic from within the loop to outside the loop does not give you satisfactory performance.
5. You do not have access to an APL "compiler". For example, STSC provides a product used in conjunction with its mainframe APL*PLUS System product which can be used to compile selected APL functions to improve their execution speed. The compiling process requires a good deal of both programmer and computer time but can produce dramatic efficiency improvements, especially on highly iterative functions.

If you choose, or are forced, to employ a looping algorithm, you may still solve the overall problem in an efficient manner by considering the context in which the loop is performed. To illustrate, let us consider the problem of computing the yield-to-maturity rates for 1000 coupon-bearing bonds.

Given the parameters which define the cash flows of a bond, it is necessary to solve for the yield by a method of successive approximations (looping). The APL solution to this problem is described in detail in the Financial Utilities chapter. For now, let us assume that we have an algorithm which can be used to determine the yield rate (to satisfactory precision) in no more than 10 iterations.

To compute the yield rates for the 1000 bonds, are we compelled to perform 10,000 iterations?

No. We are forced to loop by successive approximation (10 iterations) but we are not forced to loop by bond. To efficiently solve the problem, we may perform the 10 iterations on the parameters of all 1000 bonds at once. After 10 iterations, we will have the 1000 desired yield rates. Computing yield rates by such an "iterative" APL approach is quite efficient. The processing speed will rival or surpass that of any compiled language.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEMS: (Solutions on pages 324 to 326)

1. What are the problems with the following conditional branch expression?

```
→CALC×(X>5)?1
```

2. Assuming index origin 1, what expression will cause a branch to the line labeled NEGATIVE if N is negative, ZERO if N is zero or POSITIVE if N is positive?

3. Assuming index origin 1, write the looping logic which will add together the 100 matrices in file components 11, 14, 17, ..., 308. Assume the existence of the monadic function READ whose right argument is the number of the component to be read and whose explicit result is the matrix stored in that component (e.g. MAT←READ 11). Use each of the following techniques:

   a. Normal APL looping logic (increment, compare, branch);

   b. Precalculated label vector logic;

   c. The hypothetical looping primitive (⍬);

   d. The LOOPI, NEXTI utility functions.

   e. The each (¨) operator.

4. Write non-looping APL logic which is equivalent to the following formula:

```
OPRIN[I] = OPRIN[I-1]-(PMT-RATE×OPRIN[I-1])
```

   for I from 1 to TERM, where OPRIN[0]=LOAN.

5. Write a function CASH4 which uses another approach to perform the same task as that of functions CASH1, CASH2 and CASH3 listed in this chapter. Begin by defining a vector ACCUM:

   R1×R2×R3×... , R2×R3×R4×... , R3×R4×R5×... , ... , 1

   Perform the following transformations on the elements of BALANCE:

   A. Multiply by ACCUM

   B. Take the first difference

   C. Divide by ACCUM

   What is the result? Undo the transformations to construct the new algorithm and use it to write CASH4.

6. The following function WRAPLP modifies its character vector right argument so that it will display in the width (number of characters) specified in the left argument. The modification consists of inserting a newline (carriage return) character in place of the last blank character on each line. In that way, words (groups of contiguous nonblank characters) are not broken from one line to the next. Existing newline characters are left unaltered and are used to separate "sentences" within which the above word-wrap logic takes place. For example:

```
      ⍴TEST
70
      TEST
THIS EXAMPLE IS NOT VERY BIG.
THE FUNCTION WORKS ON LARGE VECTORS TOO.
      15 WRAPLP TEST
THIS EXAMPLE
IS NOT VERY
BIG.
THE FUNCTION
WORKS ON LARGE
VECTORS TOO.
```

   Rewrite the WRAPLP function to eliminate looping where possible.

```
                                             [WSID: WP]
    ∇ R←WID WRAPLP CVEC;⎕IO;BL;BREAK;I;L;LAST;LEN;LIM;NL;S;
      START;TCNL
[1]   ⍝ Wraps text CVEC into lines of length WID
[2]   ⍝ or less by inserting newline characters.
[3]   ⍝ Origin 1:
[4]    ⎕IO←1
[5]   ⍝ Newline character:
[6]    TCNL←⎕TCNL ⍝ APL*PLUS
[7]   ⍝ TCNL←⎕TC[2] ⍝ APL2
[8]   ⍝ TCNL←⎕AV[157] ⍝ SHARP APL
[9]   ⍝ Flag newline characters:
[10]   NL←CVEC=TCNL
[11]  ⍝ Index before start of each sentence:
[12]   START←0,NL/⍳⍴NL
[13]  ⍝ Lengths of sentences (between newlines):
[14]   LEN←¯1+(1↓START,1+⍴CVEC)-START
[15]  ⍝ Flag valid break points (blank followed by nonblank):
[16]   BL←CVEC=' '
[17]   BREAK←BL>1⌽BL
[18]  ⍝ Initialize result from argument:
[19]   R←CVEC
[20]  ⍝ Loop by sentence:
[21]   I←0
[22]   LIM←⍴LEN
[23]  LOOP1:→(LIM<I←I+1)/0
[24]   L←LEN[I]
[25]   S←START[I]
[26]  ⍝ Loop by line within sentence:
[27]  LOOP2:→(L≤WID)/LOOP1
[28]  ⍝ Find last break point within WID chars of line:
[29]   LAST←+/∨\BREAK[S+⌽⍳WID]
[30]  ⍝ Advance start to new break point:
[31]   S←S+LAST
[32]  ⍝ Insert newline:
[33]   R[S]←TCNL
[34]  ⍝ Decrement remaining length:
[35]   L←L-LAST
[36]  ⍝ Repeat:
[37]   →LOOP2
    ∇
```

Chapter 3

COMPUTER EFFICIENCY CONSIDERATIONS

Time is money. The faster an APL function will run, the less it will cost. This is true whether you are using APL on a commercial remote timesharing service or on your dedicated personal computer. In this chapter, we discuss computer efficiency: measuring time consumption and understanding the factors which affect processing efficiency.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Which expression will execute quicker on a 4000 element numeric vector V?

```
1. R←+/V÷⍴V
2. R←(+/V)÷⍴V
```

TOPIC: Timing Alternative Algorithms

The niladic APL system function ⎕AI (accounting information) returns a numeric vector of miscellaneous usage statistics. The meanings of the elements of the result vary among the different implementations of APL. One element (usually the second) measures the amount of processing time (CPU time) consumed since the current APL session began. It is usually expressed in milliseconds, or 60ths of a second or seconds. We will assume in our discussion that the index origin is 1 and that ⎕AI[2] is the measure of processing time.

Timing an algorithm is then a simple matter of checking the "stopwatch" before and after executing the algorithm:

```
TIME1←⎕AI[2]
R←(+/V)÷⍴V
TIME2←⎕AI[2]
USED←TIME2-TIME1
```
~~~~

These expressions should not be executed in immediate execution mode unless they are executed all at once on a single line:

```
TIME1←⎕AI[2] ⋄ R←(+/V)÷⍴V ⋄ TIME2←⎕AI[2] ⋄ USED←TIME2-TIME1
```

If your implementation of APL does not have a statement separator (e.g. ⋄), the expressions should be specified as lines of a function. The reason for avoiding immediate execution mode is that CPU time is being consumed as you are typing each expression. In fact, on a dedicated (i.e. personal) computer, the measure of CPU time is equivalent to the measure of clock time. That is, the value for ⎕AI[2] increases by 60 seconds every minute whether or not APL expressions are being executed. Therefore, the measured time will include typing time.

Let us solve the problem above:

```
      V←4000?4000
      T←⎕AI[2] ⋄ R←+/V÷⍴V ⋄ ⎕AI[2]-T
675
      T←⎕AI[2] ⋄ R←(+/V)÷⍴V ⋄ ⎕AI[2]-T
162
```

From this example, we can begin to see the importance of well placed parentheses. In the first algorithm, the computer performs 4000 divisions and 4000 additions. In the second, it performs 4000 additions and 1 division.

What happens if we time the algorithms again?

```
      T←⎕AI[2] ⋄ R←+/V÷⍴V ⋄ ⎕AI[2]-T
 692
      T←⎕AI[2] ⋄ R←(+/V)÷⍴V ⋄ ⎕AI[2]-T
 155
```

We get the same approximate results but they are not exactly the same. Why? On a multi-user computer, the results will vary primarily because of the varying requirements of other users at the moment of execution. Even on a dedicated computer, the results may vary because of "house-cleaning" operations performed automatically and sporadically by the APL system and because of imprecise clock resolution. Therefore, if the accuracy of your timings is important, you should perform several timings and average the results.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Write a dyadic function TIMER to time the execution of a specified algorithm. The algorithm is provided as an executable character vector right argument. TIMER runs the algorithm N times, where N is the left argument, and returns the average CPU time consumed. For example:

```
          25 TIMER 'R←+/V÷ρV'
     638.44
```

TOPIC: A Utility Function for Timing Algorithms

In writing the TIMER function, we will attempt to isolate the time consumed during the execution of the algorithm. Any time consumed during the overhead of the timing process itself will be deducted. In this way, and by averaging many samples, we can get timing results which are as precise as possible.

There are two methods in APL for executing a character vector expression under program control. The first is to use the execute (⍎) primitive and the second is to construct and execute a local function which has the expression as one of its lines.

The first approach (⍎) is simpler but not as accurate. When a character vector expression is executed, the expression must first be "parsed" so that the APL interpreter may correctly identify variables, APL primitive functions, character constants and numeric constants. It is during this parsing phase that variable names and function names are translated into pointers and addresses, the natural vocabulary of the computer. This parsing takes place when you enter an expression in immediate execution mode or in function definition mode or when you define a function under program control. Therefore, when you execute the expression as the line of a function, it has already been parsed and will execute quicker than if the expression is executed as the argument to the execute (⍎) primitive.

We will therefore use the second method, constructing and executing a local function, to time the specified algorithm. Our task is to define a function local to TIMER under program control which looks something like the following (say, to time R←(+/V)÷ρV):

```
      ∇ ELAPSED←RUN1 N;I
[1]     ELAPSED←⎕AI[2]
[2]     I←0
[3]    LOOP:→(N<I←I+1)ρEND
[4]    DOIT:R←(+/V)÷ρV
[5]     →LOOP
[6]    END:ELAPSED←⎕AI[2]-ELAPSED
      ∇
```

This function will run the specified algorithm (on line [4]) N times, where N is the right argument (e.g. RUN1 25). It will return the amount of elapsed processing time consumed during its execution.

We will define a second local function RUN2 which is identical to RUN1 except for line [4], which is defined to do nothing:

```
[4]  DOIT:
```

We may then execute both RUN1 and RUN2 with the same argument, subtract the results (to eliminate the non-algorithm overhead) and divide by the number of iterations to determine the average processing time for a single execution of the specified algorithm. Therefore, TIMER will look something like:

```
      ∇ R←N TIMER CVEC;RUN1;RUN2
   :
   :   define RUN1 and RUN2
   :
  [7]   R←(RUN1 N)-RUN2 N
  [8]   R←R÷N
      ∇
```

How do we define RUN1 and RUN2? There are two popular methods for defining functions under program control. One is to build a character matrix which "looks" like the function, less the dels (∇) and the bracketed line numbers. Each function line, including the header, occupies exactly one row of the character matrix. Each function line is padded with blanks to have as many characters as the longest function line. Such an array is called the "canonical representation" of the function and may be used as the definition of a new function in the workspace. The function is defined by a system function (⎕FX in APL2, ⎕DEF in APL*PLUS or ⎕FD in SHARP APL).

The second method is to build a character vector which "looks" exactly like the function. The lines of the function are separated by the newline (i.e. carriage return) character. Such an array is called the "visual representation" of the function. The function is defined by a system function (⎕DEF in APL*PLUS or ⎕FD in SHARP APL).

There is one final comment to make before defining the TIMER function. Since the algorithm being timed may involve variables or functions having any valid names, it is possible that these identifiers may coincidentally be the same as the variables local to TIMER and RUN1. We should take some effort to name the local variables so that the chances of a name conflict are minimized. The RUN1 function we will construct will thus look like:

```
     ∇ ⍙E⍙←⍙F⍙ ⍙N⍙;⍙I⍙
[1]    ⍙E⍙←⎕AI[1+⎕IO]
[2]    ⍙I⍙←0
[3]   ⍙L⍙:→(⍙N⍙<⍙I⍙←⍙I⍙+1)⍴⍙Z⍙
[4]   ⍙D⍙:R←(+/V)÷⍴V
[5]    →⍙L⍙
[6]   ⍙Z⍙:⍙E⍙←⎕AI[1+⎕IO]-⍙E⍙
     ∇
```

Let us define the TIMER function using the "visual representation" method. We will leave the "canonical representation" method as an exercise at the end of the chapter. We shall use the monadic ⎕DEF system function to define the function. Its result is the character vector name of the function defined. The niladic system function ⎕TCNL returns a character scalar newline character.

```
                                                  [WSID: TIMING]
     ∇ ⍙R⍙←⍙N⍙ TIMER ⍙C⍙;⍙A⍙;⍙B⍙;⍙F⍙;⍙G⍙;⍙NL⍙
[1]   ⍝ Times the execution of the character vector
[2]   ⍝ ⍙C⍙ by running it ⍙N⍙ times.  Returns a numeric
[3]   ⍝ scalar of the average CPU time consumed per run.
[4]   ⍝
[5]   ⍝ Prepare to build local functions...
[6]   ⍝ Newline character:
[7]    ⍙NL⍙←⎕TCNL ⍝ APL*PLUS
[8]   ⍝ ⍙NL⍙←⎕AV[156+⎕IO] ⍝ SHARP APL
[9]    ⍙A⍙←'∇⍙E⍙←⍙F⍙ ⍙N⍙;⍙I⍙',⍙NL⍙,'[1]⍙E⍙←⎕AI[1+⎕IO]'
[10]   ⍙A⍙←⍙A⍙,⍙NL⍙,'[2]⍙I⍙←0',⍙NL⍙
[11]   ⍙A⍙←⍙A⍙,'[3]⍙L⍙:→(⍙N⍙<⍙I⍙←⍙I⍙+1)⍴⍙Z⍙'
[12]   ⍙A⍙←⍙A⍙,⍙NL⍙,'[4]⍙D⍙:'
[13]   ⍙B⍙←⍙NL⍙,'[5]→⍙L⍙',⍙NL⍙
[14]   ⍙B⍙←⍙B⍙,'[6]⍙Z⍙:⍙E⍙←⎕AI[1+⎕IO]-⍙E⍙∇'
[15]  ⍝
[16]  ⍝
[17]  ⍝ Define local fn ⍙F⍙ to run ⍙C⍙:
[18]   ⍙R⍙←⎕DEF ⍙A⍙,⍙C⍙,⍙B⍙ ⍝ APL*PLUS
[19]  ⍝ ⍙R⍙←3 ⎕FD ⍙A⍙,⍙C⍙,⍙B⍙ ⍝ SHARP APL
[20]  ⍝
[21]  ⍝ Define local fn ⍙G⍙ to run nothing:
[22]   ⍙A⍙[⍙A⍙⍳'F']←'G'
[23]   ⍙R⍙←⎕DEF ⍙A⍙,⍙B⍙ ⍝ APL*PLUS
[24]  ⍝ ⍙R⍙←3 ⎕FD ⍙A⍙,⍙B⍙ ⍝ SHARP APL
[25]  ⍝
[26]  ⍝ Run the functions (disallow negative result):
[27]   ⍙R⍙←0⌈(⍙F⍙ ⍙N⍙)-⍙G⍙ ⍙N⍙
[28]  ⍝ Return the average:
[29]   ⍙R⍙←⍙R⍙÷⍙N⍙
     ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Define a procedure whereby the individual lines of a specified function may be timed so that possible inefficiencies can be quickly located within the function. The ideal end result of such a procedure will be a display like the following (for a 5 line function):

LINE	TIMES RUN	TOTAL CPU	AVG CPU	MIN CPU	MAX CPU
1	3	450	150	122	171
2	3	15	5	3	7
3	153	918	6	3	8
4	150	9,280	62	55	66
5	3	1,065	355	240	380
	312	11,728	38		

TOPIC: Fine-tuning Production Applications for Efficiency

After designing and implementing an application system in APL, you may find that it operates slower than you anticipated. In fact, the system may be so slow or so expensive that it is infeasible to operate. What can you do?

A procedure such as the one suggested in the problem above allows you to examine the functions for bottlenecks. You begin with the highest level cover functions and work your way into suspicious subfunctions. Having identified the major inefficiencies, you are in an ideal position to correct them. A discussion of the causes and cures for some of the inefficiencies encountered in APL is contained later in this chapter.

To aid in our discussion, suppose the function we wish to time is the following:

```
      ∇ MODEL
[1]    SETUP
[2]    LIM←50 ◇ I←0
[3]   LOOP:→END IF LIM<I←I+1
[4]    PROCESS ◇ →LOOP
[5]   END:CLOSE
      ∇
```

To time the lines of this function, we need to click our "stopwatch" at the beginning and end of each line. This suggests the placement of timer functions at the start and end of each line. For example:

```
[1]    START ◇ SETUP ◇ STOP
```

This idea breaks down for lines which involve branches:

```
        [4]    START ⋄ PROCESS ⋄ →LOOP ⋄ STOP
```

In this example, the branch to the line labeled LOOP occurs before the STOP function is executed.

Since functions placed at the end of function lines are not reliably executed (due to branching), we must be satisfied with placing timer functions only at the start of function lines. The timer function must then perform two tasks: to stop the stopwatch for the previous line (which may not be the line directly above the current line) and to start the stopwatch for the current line. If we call our timer function ⍙, we may be able to time the MODEL function above by placing the timer function as follows:

```
          ∇ MODEL
   [1]     ⍙ ⋄ SETUP
   [2]     ⍙ ⋄ LIM←50 ⋄ I←0
   [3]    LOOP:⍙ ⋄ →END IF LIM<I←I+1
   [4]     ⍙ ⋄ PROCESS ⋄ →LOOP
   [5]    END:⍙ ⋄ CLOSE
   [6]     ⍙
          ∇
```

Notice that a new line must be added (line 6) to stop the stopwatch for the last line of the function (line 5).

If your implementation of APL does not support statement separators (e.g. ⋄), you are compelled to insert the timer function on the line before each function line:

```
          ∇ MODEL
   [1]     ⍙
   [2]     SETUP
   [3]     ⍙
   [4]     LIM←50
   [5]     ⍙
   [6]     I←0
   [7]    LOOP:⍙
   [8]     →END IF LIM<I←I+1
   [9]     ⍙
   [10]    PROCESS
   [11]    ⍙
   [12]    →LOOP
   [13]   END:⍙
   [14]    CLOSE
   [15]    ⍙
          ∇
```

It is the job of the timer function (⍙) to do the following:

1. Record ⎕AI[2].

2. Look at a global variable (say ⍙T) which contains the value of ⎕AI[2] when ⍙ was last executed. Subtract this value from the value recorded in step 1. The result is the time consumed by the previously executed line.

3. Look at a global variable (say ⍙L) which contains the number of the line for which ⍙ was last executed. Using this line number and the consumption value computed in step 2, update the global variable accumulation matrix (say ⍙M) which has one row per function line and 4 columns: times run, total CPU, minimum CPU, maximum CPU.

4. Update ⍙L to contain the number of the current line.

5. Update ⍙T to contain the value of ⎕AI[2] at the start of the current line.

To write the timer function (⍙), we will assume the following initial values for the required global variables:

```
⍙L←0
⍙M←(N,4)⍴0 0,(⌊/⍳0),0
```

No initial value is set for ⍙T since ⍙ will not refer to it when it is first executed (on line 1) since ⍙L is 0. The N used in the assignment of ⍙M is the number of lines in the function being timed. The (⌊/⍳0) is used to generate the largest possible number (the identity element for mimimum) for your APL system. We cannot initialize the minimum value to 0 since the 0 will remain as the minimum value. No timing result could be less.

Let us write the timer function (assuming statement separators):

```
                                        [WSID: TIMING]
     ∇ ∆;TIME;USED;⎕IO
[1]    TIME←⎕AI
[2]   ⍝ Records time since last called and resets
[3]   ⍝ stopwatch. Checks the 'stopwatch' before
[4]   ⍝ anything else.
[5]    ⎕IO←1
[6]   ⍝ Branch if first time called:
[7]    →(×∆L)↓L1
[8]   ⍝ Compute time consumed since last called:
[9]    USED←TIME[2]-∆T
[10]  ⍝ Update accumulation matrix:
[11]   ∆M[∆L; 1 2]←∆M[∆L; 1 2]+1,USED
[12]   ∆M[∆L;3]←∆M[∆L;3]⌊USED
[13]   ∆M[∆L;4]←∆M[∆L;4]⌈USED
[14]  ⍝ Update line number:
[15]  L1:∆L←⎕LC[2]
[16]  ⍝ Set ∆L to 0 if bottom of function:
[17]   ∆L←∆L×∆L≤(⍴∆M)[1]
[18]  ⍝ Update 'stopwatch' as last step:
[19]   ∆T←⎕AI[2]
     ∇
```

Notice the use of ⎕LC (line counter) to compute the number of the current line on which ∆ is being called. If your implementation of APL does not support statement separators, and the number of lines in your function has been doubled because of the insertions of ∆, the computation of ∆L would be changed to:

```
∆L←(1+⎕LC[2])÷2
```

Along with the ∆ timer function, we need 3 other functions:

TIME∆DEFINE 'MODEL' — The TIME∆DEFINE function modifies the function named in its character vector right argument by inserting the ∆ timer function before each function line. It also initializes the global variables ∆L and ∆M.

TIME∆DISPLAY — The niladic TIME∆DISPLAY function generates and displays a formatted report of the contents of the global accumulation matrix ∆M.

TIME∆RESET — The niladic TIME∆RESET function resets the global variables ∆L and ∆M to their initial (zeroed out) settings. Then, the function whose lines are being timed may be rerun and the results redisplayed.

The definition of the TIME∆DEFINE function is fairly complex, especially if you attempt to write it without looping. Techniques for writing the function are discussed in a subsequent chapter (Boolean Techniques). The definition of TIME∆DEFINE is a problem at the end of that chapter.

The definition of the TIME∆DISPLAY function is a straightforward formatting problem.

```
                                        [WSID: TIMING]
     ∇ TIME∆DISPLAY;A;B;M;⎕IO
[1]   ⍝ Displays timing data stored in global arrays.
[2]    ⎕←'          TIMES   TOTAL    AVG     MIN     MAX'
[3]    ⎕←' LINE      RUN     CPU     CPU     CPU     CPU'
[4]    ⎕←' ----    -----   -----   -----   -----   -----'
[5]    ⎕IO←1
[6]   ⍝ Squeeze out rows of ⍙M not updated:
[7]    M←⍙M[(⍙M[;1]≠0)/⍳1↑⍴⍙M; 1 1 2 2 3 4]
[8]    A←M[;2]
[9]    B←M[;3]
[10]   M[;1]←⍳⍴A
[11]   M[;4]←B÷A
[12]  ⍝ APL*PLUS, SHARP APL:
[13]   ⎕←'I4,CBI8,X1,4BCK3I8' ⎕FMT M
[14]  ⍝ APL2:
[15]  ⍝ ⎕←('5550 555,559 ',32⍴' 555,559')⍕M[;1 2],1000×M[;3 4
        5]
[16]   ⎕←'          -----   -----   -----'
[17]   A←+/A
[18]   B←+/B
[19]  ⍝ APL*PLUS, SHARP APL:
[20]   ⎕←'CBI12,X1,2BCK3I8' ⎕FMT 1 3 ⍴A,B,B÷A
[21]  ⍝ APL2:
[22]  ⍝ ⎕←('      555,559 ',16⍴' 555,559')⍕A,B,B÷A
     ∇
```

The definition of the TIME∆RESET function is trivial:

```
                                        [WSID: TIMING]
     ∇ TIME∆RESET
[1]    ⍙L←0
[2]    ⍙M←(⍴⍙M)⍴ 0 0 ,(⌊/⍳0),0
     ∇
```

Some final notes on this line-timing procedure:

1. Since TIME∆DEFINE will permanently modify the function being timed, be sure to save a copy of the function before running TIME∆DEFINE on it.

2. Functions timed by this procedure are subject to certain constraints:

   a. Avoid sudden exits (e.g. →0) and insure that the function always exits through its bottom (last line);

   b. Avoid branches to absolute or relative line numbers (e.g. →5 or →⎕LC-1 or →NEXTL) if your implementation of APL does not support statement separators, since TIME∆DEFINE will change the line numbers.

The mainframe implementation of APL*PLUS provides a system function ⎕MF (monitor facility) which enables you to time the individual lines of a function in much the same way as the utility functions above. If you use a mainframe APL*PLUS system, you should read the documentation to learn how to use ⎕MF and its companion utility functions.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: What factors influence the efficiency of a user-defined APL function?

TOPIC: APL Efficiency Considerations

So far in this chapter we have discussed techniques for timing segments of APL code. Using these techniques, you can isolate inefficiencies in existing functions and you can choose between alternate algorithms. But how can you learn to write efficient APL functions in the first place?

You can develop a feel for the efficiencies and inefficiencies of APL.

If you were to go on a timing rampage and time every stitch of APL code in sight, some patterns would begin to emerge. You would begin to anticipate the relative speeds of algorithms without timing them. More important, you would find yourself formulating a mental model of the inner workings of the computer. Multiplication is more painful to the computer than addition and exponentiation more painful than multiplication. Out of sympathy for the machine, you will find yourself writing N+N instead of 2×N, and A×A instead of A*2.

While there is no substitute for such an encounter, below are some of the efficiency considerations which you may want to assimilate.
~~~~

1. Addition (e.g. A+¯2) is faster than subtraction (e.g. A-2) which is faster than multiplication which is faster than division which is faster than exponentiation which is faster than doing it by hand. For example, (×\Nρ2) will typically be faster than (2⋆⍳N).

2. Integers are easier (quicker) to manipulate than fractional (floating point) numbers. An array of integers will generally be stored internally in a more compact form (2 or 4 bytes per element) than an array of floating point numbers (8 bytes per element). Integers can be moved about (e.g. ⌽, ↑, [], /) quicker than floating point numbers and can be more easily operated on computationally (e.g. +, ×, ⊤, |).

3. Boolean arrays (all 0s and 1s) are stored as bits (one-eighth byte per element) in many implementations of APL. On such implementations, operations involving Boolean arrays are either lightening fast or quicksand slow depending upon the implementation. If they are fast, it is because the bits are being manipulated one byte (8 bits) or so at a time or because the CPU is optimized for Boolean operations. If they are slow, it is because each bit has to be yanked from its byte and processed by a CPU which is better suited to working with 4, 8, 16, 32 or 64 bits at a time. Examples of functions so influenced include: ∧/, ∨\, +.∧, /, ≠\.

4. Elements of arrays are stored internally in raveled order. If you picture in your mind's eye this internal "vector" representation of a matrix M, you should appreciate the reason that +/M is somewhat faster then +⌿M. If M is a Boolean matrix (in which each element occupies one-eighth byte) the difference can be dramatic.

5. Execute (⍎) is slower than branching because its argument must be parsed. For example:

```
⍎(I>99)/'M[15;]←A÷B'
```

is slower than:

```
 →(I≤99)ρL1
 M[15;]←A÷B
L1:
```

6. The workspace may be viewed as a chain of bytes. In a clear workspace, all of the bytes are "clean" (unused). As you execute expressions (e.g. A←2 or B←⍳50 or C←A+B), the bytes become occupied by variables. When variables are reassigned (e.g. A←1+A), the old value of the variable is left as so much garbage. The same fate befalls temporary results which are the products of multiple expressions. For example, the expression A←3+2×⍳5 produces the temporary results from ⍳5 and from 2×⍳5. As you proceed, the

workspace becomes cluttered with a mixture of active variables and functions and garbage (unused) variables and functions. The symbol table is used to keep track of the locations of the active objects.

Eventually, the computer will be asked to perform a function for which it cannot find sufficient clean space for the result. At that moment, the CPU will take a break from its APL function execution chores and will perform spring cleaning. All of the valid objects are shuffled back to the beginning of the workspace chain, the symbol table is updated and the remaining bytes are swept clean, ready to be reused. The CPU then resumes its APL function execution chores.

It is because of these occasional workspace cleanups that you may notice seemingly random peaks when doing timings. Logic which requires a lot of temporary storage will tend to be less efficient than that which requires less storage. For example, index assignment tends to be more efficient than catenation reassignment:

```
LOOP:VEC[I]←CRUNCH I
      vs.
LOOP:VEC←VEC,CRUNCH I
```

Consider the storage requirements of the second expression when constructing VEC to be 1000 elements, one element at a time. For example, the catenate (,) function must find space for its 932 element result while the 931 element VEC still exists. Of course, a moment later the 932 element vector is assigned the name VEC and the 931 element vector is left to smolder in the ashes. On the other hand, the index assignment approach creates a 1000 element vector just once and then changes individual elements. Much less data shuffling is involved.

7. Shape (ρ) and reshape (ρ) are the most primitive of primitives. The rank and shape of a variable are included as part of its internal representation. The shape function does not have to count its elements; it simply extracts the shape directly. Shape and reshape are extremely fast. For example, →BρL is faster than →B/L or →B↑L. Also, 1ρρMAT is faster than 1↑ρMAT. To construct a 100 by 100 identity matrix (all zeros, but ones along the diagonal), the expression (ι100)∘.=ι100 may seem simple enough to you but that's because you do not have to perform the 10,000 mindless comparisons. The less intuitive expression 100 100ρ1,100ρ0 is dramatically faster because of its use of reshape.

8. When performing scalar operations, time consumption can be measured with a ruler. Because APL is interpretive, it does not check for syntax errors, value errors or argument conformability until it executes the expression. When working with scalars, the time consumed making these checks tends to dwarf the time consumed performing the desired function. Therefore, when considering the efficiency of APL expressions dealing with scalars, it is more

pertinent to count the number of functions being executed than to dwell on the nature of the functions. Consider the expression:

```
R[I-1]←R[I-1]+(1+T[I-1])÷R[I-1]×G[I-1]*2
```

If we count the functions being performed (counting index assignment as 2 functions), the result is 16. Let us rewrite the expression (assuming T1←1+T):

```
J←I-1
R[J]←R[J]-T1[J]÷R[J]×G[J]*2
```

These expressions have the same effect as the original expression but involve 12 functions instead of 16. We can expect these expressions to run approximately 25% faster.

Let us label the approximate time consumed when performing a function on scalars a "tad". Then the original expression used 16 tads and the second used 12 tads.

It is important to develop the proper perspective on tads. APL is an efficient language and performs tads extremely quickly. A tad is a miniscule unit of time. APL can perform a hundred tads in a blink of the eye. Tads do not become important until you write functions which consume thousands or tens of thousands of tads, i.e. when you loop.

The lesson here is to avoid looping when APL's array handling capabilities can be effectively employed. Your avoidance should not develop into a mania, however. Looping in APL is fast if there are not too many iterations (say, two dozen) or if the number of tads within the loop is not too large. Do what you can to keep the tads from getting into the tens or hundreds of thousands.

9. Get to know the peculiarities of your APL implementation. For example, say you have a 100,000 element Boolean vector BV which contains only five 1s. Further, say you have only 1000 bytes of available workspace. Many implementations of APL will allow you to execute the expression I←BV/⍳⍴BV without producing a WS FULL error message. How can this be when ⍳⍴BV results in an integer vector of 100,000 elements (400,000 bytes or so)?

In one set of implementations, the APL interpreter is clever enough to construe the two symbols /⍳ as a single function. Therefore, the monadic ⍳ is never executed. Instead, the /⍳ "function" scans its Boolean left argument for 1s and returns their indices. In these implementations, it is ironic to find a section of "optimized" code like the following:

```
I←⍳⍴A
IA←A/I
IB←B/I
```

The more conventional expressions will typically be much more efficient since they employ /ι as a single function and do not require as much workspace:

```
IA←A/ιρA
IB←B/ιρB
```

In a second set of implementations, the result of monadic ι is an arithmetic progression vector and is stored internally as a "J vector". Specifically the computer stores only the vector's length, its starting value and its value-to-value increment. When executing the expression I←BV/ιρBV, the compression (/) function works with its "J" vector right argument without ever building the entire index vector.

Expressions like 2×10+ι1000 are extremely fast on APL systems which use J vectors. In this expression, a single addition and a single multiplication are performed to generate the 1000 element J vector result.

If you do not know whether your APL system employs J vectors, try V←ι1E9. If no WS FULL message is generated, you have J vectors.

10. Use "compiled" functions when available. Some vendors of APL provide workspaces of utility functions which are written in machine code rather than in APL. These functions are extremely fast and behave like regular APL utility functions. They may be copied into or erased from your workspace and they can have arguments and results. Take some time to explore available workspaces of utility functions.

In addition, much research has been conducted toward compiling APL code. For example, STSC provides a product used in conjunction with its mainframe APL*PLUS System product which can be used to compile selected APL functions to improve their execution speed. The compiling process requires a good deal of both programmer and computer time but can produce dramatic efficiency improvements when applied to bottleneck functions which consume a large portion of the processing time of an application system.

Some APL systems and some related software products allow you to run non-APL programs from within the APL environment. For example, if you have available a program written in another language (say, C or COBOL) which is very efficient and which performs a desired task, you may be able to invoke the program without ever leaving the APL workspace environment.

A final caveat. This list of computer efficiency considerations can create a distorted perspective. Your primary goal as an APL programmer is not to write APL functions which run fast. Your goal is to get the job done. If getting the job done means writing faster functions, then keep the above efficiency considerations in mind.

In any case, you should also keep in mind these nonefficiency considerations:

1. Your time is more valuable than the computer's. If you find yourself laboring to find a faster algorithm, ask yourself, "Why?" Will the savings in computer time result in a more responsive system which is less frustrating to use and which saves people time? Will the efficiency improvements result in lower computer allocation charges or lower timesharing bills? If you cannot foresee material benefits from your efforts, you are wasting your time.

In general, when writing a function which will perform a one-time task, forget efficiency. Use the code which flows most rapidly from your mind. Get the job done.

2. A readable function is better than a fast one. It is a crime against nature to insert fast, obscure, uncommented code in a production application. Any algorithm which can be understood can be adequately commented. If you do not have the inclination to insert the comments, then do not use the code. By taking a moment to include comments with your efficient algorithm, you will write code which is both fast and readable. Remember, in six months the person who cannot understand your code may be you.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 327 to 329)

1. Write a niladic function COST which will produce a display like the following:

```
        13.15 DOLLARS CONSUMED
        65.12 DOLLARS SINCE SIGNON
```

   The first line of the display will not appear the first time COST is run and will thereafter display the dollars consumed since the prior execution of COST. Assume your CPU charge is 75 cents per unit of ⎕AI[2].
~~~~

2. Write the TIMER function described in this chapter to work with the canonical representation method of local function definition.

3. In the Sorting and Searching chapter, a function CMIOTA is presented for searching through the rows of one character matrix for the location of the rows of a second. The function is designed to use one of two different algorithms depending upon the number of rows in its arguments. Time CMIOTA (as defined in that chapter) for character matrix arguments of 10, 50, 100, 500 and 1000 rows (12 columns) in all combinations (e.g. 50 row left argument and 100 row right argument). Then, change the line

```
L2:→(F≠1)⍴L4
```

to

```
L2:→(F≠1)⍴L5
```

and do all the timings again. The first set of timings uses a sorting algorithm and the second set uses a looping algorithm. Record these numbers. They are required by a problem in the Curve Fitting chapter which determines the constants to be plugged into CMIOTA for automatically choosing the fastest algorithm.

Construct your character matrices such that the rows of your left argument are distinct (or nearly so) and the rows of your right argument are found throughout the left argument. For example:

```
L←50 12⍴⎕AV[?(50×12)⍴256]        (50 row left argument)
R←L[?100⍴1⍴⍴L]                   (100 row right argument)
```

Chapter 4

POSITIONING CHARACTER DATA

Many problems in APL involve the realignment of characters. For example, the title of a report may need to be centered above the body of the report; or a character vector entered by the user may need to have any extraneous blanks deleted from it. In this chapter, we discuss techniques for positioning the nonblank character elements of an array for a variety of different applications.

∿∿∿∿ ∿∿∿∿ ∿∿∿∿ ∿∿∿∿ ∿∿∿∿ ∿∿∿∿ ∿∿∿∿ ∿∿∿∿

PROBLEM: Write the monadic functions DLB, DTB and DEB for deleting leading, trailing and extraneous blanks from a specified character vector.

TOPIC: Removing Extra Blanks from Character Vectors

The DLB, DTB and DEB functions are frequently used when accepting character input or when generating report output. For example, say you have a character matrix MONTHS of month names, left justified, an integer scalar MNO of the current month (1 to 12) and an integer scalar YR of the current year (e.g. 1987). You want to construct a character vector of the current month and year (e.g. 'JUNE 1987'). The following expression will perform the task:

```
(DTB MONTHS[MNO;]),' ',⍕YR
```

Say you want to build a 30 column character matrix NAMES of employee names by prompting for one name at a time. You want each name to be left-justified in the matrix and to contain no extra spaces between the segments of the name. Use the following expression:

```
NAMES←NAMES,[1]30↑DEB,⍞
```

The following functions will perform the desired tasks. Notice that alternative algorithms are included in each function. The relative speed of each of the algorithms depends upon the implementation of APL you use. You may want to time them (as discussed in the Computer Efficiency Considerations chapter) to determine which is fastest for your APL environment.

[WSID: FORMAT]

```
     ∇ R←DLB C
[1]   ⍝ Deletes leading blanks from character vector C.
[2]    R←(∨\C≠' ')/C
[3]   ⍝ R←(+/∧\C=' ')↓C
[4]   ⍝ R←(((C≠' ')⍳1)-⎕IO)↓C
     ∇
```

[WSID: FORMAT]

```
     ∇ R←DTB C
[1]   ⍝ Deletes trailing blanks from character vec C.
[2]    R←(+/∨\' '≠⌽C)⍴C
[3]   ⍝ R←(⌽∨\' '≠⌽C)/C
[4]   ⍝ R←(-+/∧\' '=⌽C)↓C
[5]   ⍝ R←(⎕IO-(' '≠⌽C)⍳1)↓C
[6]   ⍝ R←(1-(C=' ')⊥1)↓C
     ∇
```

[WSID: FORMAT]

```
     ∇ R←DEB C;N
[1]   ⍝ Deletes extraneous (leading, trailing,
[2]   ⍝ contiguous) blanks from character vector C.
[3]    N←C≠' '
[4]    R←(~1↑N)↓(N∨1↓N,0)/C
[5]   ⍝
[6]   ⍝ C←' ',C
[7]   ⍝ N←C≠' '
[8]   ⍝ R←1↓(N∨1⌽N)/C
     ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Write the monadic functions LJUST, CJUST and RJUST for left-justifying, centering and right-justifying the nonblank text within a specified character vector or matrix.

TOPIC: Justifying Nonblank Segments within Character Arrays

The LJUST, CJUST and RJUST functions are useful for constructing report titles and for merging character matrices. For example, to display ACME INC. centered within a width of 75 characters, use the following expression:

```
CJUST 75↑'ACME INC.'
```

As another illustration, say you have two 15 column character matrices of left-justified names, LNAMES and FNAMES. You would like to construct a 32 column character matrix of left-justified names in which the names of LNAMES precede the names of FNAMES and are separated by a comma and a single space (e.g. SMITH, JOHN). Use the following expression:

```
LJUST(RJUST LNAMES),',',' ',FNAMES
```

The following are the definitions of these functions:

```
                                        [WSID: FORMAT]
    ∇ R←LJUST C
[1]   R←(+/∧\C=' ')⌽C
    ∇
```

```
                                        [WSID: FORMAT]
    ∇ R←RJUST C
[1]   R←(+/∨\' '≠⌽C)⌽C
    ∇
```

```
                                        [WSID: FORMAT]
    ∇ R←CJUST C;B
[1]   B←C=' '
[2]   R←(⌈((+/∧\B)-+/∧\⌽B)÷2)⌽C
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Given a 20 column character matrix of employee names (in alphabetical order), construct an 80 column matrix of the names such that the names run down the resulting matrix in 4 "columns". The resulting matrix will have one-fourth (or so) as many rows as the original matrix has.

TOPIC: Restructuring Skinny Matrices into Fat Ones

Let's illustrate this problem on a simple character matrix of first names.

```
        ANNE
        BILL
        CAL
        DOT
        ED
        FRED
        GAIL          →          ANNE   FRED   KEN    RICK
        HAL           →          BILL   GAIL   LISA   VI
        IKE           →          CAL    HAL    MIKE
        JOAN          →          DOT    IKE    NED
        KEN           →          ED     JOAN   PAT
        LISA
        MIKE
        NED
        PAT
        RICK
        VI
```

In this illustration, the initial matrix has 7 columns instead of the specified 20 and the resulting matrix has 28 columns instead of 80. Still, you can see what we want to do. We will solve the problem for this simple 7 column matrix and then modify the solution to work for the specified 20 column matrix.

The brute-force approach to this problem involves breaking the matrix apart into 4 pieces and then sticking them together side-by-side. Assume the name of the character matrix is CMAT. The number of rows in the desired result is computed as:

```
NR←⌈(1↑⍴CMAT)÷4
```

NR is 5 in our illustration.

The pieces can be extracted by using the take (↑) and drop (↓) functions:

```
P1←(NR,7)↑CMAT
P2←(NR,7)↑(NR,0)↓CMAT
P3←(NR,7)↑((2×NR),0)↓CMAT
P4←(NR,7)↑((3×NR),0)↓CMAT
```

In our illustration, P2 is:

```
FRED
GAIL
HAL
IKE
JOAN
```

Notice that the last expression pads P4 at the bottom with blank rows if there are fewer than 4×NR rows in CMAT. The last step catenates the 4 pieces together:

```
R←P1,P2,P3,P4
```

A more elegant solution to this problem involves the use of dyadic transpose. We begin by padding CMAT so that its number of rows is divisible by 4:

```
NR←⌈(1↑⍴CMAT)÷4
CMAT←((4×NR),7)↑CMAT
```

Second, reshape the matrix into a 3 dimensional array:

```
CMAT←(4,NR,7)⍴CMAT
```

In our illustration, CMAT is now:

```
ANNE
BILL
CAL
DOT
ED

FRED
GAIL
HAL
IKE
JOAN

KEN
LISA
MIKE
NED
PAT

RICK
VI
```

Notice that each of the planes in CMAT corresponds to one of the "columns" of names in the desired result.

Third, use dyadic transpose to shuffle the planes and rows so that the shape changes from (4,NR,7) to (NR,4,7). Since the first coordinate (4) becomes the 2nd coordinate, the next (NR) becomes the 1st and the last (7) remains the 3rd, use 2 1 3 as the left argument (or 1 0 2 in origin 0):

```
CMAT←2 1 3⍉CMAT
```

In our illustration, CMAT is now:

```
ANNE
FRED
KEN
RICK

BILL
GAIL
LISA
VI

CAL
HAL
MIKE


DOT
IKE
NED


ED
JOAN
PAT
```

By performing this transpose, the characters of the array (if raveled) are in the same order as those in the desired result (if raveled).

Finally, reshape the array into the desired two-dimensional result:

```
R←(NR,28)⍴CMAT
```

The final solution for the 20 column problem is:

```
NR←⌈(1↑⍴CMAT)÷4
R←(NR,80)⍴2 1 3⍉(4,NR,20)⍴((4×NR),20)↑CMAT
```

The dyadic transpose approach is generally more efficient than the brute-force approach. Its work is performed primarily by the relatively efficient reshape and transpose functions. The brute-force approach makes heavy use of the less efficient take, drop and catenate functions.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write a function TITLES which will return a character matrix of titles which will be displayed at the top of a report. The left argument is an integer scalar of the width of the resulting character matrix (i.e. the width of the report). The right argument is a delimited character vector (e.g. '∩OPERATING STATEMENT∩DEC. 31, 1987∩($000''S)') whose "partitions" each begin with one of the delimiters ⊂ (left-justify), ∩ (center) or ⊃ (right-justify). The result has one row per partition. Each partition is justified within the row according to the delimiter. For example:

```
     HDG←50 TITLES'⊃PAGE 1∩OPERATING STATEMENT∩1987⊂($000s)'
     HDG
                                           PAGE 1
               OPERATING STATEMENT
                       1987
($000s)
```

TOPIC: Delimited Character Vector to Justified Matrix

Let us define the header of the TITLES function:

```
∇ R←WID TITLES CS
```

We will use origin 0 throughout:

```
⎕IO←0
```

Determine which elements of CS are justification symbols:

```
JUST←'⊂∩⊃'⍳CS
BJUST←JUST<3
```

BJUST is a Boolean vector with 1s corresponding to justification symbols.
~~~~

```
JUST←BJUST/JUST
NROWS←ρJUST
```

JUST is an integer vector with one element per justification symbol and whose values indicate which symbol (0: left; 1: center; 2: right). NROWS is the number of titles (delimiters). Determine the length (LEN) of each delimited partition (i.e. each title), excluding the justification symbol:

```
T←BJUST/ιρBJUST
LEN←(1↓T,ρBJUST)-T+1
```

Reduce (truncate) those lengths which exceed the specified width WID:

```
LEN←WID⌊LEN
```

Determine the indices into CS of the characters which start each title (i.e. the character after the justification symbol):

```
ARGSTART←1+T
```

Determine the number of leading blanks required per title to justify it (left, center or right) within the matrix result:

```
LEAD←(JUST≠0)×⌊(WID-LEN)÷1+JUST=1
```

Determine the indices into the raveled matrix result of the characters which start each title:

```
RESSTART←LEAD+WID×ιNROWS
```

Initialize the result to have the correct number of characters but to be all-blank and raveled:

```
R←(NROWS×WID)ρ' '
```

All that remains is to extract the titles from CD (we know the starting positions, ARGSTART, and the lengths, LEN, of each title), insert them into R (we know the starting positions, RESSTART, and the lengths, LEN) and then reshape R to the proper shape. If there was but one title, we could do the following:

```
R[RESSTART+ιLEN]←CS[ARGSTART+ιLEN]
```

Unfortunately, monadic ι will only work with a one element argument. Imagine an enhanced monadic ι function which exhibits the following vector behavior:

```
     ι5 2 4
0 1 2 3 4 0 1 0 1 2 3              (remember: ⎕IO=0)
```

Let us assume a function MONIOTA which will work with vectors as above. Then we can finish the function:

```
R[(LEN/RESSTART)+MONIOTA LEN]←CS[(LEN/ARGSTART)+MONIOTA LEN]
R←(NROWS,WID)⍴R
```

The definition of the MONIOTA function we need follows:

```
                                        [WSID: UTILITY]
    ∇ R←MONIOTA LEN
[1]   ⍝ Performs: (⍳LEN[1]),(⍳LEN[2]),(⍳LEN[3]),...
[2]   ⍝ In APL2: R←∊⍳¨LEN
[3]    R←LEN/-¯1↓0,+\LEN
[4]    R←R+⍳⍴R
    ∇
```

As an exercise, you should reread the TITLES logic above to see what happens when some of the partitions are empty (e.g. 80 TITLES '∩BALANCE SHEET∩∩∩DEC. 31∩').

Finally, let us redefine the TITLES function slightly to allow it to function as a typical character "vector to matrix" converter. Such a function takes a delimited character vector argument and converts it to a character matrix with one row per partition and with as few columns as possible (equal to the length of the longest partition). The rows of such a matrix result are usually left-justified (padded to the right). We will redefine the left argument of TITLES to be either the width of the resulting matrix or an empty vector if the width is to be automatically determined as the length of the longest partition.

To implement this enhancement, we need only precede the line, LEN←WID⌊LEN, by the following:

```
WID←1↑WID,⌈/LEN
```

Now the TITLES function may be used in the following way:

```
    '' TITLES '⊂RED⊂ORANGE⊂YELLOW⊂GREEN⊂BLUE'
RED
ORANGE
YELLOW
GREEN
BLUE
```

(The APL purist may prefer to express the empty vector left argument to TITLES as an empty numeric vector such as 0⍴0 rather then the empty character vector ''. In that way, the WID,⌈/LEN operation does not engender a conceptual domain error from the catenation of character and numeric data. However, since most implementations of APL "forgive" the catenation of character and numeric datatypes when one of the arguments is empty, this preference is academic.)

The completed TITLES function is listed below.

```
                                        [WSID: FORMAT]
      ∇ R←WID TITLES CS;⎕IO;ARGSTART;BJUST;JUST;LEAD;LEN;NROWS
         ;RESSTART;T
[1]    ⍝ Creates report titles from text CS within page
[2]    ⍝ width WID.  CS is delimited by '⊂∩⊃' indicating
[3]    ⍝ left, center, right justification respectively.
[4]     ⎕IO←0
[5]    ⍝ 0:left; 1:center; 2:right; 3:not a delimiter:
[6]     BJUST←3>JUST←'⊂∩⊃'⍳CS←,CS
[7]    ⍝ Select just delimiters; determine no. titles:
[8]     NROWS←⍴JUST←BJUST/JUST
[9]    ⍝ Title lengths:
[10]    T←BJUST/⍳⍴BJUST
[11]    LEN←(1↓T,⍴BJUST)-T+1
[12]   ⍝ Set WID as largest title length if empty WID
[13]   ⍝ provided; truncate titles to specified width:
[14]    LEN←LEN⌊WID←1↑WID,⌈/LEN
[15]   ⍝ Index of char following each delimiter:
[16]    ARGSTART←1+T
[17]   ⍝ Leading blanks per title, to justify:
[18]    LEAD←(JUST≠0)×⌊(WID-LEN)÷1+JUST=1
[19]   ⍝ Ind in raveled result where each segm. starts:
[20]    RESSTART←LEAD+WID×⍳NROWS
[21]   ⍝ Blank, raveled result:
[22]    R←(NROWS×WID)⍴' '
[23]   ⍝ T←MONIOTA LEN:
[24]    T←T+⍳⍴T←LEN/-¯1↓0,+\LEN
[25]    R[T+LEN/RESSTART]←CS[T+LEN/ARGSTART]
[26]    R←(NROWS,WID)⍴R
      ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 330 to 333)

1. Given a character vector TEXT which contains embedded newline characters (carriage returns) and given the scalar NL which is the newline character, what expression will return the first line of text (up to, but not including, the first newline)?
~~~~

2. Given a character vector CODE, find all occurrences of the string '/⍳'. Return a bit vector which has the same length as CODE, with a 1 in each element which corresponds to the '/' in a '/⍳' pair. All other elements are zero.

3. Write a dyadic function CENTER which returns a character vector whose length is specified by the left argument and in which the character vector right argument is centered. For example:

```
          ⍴⎕←50 CENTER 'ACME'
                       ACME
    50
```

   Test your function on each of the following:

```
          50 CENTER 'ACME'
          49 CENTER 'ACME'
          3 CENTER 'ACME'
          11 CENTER 'A'
```

4. Suppose you have a dyadic function COLFMT which formats a numeric matrix into a character matrix. Its right argument NMAT is the numeric matrix to be formatted and its left argument CTL is an integer vector with one element per column of NMAT. The integers indicate the number of decimal places, for each numeric column, to be displayed in the character matrix result CMAT. Each number is formatted in a width of <width> characters (e.g. 10), where <width> is an integer scalar global variable. For example:

```
          ⍴⎕←3 0 1 COLFMT 4 3⍴⍳12
         1.000         2       3.0
         4.000         5       6.0
         7.000         8       9.0
        10.000        11      12.0
    4 30
```

   Write a function ROWFMT which has the same syntax as COLFMT except the elements of its left argument correspond to the rows of the numeric matrix argument rather than to the columns. For example:

```
        ⍴⎕←3 0 1 2 ROWFMT 4 3⍴⍳12
     1.000      2.000      3.000
         4          5          6
       7.0        8.0        9.0
     10.00      11.00      12.00
4 30
```

ROWFMT should use COLFMT.

5. Write a dyadic function COLUMNIZE which will restructure a skinny matrix into a fat one as described in this chapter. The right argument of COLUMNIZE, CMAT, is the original skinny character matrix and the left argument is the number of "columns" of CMAT across the width of the fat character matrix result. For example, to solve the problem presented in that section, you would use:

```
      R←4 COLUMNIZE CMAT
```

   Allow a 1 or 2 element left argument. If 2 elements, the first is the number of rows per "page" and the second is the number of "columns" as discussed above. The result is a 3 dimensional character array with one plane per page. For example, using the 7 column character matrix illustrated in this chapter:

```
            2 3 COLUMNIZE CMAT
      ANNE   CAL    ED
      BILL   DOT    FRED

      GAIL   IKE    KEN
      HAL    JOAN   LISA

      MIKE   PAT    VI
      NED    RICK
```

6. Write a function HEADINGS which will behave as described below:

   SYNTAX: CMAT←WIDS HEADINGS CVEC

   DESCRIPTION:

   HEADINGS is used to convert a delimited character vector into a character matrix of column headings whose respective widths are given by the vector WIDS. Each substring of CVEC is preceded by a delimiter (∩) and may contain any number of newline delimiters (←). The newline delimiters therefore separate sub-substrings. Typically, one width (element of WIDS) is provided for each heading (substring). However, if fewer widths are provided, they are repeated to match the number of headings in CVEC. The headings are formatted into a character matrix according to the following procedure: the sub-substrings of each heading are truncated if necessary to the corresponding width for that heading; the sub-substrings are padded to the left and right with spaces to bring each sub-substring up to the width for that heading; the sub-substrings are catenated together as rows (centered with respect to one another); a row of underlines (hyphens) is catenated to the bottom of each heading; the headings are padded on the top so that each heading has the same number of rows; the headings are catenated together separating them by 2 columns of blanks (if there are more elements in WID than there are headings defined by the right argument, the remaining elements are used as the numbers of columns of blanks to be inserted between each of the pairs of headings):

```
            10 13 8 HEADINGS '∩NAMES∩HIRE←DATE∩AGE←AT←HIRE'
                                   AGE
                     HIRE           AT
      NAMES          DATE          HIRE
   ----------  -------------  --------

            10 13 8 4 1 HEADINGS '∩NAMES∩HIRE←DATE∩AGE←AT←HIRE'
                                    AGE
                       HIRE          AT
      NAMES            DATE         HIRE
   ----------    ------------- --------
```

Empty substrings in CVEC are displayed without underlines. To include an all-blank heading which is underlined, insert at least one blank character in the corresponding substring.

Chapter 5

SORTING AND SEARCHING

Many applications in the real world deal with lists of things. In APL those things are typically represented as numbers and the lists as vectors; or the things are represented as character vectors (rows) and the lists as matrices. That is, real world lists are usually represented in APL as numeric vectors or character matrices.

Since the most common operations performed on lists include sorting, searching and selecting, these too are among the most important APL operations on vectors and matrices. In this chapter, we discuss primitive and utility APL functions for performing sorting and searching. In the next chapter, we discuss selecting.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you have three 1000 element numeric vectors, ENUM (employee identification number), AGE (employee age) and OFFICE (office identification number), that these vectors are in one-to-one correspondence and that each element corresponds to a single employee. How can you reorder these three vectors such that they remain in one-to-one correspondence (i.e. the same index in each vector still corresponds to a single employee) but are sorted by office, and within office by age, and within age by employee number?

TOPIC: Major-to-minor Sorting

Sorting in APL is a two-step process: determine the "grade vector" of the vector to be sorted; and reorder the original vector by indexing the original vector with the grade vector. Therefore, to sort a vector SALARY in ascending order, you would employ the following expression:

```
SORTEDSAL←SALARY[⍋SALARY]
```
~~~~

The grade-up function (⍋) returns the grade vector and the indexing function ([]) reorders the elements. Note that while we have "sorted SALARY", the variable SALARY remains unsorted (unless we reassign it: SALARY←SALARY[⍋SALARY]).

Why does sorting require two steps in APL? Because the grade vector is required if we are dealing with several corresponding vectors whose elements must remain in one-to-one correspondence. For example, if we want to reorder ENUM, AGE and OFFICE such that the values of ENUM are in ascending order but still have corresponding elements in AGE and OFFICE, we must do the following:

```
GRADE←⍋ENUM
ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

Note that the values of AGE and OFFICE are now not in ascending order. They have simply been reordered to continue to correspond to ENUM which is in ascending order.

The sort required in the problem stated above is called a "major-to-minor" sort. It is not possible (usually) to sort all three variables and to maintain the one-to-one correspondence. Only one variable can be strictly sorted (the "major" sort variable). The other variables can at best be sorted within each of the distinct values of the major variable, since only such reordering will maintain the sorted order of the major variable's values. The second variable sorted is said to be "more minor" than the major variable. The third variable is more minor still and may be sorted only within each combination of the distinct values of the two more major sort variables. And so it goes. The last sort variable is called the "minor" sort variable.

How do you do a major-to-minor sort in APL? Backwards. Reorder all the sort variables (to maintain correspondence) by sorting the minor sort variable. Then reorder them by the next more major variable. And so on. The last variable sorted will be the major sort variable and so it will be in strictly sorted order. Since sorting does not change the relative order of the values which are equal, the effects of the earlier sorts will be preserved within each of the distinct values of the major sort variable.

The solution is therefore:

```
GRADE←⍋ENUM
ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]

GRADE←⍋AGE
ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

```
GRADE←⍋OFFICE
ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

If you study this solution, it may strike you that there is much reordering (indexing) going on needlessly. In particular, rather than reorder every variable after each grade operation, you can just reorder the grade vector. Using this approach, the solution becomes:

```
GRADE←⍋ENUM

GRADE←GRADE[⍋AGE[GRADE]]

GRADE←GRADE[⍋OFFICE[GRADE]]

ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

The processing cost using this latter solution increases linearly as the number of sort variables increases. Using the former solution, the processing cost increases exponentially. However, the latter solution lacks the clarity of the first solution and requires comments. In fact, the latter solution is sufficiently unclear that many APL programmers feel little remorse at jamming the first three lines together using embedded assignment. That solution is included here so that you will recognize it, not as an endorsement:

```
GRADE←GRADE[⍋OFFICE[GRADE←GRADE[⍋AGE[GRADE←⍋ENUM]]]]

ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

Some implementations of APL support numeric matrix right arguments to grade-up and grade-down. If so, the resulting grade vector is the result of grading the columns of the matrix (from left to right) as major-to-minor variables. If your APL implementation supports this feature, you may solve the above problem with the following expressions:

```
GRADE←⍋OFFICE,AGE,[1.5]ENUM                 (in origin 1)

ENUM←ENUM[GRADE]
AGE←AGE[GRADE]
OFFICE←OFFICE[GRADE]
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Suppose you have a 1000 row, 8 column character matrix, CMAT, of part numbers, one per row (e.g. 'AK10632B'). How can you construct a 1000 element grade vector which can be used to reorder the rows of CMAT such that the part numbers are in ascending (alphabetic) order?

TOPIC: Character Matrix Sorting

In the previous problem, all sorting was performed on numbers. When numbers are sorted up, they are in ascending order. That means smaller numbers precede bigger numbers. In this problem, we will sort characters, not numbers. What does that mean? If the characters are letters of the alphabet, it means that the earlier (in the alphabet) letters precede the later letters. For characters not in the alphabet we must decide their relative sorting "magnitude" and extend the alphabet accordingly. Such an extended alphabet is called a "collating sequence" and is used as a reference to determine which characters are bigger or smaller than a given character for sorting purposes. The following character vector represents a typical collating sequence.

```
CS←' .,:;-/0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ∆⎕'
```

A different collating sequence may define a different result when sorting a given character array. Therefore the collating sequence is a necessary parameter to the solution of the problem. To solve the problem, we will define a dyadic function CGRADEUP whose left argument is the collating sequence, whose right argument is the character matrix to be sorted and whose result is the desired grade vector:

```
∇ GRADE←CS CGRADEUP CMAT
```

If the rows of a character matrix are in sorted order, what characteristics do they have? The characters of the first column are in strictly ascending order (as defined by the collating sequence). The characters of the second column are in strictly ascending order within any distinct character in the first column. The third column is sorted within distinct combinations of values in the first and second columns. And so on. In other words, the rows of the matrix are reordered by using the columns (first to last) as the major-to-minor sort keys.

Some APL systems have defined primitive dyadic grade-up and grade-down to solve this problem directly. The left argument of ⍋ or ⍒ is the collating sequence. There is no need to define a CGRADEUP function. The solution to this problem is:

```
GRADE←CS⍋CMAT
```

On APL systems for which dyadic grade-up and grade-down are not implemented, a different approach is required. The most straightforward converts the character matrix to an integer matrix with the same shape whose values are the indices into the collating sequence of the corresponding characters. Then the integer matrix is sorted in major-to-minor order as done in the previous section. Since dyadic ⍳ is used to convert characters to indices, the characters in the character matrix which are not in the collating sequence will translate to 1 greater than the length of the collating sequence (or to the length of the collating sequence if the index origin is 0). Therefore, all characters not included in the collating sequence are treated as if they are at the end of the collating sequence.

[WSID: SORT]

```
    ∇ GRADE←CS CGRADEUP1 CMAT;I
[1]  ⍝ Returns grade vector for sorting rows of
[2]  ⍝ CMAT with collating sequence CS.
[3]  ⍝ Convert characters to indices:
[4]   CMAT←CS⍳CMAT
[5]  ⍝ Index of last column as a scalar:
[6]   I←(⍴CMAT)[1+⎕IO]-~⎕IO
[7]  ⍝ Return trivial result of no columns:
[8]   →(I≥⎕IO)⍴L1
[9]   GRADE←⍳1⍴⍴CMAT
[10]  →0
[11] ⍝ Grade rightmost (minor) column:
[12] L1:GRADE←⍋CMAT[;I]
[13] ⍝ Decrement column index and exit if done:
[14] L2:→(⎕IO>I←I-1)⍴0
[15] ⍝ Grade next more major column:
[16]  GRADE←GRADE[⍋CMAT[GRADE;I]]
[17]  →L2
    ∇
```

A more sophisticated technique packs several columns together at once so that fewer applications of grade-up (⍋) are required. For example, suppose you have a 9 column character matrix whose indices into the collating sequence are the following:

```
 3 27 16  9  8  4 15  8 33
31 30 19  9 10  8 24  2 23
 2  3 19 16 12  4 19 14 15
       :           :
       :           :
```

By grouping the matrix into 3 groups of 3 columns and by packing each group into 1 column by respectively multiplying its columns by 10000, 100 and 1 and adding, the result is:

```
 32716  90804 150833
313019  91008 240223
 20319 161204 191415
   :      :      :
   :      :      :
```

Because of the nature of major-to-minor sorting and because of the scheme used to pack these numbers, you can then determine the grade vector of this 3 column matrix (third column first) and it will be the same as that of the 9 column matrix. The approach will probably be more efficient than the original approach because only 3 grade-up operations are needed rather than the original 9.

Taking this approach to its logical extreme, you may argue to pack all 9 columns into a single column (vector) of large numbers:

```
 3271609080415083
31301909100824022 3
 20319161204191415
          :
          :
```

However, the computer internally maintains only 16 or 17 digits of precision on any number. It sees the numbers as:

```
 327160908041508 3_
313019091008240 2__
 2031916120419141_
          :
          :
```

Therefore, the last digit or two of these large packed numbers are insignificant to the computer when it is grading the vector and may produce incorrect grade indices for rows of the character matrix which are identical except in the last column or two.

So how many characters can be packed together at once? This is a function not only of the internal precision of your APL implementation but also of the length of the collating sequence. In the illustration above, the indices were packed by multiplying by consecutive powers of 100. Smaller powers (say 80) can be used to result in smaller packed numbers and to allow more columns to be packed at once. But if the powers used are too small, the indices will not always pack to distinct numbers.

For example, if the power 10 is used to pack the numbers 3 2 4 and 3 1 14, the results will be the same. This problem arises only if the range of indices is greater than the power used. Since the range of indices is one greater than the length of the collating sequence, that is the power you should use.

The following solution packs as many columns at once and performs as few grade-up operations as possible.

```
                                        [WSID: SORT]
     ∇ GRADE←CS CGRADEUP2 CMAT;I;COLS;N;P
[1]    ⍝ Returns grade vector for sorting rows of
[2]    ⍝ CMAT with collating sequence CS.
[3]    ⍝ Convert characters to origin 0 indices:
[4]     CMAT←(CS⍳CMAT)-⎕IO
[5]    ⍝ Number of columns as a scalar:
[6]     I←(⍴CMAT)[1+⎕IO]
[7]    ⍝ Return trivial result of no columns:
[8]     →(I>0)⍴L1
[9]     GRADE←⍳1⍴⍴CMAT
[10]    →0
[11]   ⍝ Compute max. no. cols. to pack (if 16 digits
[12]   ⍝ precision):
[13]   L1:COLS←⌊(P←1+⍴CS)⍟1E16
[14]   ⍝ Number of cols. to pack for first grade:
[15]    N←I⌊COLS
[16]    GRADE←⍋P⊥⍉CMAT[;(I-N)+⍳N]
[17]   ⍝ Decrement columns and exit if done:
[18]   L2:→(0≥I←I-N)⍴0
[19]   ⍝ Grade next group of more major cols.:
[20]    N←I⌊COLS
[21]    GRADE←GRADE[⍋P⊥⍉CMAT[GRADE;(I-N)+⍳N]]
[22]    →L2
     ∇
```

This solution is an improvement over the prior solution only if the packing operation is fast relative to the grade operation. The relative speeds differ among APL implementations and hardware configurations. You should time the two solutions for your implementation. Use the fastest, unless you are paid by the hour.

If you are familiar with the issue of comparison tolerance (system variable ⎕CT) , you are aware that APL systems typically do not distinguish between values which differ only beyond the 14th (or so) significant digit. Yet, here we are packing numbers out to 16 significant digits. We can do this because grade-up (⍋) and grade-down (⍒) are primitive functions which do not consider comparison tolerance (as do =, >, ⍳, ∊, ≠, etc.) If your implementation of grade-up and grade-down does consider comparison tolerance, you should modify the above function to localize ⎕CT in the header and to set ⎕CT←0 (full precision) on the first line of the function.

Finally, APL implementations store small integer numbers more compactly than large integer numbers. Because of these differences in internal storage, grade-up is faster on small integers than on large ones. This difference may be so dramatic that you should pack fewer columns and do more grade-up operations on the small integer values. If so, you should change the reference to 1E16 in the above function to 2147483647 or 32767 or whatever your largest integer is (i.e. the largest number not stored as an 8 byte floating point number).

When working with character matrices which are wide and which have rows whose values are all significantly different (e.g. names), another solution to this problem becomes practical. The approach is to work with the columns in major-to-minor order.

Sort the first column. Compare the sorted characters to their neighbor (prior and next row) characters. If both neighbor characters are different, the row is distinct and belongs in its current (sorted) position. If one or both of its neighbors have the same value as it has, we must proceed to the second column. Consider the second column for only the rows whose value is not distinct for the first column. Sort the second column within the values of the first column. Compare the sorted characters to its neighbors and again identify the rows which are still not distinct for the first two columns. And so on.

Consider each successive column until all rows are known to be distinct or until you run out of columns. As fewer and fewer nondistinct rows remain, the grade-up operation will be performed on shorter and shorter vectors. Since the grade-up operation is quicker on short vectors than on long ones, this solution can be quite fast on matrices whose row values are mostly different.

```
                                        [WSID: SORT]
      ∇ GRADE←CS CGRADEUP CMAT;C;F;G;I;M;N;R;ROWS
[1]    ⍝ Returns grade vector for sorting rows of
[2]    ⍝ CMAT with collating sequence CS.
[3]    ⍝ Index of last column as a scalar:
[4]     N←(⍴CMAT)[1+⎕IO]-~⎕IO
[5]    ⍝ Return trivial result of no columns:
[6]     →(N≥⎕IO)⍴L1
[7]     GRADE←⍳1⍴⍴CMAT
[8]     →0
[9]    ⍝ Select first column:
[10]   L1:C←CMAT[;I←⎕IO]
[11]   ⍝ Convert characters to indices and grade them:
[12]    GRADE←⍋CS⍳C
[13]   ⍝ Exit if 1 column or 1 or less rows:
[14]    →((I=N)∨1≥1⍴⍴CMAT)⍴0
[15]   ⍝ Sort characters:
[16]    C←C[GRADE]
[17]   ⍝ Flag first of groups of equal values:
[18]    F←C≠¯1⌽C
[19]   ⍝ Handle incorrect result if all values equal:
[20]    F[⎕IO]←1
[21]   ⍝ Flag values still unresolved (i.e. more than
[22]   ⍝ 1 equal value):
[23]    M←F⍲1⌽F
[24]   ⍝ Squeeze down flag-first vector:
[25]    F←M/F
[26]   ⍝ Exit if none left to resolve:
[27]    →(⍴F)↓0
[28]   ⍝ Indices into GRADE of unresolved values:
[29]    ROWS←M/⍳⍴M
```

```
    ∇ CGRADEUP (continued)
[30] ⍝ Indices into CMAT of unresolved values:
[31] LOOP:R←GRADE[ROWS]
[32] ⍝ Increment column index:
[33]  I←I+1
[34] ⍝ Select Ith column for unresolved rows:
[35]  C←CMAT[R;I]
[36] ⍝ Convert and grade characters:
[37]  G←⍋CS⍳C
[38] ⍝ Reorder grade vec to maintain sorted prior columns:
[39]  G←G[⍋(+\F)[G]]
[40] ⍝ Insert reordered grade vec:
[41]  GRADE[ROWS]←R[G]
[42] ⍝ Exit if no more columns:
[43]  →(I≥N)⍴0
[44] ⍝ Sort characters:
[45]  C←C[G]
[46] ⍝ Flag first, considering prior columns too:
[47]  F←F∨C≠¯1⌽C
[48] ⍝ Flag values still unresolved:
[49]  M←F⍲1⌽F
[50] ⍝ Squeeze down flag-first vector:
[51]  F←M/F
[52] ⍝ Exit if none left to resolve:
[53]  →(⍴F)↓0
[54] ⍝ Squeeze down unresolved indices into GRADE:
[55]  ROWS←M/ROWS
[56]  →LOOP
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Sort the following character matrix, SUBJECTS.

```
Lincoln
troops
liberty
lasting
brothers
Grant
Lee
```
~~~~

TOPIC: Uppercase/Lowercase Sorting

Since this matrix contains both uppercase and lowercase letters, the collating sequence must contain both the uppercase and lowercase alphabets. Let's try catenating them:

```
      CS←' ABCDEF...XYZabcdef...xyz'
      SUBJECTS[CS⍋SUBJECTS;]          (use CGRADEUP if
Grant                               dyadic ⍋ is unavailable)
Lee
Lincoln
brothers
lasting
liberty
troops
```

No good. Words beginning with the letter L should be together, whether the L is uppercase or lowercase. Let's try interleaving the uppercase and lowercase alphabets:

```
      CS←' AaBbCcDcEeFf...XxYyZz'
      SUBJECTS[CS⍋SUBJECTS;]
brothers
Grant
Lee
Lincoln
lasting
liberty
troops
```

Not quite. Although words beginning with the letter L are now together, those beginning with an uppercase L precede those beginning with a lowercase l, regardless of the second letter in each word. We want all Ls to be treated equally, regardless of case.

Since equality is our aim, let us promote each lowercase letter to an uppercase letter and try again. Suppose UPPERCASE is a monadic function which converts its character array argument to an array of the same shape and values except each lowercase letter has been replaced by the corresponding uppercase letter. Then, lowercase letters can be omitted from the collating sequence. The following solution does the job.

```
      CS←' ABCDEF...XYZ'
      SUBJECTS[CS⍋UPPERCASE SUBJECTS;]
brothers
Grant
lasting
Lee
liberty
Lincoln
troops
```

Notice that the character matrix is converted to uppercase letters for purposes of grading only. The original mixed case matrix is used for indexing.

The technique of converting an array to uppercase letters is also useful for searching through mixed case arrays whenever the uppercase and lowercase characteristics of letters are to be ignored. For example, to list the words which begin with "LI":

```
        ((UPPERCASE SUBJECTS[;1 2])∧.='LI')⌿SUBJECTS
Lincoln
liberty
```

The following function will perform the desired translation to uppercase letters.

```
                                            [WSID: SORT]
    ∇ R←UPPERCASE C;FOUND;IND;LOWER;UPPER
[1]   ⍝ Converts the lowercase letters in the character
[2]   ⍝ array C into the corresponding uppercase
[3]   ⍝ letters.  Useful for sorting or searching
[4]   ⍝ character arrays when the case distinction is
[5]   ⍝ to be ignored.
[6]    LOWER←'abcdefghijklmnopqrstuvwxyz'
[7]    UPPER←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[8]    R←,C
[9]   ⍝ Inds of arg in LOWER (1+last ind if not found):
[10]   IND←LOWER⍳R
[11]  ⍝ Mark those found:
[12]   FOUND←IND<⎕IO+⍴LOWER
[13]  ⍝ Insert UPPER elements in place of LOWER ones:
[14]   R[FOUND/⍳⍴FOUND]←UPPER[FOUND/IND]
[15]  ⍝ Reshape to original shape:
[16]   R←(⍴C)⍴R
[17]  ⍝
[18]  ⍝ In APL2, no need to reshape:
[19]  ⍝   IND←LOWER⍳,C
[20]  ⍝   FOUND←IND<⎕IO+⍴LOWER
[21]  ⍝   R←C
[22]  ⍝   (FOUND/,R)←UPPER[FOUND/IND]
[23]  ⍝
[24]  ⍝ Alternate algorithm...
[25]  ⍝ Construct ⎕AV of only uppercase letters:
[26]  ⍝   ∆AV←⎕AV
[27]  ⍝   ∆AV[⎕AV⍳LOWER]←UPPER
[28]  ⍝ Perform transl from lower/upper ⎕AV to upper ∆AV:
[29]  ⍝   R←∆AV[⎕AV⍳C]
    ∇
```

Implementations of APL which provide dyadic grade-up typically also provide a facility for handling this uppercase/lowercase problem directly. Specifically, the collating sequence left argument may be a matrix which contains both alphabets as two corresponding rows. For example:

```
                CS←' ABCDEF...XYZ',[0.5]' abcdef...xyz'
                SUBJECTS[CS⍋SUBJECTS;]
          brothers
          Grant
          lasting
          Lee
          liberty
          Lincoln
          troops
```

You should read your documentation for dyadic grade-up to understand the subtleties of this facility. Given the facility, the UPPERCASE function is not needed for sorting uppercase/lowercase character matrices. However, it is still useful for searching through uppercase/lowercase arrays.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given the policy numbers of 1000 existing policyholders (as a 1000 row, 12 column character matrix since the policy "numbers" may contain letters) and the policy numbers of 600 non-smokers (600 rows, 12 columns), determine the index of the existing policy (1 to 1000) to which each of the 600 non-smokers corresponds. The resulting integer vector will have 600 elements. Return the "index" 1001 if the policy number is not found.

TOPIC: Array Searching

If the two lists of policy numbers were numeric vectors rather than character matrices, the solution would be trivial. Suppose BASE is the name of the 1000 element vector of existing policy numbers, VALS is the name of the 600 element vector of non-smoker policy numbers and INDS is the desired result. The following expression will solve the problem:

```
INDS←BASE⍳VALS
```

(Note that the default comparison tolerance, i.e. ⎕CT, will need to be reduced to make accurate comparisons between numbers with more than 14 or so digits.)
~~~~

In APL implementations which support nested arrays, the dyadic ⍳ function may be used to solve this problem, even on character matrices. The first step is to convert the matrix arguments to nested vector arguments (e.g. ⊂[2]BASE in APL2). The solution then follows directly:

```
          INDS←(⊂[2]BASE)⍳(⊂[2]VALS)              (in APL2)
          INDS←(↓[2]BASE)⍳(↓[2]VALS)              (in APL*PLUS)
          INDS←(<⍤1 BASE)⍳(<⍤1 VALS)              (in SHARP APL)
```

In APL implementations which do not support nested arrays, a more creative approach is required since the dyadic ⍳ function does not operate as hoped for on character matrix lists.

One effective approach is to convert the character matrix arguments into numeric vector arguments by converting the columns of characters into columns of indices and then packing the numbers together by the techniques of the previous sections. As mentioned there, only a limited number of characters may be packed into a single number without losing precision (say 8 to 12 character columns, depending upon the length of the character vector collating sequence used to convert the characters to indices). Further, since dyadic ⍳ uses comparison tolerance, the value of ⎕CT should be set to zero to make comparisons which are as precise as possible.

The following function uses this technique to emulate dyadic ⍳ on character matrices:

```
                                                 [WSID: SEARCH]
              ∇ INDS←BASE CMIOTA1 VALS;CS;P;⎕CT
          [1]   ⍝ Returns the row indices of BASE at which the
          [2]   ⍝ rows of VALS first match.
          [3]   ⍝ Set comparison tolerance to maximum precision:
          [4]    ⎕CT←0
          [5]   ⍝ Determine collating sequence:
          [6]    CS←((⎕AV∊BASE)∨⎕AV∊VALS)/⎕AV
          [7]   ⍝ Packing factor:
          [8]    P←1+⍴CS
          [9]   ⍝ Pack and search:
          [10]   INDS←(P⊥(CS⍳⍉BASE)-⎕IO)⍳P⊥(CS⍳⍉VALS)-⎕IO
              ∇
```

Unfortunately, this technique will not work on wide character matrices. Further, in some APL implementations, the decode (⊥) function is slow. Under either of these conditions, another approach is desired.

Suppose BASE is a numeric vector and VAL is a numeric scalar. How can we find the index of the first occurrence of VAL in BASE?

```
          BASE⍳VAL
```

How can we identify (by bits) all of the occurrences of VAL in BASE?

```
BASE=VAL
```

If VALS is a vector, how can we identify (by bits) all of the occurrences of VALS in BASE?

```
BASE∘.=VALS
```

If BASE is a character matrix and VAL is a character vector (i.e. one policy number), how can we identify (by bits) all of the occurrences of VAL in BASE?

```
BASE∧.=VAL
```

If VALS is also a character matrix, how can we identify (by bits) all of the occurrences of VALS in BASE?

```
BASE∧.=⍉VALS
```

The information we seek is contained in the Boolean matrix result of this expression. Specifically, the column index of the first bit in each row is the index we seek. By using a Boolean scan, we can extract the indices:

```
INDS←⎕IO++⌿∧⍀~BASE∧.=⍉VALS
```

We will modify this algorithm somewhat to replace row-wise (e.g. ∧⍀) functions by the usually faster column-wise (e.g. ∧\) functions and to eliminate the not (~) function.

[WSID: SEARCH]

```
    ∇ INDS←BASE CMIOTA2 VALS
[1]   ⍝ Returns the row indices of BASE at which the
[2]   ⍝ rows of VALS first match.
[3]     INDS←⎕IO++/∧\VALS∨.≠⍉BASE
    ∇
```

This algorithm is an excellent illustration of the power of APL. Unfortunately, it has some drawbacks. For this example, the result of the ∨.≠ function is a 600 row, 1000 column Boolean matrix (600,000 elements) which might generate a WS FULL error message. Even if it does work, the function will consume a large amount of CPU time while making the 600,000 comparisons.

A different approach takes advantage of the speed of sorting algorithms. The following discussion assumes ⎕IO=1.

1. Combine the two arguments via catenation into a 1600 row matrix:

```
A←BASE,[1]VALS
```

2. Sort the combined matrix using the CGRADEUP function developed in a previous section (or using dyadic ⍋ if available) and using ⎕AV as the collating sequence:

```
GRADE←⎕AV CGRADEUP A
A←A[GRADE;]
```

By sorting the matrix, like rows are now contiguous.

3. Shift the rows of the matrix down one row and compare:

```
FLAG←∨/A≠¯1⊖A
```

FLAG is a 1600 element Boolean vector whose 1s flag the first of each set of contiguous like-valued rows.

4. For each of the 1600 rows, determine the index into the original unsorted catenated matrix of the first row of each set of contiguous like-valued rows:

```
FIRST←(FLAG/GRADE)[+\FLAG]
```

The 1600 elements of FIRST correspond to the rows of the sorted catenated matrix.

5. Reorder the elements so they correspond to the rows of the unsorted catenated matrix:

```
INDS←(⍴FIRST)⍴0
INDS[GRADE]←FIRST
```

6. Select only those elements of INDS which correspond to the rows of VALS (not the rows of BASE):

```
L←1↑⍴BASE
INDS←L↓INDS
```

7. Set the elements of INDS which correspond to rows of VALS for which no matching row was found in BASE to the "not found" index (1 plus the number of rows in BASE):

```
INDS←INDS⌊1+L
```

This approach is quite efficient for large arguments. However, because it requires so many steps, other algorithms may be more efficient for small arguments. In particular, inner product (∧.=) is typically quite fast when one argument is a matrix and the other is a vector. Therefore for a small (few rows) right argument of CMIOTA, it may actually be faster to loop on the rows of the right argument (using ∧.= to search through the rows of the matrix left argument) than to employ this catenating, sorting, shifting, comparing algorithm.

But how small should the right argument to CMIOTA be before we switch to a looping algorithm? Let us assume the arguments to CMIOTA are L and R:

```
I←L CMIOTA R
```

The CPU time consumed by the looping algorithm increases linearly with the number of rows in R (for a constant L) and linearly with the number of rows in L (for a constant R). Therefore, the CPU time consumed will be a function of the formula:

```
CPUL = C1+(RR×(C2+(C3×RL)))
```

where CPUL is the amount of CPU time consumed by the looping algorithm, RR and RL are the number of rows in R and L respectively, and C1, C2 and C3 are constants to be determined.

The CPU time consumed by the sorting algorithm increases linearly with the sum of the numbers of rows in R and in L. Therefore, the CPU time consumed will be a function of the formula:

```
CPUS = C4+(C5×(RR+RL))
```

where CPUS is the amount of CPU time consumed by the sorting algorithm and C4 and C5 are constants to be determined.

The values of C1, C2, C3, C4 and C5 for the formulas above will depend upon the particular machine and APL implementation. To determine them for your environment, you must time the two algorithms for a variety of arguments and then use the techniques of least squares to find the constants which define the "best" curves to fit the empirical data. There is a problem at the end of the chapter on Computer Efficiency Considerations which performs the first task and a problem at the end of the chapter on Curve Fitting which performs the second task. Work these problems and plug the derived values into the CMIOTA function below (in place of C1, C2, C3, C4, C5).

(The formulas above do not consider the number of columns in the matrix arguments nor the nature of the data, i.e. whether and where the values are found. Therefore they are not precise formulas. However, they will be sufficiently accurate to insure that the best algorithm is used in all but borderline cases.)

The following CMIOTA function uses the approaches discussed above and has been extended to handle origin 0 and to treat the trivial cases (empty or 1-row arguments) separately.

```
                                   [WSID: SEARCH]
     ∇ INDS←BASE CMIOTA VALS;A;F;G;I;L
[1]   ⍝ Returns the row indices of BASE at which the
[2]   ⍝ rows of VALS first match.
[3]   ⍝ Branch if right arg a matrix:
[4]    →(2=⍴⍴VALS)⍴L1
[5]   ⍝ Handle vec or scalar right arg:
[6]    INDS←(BASE∧.=VALS)⍳1
[7]    →0
[8]   L1:L←(⍴BASE)[⎕IO]
[9]    A←(⍴VALS)[⎕IO]
[10]  ⍝ Branch unless no rows in either arg:
[11]   →(×F←A⌊L)⍴L2
[12]  ⍝ Handle empty arg:
[13]   INDS←A⍴⎕IO
[14]   →0
[15]  ⍝ Branch if both args have more than 1 row:
[16]  L2:→(F≠1)⍴L4
[17]  ⍝ Branch unless left arg has 1 row:
[18]   →(L≠1)⍴L3
[19]  ⍝ Handle 1 row left arg:
[20]   INDS←⎕IO+VALS∨.≠,BASE
[21]   →0
[22]  ⍝ Handle 1 row right arg:
[23]  L3:INDS←,(BASE∧.=,VALS)⍳1
[24]   →0
[25]  ⍝ Branch if sort alg. costs more than looping alg.:
[26]  ⍝    (remove ⍝ after replacing C1,C2,C3,C4 by
[27]  ⍝    computed constants):
[28]  L4: ⍝→((C4+C5×L+A)>C1+A×C2+C3×L)⍴L5
[29]  ⍝ Combine args. and sort (like values together)
[30]  ⍝    (use CGRADEUP if dyadic ⍋ unavailable):
[31]   G←⎕AV⍋A←BASE,[⎕IO]VALS
[32]   A←A[G;]
[33]  ⍝ Flag 1st of distinct rows by shifting and comparing:
[34]   F←∨/A≠¯1⊖A
[35]  ⍝ Insure 1st elt is 1 (in case all rows the same):
[36]   F[⎕IO]←1
[37]  ⍝ Indices of 1st distinct rows:
[38]   I←F/G
[39]  ⍝ Replicate for each like row:
[40]   F[⎕IO]←⎕IO
[41]   I←I[+\F]
[42]  ⍝ Unsort indices (to catenated order):
[43]   INDS←I
[44]   INDS[G]←I
[45]  ⍝ Keep those corresponding to right arg:
[46]   INDS←L↓INDS
[47]  ⍝ Set 'not found' inds to 'one greater':
[48]   INDS←INDS⌊L+⎕IO
[49]   →0
[50]  ⍝ Use looping algorithm if more efficient:
[51]  L5:INDS←A⍴0
[52]   L←(A⍴L6),0
```

```
    ∇ CMIOTA (continued)
[53]  I←⎕IO
[54] L6:INDS[I]←(BASE∧.=VALS[I;])⍳1
[55]  →L[I←I+1]
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you have a 1000 element vector of ages. You wish to group the ages into the 10 ranges: 0 to 9, 10 to 19, 20 to 24, 25 to 29, 30 to 34, 35 to 39, 40 to 49, 50 to 59, 60 to 64, 65 and up. What approach would you take to translate these 1000 ages into 1000 corresponding range indices (i.e. numbers between 1 and 10)?

TOPIC: Range Searching

Suppose the 1000 element vector of ages is named AGES. Let us define a 10 element vector LOWER of the lower limits for the specified ranges:

```
LOWER←0 10 20 25 30 35 40 50 60 65
```

We need to compare each element of AGES to each element of LOWER and to determine the index into LOWER of the last element which is less than or equal to the element of AGES. Outer product may be used to solve this directly:

```
INDS←+/AGES∘.≥LOWER
```

This expression is simple and powerful but suffers from the malady of all outer product solutions. Since every element of the left argument is being compared to every element of the right argument, the number of comparisons increases exponentially as the lengths of the two arguments increase linearly. Hence, the solution is slow and expensive when performed on two long vectors.

A more efficient (for long arguments) algorithm can be developed using the same sorting technique employed in the prior section.

1. Combine the two arguments and determine the grade vector:

```
A←LOWER,AGES
GRADE←⍋A
```
~~~~

2. Rather than reorder the elements of the catenated array, reorder the elements of an array of 1s and 0s where the 1s mark elements of LOWER and the 0s mark elements of AGES:

```
FLAG←((⍴A)↑(⍴LOWER)⍴1)[GRADE]
```

3. Determine the index into LOWER of each element of the sorted catenated array:

```
FIRST←+\FLAG
```

4. The elements of FIRST correspond to the elements of the sorted catenated array. Reorder the elements so they correspond to the elements of the unsorted catenated array:

```
INDS←(⍴FIRST)⍴0
INDS[GRADE]←FIRST
```

5. Select only those elements of INDS which correspond to the elements of AGES (not the elements of LOWER):

```
INDS←(⍴LOWER)↓INDS
```

The following function LIOTA uses this approach but is extended to handle origin 0 and to return 1 greater than the largest index if the corresponding value is less than the smallest lower limit. The left argument is assumed to be in ascending order.

```
                                  [WSID: SEARCH]
    ∇ INDS←LOWER LIOTA VALS;A;F;G;I;L
[1]   ⍝ Returns the indices of LOWER at which the
[2]   ⍝ elements of VALS first match or exceed.
[3]   ⍝ Branch unless right argument empty:
[4]    →(×⍴VALS)⍴L1
[5]    INDS←⍳0
[6]    →0
[7]   ⍝ Combine arguments and sort:
[8]   L1:G←⍋A←LOWER,VALS
[9]   ⍝ Flag elements from LOWER in sorted array:
[10]   L←⍴LOWER
[11]   F←((⍴A)↑L⍴1)[G]
[12]  ⍝ Determine indices into LOWER (origin dependent):
[13]   F[⎕IO]←F[⎕IO]-~⎕IO
[14]   I←+\F
[15]  ⍝ Unsort indices (to catenated order):
[16]   INDS←I
[17]   INDS[G]←I
[18]  ⍝ Keep those corresponding to right argument:
[19]   INDS←L↓INDS
[20]  ⍝ Set 'not found' indices to 'one greater':
[21]   INDS[(INDS=⎕IO-1)/⍳⍴INDS]←L+⎕IO
    ∇
```

The solutions presented here are oriented around lower limits of ranges (in ascending order). If upper limits are considered (e.g. UPPER←9 19 24 29 34 39 49 59 64 99), the solutions must be modified accordingly:

```
    Lower limits (ascending):  +/AGES∘.≥LOWER
                          or:  LOWER LIOTA AGES

    Upper limits (ascending):  1++/AGES∘.>UPPER
                          or:  UPPER UIOTA AGES

    Lower limits (descending):  1++/AGES∘.<LOWER

    Upper limits (descending):  +/AGES∘.≤UPPER
```

The following function UIOTA works like LIOTA but requires a vector left argument of range upper limits in ascending order.

```
                                        [WSID: SEARCH]
      ∇ INDS←UPPER UIOTA VALS;A;F;G;I;L
[1]    ⍝ Returns the indices of UPPER at which the
[2]    ⍝ elements of VALS last match or are less than.
[3]    ⍝ Branch unless right argument empty:
[4]     →(×⍴VALS)⍴L1
[5]     INDS←⍳0
[6]     →0
[7]    ⍝ Combine arguments and sort:
[8]    L1:G←⍋A←VALS,UPPER
[9]    ⍝ Flag elements from UPPER in sorted array:
[10]    L←⍴UPPER
[11]    F←((-⍴A)↑L⍴1)[G]
[12]   ⍝ Determine indices into UPPER (origin dependent):
[13]    I←+\¯1↓⎕IO,F
[14]   ⍝ Unsort indices (to catenated order):
[15]    INDS←I
[16]    INDS[G]←I
[17]   ⍝ Keep those corresponding to right argument:
[18]    INDS←(⍴VALS)⍴INDS
      ∇
```

You should be aware that the LIOTA and UIOTA functions may not produce correct results when operating on floating point numeric vectors whose values are approximately equal (within comparison tolerance) to elements in the lower limit or upper limit vector. This aberration occurs because the grade-up (⍋) function used in LIOTA and UIOTA does not consider comparison tolerance when sorting. Thus, two numbers which would be treated as equal by the relational functions (say ≥ or >) are treated as distinctly different numbers by grade-up. If this is a likely problem for a particular application, you should use the appropriate outer product solution.

Let's consider an alternate algorithm for solving this range searching problem. The algorithm involves "ranking vectors". The ranking vector of a vector V is computed via ⍋⍋V and indicates the relative magnitudes of the values of V. The smallest value in V is assigned the index 1 (in origin 1), the second smallest the index 2, the third 3 and so on. For example,

```
      ⍋⍋15 5 10 15 20
3 1 2 4 5
```

Notice that in the event of ties, the earlier values receive the lower rankings. In this example, the first 15 is ranked 3rd and the next is ranked 4th.

Consider what happens to the rankings of these values when more values are catenated to the vector. For example,

```
      ⍋⍋15 5 10 15 20,13 17
4 1 2 5 7 3 6
```

Notice that the corresponding rankings (4 1 2 5 7) have increased by the number of catenated values which are less than the respective values.

```
      4 1 2 5 7-3 1 2 4 5
1 0 0 1 2
```

That is, no catenated values are less than the 5 or 10; one catenated value (13) is less than the two 15s; and two catenated values (13 17) are less than the 20.

Consider what happens when some of the catenated values are equal to values in the original vector. For example, catenating 13 15 instead of 13 17:

```
      ⍋⍋15 5 10 15 20,13 15
4 1 2 5 7 3 6
```

We get the same result. However, notice what happens when the catenated values are placed at the front of the vector:

```
      ⍋⍋13 15,15 5 10 15 20
3 4 5 1 2 6 7
      5 1 2 6 7-3 1 2 4 5
2 0 0 2 2
```

The result now indicates the number of catenated values which are less than or equal to each value, not just less than.

Given this behavior, the LIOTA and UIOTA algorithms follow directly:

```
LIOTA:  INDS←((⍴LOWER)↓⍋⍋LOWER,AGES)-⍋⍋AGES

UIOTA:  INDS←1+((⍴AGES)↑⍋⍋AGES,UPPER)-⍋⍋AGES
```

These algorithms produce correct results for origin 1. The LIOTA algorithm returns 0 (instead of 1+⍴LOWER) for values of AGES which are less than the smallest value in LOWER. The two functions listed below, LIOTA1 and UIOTA1, work like the LIOTA and UIOTA functions above but use these ranking vector algorithms. The algorithms have been modified to work correctly in either origin and to return the correct "not found" value (⎕IO+⍴LOWER).

Further, a more efficient method for computing the ranking vector is employed. When sorting a grade vector, traditional sorting logic is not needed. Index assignment will suffice. The following four sets of expressions generate equivalent results:

```
R←⍋⍋V          G←⍋V          G←⍋V           G←⍋V
               R←⍋G          R←(⍴V)⍴0       R←G
                             R[G]←⍳⍴G       R[G]←⍳⍴G
```

The last set of expressions is the most efficient. Since it is not as clear as the first expression, it should include a comment:

```
⍝ R←⍋⍋V :
 R←G←⍋V
 R[G]←⍳⍴G
```

Using this technique, the LIOTA1 and UIOTA1 functions each perform only two grade-up operations instead of four. However, the LIOTA and UIOTA functions above each perform only one grade-up operation and so will typically be the faster functions. Time them in your APL implementation.

```
                                    [WSID: SEARCH]
    ∇ INDS←LOWER LIOTA1 VALS;G;L;R;S
[1]  ⍝ Returns the indices of LOWER at which the
[2]  ⍝ elements of VALS first match or exceed.
[3]   L←⍴LOWER
[4]  ⍝ R←⍋⍋VALS :
[5]   R←G←⍋VALS
[6]   R[G]←⍳⍴G
[7]  ⍝ S←⍋⍋LOWER,VALS :
[8]   S←G←⍋LOWER,VALS
[9]   S[G]←⍳⍴G
[10] ⍝ Origin 1 indices:
[11]  INDS←(L↓S)-R
[12] ⍝ Set 'not found' indices to 'one greater':
[13]  INDS[(INDS=0)/⍳⍴INDS]←L+1
[14] ⍝ Change from origin 1 to origin 0 if needed:
[15]  →⎕IO⍴0
[16]  INDS←INDS-1
    ∇
```

```
                                        [WSID: SEARCH]
    ∇ INDS←UPPER UIOTA1 VALS;G;R;S
[1]  ⍝ Returns the indices of UPPER at which the
[2]  ⍝ elements of VALS last match or are less than.
[3]  ⍝ R←⍋⍋VALS :
[4]   R←G←⍋VALS
[5]   R[G]←⍳⍴G
[6]  ⍝ S←⍋⍋VALS,UPPER :
[7]   S←G←⍋VALS,UPPER
[8]   S[G]←⍳⍴G
[9]  ⍝ Origin 0 indices:
[10]  INDS←((⍴VALS)⍴S)-R
[11] ⍝ Change from origin 0 to origin 1 if needed:
[12]  →⎕IO↓0
[13]  INDS←INDS+1
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write a function named ∆SS (string search) which will locate every occurrence of a character vector substring (right argument) in a character vector (left argument). The result is a Boolean vector of the same length as the left argument whose 1s flag the indices at which each match begins. For example:

```
          'THIS IS A TEST' ∆SS 'IS'
    0 0 1 0 0 1 0 0 0 0 0 0 0 0
```

TOPIC: Character Substring Searching

Some APL implementations have primitive functions which solve this problem directly:

APL★PLUS:

```
    ∇ BIT←CVEC ∆SS SUB
[1]  BIT←CVEC ⎕SS SUB
    ∇
```
~~~~

APL2:

```
      ∇ BIT←CVEC ∆SS SUB
  [1]  BIT←SUB⍷CVEC
      ∇
```

If such a primitive function is unavailable to you, you must work a little to get the desired result:

```
                                        [WSID: SEARCH]
      ∇ BIT←CVEC ∆SS SUB;C;S
  [1]  ⍝ Returns bit vector of length (⍴CVEC) with 1s
  [2]  ⍝ flagging starts of substrings which match SUB.
  [3]   C←⍴CVEC
  [4]   S←⍴,SUB
  [5]   BIT←C↑(-S)↓SUB∧.=(S,C+×C)⍴CVEC
      ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write functions REPLACE and BY which will replace all occurrences of a character vector substring in a character vector by a second substring. For example:

```
          'THIS IS A TEST' REPLACE 'IS' BY 'ARE'
      THARE ARE A TEST
```

Use the function ∆SS defined in the prior section.

TOPIC: Character Substring Replacement

The BY function is used simply as a syntactic convenience to provide three arguments to the REPLACE function. One approach is to assign the right argument to a global variable (say <by>) and to return the left argument as the explicit result.

```
                                        [WSID: SEARCH]
      ∇ R←A BY B
  [1]  ⍝ Used in conjunction with REPLACE as:
  [2]  ⍝
  [3]  ⍝      'THIS IS A TEST' REPLACE 'IS' BY 'ARE'
  [4]  ⍝
  [5]   by←B
  [6]   R←A
      ∇
```
~~~~

The REPLACE function will generate a result by analyzing its two arguments and its third global "argument" <by>. When done, REPLACE erases <by> so that it will not be left global.

The approach taken by REPLACE is the following:

1. Use ∆SS to find the occurrences of the old substring in the character vector. Convert the bits to indices.

2. Create a replication vector (i.e. left argument to /) which can be used to both squeeze out the old substring and to allow room for the new substring. Perform the replication on the character vector.

3. Since the length of the character vector has changed (unless the new substring has the same length as the old substring), adjust the indices computed in step 1 to point to where the new substrings must be inserted.

4. Insert the new substrings.

```
                                          [WSID: SEARCH]
     ∇ NVEC←OVEC REPLACE SUB;BIT;IND;NHITS;REP;SIZE;⎕IO
[1]   ⍝ Replaces all occurrences of SUB in OVEC by <by> (set
[2]   ⍝ in BY), erases <by> and returns the modified OVEC.
[3]   ⍝ Requires subfn ∆SS (or ⎕SS or ⋸).
[4]   ⍝ The logic is a bit simpler using origin 0:
[5]    ⎕IO←0
[6]   ⍝ Locate the starts of the old substring:
[7]    BIT←OVEC ∆SS SUB
[8]   ⍝ Convert the bits to indices:
[9]    NHITS←⍴IND←BIT/⍳⍴BIT
[10]  ⍝ Initialize replication vector as 1s:
[11]   REP←(⍴BIT)⍴1
[12]  ⍝ Insert 0s where old substrings are:
[13]   REP[IND∘.+⍳⍴,SUB]←0
[14]  ⍝ Insert new substring length where new substrings
[15]  ⍝ will begin:
[16]   REP[IND]←SIZE←⍴,by
[17]  ⍝ Squeeze and expand OVEC with replicate:
[18]   NVEC←REP/OVEC
[19]  ⍝ Adjust old indices to get new indices:
[20]   IND←IND+(SIZE-⍴,SUB)×⍳NHITS
[21]  ⍝ Insert new substrings:
[22]   NVEC[IND∘.+⍳SIZE]←(NHITS,SIZE)⍴by
[23]  ⍝ Erase <by>:
[24]   BIT←⎕EX 'by'
     ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEMS: (Solutions on pages 334 to 336)

1. Given a 3 column integer matrix PNUM of telephone numbers (area code, phone number, extension, e.g. 213 5550123 1234), how can you sort the numbers in ascending order?

2. Modify the CMIOTA function described in this chapter to define a function IOTA which works on numeric vector arguments instead of character matrix arguments. Test it in your APL implementation. Which is faster, IOTA or dyadic ⍳ (see Computer Efficiency Considerations chapter)? What is the consequence of the dependence of dyadic ⍳ on ⎕CT (comparison tolerance) and the independence of ⍋ on ⎕CT?

3. Given a 3 column integer matrix PNUM of telephone numbers (area code, phone number, extension) and a 3 element integer vector P which represents a particular telephone number, determine the index of the first row of PNUM in which P is located.

4. Using the LIOTA function developed in this chapter, determine in which salary grouping each of the elements of the vector SALARY belong. The groupings are: (1) 1000 to 9999; (2) 10,000 to 19,999; (3) 50,000 to 69,999; (4) 100,000 and up. Return the index 5 for elements of SALARY in none of these groupings.

5. Using the ∆SS function developed in this chapter, write a monadic function DEB which will delete extraneous (leading, trailing or contiguous) blanks from its argument and will return the compressed result. For example:

```
            DEB '    TOO  MANY    SPACES.    '
      TOO MANY SPACES.
```

6. In a numeric vector NVEC, the value ¯1 represents "unknown". Display NVEC, showing each occurrence of ¯1 as the characters 'N/A' (not applicable).

7. Suppose you have a 25 column character matrix of employee names, ENAMES. Each row contains one name, left-justified. The names contain both uppercase and lowercase letters. Display the names which contain the string "son" anywhere in the name.

Chapter 6

SELECTING

This chapter deals with the task of data selection in APL. Selection is the process of extracting elements from an APL array. The reverse process, replacing the values of elements within an array, or selection assignment, is also considered. Finally, a special selection task is covered: the task of selecting those values in an array which are unique (or distinct).

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given a three element vector NVEC, what APL expression will return the first two elements? What expression will replace these elements in NVEC by the values 10 and 20?

TOPIC: Selection and Selection Assignment

There are basically 3 selection techniques available in APL.

1. Indexing. Use indexing ([]) when you know the positions within the array of the elements to be selected. For example (in origin 1):

```
NVEC[1 2]
```

2. Take/drop. Use take (↑) or drop (↓) or both when the elements to be selected are contiguous, especially at the start or end of the array. For example:

```
2↑NVEC
¯1↓NVEC
```
~~~~

3. Compression. Use compression (/) when you have a corresponding Boolean compression vector whose ones flag elements to be selected. For example:

```
1 1 0/NVEC
```

Typically, the compression vector is the result of a relational or logical expression which defines some criteria by which elements are to be selected.

Though there are 3 selection techniques, the only selection assignment technique available in APL is index assignment. For example:

```
NVEC[1 2]←10 20
```

If the nature of the selection assignment problem is oriented more toward take/drop or compression logic, you must convert the selection values to indices so that you may use index assignment. For example:

```
NVEC[2↑⍳⍴NVEC]←10 20
NVEC[¯1↓⍳⍴NVEC]←10 20
NVEC[1 1 0/⍳⍴NVEC]←10 20
```

It is because of this need to convert to indices when performing selection assignment that the APL idioms ⍳⍴ and /⍳⍴ are so common.

In APL2, the APL language has been extended to allow direct selection assignment without first converting to indices. For example:

```
NVEC[1 2]←10 20
(2↑NVEC)←10 20
(¯1↓NVEC)←10 20
(1 1 0/NVEC)←10 20
```

In fact, fairly complex selection assignment expressions are permitted. The expression,

```
(3⍴1↓⌽NVEC)←10
```

is equivalent to:

```
NVEC[3⍴1↓⌽⍳⍴NVEC]←10
```

The enhancement, when not abused, is a welcome extension to the language.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Suppose you have constructed a matrix DEPN of annual depreciation rates to be used for assets which have depreciable lives of 20 years. DEPN has 20 rows and 12 columns. DEPN[Y;M] is the fraction of the asset to be depreciated in the Yth year of its life for an asset purchased in month M (1 for January, 2 for February, and so on). Suppose YEAR is a 1000 element vector of the ages (1 to 20) of 1000 assets and MONTH is a 1000 element vector of the months of purchase (1 to 12) for the corresponding assets. Determine the annual depreciation rates for these assets.

TOPIC: Scattered Point Indexing

A common mistake made when solving this problem is to try the following:

```
DEPN[YEAR;MONTH]
```

This expression shows nicely what you want to do but unfortunately does not do it. The shape of the result of matrix indexing is the catenation of the shape of the row indices with the shape of the column indices. Since YEAR has shape 1000 and MONTH has shape 1000, the result has shape 1000 1000. These 1,000,000 elements are the rates for every combination of the elements of YEAR and the elements of MONTH.

If you can picture this 1000 by 1000 element matrix in your mind's eye, you can see that the desired rates are sitting on the diagonal. The other rates are superfluous. If you have experimented much with dyadic transpose (⍉), you know that it can return the diagonal elements of a matrix argument by providing a left argument of 1 1 (in origin 1). Therefore, a correct expression to solve this problem is:

```
1 1 ⍉DEPN[YEAR;MONTH]
```

Unfortunately, this expression requires room in your workspace for the temporary 1000 by 1000 table. This may cause a WS FULL error. Even if available workspace is not a problem, the extraction of 1,000,000 rates when you need only 1000 is extremely inefficient.

An alternate approach to this problem is to view DEPN as a vector. The vector has 240 elements and is derived by raveling the matrix DEPN. Our job is to pack the vectors YEAR and MONTH into a single vector of indices into the raveled DEPN. The desired indices may be computed by the expression,

```
MONTH+12×YEAR-1
```

Thus, the desired result may be computed from the expression,

```
(,DEPN)[MONTH+12×YEAR-1]
```

This type of problem is called a "scattered point indexing" problem because the desired elements to be selected from the matrix are scattered throughout it. Normal matrix indexing (MAT[ROWS;COLS]) is useful only when the elements to be selected are in a rectangular pattern.

Let us state the scattered point indexing solution in general terms:

```
(,MATRIX)[COLUMNINDEX+NUMCOLS×ROWINDEX-1]
```

For a 3-dimensional array, the solution is:

```
(,ARRAY)[COLUMNINDEX+(NUMCOLS×ROWINDEX-1)+(NUMCOLS×NUMROWS)×
         PLANEINDEX-1]
```

or:

```
(,ARRAY)[COLUMNINDEX+NUMCOLS×(ROWINDEX-1)+NUMROWS×
         PLANEINDEX-1]
```

When performing scattered point indexing in origin 0 (⎕IO←0), the "-1" portions of the above expressions disappear:

Matrix (origin 0):

```
(,MATRIX)[COLUMNINDEX+NUMCOLS×ROWINDEX]
```

3-D array (origin 0):

```
(,ARRAY)[COLUMNINDEX+NUMCOLS×ROWINDEX+NUMROWS×PLANEINDEX]
```

For this reason, scattered point indexing is frequently done in origin 0.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: What algorithm may be used to return the unique values (UN) from a numeric vector (NV)? The unique values (UC) from a character vector (CV)?

TOPIC: Unique (Distinct) Values

Determination of the unique, or distinct, values is a common problem in the world of data processing. For example, given 1000 sales transactions which each include the salesperson number, you may want to compile a list of the numbers of the salespeople who had sales. If only 60 salespeople accounted for all 1000 sales, you would want to determine the numbers of those 60 salespeople. (You might also want to know the number of sales and the total dollar value of the sales attributed to each salesperson. These topics are covered in the next chapter.)

To illustrate the algorithms discussed in this section, we will use the following vectors:

```
      NV
30 20 20 30 10 50 10 10
      CV
BOOKKEEPER
```

Our task is to return the vectors UN and UC:

```
      UN
30 20 10 50
      UC
BOKEPR
```

Since the problem of determining distinct values can be viewed as a searching problem, the most obvious algorithm uses the APL searching primitive, dyadic ⍳. Consider the result when you search the elements of a distinct vector for its own elements:

```
      8 9 7 15 ⍳ 8 9 7 15
1 2 3 4
```

However, if the elements are not distinct, the pattern of the result is not so regular:

```
      NV⍳NV
1 2 2 1 5 6 5 5
```

In fact, wherever the result deviates from the vector of generated indices (⍳⍴NV), the corresponding element is a repeat value, i.e. has occurred earlier in the vector. To flag the distinct values then:

```
      (NV⍳NV)=⍳⍴NV
1 1 0 0 1 1 0 0
```

And to select the distinct values:

```
          ⎕←UN←((NV⍳NV)=⍳⍴NV)/NV      * Algorithm 1 *
    30 20 10 50
```

Notice that this algorithm returns the distinct values in the same order as they first appear in the target vector. This algorithm also works on character vectors:

```
          ((CV⍳CV)=⍳⍴CV)/CV
    BOKEPR
```

Since this algorithm depends upon the behavior of dyadic ⍳, it may also be used with functions which emulate dyadic ⍳. For example, if your task is to determine the distinct rows in the character matrix NAMES, you may do so with the following expression (given CMIOTA from the previous chapter):

```
    ((NAMES CMIOTA NAMES)=⍳1⍴⍴NAMES)⌿NAMES
```

When using this algorithm on a large vector (say, 2000 or more elements), the dyadic ⍳ portion of the algorithm may require a significant amount of processing time. A more efficient algorithm may be constructed which uses the (typically very efficient) grade-up (⍋) primitive function.

Sort the vector:

```
          ⎕←SORTED←NV[⍋NV]
    10 10 10 20 20 30 30 50
```

Shift the elements of the sorted vector to the right and compare to the sorted vector to flag the first distinct value in each run of like values:

```
          ¯1⌽SORTED
    50 10 10 10 20 20 30 30
          ⎕←FIRST←SORTED≠¯1⌽SORTED
    1 0 0 1 0 1 0 1
```

Unfortunately, if the values in the vector SORTED are all the same, the vector FIRST will be all zeros. Yet the first value of FIRST should be 1. The following expression will set the first (in either origin) element to 1, and will have no effect if FIRST is an empty vector:

```
    FIRST[⍳×⍴FIRST]←1
```

Finally, select the first distinct value in each run:

```
          ⎕←UN←FIRST/SORTED
    10 20 30 50
```

Notice that this algorithm returns the distinct values in ascending order (descending if ⍒ is used). The entire algorithm follows:

```
SORTED←NV[⍋NV]                      * Algorithm 2 *
FIRST←SORTED≠¯1⌽SORTED
FIRST[⍳×⍴FIRST]←1
UN←FIRST/SORTED
```

The algorithm works on character vectors once you manage to sort the characters. If your implementation of APL supports dyadic grade-up, you may replace the first statement by:

```
SORTED←CV[⎕AV⍋CV]
```

If it does not, you may replace the first statement by:

```
SORTED←CV[⍋⎕AV⍳CV]
```

As with numeric vectors, the distinct elements in the final result are in ascending order (where ⎕AV, the atomic vector, defines the collating sequence).

You may determine the distinct rows (UNAMES) of a character matrix (NAMES) using this algorithm if you are able to sort the matrix. If your implementation of APL supports dyadic grade-up, do the following:

```
SORTED←NAMES[⎕AV⍋NAMES;]
FIRST←∨/SORTED≠¯1⊖SORTED
FIRST[×⍳⍴FIRST]←1
UNAMES←FIRST⌿SORTED
```

If your implementation does not support dyadic grade-up, use the CGRADEUP function developed in the previous chapter.

The expression CV[⍋⎕AV⍳CV] in the statement above suggests another algorithm for determining the distinct elements of a character vector. Consider the meaning of the expression CV⍳⎕AV, or better still, the expression ⎕AV∊CV. The result of the latter expression is a 256 element Boolean vector that flags the elements of ⎕AV which are in CV. Since the elements of ⎕AV are distinct by definition, the remaining step is to use the Boolean vector to select the corresponding elements from ⎕AV. The complete algorithm:

```
UC←(⎕AV∊CV)/⎕AV                     * Algorithm 3 *
```

Like algorithm 2, this algorithm returns the distinct values in ascending order (according to ⎕AV). In some implementations of APL, ∊ is extremely fast on character data. In others, it is not. If not, use algorithm 1 or 2.

Extending this algorithm to numeric (or at least integer) data will produce ridiculous expressions like:

```
UN←((⍳1E30)∊NV)/⍳1E30
```

which will hopefully generate a WS FULL error message rather than run into the next century. When we stop to consider the philosophy of the algorithm rather than its implementation, however, a very clever algorithm emerges.

The philosophy is to consider the nature of the values in NV. The expression above assumes that all elements of NV are integers between 1 and 1E30 (origin 1). A more realistic problem would have the values between, say, 1 and 100. Let us view these values as indices rather than numbers and use them as such. Initialize a Boolean vector to have 100 zeros:

```
BIT←100⍴0
```

Use the numeric (index) vector to index assign 1s into the Boolean vector:

```
BIT[NV]←1
```

Duplicates in NV are essentially discarded. The final step is similar to the final step in algorithm 3:

```
UN←BIT/⍳100
```

Notice that this algorithm returns the distinct values in ascending order, that it works only on positive integers (⎕IO and up), that the maximum value (MAX) must be known and not too large (else WS FULL on MAX⍴0). This algorithm is extremely fast on vectors of any size and for any APL implementation. The entire algorithm follows:

```
BIT←(MAX+~⎕IO)⍴0             * Algorithm 4 *
BIT[NV]←1
UN←BIT/⍳⍴BIT
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: For each of the unique-value algorithms presented in the last section, what additional logic must be included to also return the indices (INDS) of the numeric vector (NV) or character vector (CV) into the vector of distinct values (UN or UC)? For example, suppose NV and UN have the following values:

```
      NV
30 20 20 30 10 50 10 10
      UN
30 20 10 50
```

The desired value if INDS is:

```
      INDS
1 2 2 1 3 4 3 3
```

TOPIC: Translating Distinct Values to Distinct Indices

The obvious solution to this problem is:

```
INDS←UN⍳NV        or       INDS←UC⍳CV
```

These indices are useful when performing frequency counts or accumulations. These topics are covered in the next chapter.

While obvious, the dyadic ⍳ approach may not be the most efficient technique for computing these indices, depending upon the unique-value algorithm being used.

In the case of Algorithm 1, a dyadic ⍳ is already being performed so there is no need to do it again. Instead:

```
     UN←(B←(I←NV⍳NV)=⍳⍴NV)/NV        * Algorithm 1 *
     INDS←(B\⍳⍴UN)[I]
or:  INDS←(+\B)[I]-~⎕IO
```

The expansion in the second statement or the cumulative sum in the third are typically more efficient than another dyadic ⍳.

In the case of Algorithm 2, the grade-up operation removes the need to perform dyadic ⍳. Instead:

```
SORTED←NV[G←⍋NV]                  * Algorithm 2 *
FIRST←SORTED≠¯1⌽SORTED
FIRST[⍳×⍴FIRST]←1
UN←FIRST/SORTED
FIRST[⍳×⍴FIRST]←⎕IO
INDS←(⍴FIRST)⍴0
INDS[G]←+\FIRST
```

The fifth statement sets the first element of FIRST to 1 or 0 depending upon the index origin. The statement is needed only if you may be working in origin 0 since the first element of FIRST will already be 1 (unless FIRST is empty). The plus scan (+\) and indexing operations in the last statement are typically more efficient than another dyadic ⍳. The last two statements are required to "unsort" the indices so that they are in the order of the elements of NV rather than of SORTED. A simpler, though less efficient, statement may be used in place of the last two statements:

```
INDS←(+\FIRST)[⍋G]
```

In the case of Algorithm 3, there is no way to avoid the dyadic ⍳. So:

```
UC←(⎕AV∊CV)/⎕AV                  ⋆ Algorithm 3 ⋆
INDS←UC⍳CV
```

In the case of Algorithm 4, there is no need to do the dyadic ⍳ for the same reason there is no need to do any searching when determining the distinct values:

```
BIT←(MAX+~⎕IO)⍴0                 ⋆ Algorithm 4 ⋆
BIT[NV]←1
UN←BIT/⍳⍴BIT
INDS←(BIT\⍳⍴UN)[NV]
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 337 to 339)

1. What APL expression will select every other element (1st, 3rd, 5th, ...) of a vector V which has an even number of elements?

2. What APL expression will return a vector of the elements on the diagonal of the square matrix M?

3. What APL expression will return the Nth column of the matrix M as a vector, where N is the value of the last element of the vector V?
~~~~

4. In a character matrix NAMES of passenger names, the last names precede the first names and are separated from them by a slash (/). What APL expression(s) will replace the slashes with commas (,)?

5. As an actuary, you need to determine the mortality rates for a set of 500 policyholders. You have a 3 dimensional "select and ultimate" table of mortality rates named RATES. RATES has shape 2 100 16. The first dimension is sex (female, male). The second dimension is issue age (0 to 99). The third dimension is duration from issue (0 to 15 years). You have three 500 element vectors: SEX, IAGE, DUR. The elements of these vectors are in 1-to-1 correspondence with the 500 policyholders. SEX indicates the policyholder's sex (0=female; 1=male). IAGE indicates the issue age (0 to 99). DUR indicates duration from issue (0 to 15 years).

   A. Construct the 500 element vector MRATES of the mortality rates for these 500 policyholders.

   B. Suppose some of the durations from issue (elements of DUR) exceed 15. For these policies, use duration 15 (the "ultimate" duration) but increase the issue age by the amount that the duration exceeds 15. Construct MRATES.

   C. Suppose you receive a memo which informs you that 40 of the elements in RATES are incorrect and must be modified. From the information in the memo, you construct 4 vectors of length 40: NEWRATES, the correct rates; NEWSEX, the sexes for the new rates; NEWIAGE, the issue ages for the new rates; NEWDUR, the durations for the new rates. Insert the new rates into RATES.

6. Write the monadic function UNQNV which returns the distinct elements of its numeric vector argument. Write UNQCV to return the distinct elements of its character vector argument. Write UNQCM to return the distinct rows of its character matrix argument.

   Write the dyadic functions UNQI1 and UNQI0 which return the distinct elements of their index vector right arguments (origin 1 or 0 indices respectively). The left argument of UNQI1 or UNQI0 is a scalar of the number of possible indices. For example, a left argument of 5 for UNQI0 implies that all indices in the right argument are elements of the set 0 1 2 3 4.

   In addition to the distinct values, each function should compute the indices of the right argument values into the resulting distinct values. Assign the indices to the global variable <ind>. Place lamps (⍝) in front of the lines which compute <ind> so that the indices are not computed unless the lamps are removed.

Chapter 7

FREQUENCY COUNTS, ACCUMULATIONS AND CROSS-TABULATIONS

The often used APL expression +/A adds up the elements of the array A in such a way that one of the dimensions of A is eliminated, or "reduced". If A is a 1000 element vector, the result is a scalar whose one element is the sum of the 1000 elements of A. Typically, the elements of a vector represent measurements or counts of respective real world items. Then, the result of the expression +/A represents the sum of the measurements or counts for all items.

Frequently in the business world, we tend to categorize items rather than lump them together. For example, 1000 sales transactions may be categorized by retail outlet or by salesperson or by product or by day, and so on. In such an environment, we may want to "reduce" the 1000 element vector of invoice amounts into 10 sums (say, by salesperson) rather than into just a single grand total. Such a problem has traditionally been called an "accumulation" problem in APL. Naturally, to solve such a problem, we need a corresponding 1000 element vector whose values indicate the salesperson responsible for each transaction (e.g. salesperson number). Such a vector is called a "classification" vector.

By analyzing the classification vector alone, we can answer questions such as, "For how many transactions was each salesperson responsible?" Such a problem has been called a "frequency count" problem in APL.

If a second classification vector is available which represents, say, day of the week of the sale, we may want to look at sales broken down (i.e. added up) by both salesperson and day of the week. In this case, we want to reduce the 1000 element vector of invoice amounts into a 10 (salesperson) by 7 (days of the week) matrix. Such a problem has been called a "cross-tabulation" problem in APL.

To avoid ambiguity from existing terminology, we present the following terminology and definition:

"n-way plus reduction on dimension d of array A by classification 1, classification 2, ..., and classification n"

> The summarization of array A across dimension d such that dimension d is replaced by n new dimensions whose magnitudes are the number of classes defined for the corresponding classifications 1, 2, ..., n.

Let us try this terminology on an example. Suppose you have information on 1000 life insurance policies. A subset of the information is listed below:

Issue Age [0 to 99] (IAGE)	Sex [M,F] (SEX)	Underwriting Class [S,A,B,C,D] (UCLASS)	Death Benefit (DBEN)	Annual Premium (APREM)
36	M	C	150	130
27	M	S	100	75
42	F	S	80	85
50	M	B	100	210
:	:	:	:	:

The names IAGE, SEX, UCLASS, DBEN and APREM represent the names of the APL variables containing the corresponding data. Each variable is a 1000 element vector. SEX and UCLASS are character vectors and the rest are numeric vectors. Here are a few of the plus reductions that can be performed on these data:

1. The 0-way plus reduction of APREM. This is simply +/APREM.

2. The 1-way plus reduction of APREM by SEX. This is the 2 element vector (+/(SEX='M')/APREM),(+/(SEX='F')/APREM).

3. The 3-way plus reduction of DBEN by AGE, SEX and UCLASS. This is a 100 (ages) by 2 (sexes) by 5 (underwriting classes) array whose elements contain the sums of the elements of DBEN for each combination of AGE, SEX and UCLASS. Notice that the result coincidentally has 1000 elements, the same number of elements as DBEN, the vector being "reduced". The number of elements has not been reduced at all. In fact, if another classification were added, the result would contain more elements than the vector being reduced. Most of the elements of the result will be 0. The data will have become so finely classified that each "cell" (i.e. combination of classes) in the result contains at best a few policies.

4. The 1-way plus reduction of 1 (or (ρSEX)ρ1) by SEX. This is the frequency count by sex: (+/SEX='M'),(+/SEX='F').

5. The 2-way plus reduction on dimension 1 of DBEN,[1.5]APREM by AGE and UCLASS. This is a 100 (ages) by 5 (underwriting classes) by 2 (columns...the dimension not reduced) array whose elements contain the sum of the elements of DBEN (column 1) and APREM (column 2) for each combination of age and underwriting class.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: What is the frequency count (or 1-way plus reduction of 1) by issue age (IAGE) for the above insurance policies? Call the result F. Accumulate (or 1-way plus reduce) APREM by the same ages. Call the result A.

TOPIC: One-Way Plus Reductions

Let us solve this problem first by using the classical APL approach, which is simple but ignores efficiency considerations. The distinct issue ages can be determined by using an algorithm discussed in the previous chapter:

```
DIA←((IAGEιIAGE)=ιρIAGE)/IAGE
```

Next, use outer product to compare the vector of issue ages to the vector of distinct issue ages:

```
M←DIA∘.=IAGE
```

The result is a Boolean matrix with one row per distinct issue age (say 40) and one column per policy (say 1000). Each column has exactly one 1, marking the distinct age (row) to which that policy (column) corresponds. The frequency count by distinct issue age follows directly:

```
F←+/M
```

and the accumulation of APREM by distinct issue age is not far behind:

```
A←M+.×APREM
```

The dimensions of M (say 40 by 1000) and APREM (say 1000) conform along their inner coordinates allowing the matrix multiplication (+.×) which reduces the inner coordinates (1000) and returns a vector with the same number of elements as M has rows (40).

This approach is simple. However, it is inefficient and is prone to WS FULL errors. Its inefficiencies stem from the many needless
~~~~

comparisons and computations which take place in the ∘.= and +.× functions. Its WS FULL tendencies are caused by the potentially gigantic result of the outer product (∘.=).

A more efficient approach uses the sort-and-shift techniques employed in the previous chapter. Begin by sorting the ages in ascending order (retaining the grade vector):

```
GRADE←⍋IAGE
SORTED←IAGE[GRADE]
```

Shift the elements of the sorted vector to the left and compare to the sorted vector to flag the last distinct value in each run of like values:

```
LAST←SORTED≠1⌽SORTED
```

Unfortunately, if the values in the vector SORTED are all the same, the vector LAST will be all zeros. Yet the last value of LAST should be 1. To avoid this problem, you may instead use an odd expression like the following (which assumes that no policyholder has issue age ¯99):

```
LAST←SORTED≠1↓SORTED,¯99
```

Sometimes, using an arbitrary number such as ¯99 is not feasible. Perhaps the vector could contain any conceivable value. The following alternate expressions will set the last (in either origin) element to 1, and will have no effect if SORTED is an empty vector:

```
LAST←SORTED≠1⌽SORTED
LAST[(¯1+⍴LAST)+⍳×⍴LAST]←1
```

Select the last distinct value in each run:

```
DIA←LAST/SORTED
```

You may have noticed that we determined the distinct values differently than in the previous chapter. There, we constructed a bit vector FIRST which flagged the first 1 of each run of like values; here, we constructed a bit vector LAST which flagged the last 1 of each run. The reason for this minor change of algorithm is that LAST is more useful for determining the 1-way plus reductions. Consider the meaning of the expression:

```
CUM←LAST/⍳⍴LAST
```

CUM has one element per distinct issue age. The values are the cumulative frequency counts (in origin 1). If we can undo the cumulative effect, we will have the frequency counts. Cumulative sums are undone by taking the first differences:

```
F←CUM-(⍴CUM)⍴0,CUM
```

To accumulate APREM, we use the same approach. Reorder APREM to be in 1-to-1 correspondence with the sorted issue ages (SORTED); determine the cumulative sum; select the last element for each run:

```
CUM←LAST/+\APREM[GRADE]
```

Compute the first differences to get the desired result:

```
A←CUM-(ρCUM)ρ0,CUM
```

This algorithm is extremely efficient, especially on large vectors. As vectors get larger, the required processing time generally increases linearly for this grade-up (⍋) based algorithm; but it increases exponentially for the outer product (∘.=) based algorithm.

This algorithm works just as well on character classification vectors (e.g. UCLASS or SEX) as it does on numeric classification vectors (e.g. IAGE). Just begin by converting the characters to indices. For example:

```
GRADE←⍋'SABCD'⍳UCLASS

   or

GRADE←'SABCD'⍋UCLASS
```

Use the latter expression if your implementation of APL supports dyadic grade-up, and the former expression if it does not.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Generate a 3 by 10 by 2 by 5 array (SMRY) which summarizes (or 3-way plus reduces) the above insurance policies by age, sex and underwriting class. The definition of SMRY follows:

SMRY[1;;;]	Total death benefits
SMRY[2;;;]	Total annual premiums
SMRY[3;;;]	Frequency count (number of policies)
SMRY[;I;;]	Age group I (1 is 0-9; 2 is 10-19; ...; 10 is 90-99)
SMRY[;;J;]	Sex group J (1 is male; 2 is female)
SMRY[;;;K]	Underwriting group K (1 is Standard; 2 is A; 3 is B; 4 is C; 5 is D)
~~~~

TOPIC: N-Way Plus Reductions

This problem is awkward to solve using inner product and outer product techniques. The solution is confusing, WS FULL prone and inefficient.

A neater solution arises when you look at the problem backwards. Consider the result SMRY. The three elements defined by SMRY[;I;J;K] are affected by just those policies belonging to age group I, sex group J and underwriting group K. Conversely, each policy affects exactly one set of three elements in the result.

To simplify the discussion, let's consider just SMRY[3;;;], i.e. frequency counts. The first element of this array represents the number of policies in age group 1, sex group 1 and underwriting group 1. The second element of this (raveled) array represents the number of policies in age group 1, sex group 1 and underwriting group 2. And so on. The last (100th) element of this (raveled) array represents the number of policies in age group 10, sex group 2 and underwriting group 5.

By considering the result as a vector, you may treat this problem as a 1-way plus reduction (100 classes) rather than as a 3-way plus reduction (10 by 2 by 5 classes). All that remains is to pack together the values of the three classification vectors (IAGE, SEX, UCLASS) such that the resulting packed classification vector has values which distinctly identify the cell of the result affected by the corresponding policy.

The ideal packing scheme is one which converts the classification values for each policy directly into the index of the affected element in the raveled result. For example:

IAGE	SEX	UCLASS	Raveled Result Index (RRI)	
36 (4)	M (1)	C (4)	34	(origin 1)
27 (3)	M (1)	S (1)	21	
42 (5)	F (2)	S (1)	46	
50 (6)	M (1)	B (3)	53	
:	:	:	:	

The formula being used here is:

```
RRI←UCLASSINDEX+(5×SEXINDEX-1)+(10×IAGEINDEX-1)
```

When working in origin 0, the formula is a bit simpler:

IAGE	SEX	UCLASS	Raveled Result Index (RRI)	
36 (3)	M (0)	C (3)	33	(origin 0)
27 (2)	M (0)	S (0)	20	
42 (4)	F (1)	S (0)	45	
50 (5)	M (0)	B (2)	52	
:	:	:	:	

```
RRI←UCLASSINDEX+(5×SEXINDEX)+(10×IAGEINDEX)
```

The 5 in the above formula refers to the number of elements in each row of the result (i.e. the number of UCLASS classes). The 10 refers to the number of elements in each plane (i.e. the number of UCLASS classes times the number of SEX classes).

The computations of UCLASSINDEX, SEXINDEX and IAGEINDEX are straightforward:

```
⎕IO←0
UCLASSINDEX←'SABCD'⍳UCLASS
SEXINDEX←'F'=SEX
IAGEINDEX←⌊IAGE÷10
```

The elements of the vector RRI are all integers between 0 and 99. You may then use logic from the prior section to determine the distinct values in RRI and the corresponding frequencies (1-way plus reduction of 1) and 1-way plus reductions of APREM and DBEN. If you initialize the result to be an all-zero vector of the desired length (the length of the raveled result), you may simply index assign the derived frequencies and sums using the corresponding distinct indices from RRI. Finally, reshape the result to the proper shape.

The complete logic follows:

```
 ⍝ Use origin 0 throughout:
  ⎕IO←0
⍝ Index in result of column affected by policy:
  UCLASSINDEX←'SABCD'⍳UCLASS
 ⍝ Index of row affected:
  SEXINDEX←'F'=SEX
 ⍝ Index of plane affected:
  IAGEINDEX←⌊IAGE÷10
 ⍝ Index in raveled SMRY[0;;;] affected:
  RRI←UCLASSINDEX+5×SEXINDEX+2×IAGEINDEX
 ⍝ Sort result indices in ascending order:
  GRADE←⍋RRI
  SORTED←RRI[GRADE]
 ⍝ Shift to left and compare to flag last distinct values:
  LAST←SORTED≠1↓SORTED,¯1
 ⍝ Select last distinct value in each run:
  URRI←LAST/SORTED
```

```
⍝ Initialize 3 raveled all-zero arrays for SMRY[0;;;],
⍝ SMRY[1;;;] and SMRY[2;;;]:
 SMRY0←SMRY1←SMRY2←(10×2×5)⍴0
⍝ One-way plus reduce DBEN and insert into SMRY0:
 CUM←LAST/+\DBEN[GRADE]
 SMRY0[URRI]←CUM-(⍴CUM)⍴0,CUM
⍝ Ditto for APREM into SMRY1:
 CUM←LAST/+\APREM[GRADE]
 SMRY1[URRI]←CUM-(⍴CUM)⍴0,CUM
⍝ Ditto for frequency count into SMRY2 (origin 0):
 CUM←LAST/⍳⍴LAST
 SMRY2[URRI]←CUM-(⍴CUM)⍴¯1,CUM
⍝ Catenate and reshape:
 SMRY←3 10 2 5 ⍴SMRY0,SMRY1,SMRY2
 ⎕IO←1
```

The logic above can be shortened somewhat if you choose to view the problem as a single 3-way plus reduction on the last dimension of the three row matrix whose rows are DBEN, APREM and all 1s. Then, the logic is:

```
   :
   :
URRI←LAST/SORTED
SMRY←(3,10×2×5)⍴0
CUM←LAST/+\(DBEN,[0]APREM,[¯0.5]1)[;GRADE]
SMRY[;URRI]←CUM-(⍴CUM)↑0,CUM
SMRY←3 10 2 5 ⍴SMRY
⎕IO←1
```

Once this 4 dimensional array has been constructed, many questions can be answered by performing a few simple indexing and plus reduction operations. For example (origin 1):

1. How many males and females?

   ```
   +/[1]+/[3]SMRY[3;;;]
   ```

2. What is the total death benefit by sex and underwriting class (2 row, 5 column matrix)?

   ```
   +/[1]SMRY[1;;;]
   ```

3. What is the average annual premium by age group and sex (10 row, 2 column matrix)?

   ```
   ÷/[1]+/[4]SMRY[2 3;;;]
   ```

4. What is the death benefit, annual premium and frequency breakdown by age group and underwriting class, where ages are broken into 5 groups: 0 to 19; 20 to 39; ...; 80 to 99?

   ```
   +/[3] 3 5 2 5 ⍴+/[3]SMRY
   ```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Generate a 2 by 10 by 5 array (MAX) which contains the maximums of (or 3-way maximum reduces) the above insurance policies by age, sex and underwriting class. The definition of MAX follows:

MAX[1;;;] Maximum death benefits

MAX[2;;;] Maximum annual premiums

MAX[;I;J;K] Maximum death benefit and annual premium for age group I, sex group J, underwriting group K.

TOPIC: N-Way Maximum and Minimum Reductions

This problem is no different from the problem in the prior section except an n-way maximum reduction is being performed instead of an n-way plus reduction. Consequently, the same solution works, up to a point. That point in the above solution is:

```
SMRY0←SMRY1←SMRY2←(10×2×5)⍴0
CUM←LAST/+\DBEN[GRADE]
SMRY0[URRI]←CUM-(⍴CUM)⍴0,CUM
```

Unfortunately, you may not simply substitute ⌈\ for +\. The algorithm happens to work for +\ because of the nature of addition. We must find a comparable algorithm which will work for maximum (and hopefully minimum).

One such algorithm involves the grade-up (⍋) function. The idea behind the algorithm is to sort the values within their respective classes (i.e. within like values of SORTED) and then use LAST to select the maximum in each class. Stated differently, you must perform a two-key sort where RRI is the major key and the data (DBEN) is the minor key. Do the minor key first:

```
G←⍋DBEN
```

and then the major key:

```
G←G[⍋RRI[G]]
```

Use this grade vector to reorder the values, and use LAST to select the maximum in each class:

```
MAX←LAST/DBEN[G]
```
~~~~

or quicker:

```
MAX←DBEN[LAST/G]
```

The final step is to index assign MAX into the result variable (MAX0). When performing the n-way plus reduction, we initialized SMRY0 as all zeros so that those classes which were not represented by any policies (i.e. cells not index assigned) would show a plus reduction result of zero (just as +/⍳0 is 0). In mathematical terminology, the "identity element" of plus is 0. Likewise, you should initialize MAX0 as a vector filled with the identity element for maximum, which is negative infinity and is returned as nearly as possible by the expression ⌈/⍳0. The rest of the solution is:

```
MAX0←MAX1←(10×2×5)⍴⌈/⍳0
MAX0[URRI]←MAX

G←⍋APREM
G←G[⍋RRI[G]]
MAX←APREM[LAST/G]
MAX1[URRI]←MAX

MAX←2 10 2 5 ⍴MAX0,MAX1
⎕IO←1
```

A different algorithm solves the problem without the use of grade-up. It uses instead maximum scan (⌈\). In order for maximum scan to be useful, the values in each subsequent class must first be shifted up the number scale so that the maximum value of an earlier class does not shadow the maximum value of a subsequent class. To illustrate, suppose the values of LAST and DBEN[GRADE] are as follows:

```
      LAST
0  0  1  0  0  0  1  0  1  1
      DBEN[GRADE]
20 10 15 15 35 20 25 20 25 30
```

Suppose we add 100 to each value in the first class, 200 to the second class, 300 to the third class and 400 to the fourth class:

```
120 110 115 215 235 220 225 320 325 430
```

The maximum scan of this vector is:

```
120 120 120 215 235 235 235 320 325 430
```

Using LAST to select from this vector produces:

```
120 235 325 430
```

Subtracting 100, 200, 300 and 400 from the respective classes gives the desired maximums by class:

```
20 35 25 30
```

For this logic to work, you must shift (add to) the values of each class an amount which is at least equal to the difference between the maximum value in the preceding class and the minimum value in this class. Since your task is to determine these very maximums, you cannot use these precise numbers. Rather, you can determine the difference between the maximum and minimum values for the entire data vector and use that amount.

The following are expressions which implement this algorithm as well as numbers which illustrate the procedure:

```
D←DBEN[GRADE]         20 10 15 15 35 20 25 20 25 30
DIF←(⌈/D)-⌊/D         25
FIRST←¯1⌽LAST         1 0 0 1 0 0 0 1 0 1
T←+\FIRST\DIF         25 25 25 50 50 50 50 75 75 100
U←LAST/T              25 50 75 100
R←D+T                 45 35 40 65 85 70 75 95 100 130
R←⌈\R                 45 45 45 65 85 85 85 95 100 130
MAX←(LAST/R)-U        20 35 25 30
```

The solution to the above problem using this algorithm can be written as follows:

```
    :
    :
URRI←LAST/SORTED
MAX0←MAX1←(10×2×5)⍴⌈/⍳0

D←DBEN[GRADE]
DIF←(⌈/D)-⌊/D
T←+\(¯1⌽LAST)\DIF
MAX0[URRI]←(LAST/⌈\D+T)-LAST/T

D←APREM[GRADE]
DIF←(⌈/D)-⌊\D
T←+\(¯1⌽LAST)\DIF
MAX1[URRI]←(LAST/⌈\D+T)-LAST/T

MAX←2 10 2 5 ⍴MAX0,MAX1
⎕IO←1
```

You can do both DBEN and APREM at once with some minor modifications:

```
    :
    :
URRI←LAST/SORTED
MAX←(2,10×2×5)⍴⌈/⍳0

D←(DBEN,[¯0.5]APREM)[;GRADE]
DIF←(⌈/D)-⌊/D
T←DIF∘.×+\¯1⌽LAST
MAX[;URRI]←(LAST/⌈\D+T)-LAST/T

MAX←2 10 2 5 ⍴MAX
```

There are potential problems with the maximum scan algorithm. If there are many classes (i.e. 1s in LAST) and if the difference between the maximum and minimum values of the data vector is large, the values in the vector on which the maximum scan is being performed will become immense. If they become too large (beyond 16 or so significant digits), precision will be lost and the results may be incorrect. Even if precision is not lost, the numbers may get large enough to require internal floating point (usually 8 bytes per element) representation rather than integer (usually 2 or 4 bytes per element) representation. Consequently, the chances of a WS FULL error will increase and processing speed will decline since floating point numbers require more processing time than do integers.

Despite these potential problems, the maximum scan algorithm is a useful and efficient algorithm.

If the problem at the beginning of this section were stated as a minimum reduction problem rather than a maximum reduction problem, the same two approaches could have been taken, with slight modifications. Here are the solutions:

[grade-up algorithm]

```
    :
    :
URRI←LAST/SORTED
MIN0←MIN1←(10×2×5)⍴⌊/⍳0

G←⍒DBEN
G←G[⍋RRI[G]]
MIN0[URRI]←DBEN[LAST/G]

G←⍒APREM
G←G[⍋RRI[G]]
MIN1[URRI]←APREM[LAST/G]

MIN←2 10 2 5 ⍴MIN0,MIN1
⎕IO←1
```

[minimum scan algorithm]

```
    :
    :
URRI←LAST/SORTED
MIN←(2,10×2×5)⍴⌊/⍳0

D←(DBEN,[¯0.5]APREM)[;GRADE]
DIF←(⌈/D)-⌊/D
T←DIF∘.×+\¯1⌽LAST
MIN[;URRI]←(LAST/⌊\D-T)+LAST/T

MIN←2 10 2 5 ⍴MIN
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose the variable ACTIVE is a Boolean vector with the same length as IAGE, SEX, DBEN, ... whose values indicate whether the corresponding policies are active. Generate a 10 by 2 by 5 Boolean array (ALL) whose elements indicate whether (1) or not (0) all of the insurance policies (as defined above) are active within each possible age, sex and underwriting class. That is, perform the 3-way and-reduction (∧/) of ACTIVE by age, sex and underwriting class.

ALL[I;J;K] The and-reduction of ACTIVE for age group I, sex group J, underwriting group K.

TOPIC: N-Way Logical Reductions

Once again, the solution to this problem is similar to that of an n-way plus reduction. However, you need to devise an algorithm for ∧/ comparable to that for +/:

```
SMRY0←(10×2×5)⍴0
CUM←LAST/+\DBEN[GRADE]
SMRY0[URRI]←CUM-(⍴CUM)⍴0,CUM
```

Unfortunately, you may not simply substitute ∧ for + in the above logic. One simple solution may be derived from the knowledge that the ∧/ is true if the +/ equals the frequency count. As with the ⌈/ and ⌊/ problems of the last section, you must begin by initializing the result with the identity element for the reduction function. The identity element for ∧ is one, i.e. 1=∧/⍳0. The explanation for the identity element is: for any Boolean array B, (1∧B) and (B∧1) always return exactly B, so 1 is the identity element for ∧. The algorithm is:

```
ALL←(10×2×5)⍴1
CUM←(LAST/⍳⍴LAST)-LAST/+\ACTIVE[GRADE]
ALL[URRI]←CUM=(⍴CUM)⍴¯1,CUM              (¯1 for origin 0)
```

A second algorithm takes advantage of the fact that (∧/B) produces the same result as (~∨/~B). To perform the ∨/, we need not compute the frequency count. Instead, we know that the ∨/ is true if the +/ is not equal to 0. The algorithm is:

```
ALL←(10×2×5)⍴1
CUM←LAST/+\~ACTIVE[GRADE]
ALL[URRI]←CUM=(⍴CUM)⍴0,CUM
```
~~~~

A third algorithm may be employed which is based entirely on Boolean techniques. Since these techniques are discussed in the Boolean Techniques chapter, the algorithm is presented here without explanation:

```
ALL←(10×2×5)ρ1
B←ACTIVE[GRADE]
CUM←(LAST≥B)/LAST
ALL[URRI]←(LAST/B)∧CUM/¯1⌽CUM
```

Notice that this algorithm uses no arithmetic function and makes heavy use of Boolean functions (≥, ∧ and /). In some implementations of APL, Boolean functions have been optimized to be extremely fast. In such implementations, the third algorithm will dramatically out-perform the first two. You should time the alternative algorithms in your own environment before deciding among them.

If the problem at the beginning of this section were stated instead as an or-reduction (any) problem rather than an and-reduction (all) problem, similar approaches could have been taken, with slight modifications. Here are the solutions:

[sum algorithm]

```
ANY←(10×2×5)ρ0                    (note:  0=∨/ι0)
CUM←LAST/+\ACTIVE[GRADE]
ANY[URRI]←CUM≠(ρCUM)ρ0,CUM
```

[Boolean techniques algorithm]

```
ANY←(10×2×5)ρ0
B←ACTIVE[GRADE]
CUM←(LAST∨B)/LAST
ANY[URRI]←(LAST/B)≥CUM/¯1⌽CUM
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Define the syntax of utility functions for performing frequency counts, accumulations and cross-tabulations in APL.

TOPIC: N-Way Reduction Utility Functions

We will approach this problem by first imagining an extension to APL which can solve the problems of the previous sections. Imagine an
~~~~

extension to the definition of the primitive APL reduction operator (@/) so that it will accept the following dyadic syntax:

```
r←(dshape;civec1;civec2;...;civecN)@/[d]array
```

where:

array = array being reduced;

d = dimension of array being reduced;

dshape = the shape to which dimension d is "reduced"; dshape has N elements if an N-way reduction is to be performed;

civeci = the class index vector for classification i (1≤i≤N for an N-way reduction); civeci has the same length as dimension d and its values are indices (origin dependent) which identify the class indices into which the corresponding dimension d arrays are to be grouped; the values of civeci are all elements of ⍳dshape[i];

r = the N-way (where N=⍴dshape) @ reduction (where @ is any scalar dyadic function) on dimension d of array by classifications cvec1, cvec2, ..., cvecN.

This syntax calls for multiple left arguments (N+1 for an N-way reduction) where the arguments are separated by semicolons (;) and are enclosed in parentheses. Since the monadic form of reduction is a 0-way reduction, it is equivalent to the dyadic form in which a single empty vector left argument is provided.

```
r←(⍳0)@/[d]array       ←→       r←@/[d]array
```

Note that the dyadic reduction functions (except 0-way reduction) are origin sensitive since the left arguments contain indices.

We may illustrate the use of this syntax by using it to solve the problems of the previous sections. We assume origin 1.

One-Way Plus Reductions:

```
DIA←((IAGE⍳IAGE)=⍳⍴IAGE)/IAGE
IAGEIND←DIA⍳IAGE
F←(⍴DIA;IAGEIND)+/1
A←(⍴DIA;IAGEIND)+/APREM
```

N-Way Plus Reductions:

```
UCLASSIND←'SABCD'⍳UCLASS
SEXIND←'MF'⍳SEX
IAGEIND←1+⌊IAGE÷10
SMRY←(10 2 5;IAGEIND;SEXIND;UCLASSIND)+/DBEN,[1]APREM,[.5]1
```

N-Way Maximum and Minimum Reductions:

```
MAX←(10 2 5;IAGEIND;SEXIND;UCLASSIND)⌈/DBEN,[.5]APREM
MIN←(10 2 5;IAGEIND;SEXIND;UCLASSIND)⌊⌿DBEN,[1.5]APREM
```

N-Way Logical Reductions:

```
ALL←(10 2 5;IAGEIND;SEXIND;UCLASSIND)∧/ACTIVE
ANY←(10 2 5;IAGEIND;SEXIND;UCLASSIND)∨/ACTIVE
```

The syntax of dyadic reduction as described in here is adequate as a model for user-defined utility functions. Unfortunately, current APL systems do not allow multiple left arguments (unless they are packed together into a single "nested" array.) Also, optional arguments (e.g. the d in +/[d]) are not allowed. Therefore, we must make compromises.

One possible syntax is the following:

```
r←(dshape,cind1,cind2,...,cindN,{d}) PLUSRED array
                                     MAXRED
                                     MINRED
                                     ANDRED
                                     ORRED
```

where:

array = array being reduced;

d = dimension of array being reduced;

dshape = the shape to which dimension d is "reduced"; dshape has N elements if an N-way reduction is to be performed; d is optional and defaults to the last dimension of array if omitted;

cindi = the numeric suffix of the name of the global class index vector for classification i (1≤i≤N for an N-way reduction); the name of the vector is 'I',⍕cindi (e.g. I3 or I6); the class index vectors have the same length as dimension d of array and their values are indices (origin dependent) which identify the class indices into which the corresponding dimension d arrays are to be grouped; the values of the class index vectors are all elements of ⍳dshape[i];

r = the N-way (where N=⍴dshape) @ reduction (where @ is + for PLUSRED, ⌈ for MAXRED, ...) on dimension d of array by the classifications identified by cind1, cind2, ..., cindN.

In the case of a scalar right argument (e.g. 1), PLUSRED replicates the scalar to a vector with the same length as the class index vectors. The effect is to return a frequency count by class (times the value of the scalar).

We may illustrate the use of these utility functions by using them to solve the problems of the previous sections. We assume origin 1.

One-Way Plus Reductions:

```
DIA←((IAGE⍳IAGE)=⍳⍴IAGE)/IAGE
I1←DIA⍳IAGE
F←((⍴DIA),1) PLUSRED 1
A←((⍴DIA),1) PLUSRED APREM
```

N-Way Plus Reductions:

```
I2←'SABCD'⍳UCLASS
I3←'MF'⍳SEX
I4←1+⌊IAGE÷10
SMRY←(10 2 5, 4 3 2) PLUSRED DBEN,[1]APREM,[0.5]1
```

N-Way Maximum and Minimum Reductions:

```
MAX←(10 2 5, 4 3 2) MAXRED DBEN,[0.5]APREM
MIN←(10 2 5, 4 3 2, 1) MINRED DBEN,[1.5]APREM
```

N-Way Logical Reductions:

```
ALL←(10 2 5, 4 3 2) ANDRED ACTIVE
ANY←(10 2 5, 4 3 2) ORRED ACTIVE
```

Notice that the following pairs of expressions produce identical results:

```
(⍳0) PLUSRED array      ←→      +/array
 ⎕IO PLUSRED array      ←→      +⌿array
   2 PLUSRED array      ←→      +/[2]array
```

The writing of these utility functions is left as an exercise at the end of this chapter.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you wish to compute the 3-way plus reduction by age, sex and underwriting class of death benefits, annual premiums and frequency counts (as presented in the N-Way Plus Reduction section above.) How would you do this on one million policies?

TOPIC: N-Way Reductions on Files

So far, we have considered only arrays which can be easily manipulated within the active workspace. In this problem however, we would need several 1,000,000 element vectors: IAGE, SEX, UCLASS, DBEN, APREM. In many implementations of APL, the active workspace is not large enough to contain these variables. They must be broken into smaller pieces and stored on a file.

The chapter, File Design and Utilities, discusses the application of APL files for storing and manipulating large amounts of information. In that chapter, a number of alternative file organizations are described. One of them, the multi-set transposed file organization, is a likely candidate for storing the one million policies introduced above. Using that organization, the information is broken into smaller pieces and each piece is stored as a single file component. Suppose the pieces are 5000 element vectors. The file will consist of 1000 components (200 sets of 5 variables), each component containing a 5000 element vector.

To compute the required 3-way reduction, you will read in one set of the 5 variables (i.e. 5000 policies) at a time and apply the PLUSRED function on them. Accumulate the results as you go. After 200 iterations (i.e. sets), you will be done.

The file utility function EXECUTE is designed for this type of problem. It reads from file one set of information at a time for specified variables and then performs any specified computations on those variables. The left argument of EXECUTE identifies the file being used (FP) and the variables ("fields") required. The right argument of EXECUTE is a character vector representation of an APL
~~~~

expression to be executed once for each set. The variables F1, F2,... (where the n in Fn is included in the list of field numbers in the left argument) are assigned the values of the respective fields for the current set.

Suppose the variables required for this problem are located in the following fields of the file:

Field Number	Variable
3	IAGE
4	SEX
9	UCLASS
12	DBEN
13	APREM

You can add up the APREM variable (field 13) for all records on file with the following statements:

```
SUM←0
(FP,13) EXECUTE 'SUM←SUM++/F13'
```

The EXECUTE function will execute the expression SUM←SUM++/F13 once for each set on file. Before executing the expression, it will read from file the 5000 element vector of APREM values for the current set and will assign it to the variable name F13 (because the number 13 is in the left argument of EXECUTE).

To perform the desired 3-way reduction, we write the following function:

```
     ∇ SUM←XTAB;I3;I4;I9
[1]   ⍝ Returns the 3-way plus reduction by age, sex and
[2]   ⍝ underwriting class of death benefits, annual
[3]   ⍝ premiums and frequency counts.  Requires globals:
[4]   ⍝ F3 (IAGE), F4 (SEX), F9 (UCLASS), F12 (DBEN),
[5]   ⍝ F13 (APREM).  Requires fn:  PLUSRED.  Origin 1.
[6]   ⍝ Which age class?
[7]    I3←1+⌊F3÷10
[8]   ⍝ Which sex class?
[9]    I4←'MF'⍳F4
[10]  ⍝ Which underwriting class?
[11]   I9←'SABCD'⍳F9
[12]  ⍝ Perform the 3-way plus reduction:
[13]   SUM←(10 2 5, 3 4 9) PLUSRED F12,[1]F13,[0.5]1
     ∇
```

Then we use EXECUTE to execute this function once for each set of 5000 policies:

```
SUM←0
(FP,3 4 9 12 13) EXECUTE 'SUM←SUM+XTAB'
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you need to perform a 2-way plus reduction by A and B, another by B and C, and another by A and C. You could perform the 3-way plus reduction by A, B and C and then respectively plus reduce dimensions 3, 1 or 2 of the result to generate the desired arrays. Taking this approach to its extreme, you would always perform a single n-way plus reduction by every possible classification and then use monadic plus reduction to eliminate those dimensions not needed for particular reports. What are the problems which arise from taking this extreme approach? How can the problems be overcome?

TOPIC: Milky-Way Reductions

We define a new term:

Milky-Way reduction: an n-way reduction in which you reduce by every classification variable under the sun, generating a result which may have an astronomical number of elements.

Performing Milky-Way reductions can improve your productivity and reduce computer processing time consumed. Compare the following:

```
I7←1+⌊IAGE÷10             (by IAGE)
I8←'MF'⍳SEX               (by SEX)
I9←'SABCD'⍳UCLASS         (by UCLASS)
```

Milky-Way: SMRY←(10 2 5, 7 8 9) PLUSRED APREM

By SEX and UCLASS:

```
N-way:  (2 5, 8 9) PLUSRED APREM
Milky-Way:  +/[1]SMRY
```

By IAGE and UCLASS:

```
N-way:  (10 5, 7 9) PLUSRED APREM
Milky-Way:  +/[2]SMRY
```

By Sex:

```
N-way:  (2, 8) PLUSRED APREM
Milky-Way:  +/[1]+/[3]SMRY
```

Unfortunately, the number of elements in the result of a Milky-Way reduction may be astronomical. Your active workspace may not be big
~~~~

enough to contain them. Even if it can contain them, the process of plus reducing the array to a manageable size will be costly and time consuming because of the tremendous number of values.

Ironically, the majority of the values are zeros. For example, if you perform a 7-way reduction in which the 7 classifications respectively involve 5, 6, 7, 8, 9, 10 and 11 classes, the result (on a vector) will contain 1,663,200 elements (5×6×7×8×9×10×11), regardless of the length of the vector being reduced. If the vector contains 5000 elements, the result will contain at most 5000 nonzero values. If the entities represented by the 5000 elements are somewhat similar to one another, many of the entities will be classified the same. Then there will be fewer than 5000 nonzero values in the result, perhaps much fewer.

If we discard the zeros and retain just the nonzero values, the result of a Milky-Way reduction is more manageable. Of course, we must keep track of where the nonzero values belong in the result.

We propose the following functions:

```
        Δr←(dshape,cind1,cind2,...,cindN,{d}) ΔPLUSRED array
                                              ΔMAXRED
                                              ΔMINRED
                                              ΔANDRED
                                              ΔORRED
```

The meaning and syntax of these functions are identical to those of the PLUSRED, MAXRED, MINRED, ANDRED and ORRED functions introduced earlier in the chapter. The only difference between those functions and these is the result. The result of these functions is a "compressed Milky-Way array" which is a numeric vector whose elements are defined as follows (origin 0):

Δr[0]	R -- rank of the array right argument
Δr[1]	D -- dimension being reduced (origin 0)
Δr[2]	N -- shape of dshape, i.e. the N for an N-way reduction
Δr[3+ιR]	S -- shape of the reduced, but not expanded (i.e. zero-filled), result
Δr[(3+R)+ιN]	DS -- resulting shape of dimension reduced, i.e. dshape
Δr[(3+R+N)+ιS[D]]	RIND -- indices (origin 0) into the raveled dshape dimensions of the existing (i.e. nonzero) values
Δr[(3+R+N+S[D])+ι×/S]	DATA -- raveled existing (i.e. nonzero) values

Since the results of these functions are not in very useful forms, we need another set of utility functions to convert them back to the more familiar multi-dimensional forms (including zeros where appropriate). We propose the following functions:

```
r←ways ΔPLUSWAY Δr
       ΔMAXWAY
       ΔMINWAY
       ΔANDWAY
       ΔORWAY
```

The right argument of these functions is a compressed Milky-Way array as returned by the corresponding functions ΔPLUSRED, ΔMAXRED, ΔMINRED, ΔANDRED or ΔORRED. The left argument is a vector of the ways (i.e. indices from ⍳N for an N-way reduction) to be returned. The result is the normal X-way reduction (where X=⍴,ways) as returned by the functions PLUSRED, MAXRED, MINRED, ANDRED or ORRED.

For example:

```
      A←(5 6 7 8 9 10 11, 1 2 3 4 5 6 7) ΔPLUSRED NVEC
      ⎕IO←1
      ⍴2 4 6 ΔPLUSWAY A
6 8 10
      ⍴1 7 ΔPLUSWAY A
5 11
      ⍴2 ΔPLUSWAY A
6
      ⍴⍴(⍳0) ΔPLUSWAY A
0
```

Finally, we need to address the problem of performing Milky-Way reductions on files. When using PLUSRED, we execute an expression like the following, once for each set of, say, 5000 records on file:

```
SUM←SUM+A PLUSRED B
```

This solution will not work with the Milky-Way reduction functions since the values in SUM (after the first set) and the values in the result of ΔPLUSRED are not corresponding data values. They are, instead, both compressed Milky-Way arrays. We propose the following functions:

```
ΔC←ΔA ΔPLUS ΔB
      ΔMAX
      ΔMIN
      ΔAND
      ΔOR
```

These functions perform the +, ⌈, ⌊, ∧ and ∨ functions, respectively, between two compressed Milky-Way arrays, returning a compressed Milky-Way array. If either argument is an empty array, the other argument is returned.

The solution for working with files, then, needs to be modified only slightly when performing Milky-Way reductions:

```
ΔSUM←⍳0                           (before loop)
ΔSUM←ΔSUM ΔPLUS A ΔPLUSRED B      (within loop)
```

The writing of these functions is left as an exercise below.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 340 to 361)

1. Given a 500 element character vector TZONE whose values represent the time zones (E, C, M, P, H) in which each of your 500 fast food restaurants are located, how many restaurants are located in each time zone?

2. If SALES is a 500 element numeric vector whose values represent the annual sales, in dollars, of the corresponding 500 restaurants of the prior problem, what are the annual sales by time zone?

3. If TYPE is a 500 element character vector whose values represent the restaurant types (B, C, P, S) of the corresponding 500 restaurants, how many (FRQ) restaurants of each type are located in each time zone? What are the annual sales (AMT) for each of these type/zone breakdowns? How large (MAX) was the largest restaurant in each of these type/zone breakdowns?

4. Write one or more of the utility functions PLUSRED, MAXRED, MINRED, ANDRED, ORRED defined in this chapter. Compare your functions to the listings of those functions included in the solutions at the back of the book.

5. Write one or more of the utility functions ∆PLUSRED, ∆MAXRED, ∆MINRED, ∆ANDRED, ∆ORRED, ∆PLUSWAY, ∆MAXWAY, ∆MINWAY, ∆ANDWAY, ∆ORWAY, ∆PLUS, ∆MAX, ∆MIN, ∆AND, ∆OR defined in this chapter. See the listings at the back of the book.
~~~~

6. Given the 500 element vectors TZONE, SALES and TYPE defined in the questions above, suppose you also have the following data:

   STATE 500 row, 2 column character matrix of state (postal code) abbreviations indicating the states in which the corresponding 500 restaurants are located.

   MGR 500 element integer vector whose values represent the regional managers responsible for the corresponding 500 restaurants; there are 6 managers and their respective numeric codes are 301, 304, 310, 322, 329 and 333.

   FIT 500 element numeric vector whose values represent the annual federal income tax, in dollars, of the corresponding 500 restaurants.

   Using the utility functions presented in this chapter, generate the following information:

   1. Number of restaurants whose annual sales were $0 to $1 million, $1 million to $5 million, $5 million and up.

   2. Number of restaurants, total sales and total FIT by state (given a 50 row, 2 column matrix ALLSTATES of the distinct state postal codes).

   3. Total sales and total FIT by manager, annual sales volume (0-1, 1-5, 5+ million) and type of restaurant.

   4. Number of restaurants by state and type.

   5. FIT by type and sales volume.

Chapter 8

WRITING USER-FRIENDLY INTERACTIVE FUNCTIONS

An interactive function is one which "prompts" you to enter information at the keyboard. In this chapter we discuss the primitive capabilities available in APL for writing interactive functions. We also define utility functions which make the primitive functions easier to apply and friendlier to use.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write a function ASKUSER which prompts you to "ENTER EMPLOYEE NAME", assigns the character vector response to the variable NAME, prompts to "ENTER SALARY" and assigns the numeric scalar response to SALARY.

TOPIC: Primitive Interactive Functions

The primitive niladic APL functions ⎕ (quad) and ⍞ (quote-quad) allow user interaction. When invoked, both functions causes execution to pause while you type information at the keyboard. When you press the RETURN (or ENTER) key, the typed information is returned explicitly and execution resumes. Graphically, the ⎕ and ⍞ represent "windows" through which information passes to the computer from the outside world. In general, ⎕ is used to enter numeric information and ⍞ is used to enter character information.

A simple solution to the stated problem follows:

```
      ∇ ASKUSER
[1]   'ENTER EMPLOYEE NAME'
[2]   NAME←⍞
[3]   'ENTER SALARY'
[4]   SALARY←⎕
      ∇
```
~~~~

On the 1st and 3rd lines, the prompts are expressed as constant character vectors. Since the vectors are not assigned to variable names or otherwise used as arguments to functions, they are displayed. This behavior is one of the great simplifications in APL: to generate output, just construct an array (as a constant or as the result of an expression) and do not assign it to a variable or otherwise use it. This is why the expression 2+3 causes 5 to display but the expressions A←2+3 or 6×2+3 do not cause 5 to display, even though the same 2+3 operation is being performed.

This convention for generating output in APL is a mixed blessing. While it is simple to generate output, it is sometimes unclear from context whether or not output is being generated. For example, does the following function line generate output?

```
[15]  CRUNCH I
```

The answer depends upon whether or not the monadic function CRUNCH returns an explicit result. If so, the result is not being assigned and so will be displayed. If not, nothing will appear (unless output is generated during the execution of CRUNCH). Because of this lack of clarity, some APL programmers choose to show output explicitly by assigning it to ⎕. For example:

```
[15]  ⎕←CRUNCH I
```

Note that the "window" analogy still holds when using ⎕ in this context. Now information is passing from the computer to the outside world. Not only does the ⎕← convention add clarity, it also enables you to locate (under program control) occurrences of output in case you wish to direct output elsewhere (say, replace '⎕←' by 'OUTPUT ') or turn it off altogether (say, replace '⎕←' by '0 0⍴').

Using this convention, let us rewrite our simple solution:

```
     ∇ ASKUSER
[1]  ⎕←'ENTER EMPLOYEE NAME'
[2]  NAME←⍞
[3]  ⎕←'ENTER SALARY'
[4]  SALARY←⎕
     ∇
```

When executing this function, you will see the following:

```
ENTER EMPLOYEE NAME
_
```

where the "_" symbol represents the location of the cursor (or print mechanism) while the computer is awaiting your response.

Why is the cursor located at the beginning of the line below the prompt? Output in APL (via ⎕← or automatic output) is automatically followed by a "carriage return" (i.e. a newline). The only way to suppress the succeeding carriage return is by assigning the output to

⍞, the other "window". In fact, the inclusion or exclusion of the carriage return is the only difference between ⎕← and ⍞←. For example, consider the following function:

```
      ∇ DISPLAY
[1]    ⍞←'VALUES: '
[2]    ⍞←2 3 4
[3]    ⎕←'.'
      ∇
```

When you execute this function, you will see the following:

```
VALUES: 2 3 4.
      _
```

Notice that the last statement uses ⎕←, causing the cursor to move to the start of the next line for any subsequent output (including the 6 space indent you get in immediate execution mode).

How can we modify our solution to display the prompt and leave the cursor beyond the prompt on the same line?

```
ENTER EMPLOYEE NAME: _
```

Given the behavior of ⍞← described above, we are compelled to try the following:

```
[1]    ⍞←'ENTER EMPLOYEE NAME: '
[2]    NAME←⍞
```

Sure enough, it behaves as we want it to:

```
ENTER EMPLOYEE NAME: LANDER, KEVIN
```

Unfortunately, when we check the value of the variable NAME, we find that it contains not only the name but the prompt as well:

```
      ⍴NAME
34
      NAME
ENTER EMPLOYEE NAME: LANDER, KEVIN
```

The ⍞ function returns every character on the line, whether put there by ⍞← or by user entry. (Some APL systems return the ⍞← characters as blanks or stars or other designated characters.) Clearly, we need to drop the prompt characters from the result of ⍞:

```
[1]    ⍞←'ENTER EMPLOYEE NAME: '
[2]    NAME←21↓⍞
```

The number 21 is the length of the prompt (including the trailing blank).

Can we apply the same technique to the SALARY prompt? No. Quad (⎕) input causes the characters "⎕:" to display. For example:

```
ENTER SALARY
⎕:
      _
```

If we use ⍞←'ENTER SALARY: ', all we accomplish is to put the "⎕:" at the end of the prompt line:

```
ENTER SALARY: ⎕:
                      _
```

If we want to eliminate the "⎕:" from the prompt, we must use ⍞ input. However, the result will then be character valued. To convert the characters to the numbers they represent, we must use execute (⍎). Our final solution is therefore:

```
    ∇ ASKUSER
[1]   ⍞←'ENTER EMPLOYEE NAME: '
[2]   NAME←21↓⍞
[3]   ⍞←'ENTER SALARY: '
[4]   SALARY←⍎14↓⍞
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write utility functions which prompt for character vectors or numeric vectors and which handle the problems associated with each.

TOPIC: Utility Interactive Functions

First, we will define a monadic function CPROMPT (character prompt) which will display its character vector right argument as a prompt and will allow you to enter a response at the end of the same line, returning your response as a character vector.

[WSID: INPUT]

```
    ∇ R←CPROMPT PROMPT
[1]   ⍝ Displays character vector PROMPT, allows
[2]   ⍝ keyboard input on same line and returns
[3]   ⍝ character vector response.
[4]    ⍞←PROMPT
[5]    R←(⍴,PROMPT)↓⍞
    ∇
```
~~~~

Note the use of ravel (,) on the 2nd line to handle the case in which PROMPT is a scalar (e.g. R←CPROMPT '?').

Given this utility function, the solution to the problem of the last section may be rewritten:

```
     ∇ ASKUSER
[1]   NAME←CPROMPT 'ENTER EMPLOYEE NAME: '
[2]   SALARY←⍎CPROMPT 'ENTER SALARY: '
     ∇
```

This function is an improvement over the previous solution. However, problems remain. If you accidentally type a non-numeric character (e.g. 3B5) in response to the 'ENTER SALARY:' prompt, an APL error message will appear (from ⍎) and the function will suspend. This is not user-friendly behavior.

We could use ⎕ input ("evaluated input") to avoid the suspension. When an error occurs in evaluated input mode, the error message is displayed and you are reprompted. For example:

```
ENTER SALARY
⎕:
      3B5
VALUE ERROR
⎕     3B5
      ^
⎕:
      _
```

However, evaluated input mode has several disadvantages. The first is that its appearance (⎕:) is odd to a naive user. The second is that the error messages are technical APL messages (e.g. SYNTAX ERROR), not user-oriented messages (e.g. DEPARTMENT NUMBER MUST BE NUMERIC). The third is that you may inadvertently invoke other functions in the workspace. The fourth is that the response will not be accepted on the same line as the prompt.

Therefore, we will stick with ⍞ input ("character input"). We will define a function NINPUT (numeric input) which will display its character vector prompt right argument and will allow you to enter a response at the end of the same line. The response will be converted to numbers if possible and returned. If not possible, the message '** ENTER NUMBERS ONLY **' will be displayed and you will be reprompted.

How then do we convert a character vector which looks like numbers (e.g. '67 15') into a numeric vector (e.g. 67 15)? A number of APL implementations (e.g. APL*PLUS, SHARP APL) have available the companion monadic functions ⎕FI (fix input or format inverse) and ⎕VI (verify input). The ⎕FI function is just what we are looking for. It converts its character vector or scalar right argument into a numeric vector result. Each group of contiguous nonblank characters is converted into a single numeric element. If the group of characters does not represent a valid number, it is returned as 0. For example:

```
      ⎕FI '67.5 3B6 5 0 HI'
67.5 0 5 0 0
```

Since invalid groups are returned as 0, another function is required to tell the good 0s from the bad 0s. This is what ⎕VI does. The ⎕VI function converts its character vector or scalar right argument into a Boolean vector result. Each group of contiguous nonblank characters is converted into a single bit: 1 if a valid number, 0 if not. For example:

```
      ⎕VI '67.5 3B6 5 0 HI'
1 0 1 1 0
```

The ⎕FI and ⎕VI functions always return vectors, never scalars. For example:

```
      ⍴⎕FI '6'
1
      ⍴⎕FI '    '
0
```

Using ⎕FI and ⎕VI, the definition of NINPUT is:

```
                                          [WSID: INPUT]
     ∇ R←NINPUT PROMPT
[1]   ⍝ Displays character vector PROMPT, allows
[2]   ⍝ keyboard input on same line and returns
[3]   ⍝ numeric vector response.  Requires CPROMPT.
[4]   L1:R←CPROMPT PROMPT
[5]    →(∧/⎕VI R)⍴L2
[6]    ⍞←'** ENTER NUMBERS ONLY **'
[7]    →L1
[8]   L2:R←⎕FI R
     ∇
```

If ⎕FI and ⎕VI are unavailable in your implementation of APL, you must write an APL function which performs a similar function. Such a "parsing" function is quite complicated and is not included here.

In APL2, which does not have ⎕FI and ⎕VI, exception handling may be used to execute the character vector and to display appropriate messages if it can not be successfully executed. The system function ⎕EA (execute alternate) is used. It executes its right argument and returns the result. If an error occurs while executing its right argument, its left argument is executed.

In APL2, the definition of NINPUT is:

```
                                    [WSID: INPUT]
    ∇ R←NINPUT2 PROMPT
[1]   ⍝ Displays character vector PROMPT, allows
[2]   ⍝ keyboard input on same line and returns
[3]   ⍝ numeric vector response.  Requires CPROMPT.
[4]   L1:R←CPROMPT PROMPT
[5]   ⍝ Return empty numeric vector if all-blank response:
[6]    →(R∨.≠' ')⍴L2
[7]    R←⍳0
[8]    →0
[9]   ⍝ Allow only characters which may be parts of numbers
[10]  ⍝ (so that other functions will not be executed):
[11]  L2:→(∧/R∊'0123456789.¯E ')↓L3
[12]  ⍝ Make sure any E (exponential notation) is not
[13]  ⍝ preceded by a blank:
[14]   →(∨/' E'⍷' ',R)⍴L3
[15]  ⍝ Ravel when assigning to insure a vector result:
[16]   '→L3' ⎕EA 'R←,',R
[17]   →0
[18]  L3:⎕←'** ENTER NUMBERS ONLY **'
[19]   →L1
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write a function ASKUSER which prompts for employee name, salary and project codes (numeric) and assigns the respective responses to NAME, SALARY and CODES. Verify that the name is not all-blank, that the salary is a positive integer less than 100,000 and that no more than 5 project codes are entered. Terminate ASKUSER immediately if the user types END in response to any question. The explicit result of ASKUSER is 0 if END is typed and is 1 otherwise.

TOPIC: Utility Validation Functions

Let us take a reverse-engineering approach to this problem. We will write the ASKUSER function, employing imaginary utility functions as needed. Then, we will write the utility functions. Here is the finished solution:
~~~~

```
      ∇ R←ASKUSER
[1]    R←0
[2]   L1:→0 ESCAPE NAME←CPROMPTE 'ENTER EMPLOYEE NAME: '
[3]    →L1 IF (NAME∧.=' ') MESSAGE '** YOU MUST ENTER A NAME'
[4]   L2:→0 ESCAPE SALARY←1 NPROMPTE 'ENTER SALARY: '
[5]    →L2 IF ((SALARY≠⌈SALARY)∨SALARY≤0) MESSAGE '** SALARY
       MUST BE A POSITIVE INTEGER'
[6]    →L2 IF (SALARY>100000) MESSAGE '** SALARY IS
       EXCESSIVE'
[7]   L3:→0 ESCAPE CODES←0 NPROMPTE 'ENTER PROJECT CODES: '
[8]    →L3 IF (5<⍴CODES) MESSAGE '** TOO MANY PROJECTS'
[9]    R←1
      ∇
```

This function was easy to write and is easy to read. Comments are unnecessary. Our task now is to write the utility functions such that the function is "user-friendly" as well.

The CPROMPTE (character prompt with escape) function behaves like the CPROMPT function written earlier with one exception. It checks for the "escape" word END and returns the numeric scalar 1 if entered. Otherwise, it returns the character vector entered via the keyboard. The CPROMPTE function is listed below.

Since some applications permit the use of more than one escape word (e.g. END, QUIT, BACKUP, ABORT, HELP, PRINT, etc.), we have written CPROMPTE to illustrate the use of several escape words, specifically END, BACKUP, and ABORT. CPROMPTE returns the scalar 1, 2 or 3 if the respective escape word is entered.

[WSID: INPUT]

```
    ∇ R←CPROMPTE PROMPT;S
      ⍝ Displays character vector PROMPT, allows
      ⍝ keyboard input on the same line and returns
      ⍝ character vector response.  Checks for entry
      ⍝ of escape words END, BACKUP or ABORT and
      ⍝ returns corresponding numeric scalar 1, 2 or 3
      ⍝ if even partially entered.  (Modify to include
      ⍝ your own set of escape words or to use exact
      ⍝ matching.)  Requires: CPROMPT.
       R←CPROMPT PROMPT
      ⍝ Exit if empty entry:
       →(×S←⍴R)↓0
      ⍝ Branch unless 'END' partially entered:
       →(R∨.≠S↑'END')⍴L1
      ⍝ For exact (not partial) match:
      ⍝      →((3≠S)∨'END'∨.≠3↑R)⍴L1
      ⍝ Or, if ≡ is available:
      ⍝      →('END'≡R)↓L1
      ⍝ Else return scalar 1 (in origin 1):
       R←⎕IO
       →0
      ⍝ Return 2 if 'BACKUP' entered:
      L1:→(R∨.≠S↑'BACKUP')⍴L2
       R←1+⎕IO
       →0
      ⍝ Return 3 if 'ABORT' entered:
      L2:→(R∨.≠S↑'ABORT')⍴0
       R←2+⎕IO
    ∇
```

The ESCAPE function is a dyadic function which checks to see if its right argument is a scalar. If so, it returns its label left argument. If not, it returns an empty vector so that no branch will take place. The ESCAPE function is listed below.

The ESCAPE function has been written to accomodate multiple escape words. For example, the expression,

```
→(L99,L1,0) ESCAPE NAME←CPROMPTE 'ENTER EMPLOYEE NAME: '
```

will cause a branch to one of the "labels" L99, L1 or 0 if the corresponding escape word END, BACKUP or ABORT is entered. If a single label is provided as ESCAPE's left argument, that label is returned if any of the escape words is entered.

```
                                              [WSID: INPUT]
     ∇ R←LABELS ESCAPE CODE
[1]   ⍝ Used as:
[2]   ⍝
[3]   ⍝  →(L1,L2,0) ESCAPE NAME←CPROMPTE 'ENTER NAME: '
[4]   ⍝
[5]   ⍝ Returns LABELS[CODE] if code is a scalar.
[6]   ⍝ Otherwise, returns ⍳0 so no branch occurs.
[7]   ⍝ If LABELS is a singleton, it is returned for
[8]   ⍝ any scalar CODE.
[9]   ⍝ Return empty vector for non-scalar CODE:
[10]   R←⍳0
[11]   →(×⍴⍴CODE)⍴0
[12]  ⍝ Return LABELS for singleton LABELS:
[13]   R←LABELS
[14]   →(1∧.=⍴LABELS)⍴0
[15]  ⍝ Otherwise, return label for corresp escape code:
[16]   R←LABELS[CODE]
     ∇
```

The IF function is the standard conditional branching function.

```
                                              [WSID: INPUT]
     ∇ R←L IF C
[1]   ⍝ Conditional branch function.  Used as:
[2]   ⍝       →LABEL IF I>50
[3]    R←C/L
     ∇
```

The MESSAGE function is a dyadic function which returns its Boolean left argument and which displays its character vector right argument only if the left argument is 1.

```
                                              [WSID: INPUT]
     ∇ R←BIT MESSAGE CVEC
[1]   ⍝ Displays err msg CVEC if BIT=1.  Used as:
[2]   ⍝
[3]   ⍝  →ASK IF (X<0) MESSAGE 'VALUE IS NEGATIVE'
[4]   ⍝
[5]    R←BIT
[6]    →BIT↓0
[7]    ⎕←CVEC
     ∇
```

The NPROMPTE (numeric prompt with escape) function is dyadic. Its left argument is the number of numbers required. Since a check for the exact number of numbers entered (usually 1) is the most common numeric input check, we will build the check into the function. If the left argument is 0, we will accept any number of numbers. Since the result of NPROMPTE is passed as the right argument of ESCAPE, we will use CPROMPTE within NPROMPTE and will return a numeric scalar (escape) result directly instead of the normal numeric vector response.

The NPROMPTE function for APL⋆PLUS or SHARP APL follows:

```
                                        [WSID: INPUT]
      ∇ R←NUM NPROMPTE PROMPT
[1]    ⍝ Displays character vector PROMPT, allows
[2]    ⍝ keyboard input on same line and returns
[3]    ⍝ numeric vector response of length NUM
[4]    ⍝ (or of any length if NUM=0).  Returns
[5]    ⍝ numeric scalar escape code if escape word
[6]    ⍝ entered.  Requires: CPROMPTE.
[7]    L1:R←CPROMPTE PROMPT
[8]    ⍝ Exit if scalar escape code:
[9]     →(ρρR)↓0
[10]    →(∧/⎕VI R)/L2
[11]    ⎕←'** ENTER NUMBERS ONLY **'
[12]    →L1
[13]   L2:R←⎕FI R
[14]   ⍝ Exit if NUM is 0 or is length of input:
[15]    →NUM↓0
[16]    →(NUM=ρR)/0
[17]    ⎕←'** ENTER ',(⍕NUM),' NUMBER',(NUM=1)↓'S **'
[18]    →L1
      ∇
```

The NPROMPTE function for APL2 follows.

```
                                      [WSID: INPUT]
    ∇ R←NUM NPROMPTE2 PROMPT
[1]   ⍝ Displays character vector PROMPT, allows
[2]   ⍝ keyboard input on same line and returns
[3]   ⍝ numeric vector response of length NUM
[4]   ⍝ (or of any length if NUM=0).  Returns
[5]   ⍝ numeric scalar escape code if escape word
[6]   ⍝ entered.  Requires: CPROMPTE.
[7]   L1:R←CPROMPTE PROMPT
[8]   ⍝ Exit if scalar escape code:
[9]    →(⍴⍴R)↓0
[10]  ⍝ Return empty numeric vector if all-blank response:
[11]   →(R∨.≠' ')⍴L2
[12]   R←⍳0
[13]   →L4
[14]  ⍝ Allow only characters which may be parts of numbers
[15]  ⍝ (so that other functions will not be executed):
[16]  L2:→(∧/R∊'0123456789.¯E ')↓L3
[17]  ⍝ Make sure any E (exponential notation) is not
[18]  ⍝ preceded by a blank:
[19]   →(∨/' E'⍷' ',R)⍴L3
[20]  ⍝ Ravel when assigning to insure a vector result:
[21]   '→L3' ⎕EA 'R←,',R
[22]   →L4
[23]  L3:⎕←'** ENTER NUMBERS ONLY **'
[24]   →L1
[25]  ⍝ Exit if NUM is 0 or is length of input:
[26]  L4:→NUM↓0
[27]   →(NUM=⍴R)/0
[28]   ⎕←'** ENTER ',(⍕NUM),' NUMBER',(NUM=1)↓'S **'
[29]   →L1
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 362 to 363)

1. Write a function LPROMPTE (letter prompt with escape) which prompts for a single letter. The right argument is the character vector prompt. The left argument is a character vector of allowable single characters which the user may enter. The result is a one element vector index into the left argument of the
~~~~

character entered or is the numeric scalar escape code if an escape word is typed. To illustrate:

```
[10] →0 ESCAPE ACTION←'ACD' LPROMPTE 'ADD, CHANGE, DELETE: '
[11] →(ADD,CHANGE,DELETE)[ACTION]
```

2. Suppose you are writing interactive functions for a user who does not have an APL terminal. Without an APL terminal, the user cannot enter a negative symbol (¯). What modifications would you make to the utility functions described in this chapter to allow the user to enter a minus symbol (-) for negative numbers (e.g. -38)?

3. Write a niladic function PROPOSAL which generates a proposal for life insurance as follows:

```
      PROPOSAL
NAME: Fred
NUMBER OF KIDS: 3
AGES OF KIDS: 3 4 8
PRESS ENTER WHEN READY...      (press ENTER key)


Dear Fred:

As a proud parent of 3 kids (whose
average age is 5), you need insurance.


(press ENTER key)
GENERATE ANOTHER PROPOSAL? N
```

Use the utility functions developed in this chapter.

Chapter 9

MANIPULATING DATES

In the field of data processing, one of the more commonly processed forms of data is dates. Dates tell us when employees were hired, when bonds mature, when insurance policies take effect, when commissions are due, when materials must be reordered, when expenses are incurred, and so on. Despite the many uses of dates, the number of different tasks performed on dates is small. This chapter discusses those tasks: representing dates in APL, entering dates, displaying dates and manipulating dates.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given that APL supports only two datatypes (numbers and characters), how should dates be represented?

TOPIC: Representation of Dates in APL

Suppose you wish to keep track of the date March 22, 1986. What are the different possible conventions you might employ to store this date? Here are several:

```
1. DATE←'MARCH 22, 1986'        ('MONTH DD, YYYY')
2. DATE←3221986                 (MMDDYYYY)
3. DATE←19860322                (YYYYMMDD)
4. DATE←860322                  (YYMMDD)
5. DATE←1986 3 22               (YYYY MM DD)
6. DATE←1986 81                 (YYYY DDD, days from December 31
                                        of the previous year)
7. DATE←31127                   (Days from December 31, 1899)
```

In order to choose the best convention, you must consider the ways in which the date is to be used. Different representations are better for different applications.
~~~~

For example, the first representation (DATE←'MARCH 22, 1986') is ideal if all you want to do with the date is display it. However, the representation requires 14 bytes (characters) of storage which is more than any of the other representations and it does not lend itself to chronological sorting or to date arithmetic (say, adding 3 months to it).

The second representation (DATE←3221986) requires less storage (4 or 8 bytes depending upon the APL implementation) and is still fairly easy to display in a meaningful form (say 3/22/1986). However, it also requires transformation before it can be sorted or used in date arithmetic (since 3221986 is greater than 1221987 but 3/22/1986 occurs earlier than 1/22/1987).

The third representation (DATE←19860322) requires the same storage as the prior representation and is fairly easy to display in a meaningful form (say 1986/03/22) and can be sorted with or compared to other dates directly, without transformation. For example, since 19870122 is greater than 19860322, it occurs later.

The fourth representation (DATE←860322) is similar to the prior representation. Its advantage is that it displays in two fewer character positions (say 86/03/22 vs. 1986/03/22), though the storage requirements are the same. Its disadvantage is that the year is ambiguous and may not sort properly when comparing to dates in the next century (e.g. 15/03/22 for 2015/03/22).

The fifth representation (DATE←1986 3 22) has different storage requirements than the prior three representations (more or less depending upon the APL implementation). It is easier to work with for some manipulations (say, year arithmetic) but harder for others (say, comparing dates to see which is later).

The sixth representation (DATE←1986 81) makes day arithmetic easier but meaningful display harder. For example, it is simple to see that the date 50 days beyond 1986 81 is 1986 131. However, it is not as simple to see that 1986 131 is May 11, 1986.

The seventh representation (DATE←31127) makes day arithmetic, date comparisons and chronological sorting trivial operations. However, converting the date to a meaningful form (year, month, day) is a complex task.

Depending upon the specific requirements of your application, you may decide to pick any one of these or another form of date representation. If you are undecided, representations 3 (DATE←19860322) and 5 (DATE←1986 3 22) are good choices. These representations seem to provide a nice balance between the extreme forms 1 and 7.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Write a monadic function IPDATEMDY (input date in month, day, year order) which may be used to convert a date, as entered, into the internal representation of the date. The right argument of IPDATEMDY is the character vector representation of the date in month, day, year order and the result is the numeric scalar representation (YYYYMMDD) of the date. For example: DATE←IPDATEMDY CPROMPT 'ENTER DATE OF HIRE: ' (where CPROMPT returns the character vector response to the prompt provided as its right argument). The result is 0 if the right argument does not represent a valid date.

TOPIC: Entering and Validating Dates

To be as friendly as possible, the function must allow you to enter the date in any reasonable form. For example, if the date being entered is March 22, 1986, the right argument (i.e. your response) may be in any of the following forms:

```
3/22/86
3.22.1986
3 22 1986
3 22          (if the current year is 1986)
3-22
3-22-86
and so on
```

To be as safe as possible, the function must verify that the date entered is a valid date. For example, some dates which should be rejected are 13/25/86 and 2-29-86.

The function follows:

```
                                        [WSID: DATES]
    ∇ YYYYMMDD←IPDATEMDY CVEC;DD;MM;NVEC;YY;⎕IO
[1]   ⍝ Converts the character vector representation
[2]   ⍝ of a date (e.g. '6/15' or '3-22-1986' or
[3]   ⍝ '3 22' or '3.22.86') to an integer scalar
[4]   ⍝ representation (YYYYMMDD) of the date.
[5]   ⍝ The items in the right argument are in month,
[6]   ⍝ day, year order.  The result is 0 if the
[7]   ⍝ date is invalid.
[8]   ⍝
[9]    ⎕IO←1
[10]  ⍝ Ravel CVEC in case a scalar; replace '/-.'
[11]  ⍝ by space:
[12]   CVEC←,CVEC
[13]   CVEC[(CVEC∊'/-.')/⍳⍴CVEC]←' '
[14]  ⍝ Set result to 0 and exit if date not valid:
[15]   YYYYMMDD←0
[16]  ⍝ Date must contain only digits and spaces
[17]  ⍝ (once / and - are converted):
[18]   →(∧/CVEC∊' 0123456789')↓0
[19]  ⍝ Convert character vector to numeric vector:
[20]   NVEC←,⍎CVEC
[21]  ⍝ Date must have 2 or 3 elements (MM DD or
[22]  ⍝ MM DD YY):
[23]   →((⍴NVEC)∊ 2 3)↓0
[24]  ⍝ Stick on current year if omitted:
[25]   →(3=⍴NVEC)⍴L1
[26]   NVEC←NVEC,1⍴⎕TS
[27]  ⍝ Convert YY to YYYY using current century:
[28]  L1:YY←NVEC[3]
[29]   →(YY>99)⍴L2
[30]   YY←YY+100×⌊⎕TS[1]÷100
[31]  ⍝ Validate month:
[32]  L2:MM←NVEC[1]
[33]   →(MM∊⍳12)↓0
[34]  ⍝ Validate day of month:
[35]   DD←NVEC[2]
[36]   →((DD<1)∨DD>(31 29 31 30 31 30 31 31 30 31 30 31)[MM])
       ⍴0
[37]  ⍝ Check 2/29 if a leap year:
[38]   →((MM≠2)∨DD≠29)⍴L3
[39]  ⍝ Leap year every 4 years except at centuries
[40]  ⍝ (except 4th centuries):
[41]   →((0≠4|YY)∨(0=100|YY)∧0≠400|YY)⍴0
[42]  ⍝ Pack date into YYYYMMDD format:
[43]  L3:YYYYMMDD←100⊥YY,MM,DD
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Given the three element vector DATE which represents a single date (in YYYY MM DD format), write the APL expressions which will generate each of the following date formats (for DATE←1986 3 22):

a. MAR 22, 1986
b. March 22, 1986
c. 3/22/86

TOPIC: Formatting Dates for Output

In the first format (MAR 22, 1986), we must convert the month from an integer (e.g. 3) to a 3 element character vector (e.g. 'MAR'). The most direct way to do this is to first construct a 3 column character matrix which has one row per possible month:

```
MON←12 3⍴'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
```

Then we may use the month number as the row index into MON. Here is one approach (showing 1986 3 5 as MAR 05, 1986):

```
MON[DATE[2];],' ',(¯2↑'0',⍕DATE[3]),', ',⍕DATE[1]
```

This approach does not work for a matrix (3 columns) of dates. It must be modified, as in the following:

```
DAY←2 0⍕DATE[;3]
DAY[(' '=DAY)/⍳⍴DAY]←'0'
DAY←((1↑⍴DATE),2)⍴DAY
MON[DATE[;2];],' ',DAY,',',5 0⍕DATE[;,1]
```

If your implementation of APL supports the system function ⎕FMT, you may construct this date format with the following:

One date:

```
MON[DATE[2];],,'X1,ZI2,<,>,I5' ⎕FMT (DATE[3];DATE[1])
```

Matrix of dates:

```
'3A1,X1,ZI2,<,>,I5' ⎕FMT (MON[DATE[;2];];DATE[;3 1])
```

In APL2, you may do the following:

One date:

```
MON[DATE[2];],' 05, 5555'⍕DATE[3 1]
```

Matrix of dates:

```
MON[DATE[;2];],' 05, 5555'⍕DATE[;3 1]
```

In the second format (MARCH 22, 1986), the entire month name is displayed. This format differs from the first in that the length of the formatted result is not fixed but depends upon the length of the month name and upon the number of digits in the day.

One approach to selecting the month name portion of the date is to build a character matrix of month names (padded to the right with blanks), extract the appropriate row and squeeze out the blanks:

```
MONTH←12 9⍴'JANUARY  FEBRUARY MARCH    APRIL    MAY...'
MON←MONTH[DATE[2];]
MON←(MON≠' ')/MON
```

A second approach is to build a character vector of month names and a corresponding vector of the lengths of the names. The vector of lengths may be used to locate the corresponding name:

```
MONTH←'JANUARYFEBRUARYMARCHAPRILMAYJUNEJULYAUGUST...'
MLEN← 7 8 5 5 3 4 4 6 9 7 8 8
MON←MONTH[(0,+\MLEN)[DATE[2]]+⍳MLEN[DATE[2]]]
```

Using either approach, once the name is selected, the formatting of the date is simple:

```
MON,' ',(⍕DATE[3]),', ',⍕DATE[1]
```

Generally, the second format (MARCH 22, 1986) is not used when formatting many dates at once since they will have a ragged appearance:

```
MARCH 22, 1986
MAY 3, 1986
SEPTEMBER 17, 1987
JULY 4, 1988
```

On the other hand, you may choose to align the days and years:

```
    MARCH 22, 1986
      MAY 03, 1986
SEPTEMBER 17, 1987
     JULY 04, 1988
```

To construct this result, you can simply use the same approaches discussed for the first format (MAR 22, 1986), but create a 9 column character matrix of right-justified month names instead of a 3 column character matrix of abbreviated month names:

```
MON←12 9⍴'  JANUARY FEBRUARY    MARCH    APRIL      MAY...'
```

In the third format (3/22/86), the month number does not need to be translated into a month name. However, the first two digits of the year must be truncated. Here is one approach:

```
(⍕DATE[2]),'/',(¯2↑'0',⍕DATE[3]),'/',¯2↑⍕DATE[1]
```

This approach does not work for a matrix (3 columns) of dates. It must be modified, as in the following:

```
('/',4 0⍕DATE[;,1],DATE[;3]+100×DATE[;2])[;6 7 1 8 9 1 4 5]
```

If your implementation of APL supports ⎕FMT, you may do the following:

One date:

```
,'I2,</>,ZI2,</>,ZI2' ⎕FMT (DATE[2];DATE[3];100|DATE[1])
      or
,'G<Z9/99/99>' ⎕FMT 100⊥DATE[2 3],100|DATE[1]
```

Matrix of dates:

```
'I2,</>,ZI2,</>,ZI2' ⎕FMT (DATE[;2 3];100|DATE[;1])
      or
'G<Z9/99/99>' ⎕FMT (100|DATE[;1])+100×DATE[;3]+100×DATE[;2]
```

In APL2, you may do the following:

One date:

```
'56/06/05'⍕DATE[2 3 1]
```

Matrix of dates:

```
'56/06/05'⍕DATE[;2 3 1]
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Since some tasks involving dates are more easily solved from certain date representations than from others, design and write a set of date conversion functions which can be used to convert between the various date representations.

TOPIC: Manipulating Dates

Suppose you limit yourself to 4 different date representations. You will need 24 different date conversion functions to handle every possible conversion (4 types times 3 remaining types times 2 directions: to or from). If you handle 5 different date representations, you will need 40 different functions.

You can reduce the number of functions needed by assuming a "base" date representation. If you define just enough functions to be able
~~~~

to convert any date representation to or from the base representation, you can do any possible conversion, though it may require two steps instead of one. For example, to convert dates from a first representation to a second, neither of which is the base representation, you can convert the dates from the first representation to the base representation and then from the base representation to the second representation.

Let us use YYYYMMDD (e.g. DATE←19860322) as the base representation. We choose this representation because it can serve a number of functions directly, without conversions:

a. Such dates can be sorted chronologically using grade-up (⍋) or grade-down (⍒).

b. Such dates can be compared chronologically (before or after) using the relational functions (=, ≠, >, <, ≥, ≤).

c. Such dates can be displayed directly (or with minor formatting) and be readily interpreted by the reader.

If you prefer a different base date representation for your applications, you may modify the functions below to suit your needs.

Assuming the YYYYMMDD base date representation, we propose the following date utility functions:

1. MMDDYYYY←TOMDY YYYYMMDD

   Converts from YYYYMMDD format to MMDDYYYY format. For example:

```
      TOMDY 19860322 19870209
3221986 2091987
```

2. YYYYMMDD←FROMMDY MMDDYYYY

   Converts from MMDDYYYY format to YYYYMMDD format. For example:

```
      FROMMDY 3221986 2091987
19860322 19870209
```

3. DAYS←TODAYS YYYYMMDD

   Converts from YYYYMMDD format to number of days since February 29, 0000. For example:

```
      TODAYS 19860322 19870209
725393 725717
```

4. YYYYMMDD←FROMDAYS DAYS

Converts from numbers of days since February 29, 0000 to YYYYMMDD format. For example:

```
      FROMDAYS 725393 725717
19860322 19870209
```

5. QTS←TOQTS YYYYMMDD

Converts from YYYYMMDD format to 3↑⎕TS format (i.e. YYYY MM DD). The shape of the result is the catenation of 3 and the shape of the right argument. For example:

```
      TOQTS 19860322 19870209
1986  1987
   3     2
  22     9
```

6. YYYYMMDD←FROMQTS QTS

Converts from 3↑⎕TS format (i.e. YYYY MM DD) to YYYYMMDD format. The first element of the shape of the right argument must be 3. The shape of the result is all but the first element of the shape of the right argument. For example:

```
      FROMQTS 3 2 ρ 1986 1987 3 2 22 9
19860322 19870209
```

7. DAYS360←TODAYS360 YYYYMMDD

Converts from YYYYMMDD format to number of days since January 1, 0000, assuming a 30 days per month, 12 months per year, 360 days per year calendar (the 31st day of the month is treated like the 30th day). Financial institutions frequently assume 360 days per year. For example:

```
      TODAYS360 19860322 19870209
715041 715358
```

8. YYYYMMDD←FROMDAYS360 DAYS360

Converts from number of days since January 1, 0000 to YYYYMMDD format, assuming a 30 days per month, 12 months per year, 360 days per year calendar. For example:

```
      FROMDAYS360 715041 715358
19860322 19870209
```

February 29, 0000 was chosen as a base date in the TODAYS and FROMDAYS functions for computational reasons. At the end of that leap day, a 400 year cycle of leap years began. The conversion from dates to days or vice versa is easier when March 1 is considered the first day of the year. When a leap year occurs, the leap day is the last day of the year.

Let us illustrate the application of these functions by using them to solve a variety of problems. The following problems assume that the variable DATES is a vector of dates whose values are assigned in the YYYYMMDD format (e.g. DATES←19860322 19870209 19851225...).

A. How many dates occur in 1987?

```
+/(DATES≥19870101)∧DATES≤19871231     (no conversion needed)
```

B. Display in MM/DD/YYYY format the dates derived by adding 30 years to each date.

```
'G<Z9/99/9999>' ⎕FMT TOMDY 300000+DATES    (assuming ⎕FMT)
'55/55/5555'⍕TOMDY 30000+DATES             (assuming APL2)
```

C. What dates result when adding 90 days to each date?

```
FROMDAYS 90+TODAYS DATES
```

D. Which dates occur in any September?

```
(9=(TOQTS DATES)[2;])/DATES
```

E. Assuming a 360 day year (as in bond calculations), how many whole 6 month periods (i.e. semiannual coupons) are there from each date to the date July 4, 1995?

```
⌊((TODAYS360 19950704)-TODAYS360 DATES)÷180
```

F. Display (in YYYY/MM/DD format) the dates in the past (before today's date), in reverse chronological order (present to past).

```
D←(DATES<FROMQTS 3↑⎕TS)/DATES
'G<9999/99/99>' ⎕FMT D[⍒D]        (assuming ⎕FMT)
'555/55/55'⍕D[⍒D]                 (assuming APL2)
```

G. Compute the ages (age last birthday) today of people born on each of the dates.

```
TODAY←3↑⎕TS
YMD←TOQTS DATES
(TODAY[1]-YMD[1;])-(100⊥TODAY[2 3])<100⊥YMD[2 3;]
```

The definitions of these date utility functions follow. In those instances for which two substantially different algorithms are available to perform the same task, two functions have been provided, one with the name suggested above and the second with the same name followed by '∆' (e.g. FROMDAYS and FROMDAYS∆). You may want to time the alternate functions for your APL installation to determine which is faster (see the Computer Efficiency Considerations chapter).

```
                                             [WSID: DATES]
    ∇ MMDDYYYY←TOMDY YYYYMMDD
[1]   ⍝ Converts dates in form YYYYMMDD to form
[2]   ⍝ MMDDYYYY by numerical manipulations.
[3]   ⍝ The steps:  19860322 → 322 →
[4]   ⍝ (32200000000-322) → 32219860000 → 3221986
[5]   ⍝
[6]    MMDDYYYY←⌊(YYYYMMDD+99999999×10000|YYYYMMDD)÷10000
    ∇
```

```
                                             [WSID: DATES]
    ∇ MMDDYYYY←TOMDY∆ YYYYMMDD
[1]   ⍝ Converts dates in form YYYYMMDD to form MMDDYYYY
[2]   ⍝ by unpacking, rotating and re-packing the digits.
[3]   ⍝ The steps: 19860322 → 1986 322 → 322 1986 → 3221986
[4]   ⍝
[5]    MMDDYYYY← 10000 1 +.×⊖ 0 10000 ⊤YYYYMMDD
[6]   ⍝ Alternative:
[7]   ⍝ MMDDYYYY← 0 10000 ⊥⊖ 0 10000 ⊤YYYYMMDD
    ∇
```

```
                                             [WSID: DATES]
    ∇ YYYYMMDD←FROMMDY MMDDYYYY
[1]   ⍝ Converts dates in form MMDDYYYY to form
[2]   ⍝ YYYYMMDD by numerical manipulations.
[3]   ⍝ The steps:  3221986 → 1986 →
[4]   ⍝ (198600000000-1986) → 198603220000 → 19860322
[5]   ⍝
[6]    YYYYMMDD←⌊(MMDDYYYY+99999999×10000|MMDDYYYY)÷10000
    ∇
```

```
                                        [WSID: DATES]
    ∇ YYYYMMDD←FROMMDY∆ MMDDYYYY
[1]   ⍝ Converts dates in form MMDDYYYY to form YYYYMMDD
[2]   ⍝ by unpacking, rotating and re-packing the digits.
[3]   ⍝ The steps: 3221986 → 322 1986 → 1986 322 → 19860322
[4]   ⍝
[5]    YYYYMMDD← 10000 1 +.×⊖ 0 10000 ⊤MMDDYYYY
[6]   ⍝ Alternative:
[7]   ⍝ YYYYMMDD← 0 10000 ⊥⊖ 0 10000 ⊤MMDDYYYY
    ∇
```

```
                                        [WSID: DATES]
    ∇ DAYS←TODAYS YYYYMMDD;DD;YYYYMM;MM;YYYY;⎕IO
[1]   ⍝ Converts date (YYYYMMDD) to number of days since
[2]   ⍝ Feb. 29, 0000.
[3]    ⎕IO←1
[4]    DD←100|YYYYMMDD
[5]    YYYYMM←(YYYYMMDD-DD)÷100
[6]    MM←100|YYYYMM
[7]    YYYY←(YYYYMM-MM)÷100
[8]   ⍝ Treat Jan and Feb as if in prior year (to have
[9]   ⍝ leap day at end of yr)
[10]   YYYY←YYYY-MM≤2
[11]  ⍝ Days from Feb. 29, 0000 to prior Feb. 28/29 (leap
[12]  ⍝ year every 4th year, no leap year every 100th year,
[13]  ⍝ leap year every 400th year):
[14]   DAYS←(365×YYYY)+-/⌊YYYY∘.÷ 4 100 400
[15]  ⍝ Add in DD days and days from prior Feb. 28/29
[16]   DAYS←DAYS+DD+(306 337 0 31 61 92 122 153 184 214 245
       275)[MM]
    ∇
```

```
                                        [WSID: DATES]
    ∇ DAYS←TODAYS∆ YYYYMMDD;DD;MM;YYYY;YYYYMM;⎕IO
[1]   ⍝ Converts date (YYYYMMDD) to no. of days
[2]   ⍝ since Feb. 29, 0000.
[3]    DD←100|YYYYMMDD
[4]    YYYYMM←(YYYYMMDD-DD)÷100
[5]    MM←100|YYYYMM
[6]    YYYY←(YYYYMM-MM)÷100
[7]   ⍝ Treat Jan and Feb as if in prior year (to
[8]   ⍝ have leap day at end of year):
[9]    YYYY←YYYY-MM≤2
[10]  ⍝ Days from Feb. 29, 0000 to prior Feb. 28/29
[11]  ⍝ (146097, 36524, 1461, 365 days in 400, 100,
[12]  ⍝ 4, 1 year cycles):
[13]   DAYS← 146097 36524 1461 365 +.× 0 4 25 4 ⊤YYYY
[14]  ⍝ Add in DD days and days from prior Feb. 28/29:
[15]   DAYS←DAYS+DD+(306 337 0 31 61 92 122 153 184 214 245
       275)[MM]
    ∇
```

[WSID: DATES]

```
∇ YYYYMMDD←FROMDAYS DAYS;DD;IND;MM;PDAYS;RDAYS;SHAPE;Y;
  YYYY;⎕IO
 ⍝ Converts number of days since Feb. 29, 0000
 ⍝ to date (YYYYMMDD).
  ⎕IO←1
 ⍝ Work with array as a vector and reshape when done:
  SHAPE←⍴DAYS
  DAYS←,DAYS
 ⍝ Approximate year (only off for some 2/28,
 ⍝ 2/29 and 3/1 dates); 365.2425 is used because
 ⍝ there is a leap year every 4th year (+.25),
 ⍝ no leap year every 100th year (-.01), leap
 ⍝ year every 400th year (+.0025):
  YYYY←⌊DAYS÷365.2425
 ⍝ Number of days from Feb. 29, 0000 to Feb. 28/29 of
 ⍝ prior year:
  PDAYS←(365×YYYY)+-/⌊YYYY∘.÷ 4 100 400
 ⍝ Number of days from start of year to specified date:
  RDAYS←DAYS-PDAYS
 ⍝ Branch unless year may be too small by 1 (e.g. 3/1):
  →(×⍴IND←(RDAYS≥366)/⍳⍴RDAYS)↓L1
  YYYY[IND]←Y←YYYY[IND]+1
  RDAYS[IND]←DAYS[IND]-(365×Y)+-/⌊Y∘.÷ 4 100 400
 ⍝ Branch unless year too big by 1 (e.g. 2/29 looks
 ⍝ like 3/0):
 L1:→(×⍴IND←(RDAYS≤0)/⍳⍴RDAYS)↓L2
  YYYY[IND]←Y←YYYY[IND]+¯1
  RDAYS[IND]←DAYS[IND]-(365×Y)+-/⌊Y∘.÷ 4 100 400
 ⍝ Determine month no. from no. days from start of yr:
 L2:MM←(31 30 31 30 31 31 30 31 30 31 31 29 /2⌽⍳12)[
  RDAYS]
 ⍝ Determine day no. from no. days from start of mon.:
  DD←RDAYS-(306 337 0 31 61 92 122 153 184 214 245 275)[
  MM]
 ⍝ Correct for fact that Jan. and Feb. are treated
 ⍝ as if in prior yr (to have leap day at end of yr):
  YYYY←YYYY+MM≤2
 ⍝ Repack and reshape result:
  YYYYMMDD←SHAPE⍴DD+(100×MM)+10000×YYYY
∇
```

```
                                        [WSID: DATES]
    ∇ YYYYMMDD←FROMDAYSΔ DAYS;L4;L400;MM;N1;N4;N100;N400;
      YYYY;⎕IO
[1]   ⍝ Converts number of days since Feb. 29, 000
[2]   ⍝ to date (YYYYMMDD).
[3]    ⎕IO←1
[4]   ⍝ Reduce no. days by 1 so day 0 is Mar. 1, 0000:
[5]    DAYS←DAYS+¯1
[6]   ⍝ No. of 400 year cycles (146097 days) preceding
[7]   ⍝ each date:
[8]    N400←⌊DAYS÷146097
[9]   ⍝ No. days since last 400 year cycle:
[10]   DAYS←DAYS-N400×146097
[11]  ⍝ Flag 400 year leap dates (e.g. Feb. 29, 1600)
[12]  ⍝ and change to Feb. 28:
[13]   L400←DAYS=146096
[14]   DAYS←DAYS-L400
[15]  ⍝ No. of 100 year cycles (36524 days) preceding
[16]  ⍝ each date:
[17]   N100←⌊DAYS÷36524
[18]  ⍝ No. days since last 100 year cycle:
[19]   DAYS←DAYS-N100×36524
[20]  ⍝ No. of 4 year cycles (1461 days) preceding each
[21]  ⍝ date:
[22]   N4←⌊DAYS÷1461
[23]  ⍝ No. days since last 4 year cycle:
[24]   DAYS←DAYS-N4×1461
[25]  ⍝ Flag 4 year leap dates (e.g. Feb. 29, 1988)
[26]  ⍝ and change to Feb. 28:
[27]   L4←DAYS=1460
[28]   DAYS←DAYS-L4
[29]  ⍝ No. of 1 year cycles (365 days) preceding each
[30]  ⍝ date:
[31]   N1←⌊DAYS÷365
[32]  ⍝ No. days since last 1 year cycle:
[33]   DAYS←DAYS-N1×365
[34]  ⍝ Increase no. days by 1 so days are 1 to 365:
[35]   DAYS←DAYS+1
[36]  ⍝ Determine month no. from no. days from start of yr:
[37]   MM←(31 30 31 30 31 31 30 31 30 31 31 28 /2⌽⍳12)[DAYS]
[38]  ⍝ Determine day no. from no. days from start of mon.:
[39]   DAYS←DAYS-(306 337 0 31 61 92 122 153 184 214 245 275)
       [MM]
[40]  ⍝ Add back in leap days:
[41]   DAYS←DAYS+L4+L400
[42]  ⍝ Determine year from no.s of 400, 100, 4, 1 year
[43]  ⍝ cycles:
[44]   YYYY←N1+(4×N4)+(100×N100)+400×N400
[45]  ⍝ Correct for fact that Jan. and Feb. are treated as
[46]  ⍝ if in prior year (to have leap day at end of yr):
[47]   YYYY←YYYY+MM≤2
[48]  ⍝ Pack year, month, day together:
[49]   YYYYMMDD←DAYS+100×MM+100×YYYY
    ∇
```

```
                                         [WSID: DATES]
     ∇ QTS←TOQTS YYYYMMDD
[1]    ⍝ Converts date (YYYYMMDD) to ⎕TS format (YYYY MM DD).
[2]     QTS← 0 100 100 ⊤YYYYMMDD
     ∇
```

```
                                         [WSID: DATES]
     ∇ YYYYMMDD←FROMQTS QTS
[1]    ⍝ Converts date from ⎕TS format (YYYY MM DD)
[2]    ⍝ to YYYYMMDD format.
[3]     YYYYMMDD← 10000 100 1 +.×QTS
[4]    ⍝ Alternative:
[5]    ⍝ YYYY← 0 100 100 ⊥QTS
     ∇
```

```
                                         [WSID: DATES]
     ∇ DAYS←TODAYS360 YYYYMMDD
[1]    ⍝ Converts dates in form YYYYMMDD to days since
[2]    ⍝ January 1, 0000 assuming a 30 days per month,
[3]    ⍝ 12 months per year, 360 days per year calendar.
[4]    ⍝ The 31st day is treated like the 30th.
[5]    ⍝
[6]    ⍝ Change DD=31 to DD=30 and subtract 1 from all days
[7]    ⍝ and 1 from all months:
[8]     YYYYMMDD←YYYYMMDD-101+31=100|YYYYMMDD
[9]     DAYS← 360 30 1 +.× 0 100 100 ⊤YYYYMMDD
[10]   ⍝ Alternative:
[11]   ⍝ DAYS← 0 12 30 ⊥ 0 100 100 ⊤YYYYMMDD
     ∇
```

```
                                         [WSID: DATES]
     ∇ YYYYMMDD←FROMDAYS360 DAYS
[1]    ⍝ Converts days since December 30, ¯1 to dates in
[2]    ⍝ form YYYYMMDD assuming a 30 days per month, 12
[3]    ⍝ months per year, 360 days per year calendar.
[4]    ⍝
[5]    ⍝ Add 1 to all days and 1 to all months (101):
[6]     YYYYMMDD←101+ 10000 100 1 +.× 0 12 30 ⊤DAYS
[7]    ⍝ Alternative:
[8]    ⍝ YYYYMMDD←101+ 0 100 100 ⊥ 0 12 30 ⊤DAYS
     ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEMS: (Solutions on pages 364 to 365)

1. Most bonds pay semi-annual coupons (interest payments). That is, every six months the holder of the bond receives one coupon payment to compensate the holder for the use of his or her money. The bond has printed on it a maturity date, i.e. the date when the final semi-annual coupon is to be paid and when the face (par) value is to be repaid. When a bond is sold prior to maturity, the purchase date usually falls somewhere between two coupon dates. Since the seller and the buyer each hold the bond during a portion of the 6 month coupon period, each is entitled to a portion of the next coupon payment. Traditionally, the buyer pays the seller a portion (called the accrued interest) of the next coupon in addition to the agreed-upon purchase price. The number of days the bond was held by the seller is compared to the number of days the bond will be held by the buyer and the coupon is divided proportionately. The number of days is computed using a 360 day year (12 months of 30 days).

   Given two vectors PDATES and MDATES which respectively represent the purchase dates and maturity dates (in YYYYMMDD representation) of a set of bonds, determine the fractions of the coupons paid for accrued interest at purchase.

2. Suppose you borrow $1000 and agree to pay .1% (.001) of the outstanding balance per day. If the variables BDATE and RDATE represent the dates (in YYYYMMDD format) on which you respectively borrow and repay the loan, how much interest do you pay?

3. Dates stored in the YYYYDDD representation (e.g. 1986081 for March 22, 1986) are sometimes called "Julian" dates. The last three digits represent the number of days from the previous December 31. Write the utility function TOYD and FROMYD which may be used to convert dates in the YYYYMMDD representation to or from the Julian representation.

4. What expressions will return the day of week today as a character vector (e.g. 'TUESDAY')? (Hint: Feb. 29, 000 was a Tuesday.)

Chapter 10

WRITING REPORTS

Report formatting in APL is an afterthought. It was an afterthought to those who designed and implemented APL. And it is frequently an afterthought to those who use APL. The APL language excels at manipulating large multi-dimensional arrays, not at inserting dollar signs and decimal points. The task of designing and implementing reports is slow and tedious and not relished by many programmers, APL or otherwise.

Excellent report formatting capabilities have evolved in the various implementations of APL over the years. These have greatly improved the productivity of the APL programmer but probably not to the extent that report formatting is fun. The capability available in APL⋆PLUS and in SHARP APL is ⎕FMT. In APL2 it is format by example (dyadic ⍕ with character vector left argument). In unenhanced versions of APL, it is format (dyadic ⍕ with numeric vector left argument).

In this chapter, we will describe techniques and utilities which can be employed to make report formatting easier and almost enjoyable.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: How would you construct the following report?

```
                    CAMPBELL CARPET CLEANING
                         ANNUAL SUMMARY
                           12/31/1986

                                                  PERCENT
         ACCOUNT          BUDGET  ACTUAL  DIFFERENCE
   -----------------     ------  ------  ----------
   GROSS REVENUES           650     625        ¯3.8
     LESS DISCOUNTS          50      45        10.0
   NET REVENUES             600     580        ¯3.3
   EXPENSES                 500     510        ¯2.0
   NET INCOME               100      70       ¯30.0
```
~~~~

TOPIC: Viewing the Report

Imagine transcribing this report onto a piece of graph paper (evenly spaced vertical and horizontal lines) by placing each letter, digit or other character into a single square. Viewed in this way, the report appears to be a simple character matrix. Your task is to construct the character matrix.

At the simplest conceptual level, you need only use reshape (ρ):

```
REPORT←12 45ρ'          CAMPBELL CARPET....  ¯30.0   '
```

This is the most computationally direct and efficient way to construct the report. While the burden is light on the computer, however, it is enormous on you. You must count spaces and type them in precisely and you must type in the numbers whose values are probably already in the computer as vector or matrix variables. This is a tremendous waste of your time.

It is more natural to view the report as a set of submatrices which can be pieced together to form the overall report. Then your task is to construct each piece and to catenate them together to construct the whole report.

How do you subdivide the report? You should break it into as few pieces as possible where each piece may be constructed by a single straightforward procedure. Here is one possibility:

```
+----------------------------------------------+
|          CAMPBELL CARPET CLEANING            |
|               ANNUAL SUMMARY                 |
|                 12/31/1986                   |
|                                              |
+----------------------------------------------+
|                                      PERCENT |
|     ACCOUNT        BUDGET  ACTUAL DIFFERENCE |
| -----------------  ------  ------ ---------- |
+-----------------+----------------------------+
|GROSS REVENUES   |    650     625        ¯3.8 |
|  LESS DISCOUNTS |     50      45        10.0 |
|NET REVENUES     |    600     580        ¯3.3 |
|EXPENSES         |    500     510        ¯2.0 |
|NET INCOME       |    100      70       ¯30.0 |
+-----------------+----------------------------+
```

By using formatting utility functions or APL primitive functions, you can construct each of these blocks with relative ease. Once constructed, they may be pieced together by a single statement:

```
REPORT←TOP,[1]MIDDLE,[1]LEFT,RIGHT
```

If you do not have access to utility functions or are using poorly designed functions, you will be forced to break the report into smaller pieces and construct it in more steps. More steps means less productivity.

Of the four blocks above, the one which may be constructed directly by using APL primitive functions is the matrix of numbers in the lower right of the report. Assuming a 3 column numeric matrix of values named DATA, you may construct the block by the following:

```
RIGHT←(7 0 8 0 11 1 ⍕DATA),((1⍴⍴DATA),2)⍴' '
```

or:

```
RIGHT←2⌽9 0 8 0 11 1⍕DATA
```

(Note that the 2 columns of trailing blanks could have been considered a fifth block of the report.)

If your APL implementation has an enhanced formatting capability, you may choose to use it rather than ⍕. For example, with ⎕FMT, you may use:

```
RIGHT←'I7,I8,F11.1,X2' ⎕FMT DATA
```

In APL2, you may use:

```
RIGHT←'5555550 5555550 5555550.00  '⍕DATA
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Design utility functions which will construct the three remaining blocks in the report above.

TOPIC: Constructing Titles and Headings

The top block is a block of titles. When you look at that block, you see the words and date, not the blanks. You see three strings of nonblank characters on three separate lines, centered within the width of the report. The information which completely specifies this block is:

* the width of the report (45 characters)
* the fact that each line is centered within the report width
~~~~

* the nonblank strings to be used, one per line (4 lines):

```
CAMPBELL CARPET CLEANING
ANNUAL SUMMARY
12/31/1986
(empty)
```

The left block is a block of row names. It is similar to the block of titles except the nonblank strings are left-justified within the width of the block (except for one indented line). The information which specifies this block is:

* the width of the block (17 characters)

* the fact that each line is left-justified within the block

* the nonblank strings to be used, one per line (5 lines):

```
GROSS REVENUES
  LESS DISCOUNTS
NET REVENUES
EXPENSES
NET INCOME
```

The specifications for these two blocks suggest the syntax for a formatting utility function which we will call TITLES. Let us illustrate the syntax before we define it:

```
TOP←45 TITLES '∩CAMPBELL CARPET CLEANING∩ANNUAL SUMMARY
              ∩12/31/1986∩'

LEFT←17 TITLES '⊂GROSS REVENUES⊂  LESS DISCOUNTS
               ⊂NET REVENUES⊂EXPENSES⊂NET INCOME'
```

The left argument is an integer scalar of the width of the resulting character matrix. The right argument is a delimited character vector whose partitions each begin with one of the delimiters ⊂ (left-justify), ∩ (center) or ⊃ (right-justify). The result has one row per partition. Each partition is justified within the row according to the delimiter.

The TITLES function is developed and explained in the Positioning Character Data chapter.

The remaining (middle) block is a block of column headings. When you look at the block, you see four headings. The rightmost heading has two lines. The lines are centered with respect to each other. Every heading is underlined. Each set of underlines is separated from its neighbors by two spaces. The headings are centered with respect to the underlines.

We will illustrate and then define the syntax for a formatting utility function which we will call HEADINGS.

```
MIDDLE←17 6 6 10 HEADINGS '∩ACCOUNT∩BUDGET∩ACTUAL
                            ∩PERCENT←DIFFERENCE'
```

The right argument is a delimited (by leading ∩ symbols) character vector whose partitions each represent one heading. The partitions themselves may be delimited by newline delimiters (←) which indicate the points at which headings are broken into multiple lines. The left argument is an integer vector of the widths of the fields into which the headings are inserted. Typically, one width is provided for each heading (partition). However, if fewer widths are provided, they are repeated to match the number of partitions.

The subpartitions of each partition are truncated if necessary to the corresponding width for that heading. The subpartitions are centered above one another within the width for that heading. A row of underlines (hyphens) is placed below each heading across the width of the heading. The headings are separated by 2 blank columns. If a separation of more or less than 2 blank columns is desired, you may include a vector of the desired separations after the vector of widths in the left argument. They will be repeated if necessary to match the number of partitions. For example, to have one blank column between each heading in the example above:

```
MIDDLE←17 7 7 11 1 HEADINGS '∩ACCOUNT∩BUDGET∩ACTUAL...'
```

The HEADINGS function is developed as a problem at the end of the Positioning Character Data chapter.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Most primitive APL formatting capabilities are column oriented. That is, each column of a numeric matrix to be formatted is treated as a separate entity. Every row of the column is formatted in the same way as every other row. This is not true of every column. For example:

```
       6 0 6 1 6 2 ⍕ 3 3 ⍴⍳9
      1   2.0   3.00
      4   5.0   6.00
      7   8.0   9.00
```

What approach would you take to do row oriented formatting. For example, how would you generate the following?

```
      1      2      3
    4.0    5.0    6.0
   7.00   8.00   9.00
```
~~~~

TOPIC: Row Oriented Formatting

A common approach to this problem is to simply format each row separately. For example:

```
R←3 18ρ' '
R[1;]←6 0⍕MAT[1;]
R[2;]←6 1⍕MAT[2;]
R[3;]←6 2⍕MAT[3;]
```

This process may be tedious if the matrix has many rows. The problem may be solved noniteratively by transposing the data, formatting it with a column oriented formatting function and then transposing it back. Both monadic and dyadic applications of transpose (⍉) are required. The specific logic required is presented as a problem at the end of the Positioning Character Data chapter. Using the ROWFMT function developed there, the solution is:

```
width←6
R←0 1 2 ROWFMT MAT
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: You wish to print 50 checks. You are given 5 global variables: CNUM (50 element integer vector of check numbers, e.g. 305); CDATE (50 element integer vector of check dates in MMDDYY format, e.g. 121586); VENDOR (50 row, 25 column character matrix of vendor names, e.g. 25↑'ACME SUPPLIES'); CAMT (50 element integer vector of check amounts in cents, e.g. 518250); DESC (50 row, 40 column character matrix of check descriptions, e.g. 40↑'LOOSELEAF NOTEBOOKS'). Write a function to print the information on continuous form blank checks loaded in the printer. The following is an illustration of the layout and characters to be printed for a single check:

```
                                      NO. 00305   DEC 15, 1986


      TO:  ACME SUPPLIES               $5,182.50


     FOR:  LOOSELEAF NOTEBOOKS
```
~~~~

TOPIC: Formatting Multi-Row Records Using Newlines

When doing simple formatting of a numeric matrix (e.g. 7 0⍕NMAT), the character matrix result contains one row per row of the matrix being formatted. In the problem above, the result appears to contain one matrix (a check) per row (or element) of the arrays being formatted. The term "appears" is used because the "matrix" may in fact be a vector with embedded newline (carriage return) characters. Let us look at the above example in a special way:

```
' ... NO. 00305bbbbb_____   DEC 15, 1986nnn TO:  ACME SUPPLIES
  ... $5,182.50nnnFOR:  LOOSELEAF NOTEBOOKS  ...  nnnnnn'
```

The n represents a newline character and the b represents a backspace character. Notice how the backspaces are being used to underline the check number. The entire vector, including newlines, backspaces and blanks is 174 characters long. The problem becomes much simpler if you consider that your task is to format and combine the arrays into a 174 column matrix, with one row per element or row of the arrays.

Begin by breaking the CDATE variable into two parts: the month (as a 3 letter abbreviation) and the day/year portion (DDYY):

```
MON←12 3⍴'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
MON←MON[⌊CDATE÷10000;]
DDYY←10000|CDATE
```

Let us solve the problem first for those APL implementations which have ⎕FMT (e.g. APL★PLUS and SHARP APL). Given the appropriate format string (FS) left argument to ⎕FMT, the desired matrix result (CHKS) may be constructed as:

```
CHKS←FS ⎕FMT (CNUM;MON;DDYY;VENDOR;CAMT;DESC)
```

The format string left argument of ⎕FMT may contain special characters such as newline and backspace. Construct the variables NL and BS to contain the newline and backspace character scalars respectively:

```
APL★PLUS           SHARP APL
---------          ---------
BS←⎕TCBS           BS←⎕AV[158+⎕IO]
NL←⎕TCNL           NL←⎕AV[156+⎕IO]
```

The format string is constructed by carefully piecing together the control characters needed to produce the special 174 column matrix.

```
FS←'X36,<NO. >,ZI5,<',(5⍴BS),(5⍴'_'),'>,X3,3A1,'
FS←FS,'S<9?>G< ??, 19??>,<',(3⍴NL),' TO:  >,'
FS←FS,'25A1,P<$>CK¯2F15.2,<',(3⍴NL),'FOR:  >,'
FS←FS,'40A1,<',(6⍴NL),'>'
```

To print the checks, just display the variable CHKS on the printer.

Now let us solve the problem using format by example (dyadic ⍕ with character vector left argument, an APL2 enhancement). The approach is basically the same. However, ⍕ does not allow more than one array in its right argument. We must therefore build the result in sections and then combine the sections.

```
BS←⎕TC[⎕IO]
NL←⎕TC[1+⎕IO]
ROWS←⍴CNUM

S1←((36⍴' '),'NO. 05555',(5⍴BS),(5⍴'_'),3⍴' ')⍕
    (ROWS,1)⍴CNUM
S2←MON                              (MON and DDYY from above)
S3←(' 05,_5555',(3⍴NL),' TO:  ')⍕(ROWS,1)⍴1900+10000⊥
    0 100⊤DDYY
S4←VENDOR
S5←('$555,555,553.50',(3⍴NL),'FOR:  ')⍕(ROWS,1)⍴CAMT÷100
S6←DESC
S7←(ROWS,6)⍴NL

CHKS←S1,S2,S3,S4,S5,S6,S7
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Frequently, when an APL application produces multi-page reports, the reports are sent to a "print file" rather than printed directly. Why? How should such a print file be organized? How should its contents be printed?

TOPIC: Directing Report Output to Print Files

There are several reasons for directing report output to a print file rather than printing/displaying it immediately:

1. To avoid the interruption caused by printing when many different reports are to be generated. You may generate them all (to a print file) and then let them print unattended.

2. To enable simple and inexpensive restarting. If the paper jams or runs out (or the line to a remote computer drops), you may reprint the report without having to regenerate it.

3. To make multiple copies of a report. The print file may be printed repeatedly without regenerating the report.
~~~~

4. To print the report on a remote (batch) high-speed line printer. A print file must exist in order for you to submit a batch request for remote printing.

5. To spot-check a lengthy report. If a lengthy report is appended to a print file by an applicaton, selected pages may be printed and reviewed when deciding whether or not to print the entire report.

The following is a reasonable organization for a print file (assuming your APL implementation allows APL files or a reasonable emulation):

Component	Description
1 to 10	(latent, i.e. empty character matrices: 0 0⍴'')
9+2×I	Character matrix (or character vector with embedded newline characters) representation of page I (1,2,3,...) of the report. When displayed, the array will require no more lines than can be accomodated by the paper on which the page is to be printed (typically 66, i.e. 6 lines per inch on paper which is 11 inches high).
10+2×I	One column all blank character matrix with as many rows as are required at the bottom of page I to reach the bottom of the paper (typically 66 minus the number of rows in the matrix stored in component 9+2×I).

The first 10 (latent) components are included in case your implementation has a remote (batch) high-speed line printer capability. Typically, these batch facilities require several control components at the beginning of your print file. The precise significance of these control components is a function of your APL implementation. Include whatever components are needed in your environment.

After the first 10 components, the print file is organized into pairs of components, one pair per page. Two components are used per page instead of one so that your printfile will not be filled with 80 column (or so) blank rows whose only function is to move the printer to the top of the next page. In fact, if file storage is a major consideration, you should break each page into many pieces (some pieces only one line) so that excess spaces can be omitted. However, the file organization would then not allow direct access to a page in the middle of the file since you could not determine the component in which it begins except by trial and error or by maintaining a directory.

With this file organization, you can immediately tell how many pages are on the print file (.5×¯10+number of components) and you can determine exactly where any page is stored (component 9+2×I for page I).

To send the contents of a print file to a remote (batch) high-speed line printer, you must follow the directions which apply to that facility in your environment. However, if you want to print pages on your local printer (or hardcopy terminal) or just wish to spot-check pages on your CRT terminal, the following PRINT function may be useful. To use it, you tie (or otherwise activate) the print file and provide the tie number (or other file identification) as the right argument of PRINT. The dialog will then look something like this:

```
73 PAGES ON THE PRINTFILE.

BEGIN ON WHICH PAGE (OR END): 25
ALIGN PAPER TO PERFORATION AND PRESS RETURN.

     (page 25 prints)

     (page 26 prints)
           :
     (page 73 prints)

ERASE PAGES? YES
NO PAGES ON FILE.
```

If you respond YES to the ERASE PAGES? question, all pages on file will be erased. That is, all but the first 10 components of the print file will be dropped.

In this function, some attention handling code has been included (for APL⋆PLUS or SHARP APL) in case the BREAK key is pressed while the pages are printing. In that event, the printing immediately stops and you are again asked for the page number on which to begin.

```
                                      [WSID: PRTFILE]
    ∇ PRINT TIE;A;I;N;P;⎕ALX;⎕PW
[1]   ⍝ Prints some or all pages in the printfile tied
[2]   ⍝ to TIE.  Pages are in components 11, 13, 15,...
[3]   ⍝ APL*PLUS attention handling (put ⎕ALX in header):
[4]    ⎕ALX←'→L1'
[5]   ⍝ SHARP APL attention handling (put ⎕TRAP in header):
[6]   ⍝ ⎕TRAP←'∇ 1000 E →L1'
[7]   ⍝ Set print width to avoid APL wrap on long lines:
[8]    ⎕PW←250
[9]   ⍝ Number of pages in the file:
[10]   N←((⎕FSIZE TIE)[2]-11)÷2 ⍝ APL*PLUS
[11]  ⍝ N←((⎕SIZE TIE)[2]-11)÷2 ⍝ SHARP APL
[12]   ⎕←(⍕N),' PAGE',(N=1)↓'S ON THE PRINTFILE.'
[13]  ⍝ Branch if 0, 1 or many pages:
[14]   I←1
[15]   →(END,L2,L1)[1+N⌊2]
[16]  ⍝ Ask for starting page if more than one:
[17]  L1:⍞←''
[18]   ⍞←P←'BEGIN ON WHICH PAGE (OR END): '
[19]   A←(⍴P)↓⍞
[20]  ⍝ Reprompt on empty response:
[21]   →(⍴A)↓L1
[22]  ⍝ Exit if END entered (even partially):
[23]   →(A∧.=(⍴A)↑'END')⍴L4
[24]  ⍝ Convert response to numeric and validate:
[25]   I←⎕FI A ⍝ Available on APL*PLUS and SHARP APL
[26]   →((1=⍴I)∧∧/I∊⍳N)⍴L2
[27]   ⎕←'** INVALID PAGE NUMBER.'
[28]   ⎕←'** VALID PAGE NUMBERS ARE 1 THROUGH ',(⍕N),'.'
[29]   ⎕←'** TYPE ''END'' TO STOP PRINTING.'
[30]   →L1
[31]  ⍝ Align paper:
[32]  L2:⍞←P←'ALIGN TO PERFORATION AND PRESS RETURN.'
[33]   A←(⍴P)↓⍞
[34]  ⍝ Exit if END entered:
[35]   →((1≤⍴A)∧A∧.=(⍴A)↑'END')⍴L4
[36]  ⍝ Print page and spacing to next page:
[37]  L3:⎕←⎕FREAD TIE,9+2×I ⍝ APL*PLUS
[38]  ⍝ L3:⎕←⎕READ TIE,9+2×I ⍝ SHARP APL
[39]   ⎕←⎕FREAD TIE,10+2×I ⍝ APL*PLUS
[40]  ⍝ ⎕←⎕READ TIE,10+2×I ⍝ SHARP APL
[41]  ⍝ Branch if more pages:
[42]   →(N≥I←I+1)⍴L3
[43]  ⍝ Erase pages, if desired:
[44]  L4:⍞←P←'ERASE PAGES? '
[45]   →('Y'≠1↑(⍴P)↓⍞)⍴END
[46]   ⎕FDROP TIE,0⌊11-(⎕FSIZE TIE)[2] ⍝ APL*PLUS
[47]  ⍝ ⎕DROP TIE,0⌊11-(⎕SIZE TIE)[2] ⍝ SHARP APL
[48]   ⎕←'NO PAGES ON FILE.'
[49]  END:
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on page 366)

1. Using the HEADINGS function introduced in this chapter, construct the following set of column headings.

```
   LAST YEAR          THIS YEAR         GROWTH
---------------    ---------------        IN
 AVG.    TOTAL      AVG.    TOTAL       TOTAL
 SALE    SALES      SALE    SALES       SALES
------  -------    ------  -------     -------
```

2. Using the numeric scalar DATE which is today's date in the form MMDDYY and the numeric scalar PNO which is the current page number, construct the following set of titles using the TITLES function introduced in this chapter.

```
                                         PAGE 17

                FINANCIAL SUMMARY
                    12/15/86
                 WESTERN REGION
```

3. Given the following numeric matrix,

```
      NMAT
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
```

how would you format it to appear as follows?

```
      CMAT
   1      2      3      4
  5.0    6.0    7.0    8.0
  9     10     11     12
 13.0   14.0   15.0   16.0
 17     18     19     20
 21.0   22.0   23.0   24.0
```
~~~~

Chapter 11

SYSTEM DEVELOPMENT PROCEDURE

Because APL is so concise, powerful and unrestricted, almost anyone can toss together an application system. While experience and discipline are useful to have, they are not essential. This is one of the reasons why so many APL programmers do not come from traditional data processing backgrounds. (The other reason being that prolonged use of COBOL tends to rot the brain.)

If you can solve a problem in APL in one-fifth the time it takes to solve it using FORTRAN, it follows that you can develop 5 unreadable, unmaintainable APL systems in the time it takes you to develop one unreadable, unmaintainable FORTRAN system. To some computer scientists, this improvement in productivity leads to a new philosophy of system development: throw-away code. The basic idea is to write the system fast to get the job done. Then, when requirements change and the system is no longer adequate, throw it away and build a new one.

Those who foster the view that APL is the ideal language for writing throw-away code are those who would like to see APL thrown away. They also tend to kick their pets.

In many respects, APL is not unlike any other programming language. System development should be planned, documented and implemented meticulously. If done properly, the system will be a pleasure to use and to maintain. If done improperly, the system will be a living hell for all those associated with it. Documentation will be scarce, if existent. Code will be mystifying, if readable. The user will be suicidal, if not homicidal.

A well-developed system, on the other hand, is easy to recognize. It is easy to use so the user rarely needs to refer to the extensive user guide. It is reliable and efficient so the technical support person rarely needs to scan through the readable code or the extensive technical documentation. The word "enhancement" is used more frequently than "maintenance" and neither word causes the system developer to tremble with anxiety.

In this chapter we describe eight steps which should be part of any APL system development procedure:

1. Familiarization
2. Specification
3. File design
4. Workspace design
5. User documentation
6. Flow charting
7. Coding, typing, testing
8. Delivery, training

Yes, in that order. Compare this list to the procedure for the usual throw-away APL system:

1. Coding, typing, testing
2. Delivery, training
3. Familiarization (Oh! So that's what you wanted!)

I implore you. Please do not dismiss the system development procedure outlined here without trying it once. The procedure will increase your productivity, improve the quality of your system and make the development process more fun. Try it.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Familiarization

During this phase, you become familiar with the problem, not the solution. Emphasis is on the needs of the user, not the tools of the programmer.

If a manual system exists and is to be replaced, now is the time to study the manual system. If there is no manual system, you should talk with the user and "brain-storm" about an ideal system. Sketch sample reports and sample input sheets.

Where will the data come from? Is it readily available? Will the value of the system justify the installation and updating of the data? How much will the data requirements of the system grow? Will the system need to supply data to other computer systems? How frequently will reports be generated? Who will use them and why? How often will the report formats change? What is the expected life of the system?

The unasked questions which should permeate your thinking are: Is this system feasible? Does the user have a clear picture of what such a system will be like and how it will interact with the
~~~~

organization? Is the value of the finished system going to justify the time and effort required to develop it?

The familiarization phase could also be called the feasibility phase.

The phase is complete when both you and the user have a clear qualitative understanding of how the system should operate, and are both convinced that the system is a good idea.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Specification

At the heart of the word "specification" is the word "specific". That is what the specification phase is all about: getting specific. Put in writing all the details which define the system.

Bear in mind during this phase that any system has three major aspects: input, processing, output.

Input: What data items must be supplied to the system? Are there other items which are not needed now but may be needed later? How many records of data items will the system contain? How fast will it grow? From what different sources will the data come? How frequently? What is the exact record layout of any data from external media (e.g. computer tape)? What is the exact layout of the input sheets used to manually enter data? How clean will the data be? What data integrity checks must be performed (e.g. salary must be a positive integer less than 50,000)? What inter-data restrictions must be imposed (e.g. date-of-hire must be earlier than date-of-termination)?

Processing: What data items must be computed from input data items? With what formulas? What regular processing operations are to be conducted? How frequently? What steps are involved in these operations? How often will these steps change and how dramatically?

Output: What reports will need to be generated by the system? How frequently? What is the exact layout of each report? How is each report item derived from the input and computed data items? How often will the report formats change and how dramatically? Will other areas need access to certain data items? In what format?

The specification phase is complete when you have a written document (the "specification") which so thoroughly and specifically describes the system that, ideally, it could be given to any expert programmer who could then develop the system without conferring with the user.
~~~~

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: File Design

During this phase, you draw pictures of alternative file designs. You then consider the pros and cons of each design given the input and output requirements of the system.

How many file accesses are required to add one record? To add 100 records? To change a few items on one record? On 100 records? To delete one record? To delete 100 records? To display the entire contents of one record? Of 100 records? To search the entire file for records which match a set of logical criteria? To generate each of the reports included in the specification? How often will each of these operations be performed? Will the cost and response time resulting from these file accesses be acceptable?

The file design phase is complete when the structure of each file, including its name, is completely documented in written form.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Workspace Design

During this phase, you sketch sample terminal sessions. The terminal sessions illustrate the actual operation of the system including system prompts and typical user responses.

You should work closely with the user during this phase since the user must live with the system interaction being designed now.

Begin with a general flow chart of the operations to be performed by the system. Break the flowchart into functional blocks, each of which will be implemented as a single APL function. Then, for each function, write down the dialog (sample terminal session) produced by the function.

Does the dialog allow for all required input? Does it provide control of every processing step? Can any and all reports be requested easily? Can the user gracefully exit the system from anywhere without losing any input? Can the user quickly navigate among the most commonly used operations of the system? Are the prompts meaningful? Does the prompting structure allow for optimal use of the files, given their design?

The workspace design phase is complete when sample terminal sessions have been sketched for all contingencies and the user accepts the flow of the system without reservation. The major functions of the system are identified, named and documented in general terms. The file design is updated if necessary.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: User Documentation

User documentation may be written before or after the system is implemented. It is better to write it before. The only reason for writing the documentation after the system is built is that you may be able to talk the user out of any documentation at all. Not a noble reason.

By writing the user documentation before the system is implemented, you will find that the documentation is much easier to write (fewer constraints) and the system is easier to implement. For example, before implementation you can write, "Type STOP at any time to terminate the system." After implementation you must write, "Type STOP to terminate the system when adding records; type END when generating reports; type HALT when closing the accounting period; and type O-backspace-U-backspace-T if none of these works."

If you have never documented a system before implementing it, you may be reluctant to try it now. Please! Please try it! It will make the overall system development task easier and will result in a better system. It's more fun too. Try it once. What can it hurt?

The user documentation is an instruction manual which explains to an inexperienced user how to use the system. It begins with an explanation of the purpose of the system and any background information required to understand the system. After the introduction, the manual consists mainly of the sample terminal sessions along with comments explaining the various options. After reading a well-written manual, the user should be able to use every facet of the system without help from you.

As you write the manual, you should update the specifications and file design if such change is suggested by the documentation process. The user documentation phase is complete when the manual has been read by the user and accepted without reservation.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Flow Charting

During this phase, you will diagram the program logic which underlies each of the major functions identified during the workspace design phase. You will do this with the documented file structure on one side of your desk and the user documentation on the other side.

The flow charting phase can be called the "divide and conquer" phase. Divide each major function into the general steps which it must perform. Then divide the general steps into more specific steps. Continue in this fashion until the steps can be translated directly into APL code.

During this subdivision process, you will identify common steps which are required in several locations of the system. If such steps are not at the low level in which they may be translated into APL code, you may choose to label these steps as subfunctions and write their comprising steps but once. As you identify each subfunction, you should name it, define its syntax, list the variables and functions which are global to its operation and write a brief description of what it does. This will become a permanent part of the technical documentation.

The flow charting phase is complete when coding is all that remains. You will have written descriptions of all functions, subfunctions and global variables. You will have flow charts which diagram every logical step during the use of the system. The steps will be described at such a precise level of detail that they may be translated directly (without much logical reasoning) into APL code.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Coding, Typing, Testing

Without having written a symbol of APL code or pulled your chair up to your APL terminal, you are more than half done with the system. The user has been getting constant feedback from you, has a user's manual on his or her desk and has complete confidence in your ability to deliver the exact system needed. All this without a symbol of APL.

If you do not know APL, now is the time to learn it. Quickly.

During the coding, typing, testing phase, you do just that. The three tasks are clumped together because they need not each be performed to completion before starting the next task. For example, you may want to code 5 or 10 functions, type them, test them and repeat this process for the next 5 or 10 functions.
~~~~

While it is possible to code the entire system before typing a single keystroke, there are disadvantages to such extreme behavior. For one thing, a coding flaw will not be picked up until you begin testing. You may have made the same mistake dozens of times throughout the system. Second, when you are testing the code, you may not remember your intentions in a difficult piece of code. Finally, if doing nothing but writing code for 3 weeks does not drive you crazy, then doing nothing but typing APL code for 3 days will. And if that does not, then 2 weeks of testing will.

At the other extreme, you may choose to code, type and test one function at a time. There are disadvantages to this mode of programming. Some design flaws will not be uncovered until you get further into the system. Such flaws may require you to rewrite or scrap functions written earlier. Any time spent typing or testing now obsolete code will have been wasted.

The coding, typing, testing phase is complete when everything is tested and you are ready to turn the system over to the user.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Delivery, Training

During this phase, you will transfer the system from your control to the user's control. You will initialize any files which have not yet been initialized and will move the workspace or workspaces to the user's library if they need to be moved.

When the system is ready to roll, you will meet with the user to take a spin. Having read the user documentation (and helped you design the dialog), the user should require little guidance or training from you. As the system is tested, you will need to make two lists. The first list refers to bugs which are encountered. If your testing process was careful and thorough, this list will be empty.

The second list refers to suggested enhancements to the system. There is nothing like a live system to suggest what is wrong with it. The user will be happy to mention these. Since you worked together closely to design and document the system, you will not be blamed for delivering an imperfect system. Rather, you will be commended for delivering what you agreed to deliver.

The delivery and training phase is complete when the user accepts the system as is. If there are enhancements to be made to the system, you should implement them as you did the system: familiarization,
~~~~

specification, file design, etc. For simple enhancements, you may get through all the phases in a few minutes. Do not forget to update the technical and user documentation.

Chapter 12

PROGRAMMING STANDARDS

The reason for programming standards is to create a conformist world in which every programmer thinks and programs the same way. What these programmer clones lose in creativity they more than make up in productivity. After all, when picking up a program written by Clone A, Clone B has no trouble reading it. Not only is the language familiar; so too is the dialect and the handwriting.

The purpose of this chapter is to present a set of APL programming standards. They are not presented as the perfect set nor even as the author's preferred set. Rather, they are a set. Pick and choose as they suit you. The important thing here is that the set you choose be accepted by all those in your organization who will work on the same systems as you.

The standards are organized by the phases of the system development procedures presented in the prior chapter. Along with the dogma of each standard is a brief justification for it. If the justification is omitted, you may assume it is, "To improve readability by use of consistent conventions."

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Familiarization

1-1 Select, or have appointed, another programmer to review your work.

WHY: To spot design and logic flaws and otherwise help you see the forest from the trees. This is also one of the best ways to learn new design and programming techniques.
~~~~

1-2 Make certain a single user has been assigned the responsibility of working with you.

WHY: You need a single person to accept your design and to be held responsible for it.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Specification

2-1 Include hand drawn input sheets and full-screen input forms.

WHY: To insure that you and the user and you see input requirements eye to eye.

2-2 Include hand drawn reports containing exact headings, line names and number formats.

WHY: To insure that you and the user and you see output requirements eye to eye.

2-3 Have someone else review the specification.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: File Design

3-1 All file components are assigned a meaningful variable name (first letter underscored) which is used when the component is read into the workspace.

WHY: To help anyone reading the code to identify objects from file.

3-2 The file directory, if there is one, is stored in component 1 of the file.

3-3 On-line file documentation, if any, is stored in component 2 of the file.

3-4 Leave at least 10 latent (empty vector) components at the start of the file for future design modifications.

3-5 Files are documented on paper (preferably using word processing software) and include the component number, variable name, shape and description of each object in the file.

3-6 Have someone else review the file design.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Workspace Design

4-1 Along with each user prompt, list all possible error messages in the sample terminal sessions.

WHY: To insure consistent error messages.

4-2 Use the following standard user keywords when needed:

ADD: Add more data to database.

CHANGE: Replace an existing value with another.

DELETE: Remove data from database.

SHOW: Display data from database.

INSERT: Add more data among existing data in a database where order of data is important.

END: Normal termination of the current phase of the program.

HALT: Terminate program abruptly and compeltely.

WHY: To be consistent so the effect of various responses to a prompt can be anticipated.
~~~~

4-3 The system is invoked by loading an autostarted workspace (using ⎕LX). The functions do not return to immediate execution mode until the system is terminated. All input is accepted via character input (⍞) mode or full-screen input mode rather than evaluated input (⎕) mode.

WHY: To eliminate the possibility of accidentally invoking non-user functions or of getting APL system error messages (e.g. SYNTAX ERROR).

4-4 Applications are terminal independent unless special features are expressly desired.

WHY: To allow switching from one terminal to another without requiring program modifications.

4-5 All reports are directed to a printfile rather than to the terminal. The contents of the printfile must be displayed in a separate step.

WHY: To allow easy, inexpensive report restarting in the event of line noise, line drop, printer malfunction or complete crash; to allow reference by page number when printing; to allow flexibility in directing reports to terminal, line printer or remote printer.

4-6 Have someone else review the workspace design.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: User Documentation

5-1 Include all possible prompts in the documentation, along with descriptive text.

WHY: To ease the transition from text to terminal.

5-2 Write the documentation using whatever word processing software is commonly used in your department or company.

WHY: To insure professional appearance of documentation; to allow quick, easy modifications; to ease the transfer of system support since the same word processing software is used by all.
~~~~

5-3 If the dialog or options of the system are necessarily complicated, include a brief summary of the workspaces, functions, keywords, choices, and so on for quick reference.

WHY: So the user does not have to thumb through the lengthy documentation.

5-4 Include a complete Table of Contents.

5-5 Have someone else review the documentation.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Flowcharting

6-1 The purpose of each function is documented in a one-sentence description, including the function syntax and its dependence upon global variables and subfunctions. All functions have meaningful names or abbreviations.

6-2 The purpose of each global variable is documented in a one-sentence description.

6-3 Functions with same names in different workspaces are identical.

6-4 Subfunctions are chosen judiciously. They have well defined arguments, produce well defined results or effects and require or create a minimum number of global variables or other subfunctions. The state indicator does not get deeper than 5 levels.

WHY: All user defined functions call other functions (at a minimum, APL primitive functions). The key to readability is that the subfunctions can be understood without reference to further documentation. If there are too many subfunctions and they are not neatly defined, the reader will spend too much time flipping back and forth among function listings instead of reading code. The state indicator in the human brain can generally go no deeper than 5 levels without losing track.
~~~~

6-5 Flowchart using words and diagrams, not APL code.

WHY: To compel the programmer to organize her thoughts and plans before getting bogged down in coding details. If coding is all that remains, flowcharting is done.

6-6 Include all error checks in flowchart.

WHY: To remember them when coding and to not underestimate the complexity of the function.

6-7 Have someone else review the flowcharts.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Coding, Typing, Testing

7-1 All workspaces contain the global variable <wsid> which contains the workspace identification (WSID) of the saved workspace.

7-2 All workspaces contain the global variables <fnums> and <fnames> which contain the file numbers and file names of the files which are assumed to always be tied.

7-3 File tie numbers are assigned to global variables whose meaningful names are prefixed with 'f' (e.g. fSMRY, fEMPL). These variables reference files which may or may not be tied (see <fnums>). The tie number of the printfile is assigned to fPRINT.

7-4 The name of the printfile is 'PRINTFILE'.

7-5 All workspaces contain the function TIEFILES which ties those files in <fnums> and <fnames>.

7-6 Workspaces which alter their ⎕LX contain the global <lx> which is assigned the original value of ⎕LX.

7-7 Variables global to the workspace have meaningful names and are completely underscored (except for global variables containing file tie numbers).
~~~~

7-8 Variables from file and localized variables used globally by other functions have meaningful names and have their first character underscored.

7-9 Functions on file, like variables on file, have meaningful names with the first character underscored.

7-10 Strictly local variables (localized and not used within subfunctions) contain no underscores in their names. Meaningful variables have meaningful names or abbreviations. Temporary, intermediate or useless (e.g. from ⎕EX) results are assigned to single letter variable names and are not used further than 5 statements beyond the assignment.

7-11 The meaning of a variable is commented when first assigned unless the comment is the name, or the variable is read from a documented file, or the meaning can be inferred from a prompt.

7-12 The first line or two of every function is a comment which explains the purpose and syntax of the function, and lists the global variables and subfunctions required by the function.

7-13 When calling a subfunction, include a comment which lists the global variables and subfunctions required by the subfunction.

7-14 The intent of every function line is commented.

WHY: To help the program maintainer quickly locate and decipher code which needs fixing or enhancing. Writing code is the process of converting the intent to the code. Reading code is the process of attempting to reconstruct the original intent based on the code. Since the intent is obvious during coding, it can be included in a fraction of the time (and mental effort) it would take to reconstruct it later. Lines that contain prompts or error messages are often self-commenting. Examples of comments which unsuccessfully and successfully comment the intent:

```
⍝ Squeeze out the flagged rows of the matrix.     (no good)
⍝ Ignore inactive profit centers.                 (good)

⍝ Set ⎕ELX to capture error messages.             (no good)
⍝ Prepare for file reservation errors.            (good)

⍝ Increment and repeat if not done.               (no good)
⍝ Loop by region.                                 (good)
```

7-15 Line labels are L1, L2, L3,... and are kept in ascending order even if not sequential. Significant labels, which segment the function or identify key steps, may have meaningful names. For example:

```
→('ACDE'=1↑R)/ADD,CHANGE,DELETE,END
```

7-16 Branching is always to a line label or empty vector (never →0 or →).

WHY: Without →0, the program must exit through its "bottom" which is better style than having many exit points. For example, you may be certain that a statement added to the end of a function will always be executed. The use of naked branch (→) removes control from cover functions which may call the function.

7-17 Recommended branching techniques are:

```
→LABEL
→CONDITION/LABEL
→CONDITIONS/LABELS
→CONDITION↓LABEL
→LABELS[INDEX]
→CONDITION⌽LFALSE,LTRUE
```

7-18 Recommended looping technique is:

```
 I←1
LOOP:→ENDLOOP IF I>LIM
 process I
 I←I+1
 →LOOP
ENDLOOP:
```

7-19 Use ¯2147483647 for numeric values which are "not applicable"; assign this constant to the variable <huge> if used frequently.

7-20 All output to the terminal is through ⎕ or ⍞. For example:

```
⎕←'ENTER YOUR CHOICE'
```

WHY: To help locate all terminal output if the need arises (e.g. to direct output to a file) and to improve readability (e.g. ⎕←PROCESS MAT vs. PROCESS MAT).

7-21 Evaluated input mode (⎕) is not used.

WHY: To avoid unintentional escapes via)LOAD or)OFF, to avoid unintentional execution of defined functions, and to avoid technical error messages (e.g. SYNTAX ERROR) to the user.

7-22 Maintain ⎕IO=1 globally. Localize ⎕IO if assigned as 0.

7-23 The local result variable is used only for the result and not for temporary values.

WHY: To avoid unintended results upon premature function termination and to avoid confusion when reading the function.

7-24 The local argument variables are never reassigned except to ravel them.

7-25 Every function line is restartable. That is, no other functions should be performed on the same function line once a function has been executed whose effect should not be repeated. For example, T←2+V←V,R is unrestartable.

WHY: So that any function line may be restarted from its beginning after it has been stopped, say by an error. For example, the following suspension cannot be properly restarted via →3 since V will have been extended twice:

```
WS FULL
MODEL[3]  T←2+V←V,R
              ∧
```

7-26 Error messages or other error handling logic immediately follow detection. For example:

```
[10]  →(X>0)/L2
[11]  ⎕←'** VALUE MUST BE POSITIVE'
[12]  →L1
[13]  L2: etc.
```

WHY: To avoid having to search through the function to find the code which handles each error condition.

7-27 Lines containing multiple statements perform a single, logical operation.

7-28 All error messages are passed through an error-displaying function named ERRMSG.

WHY: To allow consistent presentation of error messages (e.g. preceding them by two stars or beeping twice or displaying in a specified position on the screen). To enable you to find all error messages for inclusion as an appendix in the user manual.

7-29 Testing is performed methodically by stopping on every function line, not experimentally (i.e. by jumping from bug to bug).

7-30 Test all edge (e.g. empty vector) conditions.

7-31 Have someone else review the code.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

TOPIC: Delivery, Training

8-1 The contents of all workspaces are documented using whatever workspace documentation software is commonly used in your department or company. This software produces a printed, paged listing of the definitions of all functions and at least the names and shapes of all global variables.

8-2 Functions are not locked unless you have a significent reason for doing so.
~~~~

Chapter 13

WORKSPACE DESIGN AND DOCUMENTATION

For a given computer application, two different programmers will design and implement it differently. In fact, a single programmer will develop the application differently at different times in her own career. Because of the flexibility of APL, a spectrum of approaches are both possible and feasible for any problem. How then is one to choose between plausible approaches when designing an application system? In this chapter, we discuss workspace design and documentation considerations. The aims are to expand your appreciation of the trade-offs involved during the design process, and to help you document an existing application.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Develop an application which will maintain a list of employees. For each employee, maintain the employee's number (4 digits), name (last name first) and age. Do not use files for this application. Rather, store the information in the global variables ENUM (integer vector), ENAME (25 column character matrix) and EAGE (integer vector). Provide capabilities for adding, deleting and listing employees.

TOPIC: Subfunction Design

As simple as this application is, no two programmers will develop it exactly the same way. The most pronounced difference between solutions is the degree to which subfunctions are employed. At one extreme, a single function is written which calls no subfunctions. At the other extreme, a primary (or main or cover or driver) function is written which calls a variety of subfunctions which in turn call subfunctions and so on as desired.
~~~~

The following is an illustration of the APL code written to implement the application as a single function.

[WSID: FLF]

```
    ∇ EMPLOYEES;AGE;G;GOOD;NAME;NUM;P;R
     ⍝ Ask for choice on same line:
     CHOOSE:⍞←P←'ADD, DELETE, LIST OR END: '
      R←(⍴P)↓⍞
     ⍝ Branch based on 1st char of response:
      →('ADLE'=1↑R)/ADD,DELETE,LIST,END
      ⎕←'** INVALID CHOICE. CHOOSE FROM: ADLE'
      →CHOOSE
     ⍝
     ⍝
     ADD:⎕←'EMPLOYEE NUMBER (OR 0 IF DONE)'
      NUM←,⎕
     ⍝ Continue if exactly 1 number entered:
      →(1=⍴NUM)/A1
      ⎕←'** ENTER 1 NUMBER'
      →ADD
     ⍝ Branch to choice question if 0 entered:
     A1:→(0=NUM)/CHOOSE
     ⍝ Continue unless employee number already exists:
      →(NUM∊ENUM)↓A2
      ⎕←'** EMPLOYEE ',(⍕NUM),' ALREADY IN LIST'
      →ADD
     A2:⍞←P←'EMPLOYEE NAME (MAX 25 CHARACTERS): '
     ⍝ Ask for name at end of same line:
      NAME←(⍴P)↓⍞
     ⍝ Continue unless name too long:
      →(25≥⍴NAME)/A3
      ⎕←'** NAME TOO LONG'
      →A2
     A3:⎕←'EMPLOYEE AGE'
      AGE←,⎕
     ⍝ Continue if exactly 1 number entered:
      →(1=⍴AGE)/A4
      ⎕←'** ENTER 1 NUMBER'
      →A3
     ⍝ Continue if a valid age:
     A4:→((AGE=⌈AGE)∧(AGE≥17)∧AGE≤99)/A5
      ⎕←'** AGE MUST BE INTEGER FROM 17 TO 99'
      →A3
     ⍝ Catenate new values and ask for more:
     A5:ENUM←ENUM,NUM
     ⍝ Pad name to length 25:
      ENAME←ENAME,[1]25↑NAME
      EAGE←EAGE,AGE
      →ADD
     ⍝
     ⍝
     DELETE:⎕←'ENTER EMPLOYEE NUMBERS TO DELETE'
     ⍝ Ravel to insure a vector, not scalar:
      NUM←,⎕
```

```
    ∇ EMPLOYEES (continued)
[50] ⍝ Continue if all valid numbers:
[51]  →(∧/GOOD←NUM∊ENUM)/D1
[52]  ⎕←'** NOT FOUND: ',⍕(~GOOD)/NUM
[53]  →DELETE
[54] ⍝ Flag those employees to keep:
[55] D1:GOOD←~ENUM∊NUM
[56] ⍝ Squeeze out deleted employees:
[57]  ENUM←GOOD/ENUM
[58]  ENAME←GOOD⌿ENAME
[59]  EAGE←GOOD/EAGE
[60]  →CHOOSE
[61] ⍝
[62] ⍝
[63] LIST:⎕←'NUMBER   AGE   NAME'
[64]  ⎕←''
[65] ⍝ Prepare to sort employees by number:
[66]  G←⍋ENUM
[67] ⍝ Sort and display:
[68]  ⎕←(5 0 7 0 ⍕ENUM[G],[1.5]EAGE[G]),(((⍴ENUM),3)⍴' '),
      ENAME[G;]
[69]  ⎕←''
[70]  →CHOOSE
[71] ⍝
[72] ⍝
[73] END:
    ∇
```

The following is an illustration of the APL code written to implement the application in highly subfunctionized fashion. EMPLOYEES is the driver function.

[WSID: MSF]

```
    ∇ ADDEMP;AGE;NAME;NUM
[1]  A1:NUM←NINPUT 'EMPLOYEE NUMBER (OR 0 IF DONE)'
[2]  ⍝ Exit if 0 entered:
[3]   →0 IF 0=NUM
[4]  ⍝ Continue unless employee number already exists:
[5]   →A2 UNLESS NUM∊ENUM
[6]   ⎕←'** EMPLOYEE ',(⍕NUM),' ALREADY IN LIST'
[7]   →A1
[8]  A2:NAME←CINPUT 'EMPLOYEE NAME (MAX 25 CHARACTERS): '
[9]  ⍝ Continue unless name too long:
[10]  →A2 IF(25<⍴NAME)MESSAGE '** NAME TOO LONG'
[11] A3:AGE←NINPUT 'EMPLOYEE AGE'
[12]  →A3 IF((AGE≠⌈AGE)∨(AGE<17)∨AGE>99)MESSAGE '** AGE MUST
       BE INTEGER FROM 17 TO 99'
[13] ⍝ Catenate new values and ask for more:
[14]  CATEMP
[15]  →A1
    ∇
```

```
                                        [WSID: MSF]
     ∇ CATEMP
[1]    ENUM←ENUM,NUM
[2]   ⍝ Pad name to length 25:
[3]    ENAME←ENAME RCAT NAME
[4]    EAGE←EAGE,AGE
     ∇
```

```
                                        [WSID: MSF]
     ∇ R←CINPUT PROMPT
[1]   ⍝ Display prompt and ask for response on same line:
[2]    ⍞←PROMPT
[3]    R←(⍴PROMPT)↓⍞
     ∇
```

```
                                        [WSID: MSF]
     ∇ DELEMP;GOOD;NUM
[1]   L1:⎕←'ENTER EMPLOYEE NUMBERS TO DELETE'
[2]   ⍝ Ravel to insure a vector, not scalar:
[3]    NUM←,⎕
[4]   ⍝ Continue if all valid numbers:
[5]    →L2 IF∧/GOOD←NUM∊ENUM
[6]    ⎕←'** NOT FOUND: ',⍕(~GOOD)/NUM
[7]    →L1
[8]   ⍝ Flag those employees to keep:
[9]   L2:GOOD←~ENUM∊NUM
[10]  ⍝ Squeeze out deleted employees:
[11]   SQZEMP GOOD
     ∇
```

```
                                        [WSID: MSF]
     ∇ EMPLOYEES;R
[1]   ⍝ Ask for choice:
[2]   CHOOSE:R←'ADLE' SELECT 'ADD, DELETE, LIST OR END: '
[3]   ⍝ Branch based on response:
[4]    →(ADD,DELETE,LIST,END)[R]
[5]   ⍝
[6]   ADD:ADDEMP
[7]    →CHOOSE
[8]   ⍝
[9]   DELETE:DELEMP
[10]   →CHOOSE
[11]  ⍝
[12]  LIST:LISTEMP
[13]   →CHOOSE
[14]  ⍝
[15]  END:
     ∇
```

[WSID: MSF]

```
    ∇ R←LINE IF CONDITION
[1]   R←CONDITION/LINE
    ∇
```

[WSID: MSF]

```
    ∇ LISTEMP;G
[1]   ⎕←'NUMBER   AGE   NAME'
[2]   ⎕←''
[3]  ⍝ Prepare to sort employees by number:
[4]   G←⍋ENUM
[5]  ⍝ Sort and display:
[6]   ⎕←(5 0 7 0 ⍕ENUM[G],[1.5]EAGE[G]),(((⍴ENUM),3)⍴' '),
      ENAME[G;]
[7]   ⎕←''
    ∇
```

[WSID: MSF]

```
    ∇ R←CONDITION MESSAGE CVEC
[1]   R←CONDITION
[2]   →0 UNLESS CONDITION
[3]   ⎕←CVEC
    ∇
```

[WSID: MSF]

```
    ∇ R←NINPUT PROMPT
[1]  L1:⍞←PROMPT
[2]  ⍝ Ravel response to insure a vector, not scalar:
[3]   R←,⎕
[4]  ⍝ Exit if exactly 1 number entered:
[5]   →L1 IF(1≠⍴R)MESSAGE '** ENTER 1 NUMBER'
    ∇
```

[WSID: MSF]

```
    ∇ R←M RCAT V
[1]   R←M,[1](1↓⍴M)↑V
    ∇
```

[WSID: MSF]

```
    ∇ IND←CHOICES SELECT PROMPT;R
[1]  ⍝ Ask for choice:
[2]  ASK:R←CINPUT PROMPT
[3]  ⍝ Search vector of choices for 1st char of response:
[4]   IND←CHOICES⍳1↑R
[5]  ⍝ Ask again if not a valid choice:
[6]   →ASK IF(IND>⍴CHOICES)MESSAGE '** INVALID CHOICE.
      CHOOSE FROM: ',CHOICES
    ∇
```

```
                                                    [WSID: MSF]
      ∇ SQZEMP BIT
[1]     ENUM←BIT/ENUM
[2]     ENAME←BIT⌿ENAME
[3]     EAGE←BIT/EAGE
      ∇

                                                    [WSID: MSF]
      ∇ R←LINE UNLESS CONDITION
[1]     R←CONDITION↓LINE
      ∇
```

By any measure, this sample application is tiny. Yet the advantages and disadvantages of these two extreme approaches emerge even in an application of this size. As the application grows, the differences become more important, even critical. For easy reference, we will use the abbreviations FLF (few large functions) and MSF (many small functions) to refer to the two extreme approaches illustrated above. Let us discuss the pros and cons of each. These considerations should be kept in mind when developing an application system so that the cons are minimized.

1. Utility Functions

A utility function is a usually small (under 20 statements) subfunction which performs a common task and which usually gets its inputs entirely from its arguments (vs. from global variables) and returns its outputs as an explicit result. A well designed utility function will resemble a primitive APL function in its behavior. Some examples of the application of utility functions in the illustrations above include:

```
→A2 UNLESS NUM∊ENUM
→A2 IF (25<⍴NAME) MESSAGE '** NAME TOO LONG'
NAME←CINPUT 'EMPLOYEE NAME (MAX 25 CHARACTERS): '
```

The FLF approach avoids the use of utility functions while the MSF approach uses them generously. This is a pro for MSF and a con for FLF. Utility functions provide two distinct advantages, both of which improve programmer productivity. The first is that a utility function can replace several lines of common code, allowing you to write and test code faster. For example, compare the following:

```
AGE←NINPUT 'EMPLOYEE AGE'

     vs.

L1:⎕←'EMPLOYEE AGE'
 AGE←,⎕
 →(1=ρAGE)/L2
 ⎕←'** ENTER 1 NUMBER'
 →L1
L2:
```

The second advantage is that utility functions can improve code clarity, allowing you to read code faster. For example, compare the following:

```
        →A2 UNLESS NUM∊ENUM

             vs.

        →(NUM∊ENUM)↓A2
   or   →(~NUM∊ENUM)/A2
```

2. Global Changes

After coding and testing an application, you may be asked by the user to make a change which is pervasive. For example, "Remove the dollar signs from all the numbers being displayed," or "Precede all error messages by a 10 space indent and 3 stars," or "When prompting for numbers, await the response on the same line as the prompt." If you have taken the MSF approach and have used your subfunctions consistently, you may be lucky enough to only have to change a single function. For example, if all error messages are being displayed within a subfunction MESSAGE, you may be able to implement the second request above by changing the line of MESSAGE,

```
        ⎕←CVEC
to
        ⎕←'          ***',CVEC
```

This is a pro for MSF and a con for FLF. However, if you have not used your subfunctions consistently, much of the advantage will be lost. For example, if some error messages are being displayed directly and not within the subfunction MESSAGE, you will be forced to conduct an extensive search for those messages. This is exactly what you need to do if you used the FLF approach. Since FLF involves fewer functions than MSG, a slight advantage goes to FLF.

3. Function Size

There exists a school of thought in the community of APL programmers that the ideal size of a function is a page. No line should be wider than a page (i.e. 80 characters or so) and no function should have more lines than will fit on a page (i.e. 50 lines or so). The rationale for this conviction is that the eye and the brain can retain no more than a page or so at a time. Further, by confining each function to a page, the programmer is forced to discern the forest from the trees. The outline of the program logic is placed in the higher level function and the detailed logic is included in subfunctions. The FLF approach violates this standard without remorse. The MSF approach adheres to it rigorously.

Score one for MSF if you want to see the forest and not the trees. Score one for FLF if you want to see the forest and the trees. You can find the trees with the MSF approach but not without leaving the forest (i.e. flipping to another page).

At any rate, smaller function size is a definite pro for MSF when it comes to function editing. If you use a full-screen editor, a small function will fit nicely on a single screen. If you use a line-oriented editor, a large function will suffer more from line renumbering (due to line additions or deletions) than will a small function. For example, if line 15 of a 300 line function is deleted, 285 lines will be renumbered and may need to be reprinted. If line 15 of a 30 line function is deleted, only 15 lines will be renumbered.

4. Self-Containment

The issue of self-containment becomes most obvious when you load the workspace and explore its contents. For FLF:

```
          )FNS
     EMPLOYEES
```

For MSF:

```
          )FNS
     ADDEMP    CATEMP    CINPUT    DELEMP    EMPLOYEES  IF
     LISTEMP   MESSAGE   NINPUT    RCAT      SELECT
     SQZEMP    UNLESS
```

If you decided to merge this application with another, it will be easier to accomplish with the FLF approach. Since the entire application is contained within a single function, the application can be moved about (copying or erasing) as easily as moving the function. With the MSF approach however, you need to be careful that name conflicts (i.e. functions with the same name in two different applications) do not exist.

For example, if you copy the above MSF functions into another application workspace in which a different SELECT function is defined, one of the two SELECT functions will be erased (depending on whether COPY or PCOPY is used). Likewise, if you then erase the above MSF functions from the merged application workspace, the remaining application may no longer work without the IF or CINPUT functions.

Finally, any good written technical documentation will include at least a brief description of each function in an application. The task of writing that documentation is considerably simpler for FLF than for MSF since there are fewer functions to document.

5. Global Passing

Another school of thought in the community of APL programmers states that global variables should be avoided as much as possible. Data should be passed to functions as arguments. There are three reasons for this view:

A. Less confusion. When you are reading a function and you encounter an undocumented global variable, your reading flow is interrupted. What is this variable? Where did it come from? Did I just overlook its assignment? Did the programmer just forget to localize it in the header? Is it a niladic function and not a variable at all? Is it global to this function and assigned outside of it or is it local to this function and assigned within a subfunction to which it is global?

B. Less documentation. To alleviate some of the confusion associated with the use of global variables, you should document a global variable at two points: in the first few lines of any subfunction which requires the global variable; and at the point where any subfunction is called which requires the global variable. For example, the ADDEMP and CATEMP functions defined above should include the following comments:

```
      ∇ ADDEMP
      :
[∘]    ⍝ Catenate new values and ask for more:
[∘]    ⍝ (Requires globals: NUM,NAME,AGE)
[∘]    ⍝ (Modifies globals: ENUM,ENAME,EAGE)
[∘]     CATEMP

      ∇ CATEMP
[1]    ⍝ Called by ADDEMP.
[2]    ⍝ Requires globals: NUM,NAME,AGE
[3]    ⍝ Modifies globals: ENUM,ENAME,EAGE
```

C. Fewer localization problems. When a system makes extensive use of global variables, it is easier to forget to localize a variable at the proper level or to localize it at the wrong level. Poorly localized variables can cause some of the most mystifying errors.

Since the FLF approach has fewer functions then the MSF approach, it has fewer subfunction calls and less global variable passing. This is a pro for FLF and a con for MSF.

On the other hand, the MSF approach is easier to employ when you need to use a new name in a function. It is easy to scan a small function to see whether a meaningful name has already been used. With FLF, you may inadvertently re-use the name of a variable still containing valuable information.

6. State Indicator Depth

When you pick up a system written by someone else (or written by you a long time ago), you will most likely start by reading the cover function and then each subfunction as it is called. In order to retain the meaningfulness of what is going on, you must maintain a mental state indicator as you delve into subfunctions. As you finish reading each subfunction, you must remember which function called it and in what context so that you can flip back to that function and continue reading. This process is analogous to the procedure followed by the computer as it is executing the code.

Unfortunately, the human brain cannot maintain its state indicator as flawlessly as the computer. At a depth of 5 or 6, our memories get flaky. If the current state indicator is not kept on paper, you may get lost and have to start again.

In the MSF approach, the state indicator depth grows very rapidly. Even in the tiny application above, it occasionally gets 5 levels deep. For example:

```
UNLESS[1] *
MESSAGE[2]
NINPUT[5]
ADDEMP[1]
EMPLOYEES[6]
```

In an MSF system of any respectable size, the state indicator will occasionally get 10 to 15 levels deep and will average 5 to 9 levels. Such a system is extremely difficult to read until you become familiar enough with the subfunctions that you know what they do without looking into them (like primitive APL functions).

The problem of deep state indicators is especially apparent when you are called upon to handle an error in an unknown MSF system. The first natural step after generating the error is to check the state indicator. If the state indicator is 9 levels deep, the next natural step is to go to lunch.

Having explored some pros and cons of the FLF and MSF approaches, which should you use? Typically, neither. The most readable and maintainable system is one which employs a moderate number of medium-sized functions, each performing a well-defined task. Utility functions which are self-contained and well-documented should be used generously.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: How is the above application invoked?

TOPIC: Starting an Application

In deciding how to start an application, you must first decide whether or not immediate execution mode will be needed. If so, the user should load the application workspace and then execute (from immediate execution mode) whatever functions are user functions (e.g. EMPLOYEES).

If immediate execution mode is not needed, which is most often the case, you should stay out of that mode until the termination of the application. The main reasons to avoid immediate execution mode are simplicity and security. The application will be simpler to use if the user is prompted for choices rather the having to remember the names of functions. The application will be more secure if non-user functions cannot accidentally or intentionally be invoked.

To "autostart" an application, you should assign the system variable ⎕LX (latent expression) to be the name of the desired cover function before saving the application workspace. For example:

```
      ⎕LX←'EMPLOYEES'
      )SAVE EMPLOYEES
SAVED.....
```

To initiate the application, the user only needs to load the application workspace. In some installations of APL, even the loading step is unnecessary. A workspace may be specified to be automatically loaded when APL is invoked. In either case, the expression assigned to ⎕LX will be automatically executed. For example:
~~~~

```
      )LOAD EMPLOYEES
SAVED.....
ADD, DELETE, LIST OR END: _
```

In this simple application, no special steps need to be taken after the workspace is loaded and before the cover function is executed. In larger applications, this is less likely to be true. Files may need to be tied or shared variables activated and global variables may need to be assigned. In such instances, the value of ⎕LX is more likely to be 'START' or '→RESTART'.

A typical START function will have the following layout:

```
                                        [WSID: MSF]
    ∇ START
[1]  ⍝ Workspace driver function.  Used as:
[2]  ⍝
[3]  ⍝     ⎕LX←'START'
[4]  ⍝
[5]  ⍝ Display any messages.  For example:
[6]   ⎕←''
[7]   ⎕←'WELCOME TO THE EMPLOYEE MAINTENANCE SYSTEM'
[8]   ⎕←''
[9]  ⍝ Assign any global variables.  For example:
[10]  ⎕PW←150
[11]  fEMP←345
[12] ⍝ Tie any files or share any variables.  For example:
[13]  'EMPDATA' ⎕FTIE fEMP
[14] ⍝ Read any global variables from file.  For example:
[15]  ENUM←⎕FREAD fEMP,1
[16]  ENAME←⎕FREAD fEMP,2
[17]  EAGE←⎕FREAD fEMP,3
[18] ⍝ Call the cover function.  For example:
[19]  EMPLOYEES
[20] ⍝ Do any followup work.  For example:
[21]  ENUM ⎕FREPLACE fEMP,1
[22]  ENAME ⎕FREPLACE fEMP,2
[23]  EAGE ⎕FREPLACE fEMP,3
[24]  ⎕FUNTIE fEMP
[25]  ⎕←''
[26]  ⎕←'HAVE A NICE DAY'
[27]  ⎕←''
    ∇
```

A RESTART function (used as ⎕LX←'→RESTART') performs the same tasks as the START function but handles "line drops" in those APL environments which support CONTINUE workspaces. For example, suppose you are connected via terminal, modem and telephone line to a remote APL system. If the telephone line is interrupted (say due to telephone line noise or by accidentally unplugging your modem or terminal), the APL system will detect the drop and will save your active workspace into a stored workspace named CONTINUE. When you next sign on, the CONTINUE workspace will be automatically loaded and its ⎕LX executed. (In some installations of APL, you may need to load the CONTINUE workspace manually.)

In such an instance, you do not want to run the START function again. Rather, you want to redo any steps undone by the drop (e.g. retie the files or reshare the variables) and then resume execution at the line on which the function at the top of the state indicator is suspended. The RESTART function therefore checks the state indicator (via ⎕LC) and either executes START if there is no suspension or performs restart logic. The final task performed by RESTART is to explicitly return the line number of the suspended function so that execution can resume.

A typical RESTART function will have the following layout:

```
                                           [WSID: MSF]
    ∇ R←RESTART
[1]   ⍝ Workspace driver function and line drop handler.
[2]   ⍝ Used as:
[3]   ⍝
[4]   ⍝     ⎕LX←'→RESTART'
[5]   ⍝
[6]   ⍝ Return the line number of any suspended function
[7]   ⍝ (beyond RESTART):
[8]    R←1↓⎕LC
[9]   ⍝ Remove 0's from any ⍎:
[10]   R←(R≠0)/R
[11]  ⍝ Branch if restart logic is necessary:
[12]   →(×⍴R)/L1
[13]  ⍝ Otherwise, run START and return R as an empty vector:
[14]   START
[15]   →0
[16]  ⍝ Restart logic.  Display any messages.  For example:
[17]  L1:⎕←''
[18]   ⎕←'EMPLOYEE MAINTENANCE SYSTEM BEING RESTARTED'
[19]   ⎕←''
[20]  ⍝ Tie any files or share any variables.  For example:
[21]   'EMPDATE' ⎕FTIE fEMP
[22]  ⍝ Upon exit, line number result will be branched to
[23]  ⍝ (if ⎕LX←'→RESTART') and execution will resume at
[24]  ⍝ point of suspension.
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Write a niladic function QDOC (quick documentation) which will display the contents of each function in the workspace, as if you typed ∇fnname[⎕]∇ for each function, in alphabetic order.

TOPIC: Function Documentation

The first step is to determine, under program control, which functions exist in the workspace. The system function ⎕NL (name list) can be used to this end. When used monadically with the right argument 3 (for functions), ⎕NL returns a character matrix of the names of the functions in the active workspace, one row per function. If your implementation of ⎕NL does not return the names in sorted order, sort them (see the Sorting and Searching chapter).

(Some APL implementations have different system functions for returning the names of identifiers. For example, APL★PLUS has ⎕IDLIST and SHARP APL has 1 ⎕WS. However, these implementations also support ⎕NL.)

The next step is to display the functions, under program control, one at a time. Since APL systems generally do not support an expression like ⍎'∇fnname[⎕]∇', a system function must be used. The APL★PLUS system function ⎕VR (visual representation) and the SHARP APL system function 1 ⎕FD (function definition) both return a character vector "visual representation" of the function whose name is provided as the right argument to the system function. The result, when displayed, looks exactly like the display produced by ∇fnname[⎕]∇. This is possible because the character vector result contains newline (carriage return) characters at the end of each function line substring. For example:

```
      ∇ UNLESS[⎕]∇
    ∇ R←LINE UNLESS CONDITION
[1]   R←CONDITION↓LINE
    ∇

      CV←⎕VR 'UNLESS'
      ⍴CV
59
      CV
    ∇ R←LINE UNLESS CONDITION
[1]   R←CONDITION↓LINE
    ∇

      ⍝ Replace newline characters by '⊛' to see them:
      CV[(CV=⎕TCNL)/⍳⍴CV]←'⊛'
      CV
    ∇ R←LINE UNLESS CONDITION⊛[1]   R←CONDITION↓LINE⊛    ∇⊛
```

The result of ⎕VR is an empty character vector if the function specified is locked. For implementations which provide such visual representation system functions, the solution to this problem is straightforward:

```
                                                [WSID: QDOC]
    ∇ QDOC;FNS;I;N;V;⎕IO
[1]  ⍝ Displays continuous listing of all functions in ws.
[2]  ⍝ Origin 1:
[3]   ⎕IO←1
[4]  ⍝ Determine functions:
[5]   FNS←⎕NL 3
[6]  ⍝ Exclude QDOC from list:
[7]   FNS←(FNS∨.≠(1↓⍴FNS)↑'QDOC')/FNS
[8]  ⍝ FNS←(∧/FNS∨.≠⍉(2,1↓⍴FNS)↑2 5⍴'QDOC CR∆VR')/FNS ⍝ APL2
[9]  ⍝ Sort fn names if not already: FNS←FNS[⎕AV⍋FNS;]
[10] ⍝ Loop on rows of FNS:
[11]  I←0
[12]  N←1↑⍴FNS
[13] LOOP:→(N<I←I+1)/0
[14]  V←⎕VR FNS[I;] ⍝ APL*PLUS
[15] ⍝ V←1 ⎕FD FNS[I;] ⍝ SHARP APL
[16] ⍝ V←CR∆VR ⎕CR FNS[I;] ⍝ APL2
[17] ⍝ Ignore if function locked:
[18]  →(×⍴V)↓LOOP
[19]  ⎕←V
[20] ⍝ Blank line:
[21]  ⎕←''
[22]  →LOOP
    ∇
```

Please note that QDOC will not list functions in the workspace whose names happen to be the same as any of the local identifiers in QDOC (e.g. FNS or LOOP). The system function ⎕NL 3 returns the names of identifiers which are functions at the most local level. Since FNS is a variable and LOOP is a label at the local level, any global function with the same name is "shadowed" and will not be seen. Likewise, ⎕VR (or 1 ⎕FD) returns only the visual representation of identifiers which are interpreted as functions at the local level.

For implementations which do not provide a visual representation system function, you must work with the "canonical" (matrix) representation system function ⎕CR (available in SHARP APL as 2 ⎕FD). The function ⎕CR returns a character matrix representation of the function whose name is provided as the right argument. The result has one row per function line (including the header) and as many columns as the length of the longest function line. The function lines are not numbered, are left justified within their respective rows and are padded to the right with blanks. For example:

```
      CM←⎕CR 'UNLESS'
      ⍴CM
2 23
      CM
R←LINE UNLESS CONDITION
R←CONDITION↓LINE
```

The result of ⎕CR is an empty character matrix if the function specified is locked.

To write QDOC, we need a function which will convert this canonical representation result to the more aesthetic visual representation form. The following function will do the trick:

```
                                        [WSID: FNREP]
    ∇ VR←CR∆VR CR;⎕IO;C;D;KEEP;L;N;TCNL
[1]  ⍝ Converts canonical representation of fn to visual
[2]  ⍝ representation.  Return empty vector if CR empty
[3]  ⍝ (locked fn):
[4]   VR←''
[5]   →(×/⍴CR)↓0
[6]  ⍝ Use origin 1:
[7]   ⎕IO←1
[8]  ⍝ Construct newline character:
[9]   TCNL←⎕TCNL ⍝ APL*PLUS
[10] ⍝ TCNL←⎕TC[2] ⍝ APL2
[11] ⍝ TCNL←⎕AV[157] ⍝ SHARP APL
[12] ⍝
[13] ⍝ Format header, deleting trailing blanks:
[14]  VR←CR[1;]
[15]  VR←'    ∇ ',(+/∨\' '≠⌽VR)⍴VR
[16] ⍝ Characters which may begin identifiers:
[17]  L←'ABCDEFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklmnopqrstuvwx
      yz⍙⎕'
[18] ⍝ First character in each line:
[19]  C←CR[;1]
[20] ⍝ Flag comment or label lines:
[21]  D←1↓(C='⍝')∨(C∊L)∧∨/(CR=':')∧<\~CR∊L,'0123456789'
[22] ⍝ Drop header and include leading blank column:
[23]  CR←(1 ¯1 -⍴CR)↑CR
[24] ⍝ De-indent comment or label lines:
[25]  CR←D⌽CR
[26] ⍝ Number of lines:
[27]  N←1↑⍴CR
[28] ⍝ Line numbers right justified with right bracket:
[29]  L←(((3⌈⍴⍕N),0)⍕(N,1)⍴⍳N),']'
[30] ⍝ Line numbers left justified, with both brackets
[31] ⍝ and newline:
[32]  L←TCNL,'[',(L+.=' ')⌽L
[33] ⍝ Attach line numbers to lines:
[34]  CR←L,CR
[35] ⍝ Flag trailing blanks to drop from each line:
[36]  KEEP←⌽∨\' '≠⌽CR
[37] ⍝ Squeeze out trailing blanks:
```

```
    ∇ CR∆VR (continued)
[38]  CR←(,KEEP)/,CR
[39] ⍝ Include header and trailer:
[40]  VR←VR,CR,TCNL,'    ∇',TCNL
    ∇
```

Given the CR∆VR function, the QDOC function can be rewritten for ⎕CR implementations by replacing the lines:

```
FNS←(FNS∨.≠(1↓⍴FNS)↑'QDOC')⌿FNS
V←⎕VR FNS[I;]
```

in the QDOC function above by the corresponding lines:

```
FNS←(∧/FNS∨.≠⍉(2,1↓⍴FNS)↑2 5⍴'QDOC CR∆VR')⌿FNS
V←CR∆VR ⎕CR FNS[I;]
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Design a function WSDOC (workspace documentation) which will display the entire contents of the workspace.

TOPIC: Workspace Documentation

One possible solution to this problem is to design a single self-contained function WSDOC which may be copied into the workspace to be documented. Since the function is self-contained, it requires no subfunctions. Therefore, it may be copied into the workspace or erased from the workspace with minimal impact.

Let's establish the differences between the proposed WSDOC function and the QDOC function of the previous section:

1. The output of WSDOC is paged, not continuous.

2. WSDOC is monadic. The elements of its integer vector argument represent: number of rows per page (usually 66), number of columns per page (say 85), lines in top margin (say 3), lines in bottom margin (say 3), columns in left margin (say 5), columns in right margin (say 5).
~~~~

3. At the top of each page is a title which includes the workspace ID, the current date and time and the page number. For example:

```
36150 MODEL ⋆ 11/15/1986  15:43              PAGE 4
```

4. The nondefault workspace environment is included at the top of the first page. The workspace environment includes the latent expression, the index origin, the print precision, the random link, the comparison tolerance and any other programmer-controlled workspace settings. Only those settings are displayed whose current values differ from those in a clear workspace. For example:

```
NONDEFAULT WORKSPACE ENVIRONMENT:

   ⎕LX←'START'
   ⎕PP←12
```

5. After the nondefault workspace environment, the global workspace variables are listed in alphabetic order, along with their shapes and up to one line of their raveled values. For example:

```
GLOBAL WORKSPACE VARIABLES:

     NAME ←   SHAPE ρ VALUE
     ----     -----   -----
     CODE ←           'X'
   MONTHS ←    12 9 ρ 'JANUARY  FEBRUARY MARCH...'
      MSG ←      10 ρ 'THAT''S ALL'
    TABLE ← 2 99 15 ρ 1.016283 1.11984 1.61582...
      TIE ←           368
```

6. After the global workspace variables, the function names are listed in alphabetic order. For example:

```
FUNCTIONS:

   ADDEMP   EMPLOYEES   NINPUT   UNLESS
   CATEMP   IF          RCAT
   CINPUT   LISTEMP     SELECT
   DELEMP   MESSAGE     SQZEMP
```

7. Finally, the lines of each function are displayed as in QDOC. However, function lines which are too long (for the page width) are broken into multiple lines with care taken not to break a line in the middle of an identifier or numeric constant. If a function will not fit on the remainder of a page, it is started on the top of the next page. Functions which are longer than one page are broken into multiple pages with care taken not to

display a long line on more than one page. At the bottom right corner of each page is a footnote which displays the first and last functions included on the page. For example:

IF → SELECT

8. All local variables and labels within WSDOC are prefixed by ∆∆ to minimize the number of variables and functions which are not recognized because of shadowing.

Let's list the possible problems we may encounter when using such a self-contained WSDOC function:

1. A WS FULL error may occur if there is insufficient available workspace to copy WSDOC. This problem is greater in implementations of APL (such as APL2) in which a function is copied by moving its canonical (matrix) representation. The canonical representation of WSDOC is quite large if it has a lengthy header. To get around this problem, you may remove all local variables from the header and erase all variables beginning with '∆∆' on the last line of the function. Alternately, you may load the WSDOC workspace and copy the workspace to be documented. However, bear in mind that the nondefault system variables will not be copied.

2. A SYMBOL TABLE FULL error may occur if there are insufficient available entries in the symbol table for the local variables and labels in WSDOC.

3. Another object which happens to be named WSDOC will be erased and replaced by the WSDOC function when it is copied into the workspace.

If your APL implementation does not support a visual representation system function (e.g. ⎕VR or 1 ⎕FD), you will also need to copy in the function CR∆VR. In this event, WSDOC requires CR∆VR and is not strictly self-contained.

The writing of WSDOC is left as an exercise at the end of the chapter.

The use of this WSDOC function is a simple way to get a neat and thorough listing of the contents of your workspace. If your workspace documentation requirements go beyond the capabilities of this function, you may want to acquire a more comprehensive workspace documentation software package available from your APL vendor. Such packages typically include capabilities for listing cross-reference information. For example, you can list the functions in which each workspace identifier is used, or list the identifiers used within each function, or display a "tree" diagram which shows which functions are called by other functions, and so on.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: A function named IDENTIFY analyzes the visual representation of a function to determine which identifiers are used within the function and how. It then displays any known or potential errors or inconsistencies (e.g. assigning a value to a name which is also used as a label). Make a list of all such errors or inconsistencies.

TOPIC: Function Identifiers

The IDENTIFY function as described above is useful for a final validation on any function you have written, especially a large function. After using IDENTIFY, it is a simple matter to edit the function to correct the reported problems.

Here are the problems and illustrations:

1. Redundant label.

```
[3]  L6:A←35
 :
[17] L6:Q←B*2
```

2. Unused identifier localized.

```
∇ MODEL;A
```

(A is not mentioned anywhere in MODEL, though it may be used within a character constant argument to ⍎ or in a subfunction called by MODEL)
~~~~

3. Identifier localized but not assigned.

```
      ∇ MODEL;A
  :
[9]    Q←A⍳K
```

(A is not directly assigned, e.g. A←B+2, anywhere in MODEL, though it may be assigned within a character constant argument to ⍎ or in a subfunction called by MODEL)

4. Redundant local variable.

```
      ∇ R←MODEL A;B;R
           or
      ∇ MODEL;A;B;A;R
```

5. Localized label.

```
      ∇ MODEL;LOOP;I
  :
[8]   LOOP:→(LIM<I)/END
```

6. Unused label.

```
      ∇ MODEL
  :
[6]   L4:K←2÷B
```

(No reference is made to L4, e.g. →L4 or →(L3,L4,L5)[I], anywhere in MODEL, though it may be used within a character constant (e.g. ⍎(T>0)/'→L4') or may be...gasp...referenced in a subfunction called by MODEL)

7. Assigned label.

```
      ∇ MODEL
  :
[3]    END←¯99
  :
[25]  END:⎕←V
```

8. Identifier assigned but not localized.

```
      ∇ MODEL;A;J
  :
[7]    B←⍳⍴J
```

(B is not localized in MODEL, though it may be localized in a function which calls MODEL)

9. Identifier used and not localized.

```
        ∇ MODEL;A
   :
 [6]    A←2+B
```

(B is not localized in MODEL, though it may be localized in a function which calls MODEL or it may be a subfunction required by MODEL)

10. Result not assigned.

```
        ∇ R←MODEL PARAMS
```

(R is not assigned in MODEL, though it may be assigned within a character constant to ⍎ or in a subfunction called by MODEL)

11. Argument not used.

```
        ∇ R←MODEL PARAMS
```

(PARAMS is not used in MODEL, though it may be used within a character constant argument to ⍎ or in a subfunction called by MODEL)

The task of writing the IDENTIFY function is beyond the scope of this chapter. It is included as a problem in the Boolean Techniques chapter. IDENTIFY is monadic. Its right argument is the visual representation of the function to be analyzed. For example, to analyze the function MODEL, do the following:

```
IDENTIFY ⎕VR 'MODEL'            in APL*PLUS
IDENTIFY 1 ⎕FD 'MODEL'          in SHARP APL
IDENTIFY CR∆VR ⎕CR 'MODEL'      in another APL system
```

(CR∆VR is defined earlier in this chapter.)

There are three related functions also developed in the Boolean Techniques chapter which warrant mention here. They are: RELABEL, LOCALIZE and UNCOMMENT. The right argument of each function, like IDENTIFY, is the visual representation of a function. The result of each function is a modified version of the visual representation, modified to accomplish a particular task. The functions are described below.

Syntax: NEWVR←LABLIST RELABEL OLDVR

The RELABEL function changes the labels in the visual representation so that they become L1, L2, L3 and so on. In many functions, expecially large ones, labels serve simply as branch targets for downward flowing logic. It is difficult and pointless to think up a meaningful name for each label. It is more convenient to the reader to have the labels sequentially numbered so that they can be quickly located. Some labels, however, are best left as meaningful names (e.g. LOOP, END, START, CALC). The left argument LABLIST is a character vector of the names of the labels (separated by spaces) which are not to be renamed. Provide an empty character vector left argument (i.e. '') if all labels are to be renamed.

Do not use RELABEL on any function which contains local variables L1, L2, and so on. Otherwise, these names will refer to both labels and local variables. The resulting function will no longer work correctly.

RELABEL ignores all identifiers within quotes so some labels may not be modified as desired. For example, in the expression,

```
ELX←'→BELOW'
```

the reference to the label BELOW will not be detected and modified. To handle this potential problem, you may choose to write such expressions in the following way:

```
ELX←'→',⍕BELOW
```

Likewise, the names of labels included in comments are not detected by RELABEL. You should avoid placing labels in comments. For example,

```
use:   ⍝ Branch if quota exceeded
not:   ⍝ Go to L17 if quota exceeded
```

RELABEL does not correct any of the problems with labels listed by IDENTIFY.

Syntax: NEWVR←VARLIST LOCALIZE OLDVR

The LOCALIZE function changes the local variables in the header of the visual representation so that the header includes only those variables which are assigned within the visual representation. The LOCALIZE function tends to correct problems 2, 3, 4, 5 and 8 listed by IDENTIFY. Some variables, however, are assigned within a function but should be left global or are not assigned (i.e. are assigned in subfunctions) but should be localized. The left argument VARLIST is a character vector of the names of variables (separated by spaces) which are to be included in the header if not assigned or are to be excluded from the header if assigned. Provide an empty character vector left argument (i.e. '') if all and only the assigned variables are to be localized. The localized variables will be included in the header in alphabetic order.

Syntax: NEWVR←UNCOMMENT OLDVR

The UNCOMMENT function removes all comments from the visual representation. End-of-line comments are removed completely, including the comment symbol (⍝). The comment symbols which precede full-line comments are not deleted so that all function lines remain and do not renumber. Strangely, the UNCOMMENT function allows you to include more comments in functions you write. One argument for omitting or skimping on comments is that comments use up valuable workspace. The UNCOMMENT function allows you to write one set of functions which contain extensive comments (the "maintenance version") and another set which is functionally equivalent but contains no comments (the "production version").

Since the functions RELABEL, LOCALIZE and UNCOMMENT each require a visual representation right argument and each return a visual representation result, they may be "chained" together to perform several functions at once. For example:

```
NEWVR←'LOOP' RELABEL '' LOCALIZE UNCOMMENT ⎕VR 'MODEL'
```

However, the visual representation of a function is of little value to you unless you can convert it back into a function. Some APL systems have a system function which will do this directly (⎕DEF in APL★PLUS and 3 ⎕FD in SHARP APL). The right argument of the system function is the visual representation of a function and the result is the character vector name of the function defined.

Therefore, to relabel a function named MODEL:

```
N←⎕DEF '' RELABEL ⎕VR 'MODEL'            in APL★PLUS
N←3 ⎕FD '' RELABEL 1 ⎕FD 'MODEL'         in SHARP APL
```

For APL implementations which do not have such a system function, you must use the system function ⎕FX (fix). The right argument to ⎕FX is the canonical (i.e. matrix) representation of a function (as returned by ⎕CR) and the result is the character vector name of the function defined (fixed).

Our task then is to write a function VR∆CR which will convert the visual representation result of RELABEL, LOCALIZE or UNCOMMENT into a canonical representation so that the function may be defined via ⎕FX. Given the VR∆CR function, we may relabel a function named MODEL as follows:

```
N←⎕FX VR∆CR '' RELABEL CR∆VR ⎕CR 'MODEL'
```

The following VR∆CR function will perform the necessary conversion.

[WSID: FNREP]

```
     ∇ CR←VR∆CR VR;⎕IO;B;C;D;I;LEN;NL;R;TCNL
      ⍝ Converts visual representation of fn to canonical
      ⍝ representation.  Return empty matrix if VR empty
      ⍝ (locked fn):
       CR← 0 0 ⍴''
       →(⍴VR)↓0
      ⍝ Use origin 1:
       ⎕IO←1
      ⍝ Construct newline character:
       TCNL←⎕TCNL ⍝ APL*PLUS
      ⍝ TCNL←⎕TC[2] ⍝ APL2
      ⍝ TCNL←⎕AV[157] ⍝ SHARP APL
      ⍝
      ⍝ Select header line (less newline):
       CR←(I←¯1+VR⍳TCNL)⍴VR
      ⍝ Drop off header:
       VR←I↓VR
      ⍝ Delete leading ∇ and spaces from header:
       CR←(+/∧\CR∊' ∇')↓CR
      ⍝ Locate newlines which precede and follow
      ⍝ each line:
       NL←VR=TCNL
      ⍝ Flag starts and ends of contiguous digits
      ⍝ (e.g. line no.s):
       D←VR∊'0123456789'
       D←D≠(⍴D)⍴0,D
      ⍝ Flag char following ']' after line no.:
       D←¯1⌽D\¯1⌽D/¯2⌽NL
      ⍝ Flag starts and ends of contiguous blanks
      ⍝ (e.g. after line no.s):
       B←VR=' '
       B←B≠(⍴B)⍴0,B
      ⍝ Flag first nonblank char in each line, as indices:
       D←(D>B)∨B\¯1⌽B/D
       D←D/⍳⍴D
      ⍝ Compute lengths of lines:
       LEN←(1↓¯1↓NL/⍳⍴NL)-D
      ⍝ No. of columns in result:
       C←(⍴CR)⌈⌈/LEN
      ⍝ No. of rows in result:
       R←1+⍴LEN
      ⍝ Initialize result as raveled matrix:
       CR←(R×C)↑CR
      ⍝ Construct index vector I←(⍳LEN[1]),(⍳LEN[2]),...:
       I←LEN/-¯1↓0,+\LEN
       I←I+⍳⍴I
      ⍝ Insert fn lines into raveled result:
       CR[I+LEN/C×⍳⍴LEN]←VR[I+LEN/¯1+D]
      ⍝ Reshape result to matrix:
       CR←(R,C)⍴CR
     ∇
```

This function borrows a number of the techniques discussed in the Boolean Techniques chapter. See that chapter for clarification.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Design a monadic function USEDBY whose argument is a list of functions (character matrix with one name per row or a character vector with names delimited by spaces) and which shows all subfunctions and global variables required by those functions.

TOPIC: Workspace Identifiers

When you inherit the maintenance of an APL application, there are three pieces of documentation which are invaluable to your comprehension of the system. They are:

1. Function listings. If missing, you can reconstruct them by using the WSDOC function defined in this chapter.

2. File structure documentation. If missing, you can hopefully reconstruct it by displaying the data from the files and by inferring meaning from the context in which the files are used (by reading the function listings).

3. System flow charts. If missing, you can hopefully reconstruct them by running the USEDBY function on the user level functions and by reading the function listings.

The following is an illustration of USEDBY on the EMPLOYEES function of the MSF workspace listed earlier in this chapter.
~~~~

```
          USEDBY 'EMPLOYEES'
    EMPLOYEES
       SELECT
          CINPUT
          IF
          MESSAGE
             UNLESS
       ADDEMP
          NINPUT
             IF
             MESSAGE
                 UNLESS
          IF
          UNLESS
          ENUM (global)
          CINPUT
          MESSAGE
             UNLESS
          CATEMP
             ENUM (global)
             NUM (ADDEMP - local)
             ENAME (global)
             RCAT
             NAME (ADDEMP - local)
             EAGE (global)
             AGE (ADDEMP - local)
       DELEMP
          IF
          ENUM (global)
          SQZEMP
             ENUM (global)
             ENAME (global)
             EAGE (global)
       LISTEMP
          ENUM (global)
          EAGE (global)
          ENAME (global)
```

The USEDBY function does pretty much what you would do to manually diagram the subfunction and global variable structure of a system. It starts by evaluating the visual representation of the highest level function for global identifiers referenced (i.e. all identifiers but labels, results, arguments or localized variables). For those global identifiers which are themselves functions, it evaluates each one in the same fashion. And so on it recurses deeper or less deep in the fashion of the state indicator during execution of the system.

The writing of USEDBY is left as an exercise at the end of the chapter.

The USEDBY function is extremely powerful and quite complex. It draws heavily on the materials presented in the Boolean Techniques chapter. It may help you to read that chapter before writing the function or reviewing the solution.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 367 to 382)

1. Design other "visual representation manipulation" functions which may be useful. Pattern their syntax and behavior after the IDENTIFY, RELABEL, LOCALIZE and UNCOMMENT functions described in this chapter.

2. Sketch a flowchart of the WSDOC function described in this chapter. Compare it to the flow of the WSDOC function listed in the solutions at the back of the book.

3. Sketch a flowchart of the USEDBY function described in this chapter. Compare it to the USEDBY function listed at the back of the book.
~~~~

Chapter 14

FILE DESIGN AND UTILITIES

When APL was first implemented, the language included no file capabilities. This shortcoming was quickly recognized as an obstacle to the acceptance of APL as a viable business programming language. Two approaches were taken to overcome the obstacle.

In one approach (shared variables), facilities were developed (⎕SVO, ⎕SVR, ...) to provide access to existing non-APL file structures. From APL, you can do anything with files that you can do from another programming language. While this approach enables the APL user to communicate with non-APL environments, it leaves the APL purist unsatisfied. It is difficult and disappointing to the APL programmer to work with the concise and consistent APL primitive functions on the one hand and the messy world of records, tracks, blocks, cylinders and disks on the other hand.

In the other approach (shared files), facilities were developed (⎕FCREATE, ⎕FREAD, ...) to provide access to APL file structures. In the spirit of APL simplicity, an APL file was defined as a list of APL objects residing outside of the active workspace. A file can have any number of objects (called "components") and each object can be of any type, rank or shape. The components are numbered consecutively from 1. A large object can replace a small object without the APL programmer knowing or caring how the storage is being managed on the physical storage device.

If you want to work with APL files but your APL implementation supports only shared variables, look for a public library workspace which uses shared variables to emulate APL files (e.g. the IBM workspace 2 VAPLFILE).

In this chapter, we will discuss some of the more common APL file organizations and the trade-offs between them. We will also discuss the value of quality file documentation and file utility functions.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Suppose you want to build an APL system for maintaining a database of information about insurance policyholders. For each policyholder, you will keep track of: policy number, issue age, issue date, sex, classification and face amount. Design a file organization for this application.

TOPIC: APL Database File Organization

The ideal file organization for a given application depends upon the size of the database and upon how it is used. Since this information is missing in the description of the problem, we will present several alternative organizations. In the next section, we will discuss the factors to consider when deciding among these organizations.

For simplicity, we will assume that all of the policyholder information may be expressed as numbers (e.g. sex as 0 or 1). We will refer to the items by the abbreviations: POLNO, IAGE, IDATE, SEX, CLASS, AMT.

The descriptions of 8 alternative APL file organizations follow:

1. Record Oriented

```
Comp.
 No.                    Description
-----    -----------------------------------------------
  1      POLNO, IAGE, IDATE, SEX, CLASS, AMT for 1st policy
  2        "     "     "     "     "      "   "  2nd    "
  3        "     "     "     "     "      "   "  3rd    "
  :
  :
```

2. Record Oriented with Deletion Flag

```
Comp.
 No.                    Description
-----    -----------------------------------------------------
  1      STATUS, POLNO, IAGE, IDATE, SEX, CLASS, AMT for 1st policy
            (STATUS=1 if active record, 0 if "deleted")

  2        "      "      "     "      "    "      "   "  2nd    "
  3        "      "      "     "      "    "      "   "  3rd    "
  :
  :
```

3. Directory

Comp. No.	Description
1	Numeric vector (directory) of POLNO for every policy
2	IAGE, IDATE, SEX, CLASS, AMT for 1st policy (whose POLNO is the 1st element of the directory)
3	" " " " " for 2nd policy " "
:	
:	
I	" " " " " " (I-1)th " " "

4. Directory with Deletion Flag

Comp. No.	Description
1	Boolean vector of STATUS for every policy (STATUS=1 if active record, 0 if "deleted")
2	Numeric vector (directory) of POLNO for every policy
3	IAGE, IDATE, SEX, CLASS, AMT for 1st policy (whose POLNO is the 1st element of the directory)
4	" " " " " for 2nd policy " " "
:	
:	
I	" " " " " (I-2)nd " " " "

5. Transposed

Comp. No.	Description
1	Numeric vector of POLNO for every policy
2	" " " IAGE " " "
3	" " " IDATE " " "
4	" " " SEX " " "
5	" " " CLASS " " "
6	" " " AMT " " "

6. Transposed with Deletion Flag

Comp. No.	Description
1	Boolean vector of STATUS for every policy (STATUS=1 if active record, 0 if "deleted")
2	Numeric vector of POLNO for every policy
3	" " " IAGE " " "
4	" " " IDATE " " "
5	" " " SEX " " "
6	" " " CLASS " " "
7	" " " AMT " " "

7. Multi-Set Transposed

Comp. No.	Description
1	Numeric vector of POLNO for 1st 2000 policies
2	" " " IAGE " " " "
3	" " " IDATE " " " "
4	" " " SEX " " " "
5	" " " CLASS " " " "
6	" " " AMT " " " "
7	Numeric vector of POLNO for 2nd 2000 policies
8	" " " IAGE " " " "
:	
:	
:	
:	
(1+6xI-1)	Numeric vector of POLNO for Ith 2000 policies
(2+6xI-1)	" " " IAGE " " " "
:	
:	

8. Multi-Set Transposed with Deletion Flag

Comp. No.	Description
1	Boolean vector of STATUS for 1st 2000 policies (STATUS=1 if active record, 0 if "deleted")
2	Numeric vector of POLNO for 1st 2000 policies
3	" " " IAGE " " " "
4	" " " IDATE " " " "
5	" " " SEX " " " "
6	" " " CLASS " " " "
7	" " " AMT " " " "
8	Boolean vector of STATUS for 2nd 2000 policies
9	Numeric vector of POLNO for 2nd 2000 policies
10	" " " IAGE " " " "
:	
:	
(1+7xI-1)	Boolean vector of STATUS for Ith 2000 policies
(2+7xI-1)	Numeric vector of POLNO for Ith 2000 policies
(3+7xI-1)	" " " IAGE " " " "
:	
:	

This list of file organizations is not exhaustive. It merely illustrates some typical APL file organizations.

At the two extremes are the record oriented and the transposed organizations. The directory organization is a hybrid of the two. The multi-set transposed organization is a modification of the transposed organization designed to avoid WS FULL errors when working with large databases.

The "deletion flag" alternative exists for any file organization. When you need to delete records from a database, you have two alternatives: delete the record now (shifting other records if necessary to fill the void); or flag the record to be deleted but do not delete it until later (by a procedure which restructures the file to remove all flagged records or by the gradual process of replacing flagged records by new records as they are added).

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: List the factors to consider when choosing among alternative APL file organizations.

TOPIC: File Design Considerations

To choose among file organizations, you must know how much information is to be stored and how it is to be used. For each type of task, consider how well each file organization will stand up to the demands made upon it. In particular, ask yourself:

1. How may file accesses (i.e. read or write operations) will be required? These take time.

2. How CPU efficient will the task be? Does the organization require significant amounts of processing?

3. How efficient will the task be in terms of workspace storage? Are WS FULL errors likely?

4. How complex is the file structure? Will the programs be difficult to write, to read and to debug?

5. Is redundant file storage required? If so, might the file become excessively large? Could its data get out of synch?

The following is a list of representative tasks which are performed on databases. When choosing a file organization, consider each task. Will this task be performed in this application? How often? How well is it performed in this file organization given the performance measures suggested above?

1. Add 1 record
2. Add 100 records
3. Find/change 1 record (1 item)
4. Find/change 1 record (all items)
5. Find/change 100 records (1 item)
6. Find/change 100 records (all items)
7. Find/delete 1 record
8. Find/delete 100 records
9. Find/list 1 record (5 items)
10. Find/list 1 record (all items)
11. Find/list 100 records (5 items)
12. Find/list 100 records (all items)
13. Summarize all records (1 item)
14. Summarize all records (10 items)

The chart below rates the 8 file organizations presented in the last section for each of these 14 tasks. The letters A (excellent) to F (horrible) are used for rating. These ratings are subjective and will vary from application to application but this chart is a good guideline.

File Organization								
1	2	3	4	5	6	7	8	Task
A	A	A	B	C	D	C	D	Add 1 record
A	A	A	B	A	B	A	B	Add 100 records
F	F	A	A	A	B	A	B	Change 1 record, 1 item
F	F	A	A	B	B	B	B	Change 1 record, all items
E	E	C	C	A	B	A	B	Change 100 records, 1 item
E	E	B	B	B	B	B	B	Change 100 records, all items
F	F	B	A	B	A	B	A	Delete 1 record
F	E	C	A	B	A	B	A	Delete 100 records
F	F	A	A	B	B	B	B	List 1 record, 5 items
F	F	A	A	C	C	C	C	List 1 record, all items
E	E	C	C	B	B	B	B	List 100 records, 5 items
E	E	B	B	B	B	B	B	List 100 records, all items
F	F	F	F	A	A	A	A	Summarize all records, 1 item
E	E	E	E	A	A	A	A	Summarize all records, 10 items

Several conclusions may be drawn from this chart:

1. If you intend to do much summarizing or cross-tabulating, you should choose a transposed file organization.

2. Unless you intend to add records and do nothing else, you should avoid a record oriented file organization.

3. No file organization is ideal for all tasks. The best file organization is frequently the one which has the fewest and least severe shortcomings rather than the most strengths. Sometimes a hybrid organization will be the best solution for a given application.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: For a 1,000,000 record APL database, it is critical that a specified record (e.g. policy) be located instantly. How would you organize the file?

TOPIC: Efficient Record Location

The record oriented file organization is out. We do not have the time to do up to 1,000,000 file read operations. Even the multi-set transposed file organization has problems. If blocked at 2000 records per set of components, there will need to be up to 500 file read operations. That is fine for ad hoc file analyses but is unacceptable for instant access.

To solve this problem, we need to utilize the information contained within the key value (record identifier) itself. For example, suppose our records are insurance policies and the record identifier is a policy number. We must use a portion of that number to get us quickly to the vicinity in which the record is located. Consider the following "inverted" directory file:

Comp. No.	Description
1	Two-row matrix with one column per policy whose policy number ends with 000: [1;] policy number [2;] record index where policy data are stored
2	Ditto for policies ending 001
3	Ditto for policies ending 002
:	
:	
1000	Ditto for policies ending 999

This file could be a companion file for any of the file organizations discussed above. The meaning of "record index" depends upon which file organization is used. For example, if the record oriented file organization is used, the record index can simply be the number of the component in which the record is stored. If a transposed file organization is used, the record index can be a number whose format is SSSSIIII where SSSS is the number (index) of the set of components in which the record resides and IIII is the exact index within the components of that set where the record is located.

Given this directory file, any one of the 1,000,000 policies may be located with a single file read operation. For example, to find policy 613821904, read component 905 (i.e. 1+904) of the directory file and search its first row for this number. The corresponding element of the second row contains the record index for the policy.

The term "inverted" is used to refer to a file which stores record indices (or pointers) rather than data values. The trade-off for realizing such rapid record location is that the directory must be set up initially and must be updated as records are added or deleted (or their policy numbers changed). This will slow down the record maintenance process somewhat and will make it more complex. Consequently, such a directory should be included only if essential.

An alternative to the inverted file organization is the "layered" file organization. Suppose the file is layered by the last three digits of the policy number. Rather than maintaining a list of the record indices for each possible value (000 to 999), the records are physically segregated by the values. For example, all records whose policy number end with 904 are kept together on file.

This "layering" is fairly easily accomplished with the multi-set transposed file organization. Each set of components contains records for only a single layer value. For example, the first set of components could contain the information for policies whose policy number ends with 625, the second set with 904, the third set with 707, and so on. If there are more policies with numbers ending in 904 than you can place in one set, use more than one set for the records with that layer value.

Suppose you employ a multi-set transposed file organization blocked at 2000 (maximum) records per set of components and layered by the last three digits of the policy number to store the 1,000,000 records discussed above. On average, each set will contain 1000 records. Some more. Some less. If any layer value (e.g. 000) is so popular that it belongs to more than 2000 records, the records of that layer will occupy more than one set. A directory of layer values is maintained as a vector with one element per set and is stored as a single component of the file.

Given this file organization, any one of the 1,000,000 policies may be located with 2 or 3 or so file read operations. For example, to find policy 613821904, read the layer values vector and search it for 904. The matching element(s) identify the set(s) whose records have policy numbers ending with 904. Then, read the policy number component for that set (or sets) and search for the policy number. The result is the index within the set at which that record is located.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: What should be included in written file documentation?

TOPIC: File Documentation

Since a file has no value except as employed in an application, its documentation should be couched in terms relevant to the application. For example, if the file is activated by "tying" it to an arbitrary number, show the tie number which is actually used in the application.

When components are read into the active workspace from the file, they may technically be assigned to any variable name. Show the names which are actually used in the application. When describing a file component, indicate its shape and type and the significance of its value.

The following is an illustration of proper file documentation. Do not waste your time studying its intricacies. Rather, use it to become comfortable with the general structure of good file documentation.

-- -

FILE NAME: POLICY TIE NUMBER: 321

DESCRIPTION: Contains policyholder information

Comp. No.	VARIABLE	DESCRIPTION
1	TDATE	Integer scalar of the transaction date (YYYYMMDD) when policyholder information was last added to the file from the administration system.
2	CTYPES	Integer vector of the available underwriting classification codes.
3	CNAMES	10 column character matrix of brief names for the underwriting classes; the rows of CNAMES are in 1-to-1 correspondence with the elements of CTYPES.
4	FIV	Field identification vector. Integer vector with one element per field of information on file (e.g. policy number, issue age, sex, ...). The value indicates the type of array

		used to store the information: 10: Deletion flag Boolean vector (1: active record) 11: Boolean vector 12: Character vector 13: Integer vector 14: Floating point vector nn2: Character matrix with nn columns
5	FP	File parameters vector. FP[1]: Number of active records FP[2]: Number of records (including deleted) FP[3]: Number of fields (i.e. ρFIV) FP[4]: Maximum number of records, per set of FP[3] components FP[5]: Number of sets of FP[3] components
6	ARPS	Active records per set. Integer vector with one element per set (FP[5]) of the number of active (not deleted) records per set. Note: (+/ARPS)=FP[1]
7	RPS	Records per set. Integer vector with one element per set (FP[5]) of the number of records (including deleted) per set. Note: (+/RPS)=FP[2]
8-10	(latent)	Empty numeric vector
F+10+FP[3]xS-1		Array of data for field F (1 to FP[3]) in set S (1 to FP[5]). The type and rank of this array is defined by FIV[F]. The length of its first dimension is RPS[S].

-- -

The fields of data are:

FIELD NO.(F)	VARIABLE NAME	FIV	DESCRIPTION
1	STATUS	10	Deletion flag (1=active; 0=deleted)
2	POLNO	14	Policy number (up to 13 digits)
3	IAGE	13	Issue age (NN)
4	IDATE	13	Issue date (YYYYMMDD)
5	SEX	12	Sex ('M' or 'F' or ' ' if unknown)
6	CLASS	13	Underwriting classification code (an element of CTYPES)
7	AMT	13	Face amount (cents)

-- -

If you inherit the maintenance responsibility for an application system which has no file documentation, your first task is to reconstruct the file documentation. This is usually possible by carefully reviewing the functions which access the files and by reviewing the file components themselves. To display the functions for your review, use the QDOC or WSDOC functions defined in the Workspace Design and Documentation chapter.

To display the file components, use the FILEDOC function described here. The right argument is the same as that of WSDOC: page height, width, top margin, bottom margin, left margin, right margin (e.g. 66 85 3 3 10 5). The left argument identifies the file to be documented (e.g. file tie number). The output is paged and looks like:

```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -

FILE: 21368 POLICY (521 COMPONENTS) * 11/27/86  12:06          PAGE 1

COMPONENT   SHAPE ρ VALUE
---------   ----- - -----
    1               19861025
    2          11 ρ 31 32 33 34 41 42 42 51 52 53 99
    3       11 10 ρ 'STANDARD   P38K4      P39K4     P3...
    4           7 ρ 10 14 13 13 12 13 13
    5           5 ρ 801625   801643   7   2000   401
    6         401 ρ 2000 1998 2000 2000 2000 2000 1995 2000...
    7         401 ρ 2000 2000 2000 2000 2000 2000 2000 2000...
  8-10          0 ρ ι0
   11        2000 ρ 13156281325 21065134890 21065338190...
    :
    :
                                                COMPONENTS 1 TO 67
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```

The writing of FILEDOC is left as an exercise at the end of the chapter.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Design a set of utility functions for accessing APL files. The functions should be intuitive (i.e. be analagous to APL primitive functions) and should have a syntax which is independent of the chosen file organization and independent of your implementation of APL files.

TOPIC: File Utility Functions

By designing and using such a set of file utility functions, you can become more productive. The procedure for working with files becomes:

1. Design the file organization for a given application;

2. Implement these utility functions for the given file organization;

3. Use the utility functions instead of primitive (e.g. ⎕FREAD or ⎕READ) file access functions.

By using a consistent set of utility functions, you can work on many application systems without having to continually reorient yourself to the different file organizations. In effect, the utility functions shelter you from the intricacies of each file organization.

Below is a recommended set of file utility functions. In each function, the variable FP (file parameters) represents a numeric scalar or vector which distinctly identifies the file to be used (e.g. file tie number or tie number and blocking factors). If your application deals with just one file and its organization is sufficiently unusual that you are unlikely to need these functions for a similar file, you may omit the FP argument altogether.

The philosophy behind the design of these file utility functions is to treat the file as a matrix in your workspace. The rows of the matrix are called "records". The columns are called "fields". Each utility function emulates some common matrix operation. For example, imagine the data stored in a workspace matrix named FILE. A common matrix operation is adding new records:

```
FILE←FILE,[1]NEWDATA
```

The corresponding file utility function has the syntax:

```
FP←FP CATREC NEWDATA
```

Along with the syntax of each utility function is listed the analogous APL expression for operating on a matrix named FILE.

There are two important distinctions to keep in mind. One is that a field does not need to represent a vector of data. It may be a matrix. For example, a field of employee names may be stored on file as a 20 column character matrix. Here, we treat the names as a single field. Hence, the analogous APL expression will treat them as a single column.

The second distinction is that "record indices" means values that are understood by the utility functions to identify particular records. They do not necessarily mean ⍳ indices. For example, for a multi-set transposed file organization, a single record index might be a two element vector whose first element is the index of the set and whose second element is the array index within the set.

It is not expected that you will implement all of the following utility functions. Rather, you should select those most useful for the application and implement them.

(STSC's File Manager product -- originally marketed as "EMMA" -- provides a comprehensive set of such file utility functions for an APL★PLUS-based multi-set transposed file organization.)

```
SYNTAX:  FP INITFILE FT
         FILE←FP⍴FT
```

INITFILE is used to build an "empty" file whose file parameters are FP and whose field types are defined in FT.

```
SYNTAX:  FP←FP CATREC MAT
         FILE←FILE,[1]MAT
```

CATREC is used to add (catenate) records to the end of the file. MAT is a matrix of information to be catenated (or inserted into records flagged for deletion). Each row of MAT represents a single new record. Each column of MAT corresponds to a single field, or column of a matrix field, of data (excluding the deletion flag field, if any). If MAT is a vector, it is treated like a one-row matrix. If a scalar or one-element array, it is catenated to the bottom of each field as a single record. The result is the modified value of FP.

```
SYNTAX:  FP←FP CATRECWS NREC
         FILE←FILE,[1]F1,F2,F3,...
```

CATRECWS is used to add (catenate) records to the end of the file. NREC is an integer scalar whose value represents the number of records to be catenated (or inserted into records flagged for deletion). The data for these new records are located in the global field variables F1, F2, F3, ... where F1 is a vector of values for the first field of the record (or a matrix with one row per record), F2 is for field 2, F3 is for field 3, and so on. One variable is required per field (excluding the deletion flag field, if any). Regardless of the magnitude of NREC, any field variable may be a one-row matrix, a vector with one element per column of the matrix field, or a scalar or one-element array. The data will be reshaped and catenated to the bottom of the field for NREC records. The result is the modified value of FP. The field variables are erased upon successful completion of the function.

```
SYNTAX:  RINDS←(FP,KFLD) IOTA VALUES
         RINDS←FILE[;KFLD]⍳VALUES
```

IOTA searches through the file (ignoring deleted records) for the first occurrences of records whose key value (e.g. policy number, employee number, transaction number) is specified in VALUES. IOTA behaves like dyadic ⍳. That is, its result contains one index per value in the right argument, in 1-to-1 correspondence. The elements of RINDS are record indices which can be used to directly locate the records. The elements of RINDS are ¯1 for those elements of VALUES not found. The left argument of IOTA may be just FP if there is only one key (identifying) field. Otherwise, the number of the key field (KFLD) is included in the left argument.

```
SYNTAX:  RINDS←IOTARHO FP
         RINDS←⍳1↑⍴FILE
```

IOTARHO returns the record indices of all active (not deleted) records in the file. IOTARHO behaves like monadic ⍳⍴. That is, its result contains all of the indices for the specified array (file). The elements of RINDS are record indices which can be used to directly locate the records.

```
SYNTAX:  RINDS←SVEC SLASHIOTARHO FP,SFLDS
         RINDS←(⍎SVEC)/⍳1↑⍴FILE
```

SLASHIOTARHO returns the record indices of all active (not deleted) records in the file which satisfy a specified criterion. SVEC is a character vector APL expression (e.g. '(F3>50)∧F2≠0') which defines the desired criterion. The expression is stated in terms of field variables F1, F2, F3, ... which represent the data stored in the 1st, 2nd, 3rd, ... fields of each record. The expression should be constructed such that when executed, its result is a Boolean vector with one element per record (i.e. per element or row of F1, F2, F3, ...) in which ones marks records selected. SLASHIOTARHO behaves like dyadic /⍳⍴. That is, its result contains all of the indices for the specified array (file) which satisfy the specified criterion. The elements of RINDS are record indices which can be used to directly locate the records. The right argument (beyond FP) is an integer vector of the indices of the field variables to be constructed before executing the expression (e.g. 2 3 for '(F3>50)∧F2≠0' or 1 3 7 for '0<PROCESS F3' where F1 and F7 are required by PROCESS as global

variables). If SVEC and SFLDS are empty, SLASHIOTARHO returns the record indices of all active records in the file.

```
SYNTAX:  FP←FP DELREC RINDS
         FILE←(~(⍳1↑⍴FILE)∊RINDS)⌿FILE
```

DELREC deletes specified records from the file. The elements of RINDS are the indices of the records to be deleted (as returned by IOTA, IOTARHO or SLASHIOTARHO). The result is the modified value of FP (if modified).

```
SYNTAX:  FP←SVEC COMPRESS FP,SFLDS
         FILE←(⍎SVEC)⌿FILE
```

COMPRESS deletes all records in the file which do not satisfy a specifed set of criteria.

```
FP←SVEC COMPRESS FP,SFLDS
```

has the same effect as

```
FP←FP DELREC ('~',SVEC) SLASHIOTARHO FP,SFLDS
```

Notice that the former expression does not need to construct intermediate record indices for the selected records and so is more efficient than the latter expression. If SVEC and SFLDS are empty, no records are deleted.

```
SYNTAX:  MAT←RINDS INDEX FP,FLDS
         MAT←FILE[RINDS;FLDS]
```

INDEX is used to retrieve from file the data (MAT) from selected fields (FLDS) for specified records (RINDS). The elements of RINDS are the indices of the records to be retrieved (as returned by IOTA, IOTARHO or SLASHIOTARHO). The elements of FLDS are indices of the fields to be retrieved. The result is a matrix of the retrieved data with one row per element of RINDS and one column per element of FLDS.

```
SYNTAX:  RINDS INDEXWS FP,FLDS
         F1←FILE[RINDS;1] ◇ F2←FILE[RINDS;2] ◇ ...
```

INDEXWS is used to retrieve from file the data from selected fields (FLDS) for specified records (RINDS). The elements of RINDS are the indices of the records to be retrieved (as returned by IOTA, IOTARHO or SLASHIOTARHO). The elements of FLDS are indices of the fields to be retrieved. The retrieved data are assigned to global variables named Fn where n is the number of the field retrieved (e.g. F3 and F7 for FLDS←3 7). The global variables are vectors with one element (or matrices with one row) per element of RINDS.

```
SYNTAX:  MAT←SVEC SELECT FP,FLDS,0,SFLDS
         MAT←(⍎SVEC)⌿FILE[;FLDS]
```

SELECT is used to retrieve from file the data (MAT) from selected fields (FLDS) for all active (not deleted) records in the file which satisfy a specified set of criteria (SVEC).

```
          MAT←SVEC SELECT FP,FLDS,0,SFLDS

              has the same effect as

          MAT←(SVEC SLASHIOTARHO FP,SFLDS) INDEX FP,FLDS
```

Notice that the former expression does not need to construct intermediate record indices for the selected records and so is more efficient than the latter expression. If SVEC and SFLDS are empty, SELECT retrieves data for all active records in the file.

```
SYNTAX:  SVEC SELECTWS FP,FLDS,0,SFLDS
         F1←(⍎SVEC)⌿FILE[;1] ◇ F2←(⍎SVEC)⌿FILE[;2] ◇ ...
```

SELECTWS is used to retrieve from file the data from selected fields (FLDS) for all active (not deleted) records in the file which satisfy a specified set of criteria (SVEC).

```
          SVEC SELECTWS FP,FLDS,0,SFLDS

              has the same effect as

         (SVEC SLASHIOTARHO FP,SFLDS) INDEXWS FP,FLDS
```

Notice that the former expression does not need to construct intermediate record indices for the selected records and so is more efficient than the latter expression. If SVEC and SFLDS are empty, SELECTWS retrieves data for all active records in the file.

```
SYNTAX:  FP←RINDS INDEXA (FP,FLDS) ASSIGN MAT
         FILE[RINDS;FLDS]←MAT
```

INDEXA is used to replace on file the data in selected fields (FLDS) for specified records (RINDS). The elements of RINDS are the indices of the records to be replaced (as returned by IOTA, IOTARHO or SLASHIOTARHO). The elements of FLDS are indices of the fields to be replaced. MAT is a matrix of the data to be replaced with one row per element of RINDS and one column per field (or per column of a matrix field) identified in FLDS. Mat may be a vector if FLDS identifies a single vector field. Regardless of the number of records identified by RINDS, MAT may be a one-row matrix or vector with one element per column of the fields, or a scalar or one-element array. The data will be reshaped and assigned to the specified records. The result is the modified value of FP. The ASSIGN function simply assigns its right argument to <assign> and returns its left argument. INDEXA erases the variable <assign> when done with it.

```
SYNTAX:  FP←RINDS INDEXWSA FP,FLDS
         FILE[RINDS;1]←F1 ◇ FILE[RINDS;2]←F2 ◇ ...
```

INDEXWSA is used to replace on file the data in selected fields (FLDS) for specified records (RINDS). The elements of RINDS are the indices of the records to be replaced (as returned by IOTA, IOTARHO or SLASHIOTARHO). The elements of FLDS are indices of the fields to be replaced. The replaced data is taken from global field variables named Fn where n is the number of the field replaced (e.g. F3 and F7 for FLDS←3 7). The global variables are vectors with one element (or matrices with one row) per element of RINDS. If the field variable is a one-row matrix or a vector with one element per column of the matrix field, it will be applied across all records identified by RINDS. If the field variable is a scalar or one-element array, it will be applied across all records and all columns of the field. The result is the modified value of FP. The field variables are erased upon successful completion of the function.

```
SYNTAX:  FP←(FP,XFLDS) EXECUTE XVEC
         ⍎XVEC
```

EXECUTE is used to execute a specified (XVEC) character vector APL expression (e.g. 'SUM←SUM++/F4'). The expression is stated in terms of field variables F1, F2, F3, ... which represent the data stored in the 1st, 2nd, 3rd, ... fields of each active (not deleted) record. The expression is executed once for each block of records on file (each active record in a record oriented file organization or each set of records in a multi-set transposed file organization). The left argument (beyond FP) is an integer vector of the indices of the field variables involved in the expression. Positive indices indicate fields to be read from file before execution of the expression; negative indices indicate fields to be replaced on the file after execution of the expression. The result is the modified value of FP. For example:

```
 FP←(FP,¯7 7 2 4) EXECUTE 'F7←F7⌈F2÷F4'

 SUM←0
⍝ XTAB is a function which assumes globals F3, F6, F9:
 FP←(FP,3 6 9) EXECUTE 'SUM←SUM + XTAB'
```

SYNTAX: FP←(FP,XFLDS,0,SFLDS) EXECUTE XVEC FOR SVEC
⍎XVEC

In this context, EXECUTE is used to execute a specified (XVEC) character vector APL expression for only those active (not deleted) records which satisfy a specified set of criteria (SVEC). The FOR function simply catenates and returns its two character vector arguments, separating them by a newline character. Both expressions are stated in terms of field variables F1, F2, F3, ... which represent the data stored in the 1st, 2nd, 3rd, ... fields of each record. SVEC should be constructed such that, when executed, its result is a Boolean vector with one element per record (i.e. per element or row of F1, F2, F3, ...) in which ones mark records to be selected for subsequent construction of the field variables to be included in the execution of XVEC. XFLDS and SFLDS are integer vectors of the indices of the field variables involved in the respective expressions XVEC and SVEC. Negative elements of XFLDS indicate fields to be replaced after execution of XVEC. The result is the modified value of FP. For example:

```
FP←(FP,¯7 7 2 4 0 3) EXECUTE 'F7←F7⌈F2÷F4' FOR 'F3>1'
```

SYNTAX: RINDS←(FP,KFLD) IOTA VALUES LAYERS Z
RINDS←IOTARHO FP LAYERS Z
RINDS←SVEC SLASHIOTARHO FP,SFLDS LAYERS Z
FP←SVEC COMPRESS FP,SFLDS LAYERS Z
MAT←SVEC SELECT FP,FLDS,0,SFLDS LAYERS Z
SVEC SELECTWS FP,FLDS,0,SFLDS LAYERS Z
FP←(FP,XFLDS) EXECUTE XVEC LAYERS Z
FP←(FP,XFLDS,0,SFLDS) EXECUTE SVEC FOR SVEC LAYERS Z

LAYERS can be used when the file is layered (see Efficient Record Location section in this chapter). When used, the scope of the file utility function is limited to just those records whose layer value is in the list of layer values Z. All other records are ignored. The LAYERS function simply assigns its right argument to <layers> and returns its left argument. The file utility function erases the variable <layers> when done with it.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Design and document a precise file layout to implement the file utility functions of the previous section for a multi-set transposed file organization. Keep the layout general enough to allow a deletion flag or not, and to allow layers or not.

TOPIC: Multi-Set Transposed File Organization

Here is a possible file layout for a multi-set transposed file organization. A discussion follows it.

```
     FILE NAME: up to you                  TIE NUMBER: up to you

     DESCRIPTION: Multi-set transposed file with optional record
                  deletion flags and optional layers.

     COMP.
      NO.   VARIABLE   DESCRIPTION
     -----  --------   ------------------------------------------
       1      DOC      Character matrix or vector (with embedded
                       newlines) description of the purpose and
                       contents of this file. [optional]

       2      FN       Character matrix of abbreviated field names
                       (as valid identifier names) with one left
                       justified name per row. [optional]

                              (1↑⍴FN)=(1↓⍴FT)

       3      FD       Character matrix of full field descriptions
                       with one description per row. [optional]

                              (1↑⍴FD)=(1↓⍴FT)

       4      FT       Two row integer matrix of field type
                       information with one column per field (FP[3]
                       columns).  The meanings of the values are:

                         FT[1;] Field width.  If 0, the field is
                                inactive (latent).  If 1, the field is
                                a vector field.  Otherwise, the field
                                is a matrix field with this many
                                columns.

                         FT[2;] Field datatype.  Options: 1 (Boolean),
                                2 (character), 3 (integer),
                                4 (floating point).

      5-6   (latent)   Available for custom requirements. Contain: ⍳0.
```

COMP. NO.	VARIABLE	DESCRIPTION
7	FP	Integer vector of file parameters. The meaning of the values are: FP[1] Field number of layer value field (origin 1) or 0 if file not layered. FP[2] Tie number used when accessing file FP[3] Number of fields, including latent fields. FP[3]=(1↓ρFT). FP[4] Number of components preceding the data components. FP[4]≥10. FP[5] Maximum number of records per set of components. ∧/FP[5]≥RPS. FP[6] Number of sets, including sets with no active records. FP[6]=(ρRPS)=(ρARPS). FP[7] Number of records, including records flagged for deletion. FP[7]=(+/RPS). FP[8] Number of active (not latent) fields, including the deletion flag field if one exists. FP[8]=(+/×FT[1;]) FP[9] Number of active (some active records) sets. FP[9]=(+/0≠ARPS). FP[10] Number of active (not flagged for deletion) records. FP[10]=(+/∣ARPS). FP[11] Magnitude is field number of deletion flag Boolean vector field (origin 1) or 0 if no deletion flag. If nonzero: 1∧.=FT[;∣FP[11]]. If negative: FP[5] inactive records are added for each new set, i.e. FP[5]∧.=RPS.
8	RPS	Integer vector with one element per set (FP[6] elements) of the number of records, including those flagged for deletion, in each set.
9	LV	Layer values. LV is a vector with one element per set (FP[6] elements) if FT[1;FP[1]]=1 or a matrix with one row per set if FT[1;FP[1]]>1. LV is not used (empty numeric vector) if FP[1]=0.

COMP. NO.	VARIABLE	DESCRIPTION
10	ARPS	Integer vector with one element per set (FP[6] elements) whose magnitude is the number of active (not flagged for deletion) records in each set. If positive, the active records in the corresponding set are the leading records (no interspersed deleted records). If negative, the active records are not exclusively the leading records. This component is not used (empty numeric vector) if FP[11]=0.
11 to FP[4]	(latent)	Available for custom requirements. Contain: ⍳0.
(F+FP[4]) +(FP[3]× S-1) where: 1≤F≤FP[3] 1≤S≤FP[6]	DATA	Array of data for field F in set S. The array is an empty numeric vector (latent) if FT[1;F]=0. It is a vector if FT[1;F]=1. It is an n-column matrix (n←FT[1;F]) if n≠1. The array (if not latent) has RPS[S] elements or rows, of which |ARPS[S] are active, i.e. not flagged (by a corresponding 0 in field |FP[11]) for deletion. The array will never have more than FP[5] elements or rows (i.e. records), and will always have exactly FP[5] elements or rows if FP[11]<0.

Components 1 (DOC), 2 (FN) and 3 (FD) are not essential pieces of the file. However, since they serve to document the file, they are included and recommended.

Components 4 (FT) and 7 (FP) completely define the structure of the file. They are constructed to satisfy the requirements of the particular application. After creating an empty file (via ⎕FCREATE or ⎕CREATE) and assigning values to FT and FP, you initialize the file by calling INITFILE with FP and FT as its arguments.

Components 5 and 6 contain empty vectors and are not used. It is a good practice to leave a few "latent" components for future unanticipated requirements.

Components 8 (RPS), 9 (LV) and 10 (ARPS) are used and updated as needed by the utility functions.

Components 11 to FP[4] are additional latent components in case more than 2 (components 5 and 6) are needed for the particular application.

All components beyond FP[4] are reserved for the data.

The implementation of the file utility functions for this specific file layout is left as an exercise in the problems at the end of the chapter.

This file layout is illustrated and discussed further in the next section.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Construct a file for maintaining a database of information about 45,000 insurance policyholders. For each policyholder, you will keep track of: policy number (12 digits/letters), issue age, issue date, sex (M or F), classification (S,A,B,C,D) and face amount. Layer the file by classification.

TOPIC: An Illustration of File Utilities

The first step is to define the fields for this file. They are:

1. Policy number
2. Issue age
3. Issue date
4. Sex
5. Classification
6. Face amount
7. Deletion bit

The last field, deletion bit, is optional. You should consider how the file will be used before you decide whether or not to employ a deletion bit field. If employed, record deletion is quick because deleted records are not physically removed from file. They are simply flagged for deletion (bit=0). However, record searching and retrieval will be slower because "deleted" records must be detected and ignored.

For this illustration, we will employ the deletion bit field.

We have defined 7 fields but let's allow room for another 3 fields for future expansion:

8. (latent)
9. (latent)
10. (latent)
~~~~

The next step is to specify the nature of each field precisely by constructing FT, which will be component 4 of the file.

```
WIDTH←12 1 1 1 1 1 1 0 0 0
TYPE ← 2 3 3 2 2 4 1 1 1 1
FT←WIDTH,[.5]TYPE
```

Since the policy numbers (field 1) may contain letters as well as digits, they will be stored in a 12 column character (type 2) matrix field. Issue age (field 2) and issue date (field 3) will be stored as integer (type 3) vector (width 1) fields. The dates will be stored in YYYYMMDD format. Sex (field 4) and classification (field 5) will be stored as character vector fields. Face amount (field 6) will be stored as a floating point (type 4) vector field since the values may contain cents but will be stored in dollar units. Deletion bit (field 7) must be stored as a Boolean (type 1) vector field. Fields 8 to 10 are stored as latent (width 0) Boolean fields.

Next, define the elements of FP, which will be component 7 of the file. The file will be layered by classification which is field 5:

```
LAYER←5
```

Let's pick a tie number to which the file will be tied when used:

```
TIE←987
```

There are 7 active fields and 10 fields in all:

```
AFLDS←7
INCR←10
```

The file begins with 10 reserved components. Let's include another 40 latent components (11 through 50) in case we later need a place to store related items.

```
DISP←50
```

The file will contain 45,000 or so records. Let's keep our components down to some manageable size so WS FULL errors are kept to a minimum and so new records may be added without having to read giant objects. Let's arbitrarily limit each data object to 2000 records. This means the 45,000 record file will contain at least 23 sets of data components. To read one field for all records will require at least 23 file read operations. By increasing the blocking factor, the number of sets decreases, and vice versa. For some applications, you may want a block size of 30,000. For others, a block size of 10 may be ideal. Here, we will use 2000.

```
BLK←2000
```

Since we are using a deletion bit field, we can choose either of two methods for adding new sets. After the 2000th record is added to the last set of the file and another record is to be added, a new set must be appended to the file. This set mayh be appended with just the single new record, or with 2000 records, 1999 of which are flagged deleted. If the latter approach is taken, subsequent new records will simply replace "deleted" records.

From your point of view, as the programmer using the file utility functions, the utility functions behave the same regardless of the approach chosen. Which approach you should use is a function of your APL file system implementation. Some systems behave poorly (i.e. gobble up much disk storage) when asked to replace an object in a component by a larger object. On such systems, you should choose the latter approach so that all the records of the set are added at once. Then, the components in that set will not grow.

Here, we will choose the latter approach by specifying the number of the deletion field as a negative number:

```
          DFLD←¯7
```

Let's construct FP:

```
          FP←LAYER,TIE,INCR,DISP,BLK,0,0,AFLDS,0,0,DFLD
```

We will name the file 'POLDATA'. Create the file and run INITFILE:

```
          'POLDATA' ⎕FCREATE TIE          (or ⎕CREATE on SHARP APL)
          FP INITFILE FT
```

At this point the file has 50 (i.e. DISP) components, no records and no sets. Before we start adding records, let's take a valuable moment to construct and replace the 3 documentation components:

```
          DOC←'Insurance policyholder database'
          FN←7 5⍴'PNUM IAGE IDATESEX  CLASSFACE DBIT '
          FD←(15↑'Policy number'),[1]...,[.5]15↑'Deletion bit'
          DOC ⎕FREPLACE TIE,1             (or ⎕REPLACE on SHARP APL)
          FN ⎕FREPLACE TIE,2
          FD ⎕FREPLACE TIE,3
```

From this point on, we can simply use the file utility functions. Let's catenate 4 records:

```
          F1←(12↑'ABCD'),[1](12↑'A1'),[1](12↑'XYZ99'),[.5]12↑'P55'
                                                   (policy number)
          F2←25 55 45 35                           (issue age)
          F3←19820715 19850123 19851230 19860402   (issue date)
          F4←'M'                                   (sex, all male)
          F5←'SASS'                                (classification)
          F6←50000 30000 40000 25000               (face amount)

          FP←FP CATRECWS 4
```

Which ones are 45 or older?

```
      'F2≥45' SELECT FP,1 0 2
XYZ99
A1
```

Make the 45 year old a female:

```
      FP←((FP,2) IOTA 45) INDEXA FP,4 ASSIGN 'F'
```

Increase each face amount by a factor of 100 for those with standard (S) classification:

```
      FP←(FP, ¯6 6) EXECUTE 'F6←100×F6' LAYERS 'S'
```

Return the ages of those with standard (S) classification:

```
      ,'' SELECT FP,2 LAYERS 'S'
25 45 35
```

Delete policyholder records with issue dates in 1985:

```
      FP←'1985≠⌊F3÷10000' COMPRESS FP,3
```

When done using the file, untie it:

```
      ⎕FUNTIE FP[2]                    (or ⎕UNTIE on SHARP APL)
```

To use the file again later, retie the file and read the file parameters vector from component 7:

```
      'POLDATA' ⎕FTIE 987              (or ⎕TIE on SHARP APL)
      FP←⎕FREAD 987 7                  (or ⎕READ on SHARP APL)
```

You can then use the file utility functions again.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 383 to 448)

1. Suppose you have developed an application and are encountering frequent WS FULL error messages. Perhaps there are too many functions in the workspace. Design and implement a set of file utility functions which may be used to store some of these functions on file and to retrieve them when needed.
~~~~

2. Sketch a flowchart of the FILEDOC function described in this chapter. Compare it to the flow of the FILEDOC function listed in the solutions at the back of the book.

3. In the Workspace Design and Documentation chapter, a simple application system is developed for maintaining a list of employees. Rewrite the EMPLOYEES function (the large one) to assume that employee information will be kept on file rather than in global workspace variables. Do not design a precise file organization. Rather, use the file utility functions introduced in this chapter.

4. Try your hand at writing one or more of the file utility functions for the specific multi-set transposed file layout presented in this chapter. Compare your function to the listing of that function included in the solutions at the back of the book. The functions listed require an APL*PLUS system. If you use SHARP APL or APL2, see the next problem.

5. What general modifications must be made to the APL*PLUS system file utility functions written in the previous problem so that they will work on a SHARP APL system? On an APL2 system?

Chapter 15

BOOLEAN TECHNIQUES

The treatment of logical conditions in APL is simple and powerful. The concept of "true" is represented by the numeric value 1 and "false" by 0. These values may be manipulated with the same ease as those of any other numeric values. An array which contains only 1s and 0s is called a "Boolean" array (after the mathematician George Boole) or a "bit" array (after the unit of computer storage).

The APL language contains 7 primitive functions which return exclusively Boolean results (=,≠,>,≥,<,≤,∊). These are called relational functions. There are 5 primitive functions which not only return Boolean results, but also require exclusively Boolean arguments (~,∧,∨,⍲,⍱). These are called logical functions.

On the surface, these 12 functions provide an adequate, but not wonderful, set of capabilities for working with logical data. However, by applying the APL operators (such as reduction and scan) to these functions and by utilizing the "shift and compare" techniques available in APL, you can begin to appreciate the rich Boolean functionality of APL. Extremely complex problems can be solved directly using noniterative Boolean techniques that would never even be considered in another programming language.

This chapter presents no new or complex APL primitive functions. Rather it presents deeper interpretations of existing simple functions. The aim of the chapter is to develop your Boolean vocabulary and to broaden your thinking when faced with Boolean problems.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: The dyadic logical functions ∧, ∨, ⍲, ⍱ require Boolean arguments and return Boolean results. Since they are scalar functions, the result is a scalar when both arguments are scalars. For the set of four possible pairs of arguments (0 and 0; 0 and 1; 1 and 0; 1 and 1), each function returns its four distinct Boolean results. For example, the results of the ∧ (and) function may be expressed in the table:

Right Argument

(∧)		0	1
Left Argument	0	0	0
	1	0	1

Viewing this table of ∧ results as a vector (0 0 0 1), you can see that the results for ∨, ⍲, ⍱ are respectively: 0 1 1 1, 1 1 1 0, 1 0 0 0. There are 16 possible combinations of 4 bits. These 4 functions produce only 4 of them. What functions may be used to generate the rest? What logical interpretations can be given to each of these functions?

TOPIC: Logical Scalar Functions

For a left argument L←0 0 1 1 and a right argument R←0 1 0 1, the 16 possible combinations of results may be generated by the following expressions:

Result	Expression (L←0 0 1 1) (R←0 1 0 1)	Interpretation	Equivalent
0 0 0 0	(ρL)ρ0	Always false	
0 0 0 1	L∧R	And (both; "multiplication")	
0 0 1 0	L>R	Except (unless; and not; "subtraction")	L∧(~R)
0 0 1 1	L	Left argument	
0 1 0 0	L<R	Nor not	(~L)∧R
0 1 0 1	R	Right argument	
0 1 1 0	L≠R	Toggle if (exclusive or)	
0 1 1 1	L∨R	Or ("addition")	
1 0 0 0	L⍱R	Nor (neither)	~(L∨R)
1 0 0 1	L=R	Toggle if not	
1 0 1 0	~R	Not right argument	
1 0 1 1	L≥R	Or not	L∨(~R)
1 1 0 0	~L	Not left argument	
1 1 0 1	L≤R	Nand not	(~L)∨R
1 1 1 0	L⍲R	Nand (not both)	~(L∧R)
1 1 1 1	(ρL)ρ1	Always true	

To illustrate the use of these logical expressions, let us solve some problems.

A. Given a vector DEP of bank deposits, how many deposits are between (inclusive) 100 and 200?

```
+/(DEP≥100)∧DEP≤200            (and)
+/(DEP<100)⍱DEP>200            (nor)
```

B. How many are greater than 250, ignoring those which are exactly 500?

```
+/(DEP>250)>DEP=500            (except, and not)
+/(DEP=500)<DEP>250            (nor not)
```

(It is easy to see why the second expression is typically read "how many deposits where (DEP>250) except where (DEP=500)", rather than "how many deposits where (DEP=500) nor not where (DEP>250)".)

C. How many are either 100 or greater than 250, ignoring those which are exactly 500?

```
+/(DEP>250)≠DEP∊100 500        (toggle if)
+/(DEP≤250)=DEP∊100 500        (toggle if not)
```

D. How many are smaller than 10 or larger than 1000?

```
+/(DEP<10)∨DEP>1000          (or)
+/(DEP≥10)⍲DEP≤1000          (nand)
+/(DEP<10)≥DEP≤1000          (or not)
+/(DEP≥10)≤DEP>1000          (nand not)
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: There are 10 scalar dyadic relational or logical functions (=, ≠, >, ≥, <, ≤, ∧, ∨, ⍲, ⍱). All of these may be used with the reduction or scan operators to derive functions which can operate on Boolean arrays. Which of these 20 derived functions have useful interpretations? What are the interpretations?

TOPIC: Logical Reductions and Scans

Obviously, the word "useful" is subjective. So let us be subjective. Only two of the reductions have useful interpretations. However, we will throw in a third reduction (+/) since its interpretation becomes "how many" rather than "add up" when its argument is Boolean.

Expression	Interpretation
+/R	How many
∧/R	All
∨/R	Any

To illustrate these functions, let us solve some problems.

A. Given a character vector CVEC, how many nonblank elements does it contain?

```
+/CVEC≠' '             (how many)
```

B. Are all of the elements nonblank?

```
∧/CVEC≠' '             (all)
~∨/CVEC=' '            (not any; none)
```
~~~~

Six of the scans have useful interpretations.

Expression	Interpretation
∧\R	Leading
∨\R	Not leading not (leading 0s)
<\R	First
≤\R	Not first not (first 0)
≠\R	Leading 1-poles to 1-maps
=\R	Leading 0-poles to 0-maps

Let us solve some problems using these functions.

A. How many leading blanks (blanks before the first nonblank) exist in the character vector CVEC?

```
+/∧\CVEC=' '                                    (how many leading)
```

B. Delete the leading blanks from CVEC, returning the elements beyond the leading blanks.

```
(∨\CVEC≠' ')/CVEC                               (leading 0s)
```

C. Is the first nonblank character in CVEC a digit?

```
∨/((<\CVEC≠' ')/CVEC)∊'0123456789'              (first)
```

(∨/, i.e. any, is used in case there is no first nonblank)

D. Delete the first ',' from CVEC, returning the rest of CVEC.

```
(≤\CVEC≠',')/CVEC                               (first 0)
```

The remaining two scans (≠\ and =\) require some explanation before using them to solve problems. A "maps" vector is a Boolean vector which consists of sets of contiguous 1s (1-maps) separated by one or more 0s (0-maps). For example, the following bit vector contains 3 1-maps (each of which is underlined) and 4 0-maps:

```
0 0 1 1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0
    -------       ---     ---------
```

A "leading 1-poles" vector is a Boolean vector which consists of pairs of 1s, separated by zero or more 0s. The 1s are called "poles". The left pole in each pair may be viewed as the starting element of a set of contiguous elements. The right pole in each pair may be viewed as the next element beyond the ending element of the set. For example, the following bit vector contains 3 pairs of leading 1-poles.

```
0 0 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0
    -------       ---     ---------
```

Notice that 1-maps and leading 1-poles are alternate means of conveying the same information. Specifically, they each identify spans of contiguous elements. 1-maps do so by using 1s to flag the elements within the spans. Leading 1-poles do so by using 1s to flag the starts of spans and the starts of non-spans (hence the word "leading").

The ≠\ function converts bit vectors from the leading 1-pole representation to the 1-maps representation:

```
      ⎕←≠\⎕←0 0 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0
0 0 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0
0 0 1 1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0
```

The =\ function converts bit vectors from the leading 0-poles representation (use 0s as poles instead of 1s) to the 0-maps representation (use 0s as maps instead of 1s).

```
      ⎕←=\⎕←1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 0 1
1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 0 1
1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 1 1
```

Let us solve some problems.

E. Given a character vector CVEC, return all characters within quotes.

```
(≠\CVEC='''')/CVEC
```

(This expression also returns the leading quote of each quote pair and the second quote from each pair of "doubled" quotes within quote pairs.)

F. Return everything in CVEC except quote characters or characters within quotes.

```
T←=\CVEC≠''''
(T∧¯1⌽T)/CVEC
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: In the last expression of the final illustration in the prior section, a shift-and-compare operation (T∧¯1⌽T) is performed to produce the effect of extending each 0-map to the right by one element. What other shift-and-compare operations have useful interpretations when applied on Boolean arrays? What are the interpretations?

TOPIC: Logical Shift-and-Compare (Map) Operations

In the following list of shift-and-compare operations, the catenate and drop technique (e.g. ¯1↓0,B) is used instead of the rotate technique (e.g. ¯1⌽B). The reason for this choice is that the rotate technique has the undesirable effect of "filling" the first element (or last element if 1⌽B) with the arbitrary value of the last (or first) element of the array, rather than with the 1 or 0 needed to make the comparison work in every case.

Also notice that the catenate is done before the drop (e.g. ¯1↓0,B) instead of the other way around (e.g. 0,¯1↓B) so that the expression will behave correctly when the argument is empty.

Expression	Interpretation
R≠¯1↓0,R	1-maps to leading 1-poles
R=¯1↓1,R	0-maps to leading 0-poles
R>¯1↓0,R	1-maps to first 1 bits
R≥¯1↓1,R	0-maps to first 0 bits
R∨¯1↓0,R	extend 1-maps to right by 1 (shorten 0-maps from left by 1)
R∧¯1↓1,R	extend 0-maps to right by 1 (shorten 1-maps from left by 1)

Since each shift-and-compare operation transforms a map, these operations are sometimes called "map" operations.

Let us solve some problems using these operations.

A. Given a character vector SENTENCE, return the lengths of the words in it. A word is any set of contiguous letters.

```
LETTERS←'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef...xyz'
MAPS←(SENTENCE∊LETTERS),0          (0 to insure last element
                                    not a letter)
POLES←MAPS≠¯1↓0,MAPS               (1-maps to leading 1-poles)
INDS←POLES/⍳⍴POLES                 (indices of poles)
INDS←(((⍴INDS)÷2),2)⍴INDS          (reshape to 2 column matrix)
-/⌽INDS                            (subtract in pairs)
```

B. Determine the indices of the first letter of each word.

```
MAPS←SENTENCE∊LETTERS
(MAPS>¯1↓0,MAPS)/⍳⍴MAPS        (1-maps to first 1 bits)
```

C. Given a numeric vector BAL, how many times do the values go from positive to negative or vice-versa?

```
MAPS←BAL<0
+/1↓MAPS≠¯1↓0,MAPS             (1-maps to leading 1-poles)
```

D. Delete the extraneous (leading, trailing or redundant) blanks from a character vector CVEC.

```
NB←CVEC≠' '
CVEC←(NB∨¯1↓0,NB)/CVEC         (extend 1-maps to right by 1)
CVEC←(-' '=¯1↑CVEC)↓CVEC       (drop last element if blank)
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Some applications, such as text processing, are well-suited to the Boolean techniques discussed in the prior sections. These techniques allow you to analyze an array without iterating by character. For example, (+/∧\R=' ') tells you how many leading blanks are in the array R. At times, you will want to work with an array which is the catenation of several arrays (e.g. the sentences in a paragraph). Then you may want to apply the Boolean operations on each respective "subarray" (e.g. the number of leading blanks in each sentence of a paragraph). Design and implement a set of Boolean utility functions which will perform the operations described in the prior sections, for each of the subarrays in a specified array.

TOPIC: Logical Partition Operations

We will use the term "partition" to refer to each subarray of an array. For example, we may view the following character vector as consisting of 3 partitions (sentences):

```
'HELLO.  THIS IS A SAMPLE.  WHAT DO YOU THINK?'
```

There are a number of different methods which may be used to define where in this character vector each of the partitions begins and
~~~~

ends. For example, we could specify the indices of the starting elements and of the ending elements for each partition; or we could specify the starting indices and the lengths of each partition.

Since we will be using Boolean techniques discussed above, we will define the locations of the partitions by specifying a Boolean vector which has as many elements as the character vector and which has 1s in the indices which correspond to the first element of each partition. Such a Boolean vector will be called a "partition vector".

A partition vector for the character vector above is defined below (with spaces removed from the display of the partition vector for clarity):

```
CVEC←'HELLO.  THIS IS A SAMPLE.  WHAT DO YOU THINK?'
PV  ← 10000010000000000000000010000000000000000000
```

We desire a set of functions which will perform each of the Boolean operations described in prior sections, but will do so independently on each of the partitions of a specified array. Since the functions require the knowledge of how the array is partitioned, the partition vector must be an argument to each function. For example, if +/ and ∧\ were defined to permit a partition vector left argument, we could determine the number of leading blanks in each sentence of CVEC with the following expression:

```
PV+/PV∧\CVEC=' '
```

The result would contain one element per partition (e.g. 0 2 2).

Since +/ and ∧\ do not, in fact, accept partition vector left arguments, we will design our own set of "partition functions".

Non-Partition Expression	Partition (PV) Expression
+/R	PV pPLUSRED R
∧/R	PV pANDRED R
∨/R	PV pORRED R
∧\R	PV pANDSCAN R
∨\R	PV pORSCAN R
<\R	PV pLTSCAN R
≤\R	PV pLESCAN R
≠\R	PV pNESCAN R
=\R	PV pEQSCAN R
R≠¯1↓0,R	PV pNEMAP R
R=¯1↓1,R	PV pEQMAP R
R>¯1↓0,R	PV pGTMAP R
R≥¯1↓1,R	PV pGEMAP R
R∨¯1↓0,R	PV pORMAP R
R∧¯1↓1,R	PV pANDMAP R

(Notice that the logical scalar functions (e.g. L>R) are not included here since the scalar functions work correctly whether or not their arguments are partitioned.)

The number of leading blanks in each sentence in CVEC is:

```
PV pPLUSRED PV pANDSCAN CVEC=' '
```

The use of these functions will be further illustrated in the next section.

The definitions of these functions make heavy use of the Boolean techniques described in prior sections. You may want to study the definitions of some of the functions to become better acquainted with actual applications of Boolean techniques.

(The algorithms underlying many of these functions were conceived by Robert A. Smith of STSC and are introduced in the publication, Boolean Functions and Techniques, 1975, Scientific Time Sharing Corporation.)

[WSID: BOOLEAN]

```
    ∇ R←P pPLUSRED B;T
[1]   ⍝ Returns +/S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝
[6]   ⍝ Works on Boolean B only.  For numeric B:
[7]   ⍝    R←(1⌽P)/+\B
[8]   ⍝    R←R-¯1↓0,R
[9]   ⍝
[10]  ⍝ Compress partition vec for just 1 bits and
[11]  ⍝ leading bits:
[12]   T←⍴R←(P∨B)/P
[13]  ⍝ Convert to indices:
[14]   R←R/⍳T
[15]  ⍝ Lengths of compressed partitions:
[16]   R←(1↓R,⎕IO+T)-R
[17]  ⍝ Deduct 1 for partitions with leading 0 bit:
[18]   R←R-~P/B
    ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pANDRED B
[1]   ⍝ Returns ∧/S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ Compress partition vec for just 0 bits and
[6]   ⍝ leading bits:
[7]    R←(P≥B)/P
[8]   ⍝ Which partitions have no 0s beyond leading bit?
[9]    R←R/1⌽R
[10]  ⍝ ...and have a leading 1 bit:
[11]   R←R∧P/B
     ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pORRED B
[1]   ⍝ Returns ∨/S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ Compress partition vec for just 1 bits and
[6]   ⍝ leading bits:
[7]    R←(P∨B)/P
[8]   ⍝ Which partitions have no 1s beyond leading bit?
[9]    R←R/1⌽R
[10]  ⍝ Leading 1 bit or any trailing 1s:
[11]   R←R≤P/B
     ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pANDSCAN B;T
[1]   ⍝ Returns ∧\S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.  Uses fact
[5]   ⍝ that ∧\A ←→ ~∨\~A.
[6]   ⍝ Consider just 0 bits and leading bits:
[7]    T←P≥B
[8]   ⍝ All 1s except leading 1 bits (as 0s):
[9]    R←~T/B
[10]  ⍝ 1-maps to 1-poles and expand:
[11]   R←T\R≠¯1↓0,R
[12]  ⍝ 1-poles to 1-maps and toggle:
[13]   R←~≠\R
     ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pORSCAN B;T
      ⍝ Returns ∨\S for each partition S of B,
      ⍝ where P is the corresponding Boolean
      ⍝ partition vector whose 1s mark the first
      ⍝ element of each partition.
      ⍝ Consider just 1 bits and leading bits:
       T←P∨B
      ⍝ All 1s except leading 0 bits:
       R←T/B
      ⍝ 1-maps to 1-poles and expand:
       R←T\R≠¯1↓0,R
      ⍝ 1-poles to 1-maps:
       R←≠\R
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pLTSCAN B;T
      ⍝ Returns <\S for each partition S of B,
      ⍝ where P is the corresponding Boolean
      ⍝ partition vector whose 1s mark the first
      ⍝ element of each partition.
      ⍝ Consider just 1 bits and leading bits:
       T←P∨B
      ⍝ All 1s except leading 0 bits:
       R←T/B
      ⍝ 1-maps to leading 1 bits and expand:
       R←T\R>¯1↓0,R
      ⍝ Set leading 1 bits to 1:
       R←R∨P∧B
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pLESCAN B;T
      ⍝ Returns ≤\S for each partition S of B,
      ⍝ where P is the corresponding Boolean
      ⍝ partition vector whose 1s mark the first
      ⍝ element of each partition.
      ⍝ Consider just 0 bits and leading bits:
       T←P≥B
      ⍝ All 1s except leading 1 bits (as 0s):
       R←~T/B
      ⍝ 1-maps to leading 1 bits and expand:
       R←T\R>¯1↓0,R
      ⍝ Set leading 0 bits to 0; subtract other
      ⍝ leading 1 bits:
       R←R<B≥P
    ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pNESCAN B
[1]   ⍝ Returns ≠\S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 1-poles to 1-maps, shift right, mark overlap
[6]   ⍝ leading bits:
[7]    R←P/¯1↓0,≠\B
[8]   ⍝ 1-maps to 1-poles, expand, marking leading
[9]   ⍝ bits to toggle:
[10]   R←P\R≠¯1↓0,R
[11]  ⍝ Toggle selected leading bits, 1-poles to 1-maps:
[12]   R←≠\B≠R
     ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pEQSCAN B
[1]   ⍝ Returns =\S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 0-poles to 0-maps, shift right, mark overlap
[6]   ⍝ leading bits, toggle:
[7]    R←∼P/¯1↓1,=\B
[8]   ⍝ 1-maps to 1-poles, expand, marking leading
[9]   ⍝ bits to toggle:
[10]   R←P\R≠¯1↓0,R
[11]  ⍝ Toggle selected leading bits, 0-poles to 0-maps:
[12]   R←=\B≠R
     ∇
```

[WSID: BOOLEAN]

```
     ∇ R←P pNEMAP B
[1]   ⍝ Returns S≠¯1↓0,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 1-maps to leading 1-poles:
[6]    R←B≠¯1↓0,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]    R←R≠P∧B≠R
     ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pEQMAP B
[1]   ⍝ Returns S=¯1↓1,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 0-maps to leading 0-poles:
[6]     R←B=¯1↓1,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]     R←R≠P∧B≠R
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pGTMAP B
[1]   ⍝ Returns S>¯1↓0,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 1-maps to first 1 bits:
[6]     R←B>¯1↓0,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]     R←R≠P∧B≠R
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pGEMAP B
[1]   ⍝ Returns S≥¯1↓1,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ 0-maps to first 0 bits:
[6]     R←B≥¯1↓1,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]     R←R≠P∧B≠R
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pORMAP B
[1]   ⍝ Returns S∨¯1↓0,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ Extend 1-maps to right by 1:
[6]     R←B∨¯1↓0,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]     R←R≠P∧B≠R
    ∇
```

[WSID: BOOLEAN]

```
    ∇ R←P pANDMAP B
[1]   ⍝ Returns S∧¯1↓1,S for each partition S of B,
[2]   ⍝ where P is the corresponding Boolean
[3]   ⍝ partition vector whose 1s mark the first
[4]   ⍝ element of each partition.
[5]   ⍝ Extend 0-maps to right by 1:
[6]    R←B∧¯1↓1,B
[7]   ⍝ Toggle leading bits which are not correct:
[8]    R←R≠P∧B≠R
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Write a monadic function RELABEL which will locate and modify all of the line labels (and references to those labels) in any specified function (whose character vector visual representation is provided as the argument to RELABEL) such that the labels become L1, L2, L3,... The result of RELABEL is the modified visual representation.

TOPIC: An Illustration of Boolean Techniques

The visual representation of a function is a character vector which, when displayed, looks just like the display produced by the command ∇FNNAME[⎕]∇. In order to be a character vector, not matrix, which displays on several lines, the visual representation necessarily contains "newline" (carriage return) characters. For example, suppose VR is the visual representation of the simple AVG function:

```
      ρVR
36
      VR
    ∇ R←AVG V
[1]  R←(+/V)÷ρV
    ∇
```

To get a more revealing view of VR and why it has the shape displayed above, we can replace all newline characters by the "@" symbol, and all blank characters by the "_" symbol. Suppose NL is a scalar newline character.
~~~~

```
              T←VR
              T[(T=NL)/⍳⍴T]←'@'
              T[(T=' ')/⍳⍴T]←'_'
              T
          ____∇_R←AVG_V@[1]__R←(+/V)÷⍴V@____∇@
```

The visual representation of a function may be generated by using the ⎕VR system function in APL★PLUS, or 1 ⎕FD in SHARP APL, or the CR∆VR function (and ⎕CR) developed in the Workspace Design and Documentation chapter. The newline character may be generated by using ⎕TCNL in APL★PLUS, ⎕TC[2] (origin 1) in APL2 or by selecting the newline character from ⎕AV (the index of the element depends upon the APL implementation; it is 157 in origin 1 for SHARP APL).

Let us define the header of the desired RELABEL function:

```
            ∇ NEWVR←RELABEL VR
```

We will find it convenient to view the visual representation as a partitioned array, where each line of the function is a partition. We will construct a partition vector whose 1s flag the first character on each line. Since every line is preceded by a newline, except the header line, and since the last character in the visual representation is a newline, we may construct the partition vector as follows:

```
        [1]  NL←¯1⌽VR=⎕TCNL
```

(Use ⎕TC[2] in APL2 or ⎕AV[157] in SHARP APL instead of ⎕TCNL.)

To solve this task, we must locate all identifiers, including labels, referred to in the visual representation. Identifiers are consecutive strings of alphanumeric characters whose first letter is alphabetic. However, we must be careful to ignore all such strings located within comments or within quotes, since such strings are probably words and not identifiers. Further, we cannot simply ignore everything beyond the first comment symbol (⍝) on each line. The first comment symbol may be located within quotes and so will not actually represent the beginning of a comment. (Note: many APL implementations now permit end-of-line comments, in addition to full-line comments, and so we cannot assume that the comment symbol will appear in a specific position on the line.)

Our first step is to find and ignore all characters within quotes (even quotes within comments). Then, the comment symbols not within quotes begin genuine comments, so we can use them to find and ignore all characters within comments. Finally, by ignoring all characters either within quotes or within comments, we can proceed with our job of finding identifiers.

As we saw in a prior section, =\ can be used to ignore characters within quotes, using an expression like:

```
        (=\CVEC≠'''')/CVEC
```

However, this expression cannot be blindly applied to VR since a comment may contain an odd quote character (e.g. ⍝ YOU DON'T SAY!) which will cause mismatching of quote characters. Instead, we must perform the =\ (0-poles to 0-maps) on each partition (function line) of VR individually.

```
[2]  NQ←VR≠''''
[3]  NCCON←NL pEQSCAN NQ
```

NQ flags the non-quote characters. NCCON is a map vector of the characters which are not within quote pairs (i.e. character constants) within each function line. Note that the closing quotes of each quote pair are included in the map while the opening quote is not.

We can now use NCCON to help us locate all valid comment symbols.

```
[4]  NC←NCCON⍲VR='⍝'
```

NC flags the non-comment characters (0s flag the valid comment symbols). For each line of the visual representation, we wish to propogate the 0 which flags the comment symbol in NC so that all following characters are flagged with 0s (to subsequently ignore them). For a single partition, we can use ^\. For all partitions, we must use pANDSCAN:

```
[5]  NCMT←NL pANDSCAN NC
```

NCMT is a map vector of the characters which do not follow a valid comment symbol. Note that the comment symbols themselves are not included in the map.

It is now a simple matter to construct a map vector, PARSE, of the characters which are not within quote pairs and which do not follow a comment symbol within each function line:

```
[6] PARSE←NCMT^NCCON
```

Our next thrust is to look at the characters flagged by PARSE and to construct a pole vector whose poles (pairs of 1s) flag the identifiers. First, flag the digit characters, the letters, the alphanumeric characters and blanks:

```
[7]  NUM←PARSE^VR∊'0123456789'
[8]  ALP←PARSE^VR∊'ABCD...XYZ∆abcd...xyz⍙'
[9]  AN←NUM∨ALP
[10] BL←PARSE^VR=' '
```

Then, construct a pole vector of the alphanumeric maps.

```
[11] PAN←AN≠¯1↓0,AN
```

Some of these pole pairs flag identifiers (e.g. COUNT or AMT85 or J) while others do not (e.g. 58 or 1E6). We must "turn off" the pole pairs whose first pole does not correspond to an alphabetic character. The technique used to do this follows:

```
[12] T←PAN/ALP
[13] PID←PAN\T∨¯1⌽T
```

Notice that the compression (/) places the pole values next to each other, the map operation extends the value of the first pole (1 or 0) into the second pole, and the expansion (\) replaces the poles (some 0s now) into their original positions. PID is a pole vector which flags identifiers.

Since some identifiers begin with the '⎕' symbol (e.g. ⎕IO or ⎕PP or ⎕EX), we must adjust the poles in PID to include the '⎕'.

```
[14] T←1⌽PID
[15] T←T\'⎕'=T/VR
[16] PID←T∨PID>¯1⌽T
```

Now that we have located all identifiers within the visual representation, we must locate the labels. Labels are identifiers which immediately follow the bracketed function line number (ignoring spaces) and which immediately precede a colon (:).

Let us find the line numbers. Construct a pole vector of the numeric digit maps.

```
[17] PNUM←NUM≠¯1↓0,NUM
```

If the first pole in the PNUM pole pairs is exactly 2 characters after a newline character, that pair of poles identifies a line number. Flag the character following the "]" character after the line number at the beginning of each line.

```
[18] START←¯1⌽PNUM\¯1⌽PNUM/¯1⌽NL
```

Notice that / and \ are again used to place the pole values next to each other. In this case, the rotate operation changes 1 0 poles to 0 1 poles so that the last pole of the numeric pole vector (corresponding to the "]") is flagged.

The next step is to move the bits in START to the right so that they correspond to the first nonblank after the bracketed line number. Construct a pole vector of the blank maps.

```
[19] PBL←BL≠¯1↓0,BL
```

Flag the first nonblank character in each line:

```
[20] START←(START>BL)∨PBL\¯1⌽PBL/START
```

The (START>BL) term is included for lines which do not have any blanks between the bracketed line number and the next nonblank character.

By matching up START and PID, we can construct a pole vector of identifiers which begin with the first nonblank character of a function line:

```
[21] T←PID/START
[22] PSID←PID\T∨¯1⌽T
```

Construct a pole vector of labels (i.e. identifiers at the beginnings of function lines which are followed by a colon):

```
[23] T←':'=PSID/VR
[24] PLAB←PSID\T∨1⌽T
```

Using the pole vector, determine the starting and ending (plus 1) indices of the identifiers in the function (as a 2 column matrix, one row per identifier):

```
[25] IND←PID/⍳⍴PID
[26] NID←(⍴IND)÷2
[27] IND←(NID,2)⍴IND
```

Determine the length of each identifier name:

```
[28] IDSTART←IND[;⎕IO]
[29] IDLEN←IND[;1+⎕IO]-IDSTART
```

From here on, Boolean techniques are not required. The techniques used are discussed in other chapters, most notably the Positioning Character Data chapter and the Sorting and Searching chapter.

The RELABEL function is presented below in its entirety. The function uses the Boolean techniques described above. However, it has been modified in a number of ways. The logical partition functions (e.g. pANDSCAN) have been replaced by the equivalent logic so that RELABEL does not require their presence. The variables have been localized. RELABEL also has a left argument: a character matrix or vector (blank-delimited) of the names of the labels which are not to be renamed. Provide an empty character vector if all labels are to be renamed. For example, to relabel the function MODEL:

```
⎕DEF '' RELABEL ⎕VR 'MODEL'                 (on APL*PLUS)
3 ⎕FD '' RELABEL 1 ⎕FD 'MODEL'              (on SHARP APL)
⎕FX VR∆CR '' RELABEL CR∆VR ⎕CR 'MODEL'      (otherwise)
```

```
                                        [WSID: FNIDS]
    ∇ R←KEEP RELABEL VR;ALP;AN;BL;COLS;FOUND;IDLEN;IDS;
      IDSTART;IND;KLABS;KLEN;KSTART;LABS;NB;NC;NCCON;NCMT;
      NEW;NID;NKEEP;NL;NLAB;NLEN;NQ;NSTART;NUM;PAN;PARSE;PBL
      ;PID;PLAB;PNUM;PSID;S;START;T;⎕IO
[1]   ⍝ Modifies the vector representation (VR) of a
[2]   ⍝ function such that its labels and references to
[3]   ⍝ same are changed to L1, L2, L3,... for all labels
[4]   ⍝ but those specified in KEEP, a character matrix
[5]   ⍝ or vector (blank delimited) of label names.
[6]   ⍝ Requires subfunction: CMIOTA.
[7]    ⎕IO←0
[8]   ⍝ Flag newline chars (Boolean partition vector):
[9]    NL←¯1⌽VR=⎕TCNL ⍝ APL*PLUS
[10]  ⍝ NL←¯1⌽VR=⎕TC[1] ⍝ APL2
[11]  ⍝ NL←¯1⌽VR=⎕AV[156] ⍝ SHARP APL
[12]  ⍝ Flag nonquotes:
[13]   NQ←VR≠''''
[14]  ⍝ Map of chars not in quote pairs (i.e. char constants)
[15]  ⍝ within each fn line (NCCON←NL pEQSCAN NQ):
[16]   NCCON←=\NQ≠NL\T≠¯1↓0,T←~NL/=\¯1↓1,NQ
[17]   NQ←0
[18]  ⍝ Flag non-⍝ chars (includes ⍝s in quotes):
[19]   NC←NCCON⍲VR='⍝'
[20]  ⍝ Map of chars which do not follow a ⍝ (ignoring ⍝s
[21]  ⍝ within quotes) within each fn line. ⍝s are flagged 0.
[22]  ⍝ (NCMT←NL pANDSCAN NC):
[23]   S←NL≥NC
[24]   NCMT←~≠\S\T≠¯1↓0,T←~S/NC
[25]   S←T←NC←0
[26]  ⍝ Map of chars which are not included within ⍝s or '':
[27]   PARSE←NCMT∧NCCON
[28]   NCCON←NCMT←0
[29]  ⍝ Flag digits, letters, blanks:
[30]   NUM←PARSE∧VR∊'0123456789'
[31]   ALP←PARSE∧VR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklm
       nopqrstuvwxyz⍙'
[32]   BL←PARSE∧VR=' '
[33]   PARSE←0
[34]  ⍝ Flag alphanumeric chars:
[35]   AN←NUM∨ALP
[36]  ⍝ Pole vec of contiguous digits:
[37]   PNUM←NUM≠¯1↓0,NUM
[38]   NUM←0
[39]  ⍝ Pole vec of contiguous digits/letters:
[40]   PAN←AN≠¯1↓0,AN
[41]   AN←0
[42]  ⍝ Pole vec of identifiers:
[43]   PID←PAN\T∨¯1⌽T←PAN/ALP
[44]   ALP←PAN←0
[45]  ⍝ Flag '⎕' before identifiers (⎕names):
[46]   T←1⌽PID
[47]   T←T\'⎕'=T/VR
```

```
    ∇ RELABEL (continued)
[48] ⍝ Shift leading poles of ⎕names to include ⎕:
[49]  PID←T∨PID>¯1⌽T
[50]  T←0
[51] ⍝ Flag char following ] after line no.:
[52]  START←¯1⌽PNUM\¯1⌽PNUM/¯1⌽NL
[53]  NL←PNUM←0
[54] ⍝ Pole vec of contiguous blanks:
[55]  PBL←BL≠¯1↓0,BL
[56] ⍝ Flag 1st nonblank char in each line:
[57]  START←(START>BL)∨PBL\¯1⌽PBL/START
[58]  BL←PBL←0
[59] ⍝ Pole vec of identifiers at start of line:
[60]  PSID←PID\T∨¯1⌽T←PID/START
[61]  START←0
[62] ⍝ Pole vec of labels:
[63]  PLAB←PSID\T∨1⌽T←':'=PSID/VR
[64]  PSID←0
[65] ⍝ Start and end (+1) indices of identifiers:
[66]  IND←PID/⍳⍴PID
[67] ⍝ No. of identifiers:
[68]  NID←(⍴IND)÷2
[69]  IND←(NID,2)⍴IND
[70] ⍝ Start indices of identifiers:
[71]  IDSTART←IND[;0]
[72] ⍝ Lengths of identifiers:
[73]  IDLEN←IND[;1]-IDSTART
[74]  IND←0
[75] ⍝ Map vec of nonblanks in labels to keep:
[76]  NB←' '≠KEEP←,' ',KEEP
[77] ⍝ Start indices in KEEP of identifiers:
[78]  NKEEP←⍴KSTART←(NB>¯1⌽NB)/⍳⍴NB
[79] ⍝ Lengths of KEEP identifiers:
[80]  KLEN←(1+(NB>1⌽NB)/⍳⍴NB)-KSTART
[81] ⍝ Length of longest identifier:
[82]  COLS←(⌈/KLEN)⌈⌈/IDLEN
[83] ⍝ Raveled blank matrix of identifier names:
[84]  IDS←(NID×COLS)⍴' '
[85] ⍝ Fill them in (T←MONIOTA IDLEN):
[86]  T←T+⍳⍴T←IDLEN/-¯1↓0,+\IDLEN
[87]  IDS[T+IDLEN/COLS×⍳NID]←VR[T+IDLEN/IDSTART]
[88] ⍝ Reshape to mat of identifiers:
[89]  IDS←(NID,COLS)⍴IDS
[90] ⍝ Mat of label names:
[91]  LABS←(((NID,2)⍴PID/PLAB)[;0])⌿IDS
[92]  PID←PLAB←0
[93] ⍝ Raveled blank matrix of labels to keep:
[94]  KLABS←(NKEEP×COLS)⍴' '
[95] ⍝ Fill in (T←MONIOTA KLEN):
[96]  T←T+⍳⍴T←KLEN/-¯1↓0,+\KLEN
[97]  KLABS[T+KLEN/COLS×⍳NKEEP]←KEEP[T+KLEN/KSTART]
[98] ⍝ Reshape to mat of labels to keep:
[99]  KLABS←(NKEEP,COLS)⍴KLABS
```

```
    ∇ RELABEL (continued)
[100] ⍝ Squeeze out labels in KEEP:
[101]  LABS←(NKEEP=KLABS CMIOTA LABS)⌿LABS
[102] ⍝ Row indices in LABS where rows of IDS are found:
[103]  IND←LABS CMIOTA IDS
[104] ⍝ No. of labels:
[105]  NLAB←1⍴⍴LABS
[106] ⍝ Flag identifiers which are labels:
[107]  FOUND←IND<NLAB
[108] ⍝ Start indices of label identifiers:
[109]  IDSTART←FOUND/IDSTART
[110] ⍝ Lengths of same:
[111]  IDLEN←FOUND/IDLEN
[112] ⍝ Indices into LABS of same:
[113]  IND←FOUND/IND
[114] ⍝ Building new labels: 'L1L2L3...':
[115]  NEW←' ',⍕1+⍳NLAB
[116] ⍝ Start indices in NEW of new labels:
[117]  NSTART←(NEW=' ')/⍳⍴NEW
[118]  NEW[NSTART]←'L'
[119] ⍝ Lengths of new labels:
[120]  NLEN←(1↓NSTART,⍴NEW)-NSTART
[121] ⍝ Lengths replicated for all label identifiers:
[122]  NLEN←NLEN[IND]
[123] ⍝ Start indices replicated as well:
[124]  NSTART←NSTART[IND]
[125] ⍝ Initialize replication vec:
[126]  R←(⍴VR)⍴1
[127] ⍝ Use 0s to squeeze out old identifiers
[128] ⍝ (T←MONIOTA IDLEN):
[129]  T←T+⍳⍴T←IDLEN/-¯1↓0,+\IDLEN
[130]  R[T+IDLEN/IDSTART]←0
[131] ⍝ Insert lengths to expand for new identifiers:
[132]  R[IDSTART]←NLEN
[133] ⍝ Squeeze/expand vis rep as needed:
[134]  R←R/VR
[135] ⍝ Adjust for new lengths:
[136]  IDSTART←IDSTART++\¯1↓0,NLEN-IDLEN
[137] ⍝ Insert new labels (T←MONIOTA NLEN):
[138]  T←T+⍳⍴T←NLEN/-¯1↓0,+\NLEN
[139]  R[T+NLEN/IDSTART]←NEW[T+NLEN/NSTART]
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEMS: (Solutions on pages 449 to 458)

1. What expression will tell you whether all of the elements of the numeric vector NVEC are integers?

2. Delete the trailing blanks from the character vector CVEC.

3. Left justify the rows of the character matrix NAMES (i.e. shift each row left until its first character is a nonblank).

4. How many numbers (set of contiguous digit characters) are there in the character vector INPUT?

5. Write the function TIME∆DEFINE as described in the Computer Efficiency Considerations chapter.

6. Write one or more of the functions IDENTIFY, LOCALIZE and UNCOMMENT as described in the Workspace Design and Documentation chapter.

Chapter 16

IRREGULAR ARRAYS

APL derives much of its power from its conciseness and consistency. Unfortunately, the real world is not nearly so concise and consistent. While APL sees the world as a set of rectangular arrays of data, the world is nonrectangular by nature.

In this chapter, we deal with irregular (nonrectangular) arrays. We will present a typical problem which involves irregular arrays and will attempt to perform a variety of tasks on these arrays. We will examine alternative methods available in APL for performing these tasks on the irregular arrays.

As an illustration of irregular arrays, suppose you wish to keep track of customer information for a business you operate. The following table shows some of the information.

CUSTOMER INFORMATION

			UNPAID BILLS		
ID NO.	TERR NO.	NAME	DATE	INV. NO.	AMOUNT
3	5	ACME CORPORATION	3/15/86	372	586.25
			4/10/86	395	406.15
			4/25/86	407	100.00
65	3	FASTENERS INC.	4/20/86	405	802.16
74	5	KLINGLEY & SONS, INC.			
89	2	GHR CORP.	2/12/86	350	5.25
			4/10/86	396	59.60
:	:	:	:	:	:

How would you store this information in APL variables?

The ID numbers can be assigned as a vector with one element per customer:

```
ID←3 65 74 89...
```

Likewise for territory numbers:

```
TERR←5 3 5 2...
```

The customer names may be stored as a character matrix with one row per customer and as many columns as the widest customer name (or an arbitrary maximum number of columns). Assuming 500 customers and a width of 25 character columns,

```
NAME←500 25⍴(25↑'ACME CORPORATION'),(25↑'FASTENERS INC.')...
```

While this approach to storing the customer names in a matrix is tolerable, it has some disadvantages. First, if a customer's name is longer than the allowable width (here 25), it must be abbreviated or truncated. Second, short customer names must be padded with blanks to the maximum width, implying a waste of storage. Third, when extracting a customer's name from this character matrix, as for a form letter, the trailing blanks may need to be deleted, requiring more complex programming and more processing time.

If these disadvantages are not major, you will do well to store the names in a character matrix and tolerate the disadvantages. If the disadvantages are great enough to be unworkable, you must use some other method to work with this irregular information.

Conceptually, the customer names define an irregular array, a "vector" of character vectors. Each "element" of the "vector" is itself the character vector name for a single customer. In this chapter, we will refer to this type of array as a "nest of character vectors" or simply a "character nest". We will refer to the character vector "elements" as "items".

Likewise, the unpaid bills information cannot be fit neatly into a conventional APL array. If we assign the values to a 3 column numeric matrix, there will not be one row per customer (as there are in a character matrix of customer names) and so we must maintain additional information which tells us which rows belong to which customer.

Conceptually, the unpaid bills information defines an irregular array, a "vector" of 3 column numeric matrices. Each "element" of the "vector" is itself the 3 column numeric matrix for a single customer. In this chapter we will refer to this type of array as a "nest of numeric matrices" or simply a "matrix nest". We will refer to the numeric matrix "elements" as "items".

In this chapter, we will present 4 different approaches to the problem of working with irregular arrays:

1. APL2 nested arrays
2. APL*PLUS nested arrays
3. SHARP APL nested arrays
4. Conventional APL arrays

You will notice that the APL2 and APL*PLUS implementations of nested arrays are similar. The primary differences are in the function symbols chosen for the particular nested array operations.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Construct a character nest of customer names and a matrix nest of unpaid bills information.

TOPIC: Constructing Irregular Arrays

In APL2, APL*PLUS and SHARP APL, character nests and matrix nests may be stored as nested arrays. In particular, they are stored as vectors of items, where the items are character vectors (character nests) or 3 column numeric matrices (matrix nests). The conventions for constructing these arrays are different for each of the different implementations of APL.

In APL2 and APL*PLUS, the arrays may be constructed by "vector notation" or "strand notation":

```
NAME←'ACME CORPORATION' 'FASTENERS INC.' ...
UBILLS←(3 3⍴31586 372 58625 41086 395 40615 42586 407 10000)
      (1 3⍴42086 405 80216) (0 3⍴0) ...
```

Alternately, the arrays may be initialized to have the correct number of items, after which the items are individually index assigned (assume 500 customers):

```
NAME←UBILL←500⍴0
NAME[1]←⊂'ACME CORPORATION'
NAME[2]←⊂'FASTENERS INC.'
   :
UBILLS[1]←⊂3 3⍴31586 372 58625 41086 395 40615 42586 407
          10000
UBILLS[2]←⊂1 3⍴42086 405 80216
UBILLS[3]←⊂0 3⍴0
   :
```

The monadic enclose (⊂) function converts its right argument into a "nested scalar". In APL2, a tidier notation may be employed to perform the index assignment:
~~~~

```
NAME←UBILLS←500⍴0
(1⊃NAME)←'ACME CORPORATION'
(2⊃NAME)←'FASTENERS INC.'
   :
(1⊃UBILLS)←3 3⍴31586 372 58625 41086 395 40615 42586 407
           10000
(2⊃UBILLS)←1 3⍴42086 405 80216
(3⊃UBILLS)←0 3⍴0
   :
```

In SHARP APL, the arrays may be constructed by repeatedly applying the link (⊃) function:

```
NAME←'ACME CORPORATION'⊃'FASTENERS INC.'⊃ ...
UBILLS←(3 3⍴31586 372 ... 407)⊃(1 3⍴42086 405 80216)⊃
       (0 3⍴0)⊃ ...
```

Alternately, the arrays may be initialized to have the correct number of nested items, after which the items are individually index assigned:

```
NAME←UBILLS←500⍴<0
NAME[1]←<'ACME CORPORATION'
NAME[2]←<'FASTENERS INC.'
   :
UBILLS[1]←<3 3⍴31586 372 58625 41086 395 40615 42586 407
           10000
UBILLS[2]←<1 3⍴42086 405 80216
UBILLS[3]←<0 3⍴0
   :
```

Notice that the less than (<) symbol is used to perform the enclose function.

If you already have your customer names in the form of a character matrix (CMAT) with one row per name, you may convert the matrix directly into a character nest by one of the following:

```
NAME←⊂[2]CMAT                (APL2)
NAME←↓[2]CMAT                (APL*PLUS)
NAME←<⍤1 CMAT                (SHARP APL)
```

Each of the items of the resulting nest will be a character vector of the same length, namely the number of columns in the character matrix (CMAT). To delete the trailing blanks from each character vector item of a character nest, you must do the following:

```
NAME←DTB¨ NAME               (APL2, APL*PLUS)
NAME←(+/∧\' '≠⌽CMAT)⍴¨>NAME  (SHARP APL)
```

where DTB (delete trailing blanks) is a user-defined monadic function which deletes the trailing blanks from its character vector right argument. For example:

```
        ∇ R←DTB CVEC
   [1]  R←(+/∧\' '≠⌽CVEC)⍴CVEC
        ∇
```

When generating reports which include the customer names, you may need to convert the character nest back into a character matrix with a specified number of columns. Each name must be left-justified in a single row of the matrix. Suppose you want a 25 column matrix. You may do this as follows:

```
       CMAT←⊃[2]25↑¨NAME             (APL2)
  or:  CMAT←25↑[2]⊃[2]NAME           (APL2)
       CMAT←↑[1.5]25↑¨NAME           (APL*PLUS)
       CMAT←25↑ö>NAME                (SHARP APL)
```

The second APL2 algorithm is more efficient that the first but will require more workspace storage than the first if any name is longer than 25 characters. If much longer, a WS FULL may occur.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: How can you work with irregular arrays if your implementation of APL does not support nested arrays?

TOPIC: Emulating Nested Arrays on Non-Nested Systems

Skip this section if your implementation of APL has nested arrays.

To solve the tasks above using conventional APL, we must devise a scheme wherein a character nest or a matrix nest may be stored as one or more conventional APL arrays. We will deal first with the character nest problem.

One approach is to catenate the names together, preceding each name by a delimiter character. For example:

```
NAME←'⍟ACME CORPORATION⍟FASTENERS INC.⍟KLINGLEY...'
```

(In SHARP APL, NAME can be converted into the nested array by the expression: ¯1ö< NAME).

Unfortunately, this scheme does not allow an efficient means to directly access the name for a specified customer since the array must first be searched for the occurrences of the delimiter character. For example, the name of the fifth customer may be extracted with the expression:
~~~~

```
1↓(5=+\NAME=1ρNAME)/NAME
```

which works but is inefficient.

If the delimiter character is not going to be used to locate the substrings of the character nest, you may omit them altogether. Let us catenate the names together as a single character vector:

```
NAME←'ACME CORPORATIONFASTENERS INC.KLINGLEY...'
```

We need a second array to tell us where each name starts. One possibility is a Boolean "partition" vector which has one element per element of NAME and whose 1s mark the corresponding first elements of each name:

```
NPV←1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0...
```

Together, these two arrays contain all the information you need to determine the name for a specified customer. For example, the name of the fifth customer may be extracted with the expression:

```
(5=+\NPV)/NAME
```

(In SHARP APL, NPV and NAME can be converted into the nested array by the expression: NPV 1⍤< NAME).

Unfortunately, the partition vector approach can be awkward to work with and has an inherent flaw: it will not handle an empty name. For example, if one of the customer names is unknown, you would naturally store it as an empty character vector. However, the partition vector requires each name to have at least one element (for the 1 in the partition vector).

Instead of constructing a Boolean partition vector to keep track of where the names start and end in the vector NAME, construct a vector of the lengths of each name (including 0s for empty names).

```
NLEN←16 14 21 9 15 0 11...
```

The sum of these elements should be the same as the number of elements in NAME, i.e. (+/NLEN)=ρNAME. Using NLEN, you may extract the name of the fifth customer with the expression:

```
NAME[(¯1↓0,+\NLEN)[5]+ιNLEN[5]]
```

This expression works properly even for empty customer names (i.e. for elements of NLEN which are 0).

Note that the (¯1↓0,+\NLEN) portion of the expression is merely computing the starting indices (i.e. the index before the first character) of the names. To improve the efficiency of the name extraction process, you may perform this operation once:

```
NSTART←¯1↓0,+\NLEN
```

after which the extraction process (for customer 5) becomes:

```
NAME[NSTART[5]+⍳NLEN[5]]
```

Unfortunately, the price you pay for the greater efficiency is greater complexity. You must maintain 3 arrays (NAME, NLEN, NSTART) when working with customer names. Any utility functions you write to work with this character nest (e.g. to emulate the capabilities illustrated above on nested array systems) must cope with at least these 3 arguments.

Utility function design will be greatly simplified if we can weld these three arrays into a single array. Unfortunately, numbers (NLEN and NSTART) do not cohabitate well with characters (NAME) in the arrays of conventional APL systems. (Note: the APL2 and APL*PLUS nested array implementations allow "heterogeneous" arrays which contain both character and numeric elements.) You must first convert them to a common datatype. Suppose we convert the character vector NAME into a numeric vector (e.g. ⎕AV⍳NAME) and then catenate the three arrays together, along with the number of customers:

```
CNEST←(⍴NLEN),NLEN,START,⎕AV⍳NAME
```

This array is a single object which contains all of the customer name information. Since it is a single array, you may design utility functions which take CNEST as one argument and the other parameters of the problem as the other argument. For example, you can write a function EXTRACT whose left argument is this "character nest" array and whose right argument is the index of the nest for which the character vector name is to be returned:

```
      CNEST EXTRACT 3
KLINGLEY & SONS, INC.
```

The definition of the EXTRACT function is straightforward:

```
    ∇ R←CNEST EXTRACT I;N
[1]   N←1↑CNEST
[2]   R←⎕AV[CNEST[(CNEST[1+N+I]+1+N+N)+⍳CNEST[1+I]]]
    ∇
```

Unfortunately, this definition of a character nest is probably no improvement over padded character matrices. On an APL system in which small integers (i.e. 1 to 256) are stored with 4 bytes per element, each name in this character nest requires 8+4×C bytes, where C is the number of characters in the name. Since concern for storage is one of the reasons for exploring character nests, a better solution is needed.

A big improvement can be made if you can manage to squeeze more than one character into an integer. Since integers are stored in 2 bytes or 4 bytes, depending upon your implementation of APL, it is possible to translate 2 or 4 characters into a single integer. For example, in APL implementations which store integers in 4 bytes, the range of integers is ¯2,147,483,648 to 2,147,483,647. Beyond this range, integers are stored in 8 bytes. The procedure for converting 4 characters into one integer is to convert each character into an integer from 1 to 256 (by finding its index in the atomic vector, ⎕AV) and then pack these 4 numbers into a number between ¯2,147,483,648 and 2,147,483,647. Given a 4 element character vector CVEC4, a single integer may be produced as follows:

```
⎕IO←0                                (origin 0 is simpler)
INT←⌈¯2147483648+256⊥⎕AV⍳CVEC4
```

To convert the integer back to a 4 element character vector:

```
⎕IO←0
CVEC4←⎕AV[(4⍴256)⊤INT+2147483648]
```

Using such a packing and unpacking algorithm, each name in the character nest will require only 8+4×⌈C÷4 bytes, where C is the number of characters in the name. This is a big improvement in storage but comes at the expense of processing efficiency. The packing and unpacking takes time.

Some APL implementations provide primitive functions for converting between datatypes, e.g. for converting one 4-byte integer into 4 1-byte characters or vice versa. Such functions are extremely efficient since no modification is made to the internal representation of the data, only to the "header" of the variable which indicates its shape and datatype. Such implementations are ideal candidates for manipulations of character nests by conventional means (i.e. not nested arrays).

APL⋆PLUS PC is one such implementation. Integers are stored in 2 bytes (¯32768 to 32767) per element. The system function ⎕DR (data representation) converts between datatypes. For example, the expression

```
NVEC←163 ⎕DR CVEC
```

converts an N element character vector to an N÷2 element integer vector. The expression

```
CVEC←82 ⎕DR NVEC
```

converts an M element integer vector to an M×2 element character vector.

Here is one possible design for storing character nests on an APL⋆PLUS PC system (origin 1):

CNEST[1]	number of character vector substrings (N)
CNEST[2 to N+1]	index (⎕IO=0) into this vector of the starts of the substrings (S)
CNEST[S]	length of this character substring (L)
CNEST[S+1 to S+⌈L÷2]	integer representation (163 ⎕DR) of this character substring, padding with 1 space if necessary to produce an even number of characters

For example, the character nest which stores the 3 character substrings 'TREE', 'DOG' and 'ELEPHANT' is:

```
CNEST←3 4 7 10 4 21076 17733 3 20292 8263 8 19525
      20549 16712 21582
```

Note that:

```
      163 ⎕DR 'TREEDOG ELEPHANT'
21076 17733 20292 8263 19525 20549 16712 21582
```

Given this design, you may write utility functions which will allow you to work with character nests with the same (or greater) ease and efficiency as when working with nested array systems. For example, to construct the character nest from a character matrix of customer names:

```
NAME←CNEST CMAT
```

(The CNEST function and all other character nest utility functions suggested below are written for the APL*PLUS PC implementation of APL as exercises at the end of the chapter.)

To convert a character nest back into a character matrix with a specified number of columns (say 25), use the CN∆M function:

```
CMAT←25 CN∆M NAME
```

We will deal now with the matrix nest problem. In one regard, the problem is more difficult to work with than the character nest problem and in another regard it is simpler. It is more difficult because the items are matrices instead of vectors and it is simpler because the values are numeric, not character, and so do not need to be converted into numbers.

View the matrices as vectors (i.e. ravel them). The number of elements in each vector is a multiple of 3 since the matrices have 3 columns. Viewed in this way, the nest of numeric matrices becomes a nest of numeric vectors, which we will call a "numeric nest".

One possible design for storing numeric nests is patterned after the design proposed above for character nests (origin 1):

NNEST[1]	number of numeric vector segments (N)
NNEST[2 to N+1]	index (⎕IO=0) into this vector of the starts of the segments (S)
NNEST[S]	length of this numeric segment (L)
NNEST[S+1 to S+L]	numeric segment

For example, the numeric nest which stores the 3 segments (30 17 15), (25) and (82 93 95 98) is:

```
NNEST←3 4 8 10 3 30 17 15 1 25 4 82 93 95 98
```

Given this design, you may write utility functions for working with numeric nests. For example, to construct the numeric nest of unpaid bills information:

```
⍝ Number of elements per customer/segment:
 LEN←3×3 1 0 2...
⍝ Values:
 VAL←31586 372 58625 41086 395 40615 42586 407 10000 42086...
⍝ Construct numeric nest:
 UBILLS←LEN NNEST VAL
```

(The NNEST function and all other numeric nest utility functions suggested below are written as exercises at the end of the chapter.)

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Add to your customer information database a new customer, number 33, TOP DOG LTD., territory 4, which has no unpaid bills.

TOPIC: Catenating to Irregular Arrays

Use catenation to update the two regular arrays:

```
ID←ID,33
TERR←TERR,4
```
~~~~

Likewise, use catenate to update the two irregular arrays if they are stored as nested arrays:

```
NAME←NAME,⊂'TOP DOG LTD.'          (APL2, APL★PLUS)
NAME←NAME,<'TOP DOG LTD.'          (SHARP APL)
UBILLS←UBILLS,⊂0 3⍴0               (APL2, APL★PLUS)
UBILLS←UBILLS,<0 3⍴0               (SHARP APL)
```

The enclose function (⊂ or <) must be used to construct a nested scalar before it is catenated to the nested array. The enclose function is not needed when catenating more than one customer since vector notation (or the SHARP APL link function) performs the enclosing. For example:

```
NAME←NAME,'TOP DOG LTD.' 'BOTTOM CAT CORP.' (APL2, APL★PLUS)
NAME←NAME,'TOP DOG LTD.'⊃'BOTTOM CAT CORP.' (SHARP APL)
```

Using conventional APL, you may write a function CNCAT which will catenate a character nest to a character vector, "enclosing" the character vector and catenating it as a new item:

```
NAME←NAME CNCAT 'TOP DOG LTD.'
```

If both arguments to CNCAT are character vectors, the result is a 2 item character nest in which each item comes from a corresponding argument. This allows you to catenate more than one customer name at a time via the notation:

```
NAME←NAME CNCAT 'TOP DOG LTD.' CNCAT 'BOTTOM CAT CORP.'
```

Likewise, a function NNCAT may be written which will catenate two numeric nests. Unfortunately, the function cannot readily discern between an argument which is a numeric nest and an argument which is a simple numeric vector. (With CNCAT, a character nest is numeric and a character vector is character.) One way to solve this problem is to write 4 functions (NNCATSS, NNCATVS, NNCATSV, NNCATVV) which will respectively handle the 4 possible combinations of arguments. For example, NNCATVS will catenate a numeric nest ("vector") left argument to a simple numeric vector ("scalar") right argument:

```
UBILLS←UBILLS NNCATVS ,0 3⍴0
```

You may catenate the unpaid bills information for more than one customer at a time via the notation:

```
UBILLS←UBILLS NNCATVV (,0 3⍴0) NNCATSS ,1 3⍴40986 400 1000
```

Another approach is to provide the "scalar" (i.e. not numeric nest) argument as a matrix. Then, the function can tell its numeric nest (vector) arguments from its non-numeric-nest (matrix) arguments:

```
UBILLS←UBILLS NNCAT (0 3⍴0) NNCAT 1 3⍴40986 400 1000
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Return the name and unpaid bills information for the 5th customer, as a character vector and a 3 column numeric matrix. Return the names and unpaid bills information for the 2nd, 8th and 12th customers, as a 3 item character nest and a 3 item matrix nest, respectively.

TOPIC: Selecting from Irregular Arrays

Using nested arrays, the 5th name and matrix of unpaid bills information may be returned by selecting and disclosing (converting from a nested scalar to a simple array):

```
N←⊃NAME[5]             (APL2, APL⋆PLUS)
U←⊃UBILLS[5]

N←>NAME[5]             (SHARP APL)
U←>UBILLS[5]
```

The select-and-disclose operation is performed so frequently when working with nested arrays that a specific function ("pick") is available in APL2 and APL⋆PLUS to do just that:

```
N←5⊃NAME               (APL2, APL⋆PLUS)
U←5⊃UBILLS
```

To select several items from a nested vector to return a subset nested vector, you may use indexing directly:

```
N3←NAME[2 8 12]        (APL2, APL⋆PLUS, SHARP APL)
U3←UBILLS[2 8 12]
```

Using conventional APL, you may write functions CNIDX and NNIDX which will extract the desired information:

```
N←NAME CNIDX 5
U←UBILLS NNIDX 5
U←(((ρU)÷3),3)ρU

N3←NAME CNIDX 2 8 12
U3←UBILLS NNIDX 2 8 12
```

These functions return nests (character or numeric) if the right argument is a vector and simple arrays (character or numeric vector) if the right argument is a scalar. Since the unpaid bills information is stored as a vector, the third statement above is necessary to reshape the resulting vector into a 3 column matrix.
~~~~

PROBLEM: Replace the name and unpaid bills information of the 5th customer by the character vector 'NEW CORP.' and by the 3 column numeric matrix NEWBILLS. Replace the names and unpaid bills information of the 2nd, 8th and 12th customers by those of the 4th, 5th and 6th customers.

TOPIC: Replacing Items of Irregular Arrays

Using nested arrays, the 5th name and matrix of unpaid bills information may be replaced by any of the following:

```
NAME[5]←⊂'NEW CORP.'            (APL2, APL*PLUS)
UBILLS[5]←⊂NEWBILLS

(5⊃NAME)←'NEW CORP.'            (APL2)
(5⊃UBILLS)←NEWBILLS

NAME[5]←<'NEW CORP.'            (SHARP APL)
UBILLS[5]←<NEWBILLS
```

The 2nd, 8th and 12th customers may be modified directly by indexing and index assignment:

```
NAME[2 8 12]←NAME[4 5 6]        (APL2, APL*PLUS, SHARP APL)
UBILLS[2 8 12]←UBILLS[4 5 6]
```

Using conventional APL, you may write functions CNIDXA, NNIDXA and ASSIGN which will replace the desired information:

```
NAME←NAME CNIDXA 5 ASSIGN 'NEW CORP.'
UBILLS←UBILLS NNIDXA 5 ASSIGN ,NEWBILLS

NAME←NAME CNIDXA 2 8 12 ASSIGN NAME CNIDX 4 5 6
UBILLS←UBILLS NNIDXA 2 8 12 ASSIGN UBILLS NNIDX 4 5 6
```

The function ASSIGN is a simple function provided for your convenience. The CNIDXA and NNIDXA functions require 3 arguments (original nested array, indices to be replaced, simple or nested array to be inserted). Since APL syntax allows only 2 arguments, the 3rd argument must be passed as a global variable. The function ASSIGN assigns its right argument to the global variable <assign> and returns its left argument as its explicit result.

The CNIDXA and NNIDXA functions require a nest (in <assign>) if their right argument is a vector; they require a simple array if their right argument is a scalar.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: What is the average length (ANLEN) of customer names? How many unpaid bills (NUBILLS) exist?

TOPIC: Determining Shapes of Irregular Items

Using nested arrays, you must determine the shape of each item and then manipulate the resulting vector accordingly:

```
ANLEN←(+/∊⍴¨NAME)÷⍴NAME          (APL2)
ANLEN←(⊃+/⍴¨NAME)÷⍴NAME          (APL*PLUS)
ANLEN←(+/,⍴⍤>NAME)÷⍴NAME         (SHARP APL)
```

Each of these expressions requires one more step than intuition suggests (∊, ⊃ and ,). This additional step is needed because the result after performing the ⍴¨ (or ⍴⍤>) function is still a nested vector (or 1 column matrix), whose items (or rows) are each one element vectors, since monadic ⍴ always returns a vector. The array of one element vectors must be converted into a simple vector of scalars.

The solutions for the second part of the problem are similar:

```
NUBILLS←+/↑¨⍴¨UBILLS             (APL2)
NUBILLS←+/⊃¨⍴¨UBILLS             (APL*PLUS)
NUBILLS←+/,0 ¯1↓⍴⍤>UBILLS        (SHARP APL)
```

Since the items of UBILLS are 3 column matrices, logic is included (↑¨, ⊃¨ and ,0 ¯1↓) to look at only the number of rows in each matrix (i.e. the first element of the shape of each matrix).

Using conventional APL, you may write functions CNLEN and NNLEN which return a vector of the lengths of the items in the specified character or numeric nest:

```
ANLEN←(+/CNLEN NAME)÷1↑NAME
NUBILLS←(+/NNLEN UBILLS)÷3
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Sort the database of customer information by customer name, alphabetically.

TOPIC: Sorting Character Nests

To sort the database of customer information, we need the grade-up vector which would put the names in alphabetically sorted order. Given this vector, say GRADE, the database may be reordered as follows:

```
ID←ID[GRADE]
TERR←TERR[GRADE]

NAME←NAME[GRADE]               (nested arrays)
UBILLS←UBILLS[GRADE]

NAME←NAME CNIDX GRADE          (conventional APL)
UBILLS←UBILLS NNIDX GRADE
```

How do we determine GRADE? Using nested arrays, you must first convert NAME into a character matrix and then apply ⍋ or CGRADEUP (see Searching and Sorting chapter) on the character matrix.

```
ALP←' .,;:-/ABCDEFGHIJKLMNOPQRSTUVWXYZ'

GRADE←ALP⍋⊃NAME                        (APL2)
GRADE←ALP⍋⍉⍪NAME                       (SHARP APL)
GRADE←ALP⍋↑(⊃⌈/⍴¨NAME)↑¨NAME           (APL*PLUS)
```

Using conventional APL, you may write a function CNGRADEUP which returns the grade vector that may be used to reorder the character vector items of a specified character nest into sorted order.

```
GRADE←ALP CNGRADEUP NAME
```

The CNGRADEUP function may be written employing techniques discussed in the Searching and Sorting chapter. As such, it does not need to convert its character nest right argument into a character matrix with as many columns as the longest item. In this way, the CNGRADEUP function is superior to the algorithms listed above for nested arrays. Each of these algorithms constructs a character matrix from the nest. If one name is very long, a WS FULL error may be generated.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: How many times does the name 'FASTENERS INC.' occur as a customer name? Which customer is 'ABC CORP.'? Which is 'DEF CORP.'? Which is 'GHR CORP.'?

TOPIC: Searching Character Nests

Using nested arrays, the number of times the name 'FASTENERS INC.' occurs may be determined by one of the following:

```
+/NAME∊⊂'FASTENERS INC.'            (APL2, APL*PLUS)
+/NAME∊<'FASTENERS INC.'            (SHARP APL)
```

Using conventional APL, you may write a function CNEQ which compares each item of a specified character nest to a specified character vector and returns a bit vector with one element per item in the character nest, the 1s corresponding to items which match the character vector.

```
+/NAME CNEQ 'FASTENERS INC.'
```

The CNEQ function may take two character nest arguments, or one character nest and one character vector argument (in either order) or two character vector arguments (returning a bit scalar). In other words, its behavior is parallel to that of the primitive scalar dyadic function equals (=).

The second part of the problem begs for an index result. Therefore, dyadic iota (⍳) is the logical choice. Using nested arrays,

```
NAME⍳'ABC CORP.' 'DEF CORP.' 'GHR CORP.'    (APL2, APL*PLUS)
NAME⍳'ABC CORP.'⊃'DEF CORP.'⊃'GHR CORP.'    (SHARP APL)
```

Using conventional APL, you may write a function CNIOTA which searches through its character nest left argument for the first occurrence of each of the items in its character nest right argument. If the right argument is a character vector, the result is a scalar.

```
NAME CNIOTA 'ABC CORP.' CNCAT 'DEF CORP.' CNCAT 'GHR CORP.'
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: Determine the total amount of unpaid bills for each customer, returning a numeric vector AMT with one element per customer.

TOPIC: Reducing Numeric Nests

Using nested arrays, the "each", "on" or "with" operators are needed:

```
AMT←3⊃¨+⌿¨UBILLS                                  (APL2, APL★PLUS)
AMT←+/(⌈/,0 ¯1↓⍴ö>UBILLS)↑ö>,ö>(<0 2)↓¨>UBILLS    (SHARP APL)
```

Using conventional APL, you may write a function NNSUMCOL which will sum the Nth column of each M-column matrix item stored as a numeric vector item in the specified numeric nest.

```
AMT←UBILLS NNSUMCOL 3 3
```

The first 3 in the right argument of NNSUMCOL is the number of columns in the raveled matrix items. The second 3 is the index of the column to be summed.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 459 to 473)

1. Suppose you have a numeric nest (vector of numeric vectors) named SALES which contains one item per customer and for which each item is a vector of the customer's monthly sales. The length of each vector is a function of how long the customer has been generating sales. What expression will produce a vector (AVG) with one element per customer, where each element is the average monthly sales for the corresponding customer?

2. Given a character nest NAMES, what expression will return a character nest of its distinct (unique) items (UNAMES)?
~~~~

3. Write one or more of the character nest functions mentioned in this chapter for your implementation of APL (if it does not have nested arrays). The functions are: CNEST, CN∆M, CNCAT, CNIDX, CNIDXA, ASSIGN, CNLEN, CNGRADE, CNEQ, CNIOTA.

4. Write one or more of the numeric nest functions mentioned in this chapter if your implementation of APL does not have nested arrays. The functions are: NNEST, NNCATSS, NNCATVS, NNCATSV, NNCATVV, NNCAT, NNIDX, NNIDXA, ASSIGN, NNLEN, NNSUMCOL.

Chapter 17

CURVE FITTING

In both the business and scientific disciplines, the need occasionally arises to fit mathematical curves to empirical data. The process of curve fitting is a wonderful, magical process. Like most magic, curve fitting is illusion based upon reality. The reality is that rigorous mathematical algorithms are applied by the computer to determine the precise parameters of a "best" curve. The illusion is that this "best" curve somehow possesses more knowledge than the data; that it can be used to predict the future. This can be a dangerous and foolhardy view.

This is not to say that curve fitting is useless. Far from it. Curve fitting is not only useful; it is fun. Just be careful that you are not lured into thinking that the computer's intuition is better than your own.

In this chapter, we discuss the most complex of all the APL primitive functions, quad-divide (⌹). The chapter does not presume that you understand the concepts of numerical analysis and linear algebra needed to appreciate the algorithms applied within quad-divide.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given the following sample data and the function of a curve, find the coefficients of the curve.

T	X	Y	Z
9	0	3	0
20	1	2	3
19	4	1	2

Curve: T=(A×X)+(B×Y)+(C×Z)

Problem: find scalars A, B, C
~~~~

TOPIC: Using Quad-Divide

To provide some meaning to this example, suppose the three observations represent three salespeople. The first salesperson received no base salary (X) last year, was paid $3 in stock options (Y) and received no bonuses (Z). He generated 9 new customers (T). The second salesperson received $1 in base salary, $2 in stock options, $3 in bonuses and generated 20 new customers. The third salesperson received $4 in base salary, $1 in stock options, $2 in bonuses and generated 19 new customers.

We have a hypothesis that the number of new customers generated by a salesperson (T) is a direct function of the base salary (X), stock options (Y) and bonuses (Z), and that the function is in the form:

```
          T=(A×X)+(B×Y)+(C×Z)
```

If we can determine the constants (coefficients) A, B and C in this formula, we can predict the number of new customers (T) which will be generated by a salesperson who is paid a specified base salary, stock option amount and bonus amount. Since we have three equations (9=3×B; 20=A+(2×B)+(3×C); 19=(4×A)+B+(2×C)) in three unknowns (A, B, C), the problem can be solved by algebraic manipulations and substitutions. After tedious work, we discover that A=2, B=3, C=4.

Quad-divide (⌹) can be used to solve this problem directly. Provide the dependent values (T) as its vector left argument, and the independent values (X, Y, Z) as its 3 column matrix right argument. The result is a vector of the three desired constants (coefficients).

```
                T←9 20 19
                XYZ←3 3⍴0 3 0 1 2 3 4 1 2
                C←T⌹XYZ
                C
          2 3 4
```

Once the coefficients are known, it becomes a simple matter to apply the formula to hypothetical values. For example, how many new customers do we expect to be generated by a salesperson who is paid 0 in base salary, $3 in stock options and $4 in bonuses?

```
                0 3 4+.×C                    ⍝ (0×A)+(3×B)+(4×C)
          25
```

In order to solve problems like these (systems of linear equations), the data must satisfy certain requirements. For one thing, we must provide one equation (observation) for each unknown. Since there were three unknown constants (A, B, C) in the above example, we needed three observations. If we do not provide enough observations, there are an infinite number of solutions (values of A, B, C) which will satisfy the specified observations. Quad-divide will generate a DOMAIN ERROR.

Second, even if we provide one observation per unknown, there may be insufficient data to determine a unique solution. For example, in the following set of data, the third observation provides the same information as the second. The values are simply doubled.

T	X	Y	Z
9	0	3	0
20	1	2	3
40	2	4	6

No amount of algebraic manipulation will produce a unique solution. To have a unique solution, each observation must be independent. That is, no observation may be a linear combination of the other observations.

If observations are not independent, quad-divide will generate a DOMAIN ERROR. The matrix right argument is said to be "singular" and cannot be "inverted". No unique solution exists.

Third, if we provide more observations that there are unknowns, there will likely be contradictory data so that no single set of coefficients will satisfy all of the observations. In such an instance, the best we can hope for is a set of coefficients which defines a formula which is reasonably accurate for all of the observations, though it may not be precise for any of them.

This is, in fact, what quad-divide returns. It does so by using the method of least squares. Simply stated, it returns those coefficients which when applied against the independent values will return a dependent value (the "expected") which is as close as possible to the specified dependent value (the "actual"). One measure of "noncloseness" (called the "mean squared error") is the mean of the squares of the differences between each of the actuals and their corresponding expecteds. The algorithm within quad-divide determines the set of coefficients which produces the smallest value of this noncloseness measure (hence, "least squares").

To illustrate this "best fit" behavior of quad-divide, let us add one more salesperson to our example and then determine the coefficients:

T	X	Y	Z
9	0	3	0
20	1	2	3
19	4	1	2
11	2	2	1

```
            T←9 20 19 11
            XYZ←4 3⍴0 3 0 1 2 3 4 1 2 2 2 1
            C←T⌹XYZ
            C
      1.6667 2.6667 4.3333
```

We may determine the expecteds by applying the computed coefficients against the matrix of independent values:

```
            E←XYZ+.×C
            E
      8 20 18 13
```

While only one of these values exactly matches the actuals (T), the other values are reasonably close. The mean squared error is:

```
            ((E-T)+.*2)÷⍴E
      1.5
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: During the last 5 years, the annual sales figures for your firm have been 27, 29, 35, 41, 44. If you assume that sales are growing at a linear rate, what do you project sales to be next year?

TOPIC: Forecasting

Forecasting is just curve fitting in disguise. Instead of expressing a formula in terms of several independent variables (such as X, Y, Z), we express it in terms of time (T). In this problem, we are searching for a formula which looks like:

```
      SALES=A+(B×T)
```

where T is some measure of time (such as 1, 2, 3, 4, 5 for the last 5 years) and A and B are the constants to be determined.

If we express our data in the same form as the data in the previous section, the solution becomes obvious.
~~~~

SALES	1	T
27	1	1
29	1	2
35	1	3
41	1	4
44	1	5

The column of 1s is used to indicate that the first constant (A) is multiplied by 1 in each observation (i.e. in SALES=A+(B×T)).

```
      MAT←5 2ρ1 1 1 2 1 3 1 4 1 5
      SALES←27 29 35 41 44
      C←SALES⌹MAT
      C
21.4 4.6
```

The two elements in C are our constants A and B. To project the sales for next year, we need only plug the time 6 into our formula:

```
      1 6+.×C
49
```

If we wish to see how closely our formula tracks the past five years, we can apply the constants to the time periods 1 to 5:

```
      MAT+.×C
26 30.6 35.2 39.8 44.4
```

We can compare these values to the actual sales (27, 29, 35, 41, 44).

Suppose we assume that sales are growing at a quadratic rate instead of a linear rate. The formula instead looks like:

```
SALES=A+(B×T)+(C×T*2)
```

Our data can be expressed in an expanded table:

SALES	1	T	T*2
27	1	1	1
29	1	2	4
35	1	3	9
41	1	4	16
44	1	5	25

The solution is no more complex:

```
          MAT←(⍳5)∘.*0 1 2
          SALES←27 29 35 41 44
          C←SALES⌹MAT
          C
    22.4 3.742857 .142857
```

Nor is the projection:

```
          1 6 36+.×C
    50
```

Nor is the tracking:

```
          MAT+.×C
    26.2857 30.4571 34.9143 39.6571 44.6857
```

Notice that a quadratic formula tracks better than a linear one. This in no surprise since the linear formula is just a special case of the quadratic formula (the case in which the coefficient of the squared term is 0). If the line fits better than any other quadratic form, the third coefficient will be zero.

After realizing that a quadratic formula fits better than a linear formula, we may jump to the conclusion that a cubic formula (e.g. SALES=A+(B×T)+(C×T*2)+(C×T*3)) fits better still. And it does. However, we must bear two thoughts in mind before carrying this logic to its extreme. The first is that we must provide at least one observation per unknown constant. Otherwise, quad-divide will generate a DOMAIN ERROR because there are an infinite number of possible solutions. Hence, for our 5 observations (years), the most terms we can have in our formula is 5.

The second thing to consider is that the reason a higer-order formula fits better to the data is that it exhibits more helter-skelter behavior than a lower-order formula. It has more mood swings. Thus, the formula which is selected by quad-divide to perfectly match your 5 observations may send you into outer space when you project the sixth period.

The bottom line in this discussion is that despite the difficulty of the number-crunching task faced by the computer, your task is more difficult. It is your job, after all, to determine which formula you think your data fits. There are an infinity of possible formulas. The computer merely has to crunch out coefficients.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEM: As in the previous section, we wish to project forward one time period the sales figures from the past 5 years (27, 29, 35, 41, 44). We believe the data conforms to the formula, SALES=÷(A+(B×T)).

TOPIC: Fitting Data to a Nonlinear Formula

Our problem here is that we need to determine two constants (A and B) but the formula is not expressed as an additive combination of two terms, each multiplied by one of the constants. We must transform the formula so that it is in a form suitable to quad-divide.

If we multiply both sides of the formula by A+(B×T), we get:

```
1=(SALES×A)+(SALES×B×T)
```

From this linear form, we can solve directly for A and B:

```
      SALES←27 29 35 41 44
      TIME←⍳5
      MAT←SALES,[1.5]SALES×TIME
      C←((⍴SALES)⍴1)⌹MAT
      C
.040625 ¯3.7567
```

The projection follows directly:

```
      ÷C+.×1 6
55.295
```

While we have accomplished our projection via this transformation, we must concede that quad-divide provides us the best coefficients for the transformed formula, not for the original formula. Hopefully, this distinction is not important. However, we must bear it in mind whenever we transform a formula.

Let us try fitting the data to another formula which is nonlinear and requires a transformation.

Formula: SALES=A×*B×T

Transformed formula: (⍟SALES)=(⍟A)+(B×T)

```
Solution:   MAT←TIME∘.*0 1
            C←(⍟SALES)⌹MAT            ⍝ Returns: (⍟A) and B
            (*C[1])×*C[2]×6
     51.426
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: In the previous sections, we use four different formulas to project 5 years of sales figures one year:

1. SALES=A+(B×T)
2. SALES=A+(B×T)+(C×T⋆2)
3. SALES=÷(A+(B×T))
4. SALES=A×⋆B×T

For each of these formulas, we obtain a different set of coefficients and a different projection (sales figure for the sixth year). Which projection is the best?

TOPIC: Finding the Best Formula

The best projection is that which exactly matches the actual. Unfortunately, the actual sixth year sales figure is not yet known. Therefore, we must determine the best projection in another way. Let's choose the projection that comes from the formula which most closely fits the data. (Bear in mind that the best fitting curve is not necessarily the best projector of the data.)

Which formula fits the best? Just as the mean squared error is used to determine the best coefficients for a given formula, so too can it be used to determine the best formula for a given set of data. The formula which produces the least mean squared error is the best fitting formula. Remember: the mean squared error is the mean of the squares of the differences between each of the actuals and thier corresponding expecteds.

Let us compute the mean squared error for each formula:

```
      SALES←27 29 35 41 44
      TIME←1 2 3 4 5
```

1. SALES=A+(B×T)

```
      MAT←TIME∘.⋆0 1
      C←SALES⌹MAT
      EXP←MAT+.×C
      ((SALES-EXP)+.⋆2)÷⍴SALES
1.04
```
~~~~

2. SALES=A+(B×T)+(C×T⋆2)

```
              MAT←TIME∘.⋆0 1 2
              C←SALES⌹MAT
              EXP←MAT+.×C
              ((SALES-EXP)+.⋆2)÷⍴SALES
        0.983
```

3. SALES=÷(A+(B×T))

```
              MAT←SALES,[1.5]SALES×TIME
              C←((⍴SALES)⍴1)⌹MAT
              EXP←÷(TIME∘.⋆0 1)+.×C
              ((SALES-EXP)+.⋆2)÷⍴SALES
        1.85
```

4. SALES=A×⋆B×T

```
              MAT←TIME∘.⋆0 1
              C←(⍟SALES)⌹MAT
              EXP←(⋆C[1])×⋆C[2]×TIME
              ((SALES-EXP)+.⋆2)÷⍴SALES
        1.10
```

The smallest mean squared error for the above formulas is the one for the second formula. Therefore, the second formula is the "best" formula (of the four selected) and the best projection is the one from that formula (50).

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEMS: (Solutions on pages 474 to 475)

1. Each month, you buy 4 styles of envelopes from your paper supplier (styles A, B, C, D). The supplier has never bothered to itemize the different styles on the invoice. The invoice shows a total amount only. Suppose you bought the following quantities of envelopes during the previous 4 months:

A	B	C	D	Total Invoice
32	61	15	82	$60.62
35	104	10	82	$59.57
37	83	5	85	$56.70
25	62	14	85	$60.42
~~~~

Assuming prices have remained constant during this period and that the invoice does not include volume discounts or other credit or debit adjustments, what are the unit prices for the 4 styles of envelopes?

2. During the last 2 months, you have been making weekly checks of your inventory of muffler bearings. The results are given here:

Week:	1	2	3	4	5	6	7	8	9
On hand:	1850	1772	1705	----	1508	1490	----	----	1250

(During week 4 the plant was closed and during weeks 7 and 8 you were on vacation.) Assuming the supply of bearings is being depleted at a steady (linear) rate, when will the supply be exhausted (assuming no restocking)?

3. What is the radius and center of the circle which best fits the points (4,1), (3,2), (2,3), (4,5), (7,2), (6,4), i.e. for which X←4 3 2 4 7 6 and Y←1 2 3 5 2 4? The general formula for a circle is:

```
(R*2) = ((X-CX)*2) + ((Y-CY)*2)
```

where R is the radius, and (CX,CY) is the center.

4. In the Sorting and Searching chapter, a function CMIOTA is presented for searching through the rows of one character matrix for the location of the rows of a second. The function is designed to use one of two different algorithms depending upon the number of rows in its arguments. In the Computer Efficiency Considerations chapter, the two algorithms are timed for a variety of different size arguments (in a problem at the end of that chapter). Suppose you have constructed 4 vectors which contain the results of those timings:

L: The number of rows for left arguments;

R: The number of rows for right arguments;

T1: The average time required to run CMIOTA using the first (sorting) algorithm for a left argument with L rows and a right argument with R rows;

T2: The average time required to run CMIOTA using the second (looping) algorithm for a left argument with L rows and a right argument with R rows.

According to the logic in the Sorting and Searching chapter, the vectors are related according to the approximate formulas:

```
T1 = C4+(C5×(R+L))
T2 = C1+(R×(C2+(C3×L)))
```

Determine the values for C1, C2, C3, C4 and C5. You may replace the like-named variables in the CMIOTA function so that it will use the fastest algorithm for any combination of arguments in your APL environment.

Chapter 18

FINANCIAL UTILITIES

In a variety of business oriented computer applications, sophisticated financial calculations are required. Many of these calculations deal with the time-value-of-money concept, i.e. the concept of interest. In this chapter, we develop utilities which handle some of the more common financial calculation requirements.

This chapter does not provide a comprehensive treatment of the theory of interest or of financial analysis. It provides only as much material as is needed for you to grasp the important concepts. Nor does it provide a comprehensive library of financial software. It provides some generally useful utility functions.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: You are considering the purchase of a new machine. The amount you must borrow from the bank is $225,000 to be repaid in monthly installments at 13% interest over 10 years. What will be the monthly payment?

TOPIC: Interest and Annuities

We will work toward the solution to this problem gradually.

Suppose you deposit $100 in your bank account at the beginning of the year. If at the end of the year your balance is $112, you have earned $12 on your $100 deposit and so your "effective" (annual or APR) interest rate is .12 (12÷100). At the end of two years (if interest rates remain the same), you will have $125.44 ($112 + $112×.12 or $112×1.12). After three years, you will have $140.49 ($125.44×1.12). And so on. In general, after N years, you will have $100×1.12*N. The factor 1.12 (1 plus the interest rate) is called the annual "accumulation factor".
~~~~

Suppose we look at your banking situation in a reverse way, from the present back into the past. If you have a $100 balance today, how much must you have had in the account one year ago? If we call the desired amount AMT, we know from the logic above that AMT×1.12 = $100. Therefore, AMT = $100÷1.12 = $89.28. Two years ago, you had $79.72 ($89.28÷1.12). And so on. In general, N years ago, you had $100÷1.12*N. The factor ÷1.12 (the reciprocal of the annual accumulation factor) is called the annual "discount factor").

If instead of paying you 12% per year, the bank offers to pay you 1% per month, how much money will you have at the end of the year? Since the monthly accumulation factor is 1.01, you will have $101 ($100×1.01) after one month, $102.01 ($101×1.01) after two months, $103.03 ($102.01×1.01) after three months, ... and $112.68 ($111.57×1.01 or $100×1.01*12) after twelve months. Since you have earned $12.68 on your $100 deposit, your effective interest rate is .1268. This rate may also be called a "nominal" rate of .12, compounded monthly (.12 = .01×12 months). The term "compounded" (or converted) refers to the periodic process of applying the accumulation factor to the balance to derive the new balance.

Since an interest rate may be expressed in "effective" or "nominal" terms, it is important to understand the distinction between the two and to be able to convert between them. The relationship between them follows:

(1+EINT) = (1+NINT÷CONV)*CONV

where:

EINT = effective interest rate (e.g. .1268)
NINT = nominal interest rate (e.g. .12)
CONV = number of interest conversion (compounding) periods per year

It is instructive to note that for a given nominal interest rate, the equivalent effective interest rate increases as the conversion frequency increases. That is, the more frequently the bank compounds your nominal interest, the more you will have at the end of the year. However, the amount of increase in the effective interest rate drops off as the conversion frequency gets bigger and bigger. In fact, it makes little difference whether you compound weekly or every second. This brings up the concept of "continuous" compounding. That is, by using a bigger and bigger value of CONV in the formula above, the value of (1+EINT) will eventually (when CONV becomes infinitely large) become constant. This value (called the "force of interest") may be computed by using a large value of CONV (say, 10000) or by using the following formula (which may be derived mathematically):

(1+EINT) = *NINT (when CONV is infinitely large)

Because of marketing appeal or because of the mathematical simplicity, financial institutions occasionally quote interest rates "compounded continuously". You may convert them to equivalent effective rates by applying the above formula.

The following functions may be used to convert between nominal and effective interest rates.

[WSID: INTEREST]

```
    ∇ E←C EFFECTIVE N;R;S;T
[1]   ⍝ Converts the nominal interest rates in N,
[2]   ⍝ compounded C times per year, into effective
[3]   ⍝ (annual) interest rates.  A value of ¯1 for
[4]   ⍝ C implies continuous compounding.  C and N
[5]   ⍝ may have any shapes as long as they are
[6]   ⍝ scalar conformable.
[7]   ⍝
[8]   ⍝ Determine shape of result:
[9]    S←⍴C+N
[10]  ⍝ Construct all zero raveled result:
[11]   R←⍴E←(×/S)⍴0
[12]  ⍝ Convert arguments to same-length vectors:
[13]   C←R⍴C
[14]   N←R⍴N
[15]  ⍝ Select rates with noncontinuous compounding:
[16]   T←C≠¯1
[17]   E[T/⍳⍴T]←¯1+(1+(T/N)÷T/C)*T/C
[18]  ⍝ Select continuous rates:
[19]   T←~T
[20]   E[T/⍳⍴T]←¯1+*T/N
[21]  ⍝ Reshape result:
[22]   E←S⍴E
    ∇
```

[WSID: INTEREST]

```
    ∇ N←C NOMINAL E;R;S;T
[1]   ⍝ Converts the effective (annual) interest rates
[2]   ⍝ in E into nominal interest rates compounded C
[3]   ⍝ times per year.  A value of ¯1 for C implies
[4]   ⍝ continuous compounding.  C and E may have any
[5]   ⍝ shapes as long as they are scalar conformable.
[6]   ⍝
[7]   ⍝ Determine shape of result:
[8]    S←⍴C+E
[9]   ⍝ Construct all zero raveled result:
[10]   R←⍴N←(×/S)⍴0
[11]  ⍝ Convert arguments to same-length vectors:
[12]   C←R⍴C
[13]   E←R⍴E
[14]  ⍝ Select rates with noncontinuous compounding:
[15]   T←C≠¯1
[16]   N[T/⍳⍴T]←(T/C)×¯1+(1+T/E)*÷T/C
[17]  ⍝ Select continuous rates:
[18]   T←~T
[19]   N[T/⍳⍴T]←⍟1+T/E
[20]  ⍝ Reshape result:
[21]   N←S⍴N
    ∇
```

Let us extend our discussion to include annuities. An "annuity" is a regular series of payments. Suppose, for example, you plan to deposit $100 in your bank account at the beginning of each year for the next 10 years. If the effective interest rate is .12, what will be your balance at the end of 10 years?

The first payment will have compounded 10 times, the second 9 times, and so on. The sum of these amounts is the balance:

```
BALANCE = (100×1.12*10) + (100×1.12*9) +...+ (100×1.12*1)
```

By dividing each side of this equation by 1.12 to get a second equation, by subtracting the second equation from the first equation and by algebraic manipulation, we discover that:

```
BALANCE = 100×1.12×((1.12*10)-1)÷.12
```

By performing the calculations, we learn that the "future value" of this 10 year annuity is $1,965.46.

Suppose we look at a 10 year annuity in a reverse way, from the beginning of the annuity rather than the end. What is the value today (the "present value") of an annuity in which you deposit $100 at the beginning of each year for the next 10 years? That is, what single amount can you deposit today that will compound to $1,965.46 at the end of 10 years, i.e. that will result in the same future value as will the annuity?

If PV (present value) is the amount we are seeking, the equation we need to solve is:

```
(PV×1.12*10) = 100×1.12×((1.12*10)-1)÷.12
```

from which we get:

```
PV = 100×1.12×(1-1.12*¯10)÷.12
```

By performing the calculations, we learn that the present value of this 10 year annuity is $632.83. That is, a single payment today of $632.83 is equivalent to a 10 year annuity of $100 starting today. Each will compound to $1,965.46 at the end of 10 years.

Having presented some examples to expose you to the concepts of compounding, annuities and present value, let us now take a leap forward and work with more general annuities. First, some definitions:

TERM: the number of years during which the annuity is paid;

PAY: the annual payment amount;

PER: the number of payments per year (PAY÷PER per payment);

PDEF: the fraction of a payment period preceding (deferring) each payment; 0 if the payment occurs at the beginning of the payment period and 1 if at the end;

DEF: the number of years from the valuation date to the first period; 0 if the present value of the annuity is desired and -TERM if the future value is desired;

CONV: the number of interest conversion (compounding) periods per year;

INT: the nominal annual interest rate.

We will illustrate these terms by depicting a sample annuity on a "time" diagram. Below is the time diagram of an annuity for which the present value is desired. The annuity will be paid for 3 years (TERM=3), will be $200 per year (PAY=200) paid semiannually (PER=2) at the end (PDEF=1) of each half year payment period and will commence after 1 year (DEF=1) without payments. The present value is to be computed at a nominal interest rate of 12% (INT=.12) compounded monthly (CONV=12).

```
       ←----DEF=1----→←-----------------TERM=3-----------------→
 year: 0      0.5      1      1.5      2      2.5      3      3.5      4
       |-------|--------|------|--------|------|--------|------|--------|
       ↑                      $100     $100   $100     $100   $100     $100
    PRESENT             PDEF=1-→
   (INT=.12)            ←---PAY=200---→
   (CONV=12)            ←----PER=2----→
```

The general formula for the value of an annuity is:

```
VAL = PAY×(V⋆DEF+PDEF÷PER)×(1-V⋆TERM)÷PER×1-V⋆÷PER
```

where:

VAL: the value (present, future or otherwise) of an annuity as of DEF years before the first payment period;

V: the annual discount factor, computed from INT and CONV,

```
V = (1+INT÷CONV)⋆-CONV
```

Some modifications to this formula are needed for some possible extreme conditions. First, if TERM is infinite (such an annuity is called a "perpetuity"), the V⋆TERM term of the formula becomes 0 and may be omitted. Second, if CONV is infinite (continuous interest compounding), the definition of V becomes:

```
V = ⋆-INT
```

Third, if PER is infinite ("continuous payments", a theoretical concept), the PDEF÷PER term of the formula becomes 0 and may be omitted, and the PER×1-V⋆÷PER term becomes -⍟V.

Given this formula and these possible modifications, a niladic function VALUE can be written which computes the value of an annuity given all of its parameters as global variables. The global variables have the same names as the parameters defined above except the first letter is from the alternate character set (tERM, pAY, pER, pDEF, dEF, cONV, iNT). Infinite values are represented by the value ¯1 (for tERM, pER or cONV). The global variables may have any shape as long as they are all scalar conformable with one another, that is as long as the expression,

```
tERM+pAY+pER+pDEF+dEF+cONV+iNT
```

does not generate a RANK ERROR or a LENGTH ERROR.

All of the global values (except iNT) have default values which are used if that global variable does not exist.

```
                                   [WSID: INTEREST]
    ∇ VAL←VALUE;CONV;DEF;E;INT;N;NDPP;PAY;PD;PDEF;PER;R;
      SHAPE;TERM;V
[1]   ⍝ Returns the present or future value of a stream
[2]   ⍝ of uniform cash flows defined by the optional
[3]   ⍝ global variables:
[4]   ⍝
[5]   ⍝ default  name  description
[6]   ⍝ -------  ----  -------------------------------------
[7]   ⍝    1     tERM  no. years of payments (¯1=perpetuity)
[8]   ⍝    1     pAY   annual payment amount
[9]   ⍝    1     pER   no. payments per year (¯1=continuous)
[10]  ⍝    1     pDEF  is payment at start (0) or end (1) of
[11]  ⍝                payment period (or fractional)
[12]  ⍝    0     dEF   no. years from valuation date to
[13]  ⍝                first payment period
[14]  ⍝    1     cONV  no. interest conversion (compounding)
[15]  ⍝                periods per year (¯1=continuous)
[16]  ⍝    -     iNT   nominal annual interest rate (or force
[17]  ⍝                of interest if cONV=¯1)
[18]  ⍝
[19]  ⍝ Determine values from globals or defaults:
[20]   E←×⎕NC N←'tERM'
[21]   TERM←⍎(E/N),(~E)/'1'
[22]   E←×⎕NC N←'pAY'
[23]   PAY←⍎(E/N),(~E)/'1'
[24]   E←×⎕NC N←'pER'
[25]   PER←⍎(E/N),(~E)/'1'
[26]   E←×⎕NC N←'pDEF'
[27]   PDEF←⍎(E/N),(~E)/'1'
[28]   E←×⎕NC N←'dEF'
[29]   DEF←⍎(E/N),(~E)/'0'
[30]   E←×⎕NC N←'cONV'
[31]   CONV←⍎(E/N),(~E)/'1'
[32]   INT←iNT
[33]  ⍝
[34]  ⍝ Construct result as a vector; reshape when done:
[35]   SHAPE←⍴TERM+PAY+PER+PDEF+DEF+INT+CONV
[36]   R←×/SHAPE
[37]   TERM←R⍴TERM
[38]   PAY←R⍴PAY
[39]   PER←R⍴PER
[40]   PDEF←R⍴PDEF
[41]   DEF←R⍴DEF
[42]   INT←R⍴INT
[43]   CONV←R⍴CONV
[44]  ⍝ Convert interest to annual discount factor:
[45]   V←R⍴0
[46]  ⍝ Noncontinuous compounding:
[47]   E←CONV≠¯1
[48]   V[E/⍳⍴V]←(1+(E/INT)÷E/CONV)*-E/CONV
[49]  ⍝ Continuous compounding:
[50]   E←~E
[51]   V[E/⍳⍴V]←*-E/INT
```

```
    ∇ VALUE (continued)
[52] ⍝
[53] ⍝ Period deferral as a fraction of a year:
[54]  E←PER≠¯1
[55]  PD←E×PDEF÷PER+~E
[56] ⍝
[57] ⍝ Determine nominal (by payment period) discount:
[58]  NDPP←Rρ0
[59] ⍝ Noncontinuous payments:
[60]  NDPP[E/ιρNDPP]←(E/PER)×1-(E/V)⋆÷E/PER
[61] ⍝ Continuous payments:
[62]  E←~E
[63]  NDPP[E/ιρNDPP]←-⍟E/V
[64] ⍝
[65] ⍝ Value at valuation date:
[66]  E←TERM≠¯1
[67]  VAL←SHAPEρPAY×(V⋆PD+DEF)×(1-E×V⋆E×TERM)÷NDPP
    ∇
```

Using the VALUE function, you can now answer the problem which began this section. The repayment schedule of a loan is simply an annuity. The present value of that annuity is exactly equal to the amount borrowed from the bank. We know the following:

```
      tERM←10
      pER←12
      pDEF←1
      dEF←0
      cONV←12          (probably, but ask the bank)
      iNT←.13
```

The only unknown parameter is pAY, whose value we seek. However, we know the present value (225000) which will be the result of VALUE if we provide the correct value of pAY. We may use VALUE iteratively, modifying the value of pAY in a trial and error fashion until the result of VALUE is 225000. Alternatively, since we know that the result of VALUE is directly proportional to the value of pAY, we may arbitrarily set pAY to 12 (i.e. 1 per month) and divide the result of VALUE into 225000 to determine the factor by which pAY must be multiplied to produce the desired present value.

```
      pAY←12
      225000÷VALUE
3359.49
```

This is the monthly payment.

If you wish to experiment with the rate of interest to determine its effect upon the monthly payment, you may do it as follows:

```
      iNT←.10 .11 .12 .13 .14 .15
      225000÷VALUE
2973.39 3099.38 3228.1 3359.49 3493.49 3630.04
```

To produce a table of the monthly payments for each of these six interest rates, for 5, 10, 15 and 20 years, you may do it as follows:

```
        iNT←.10 .11 .12 .13 .14 .15 ∘.+ 0 0 0 0
        tERM←0 0 0 0 0 0 ∘.+ 5 10 15 20
        9 2⍕225000÷VALUE
 4780.59  2973.39  2417.86  2171.30
 4892.05  3099.38  2557.34  2322.42
 5005.00  3228.10  2700.38  2477.44
 5119.44  3359.49  2846.79  2636.05
 5235.36  3493.49  2996.42  2797.92
 5352.73  3630.04  3149.07  2962.78
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Having borrowed $225,000 from the bank to be repaid in monthly installments at 13% interest over 10 years, how much will you have paid after one year? Of that amount, how much will have gone toward payment of interest and how much toward repayment of the original principal of $225,000?

TOPIC: Loan Amortization Schedules

From the function presented in the prior section, we can answer the first part of the problem.

```
        tERM←10
        pER←12
        pDEF←1
        dEF←0
        cONV←12
        iNT←.13
        pAY←1
        ⎕←ANNUAL←225000÷VALUE
40313.9
```

This annual loan repayment consists of an interest portion and a principal repayment portion. To determine the two portions, we need only determine the remaining principal at the end of the year. The difference between the original principal ($225,000) and the remaining principal is the amount of principal repaid. The difference between the total annual payment ($40,313.90) and the principal repaid is the interest paid.

So how do we determine the remaining principal at the end of the year? The principal is the present value of the remaining payments.
~~~~

That is, the amount you owe the bank at any given time (the principal) is equivalent to the value at that time of the remaining payments. For this reason, the bank is indifferent (theoretically) to whether you repay the principal of the loan or continue to make the loan payments.

The principal at the end of the year may be computed by the following:

```
      tERM←9
      pAY←ANNUAL
      ⎕←REMAINING←VALUE
213252.48
```

The principal repaid and interest paid follow directly:

```
      ⎕←PRINCIPAL←225000-REMAINING
11747.52
      ⎕←INTEREST←ANNUAL-PRINCIPAL
28566.38
```

By applying the logic in this section, we may write an APL function which will generate a "loan amortization table". A loan amortization table shows the breakout of each loan payment into its interest and principal repayment portions. The monadic function SCHEDULE takes a 6 element vector right argument which defines the parameters of the loan:

[1] TERM: the number of years of loan repayment;

[2] PAY: the annual repayment amount;

[3] PER: the number of payments per year;

[4] DEF: the number of years from the loan to the first repayment period;

[5] CONV: the number of interest conversion (compounding) periods per year;

[6] INT: the nominal interest rate.

The result of SCHEDULE is a two column matrix of the loan amortization table. The matrix has one row per loan payment (TERM×PER). The values in the first column are the portions of each payment which represent principal repayment. The values in the second column are the portions which represent interest payments.

We may solve the problem above as follows:

```
      MAT←SCHEDULE 10,ANNUAL,12 0 12 .13
```

where ANNUAL was computed above by the VALUE function. MAT has 120 rows (one row per loan payment). The principal repaid and interest paid during the first year are:

```
      +⌿MAT[⍳12;]
11747.52 28566.38
```

There are two identities about the loan amortization table which should be noted. The sum of each row is the periodic payment amount:

```
      (+/MAT)∧.=ANNUAL÷12
1
```

The total of the principal repayment column is the original loan amount:

```
      225000=+/MAT[;1]
1
```

```
                                   [WSID: INTEREST]
    ∇ MAT←SCHEDULE PARAMS;CONV;DEF;INT;INTER;PAY;PER;PRIN;
      TERM;V;VAL
   ⍝ Returns a two column matrix of the loan payment
   ⍝ schedule for the loan defined by the parameters
   ⍝ in the vector right argument.  The result has one
   ⍝ row per payment:
   ⍝
   ⍝   MAT[;1]  principal repaid
   ⍝   MAT[;2]  interest paid
   ⍝
   ⍝ The parameters in the argument are:
   ⍝
   ⍝   PARAMS[1]  TERM    no. years of payments
   ⍝   PARAMS[2]  PAY     annual payment amount
   ⍝   PARAMS[3]  PER     no. payments per year
   ⍝   PARAMS[4]  DEF     no. years from date of loan to
   ⍝                      first payment period (0=no
   ⍝                      repayment deferral)
   ⍝   PARAMS[5]  CONV    no. interest conversion
   ⍝                      (compounding) periods per
   ⍝                      year (¯1=continuous)
   ⍝   PARAMS[6]  INT     nominal annual interest rate
   ⍝                      (or force of interest if
   ⍝                      CONV=¯1)
   ⍝
    TERM←PARAMS[1]
    PAY←PARAMS[2]
    PER←PARAMS[3]
    DEF←PARAMS[4]
    CONV←PARAMS[5]
    INT←PARAMS[6]
   ⍝ Convert interest to annual discount factor:
   ⍝ Branch if noncontinuous compounding:
    →(CONV≠¯1)⍴L1
   ⍝ Continuous compounding:
    V←*-INT
    →L2
```

```
    ∇ SCHEDULE (continued)
[36] ⍝ Noncontinuous compounding:
[37] L1:V←(1+INT÷CONV)*-CONV
[38] ⍝ Determine present value (principal balance) at
[39] ⍝ start of each payment period:
[40] L2:VAL←(PAY×V*÷PER)×(1-V*(⌽⍳TERM×PER)÷PER)÷PER×1-V*÷PER
[41] ⍝ Reset first value to loan amount if repayment
[42] ⍝ is deferred:
[43]  VAL[1]←VAL[1]×V*DEF
[44] ⍝ Principal repaid:
[45]  PRIN←VAL-1↓VAL,0
[46] ⍝ Interest paid:
[47]  INTER←(PAY÷PER)-PRIN
[48] ⍝ Adjust interest to keep it from exceeding
[49] ⍝ periodic payment:
[50]  INTER←(+\INTER)⌊+\(⍴INTER)⍴PAY÷PER
[51]  INTER←INTER-¯1↓0,INTER
[52]  PRIN←(PAY÷PER)-INTER
[53]  MAT←PRIN,[1.5]INTER
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: You have been presented an opportunity to invest \$10,000 of your money now in exchange for the promise of several future cash payments. The schedule of payments follows:

May 12, 1987	You pay \$10,000
Jan. 1, 1989	You receive \$2,000
Jan. 1, 1990	\$3,000
Jan. 1, 1991	\$4,000
Jan. 1, 1992	\$5,000
May 12, 1992	\$1,000

Is this a good investment? Are you better off leaving your money in the bank at 10%?

TOPIC: Internal Rate of Return

One approach to this problem is to explore two scenarios. In the first scenario, you deposit \$10,000 in your bank now (May 12, 1987) and leave it there. At the end of 5 years (May 12, 1992), at 10% effective interest, your money has compounded to \$16,105.10 (\$10,000×1.10*5). In the second scenario, you deposit nothing now but instead deposit \$2,000 on Jan. 1, 1989, \$3,000 on Jan. 1, 1990,
~~~~

and so on as in the schedule above. Compute the balance as of May 12, 1992.

To compute the balance for the second scenario, you must convert the dates to more manageable measurements of time. By using the TODAYS function presented in the Manipulating Dates chapter, you can translate the dates into days from May 12, 1992:

```
      DAYS←TODAYS 19890101 19900101 19910101 19920101
           19920512
      ⎕←DAYS←DAYS[5]-DAYS
1227 862 497 132 0
```

We can translate these days into "years" by dividing by 365. Then, it is a simple matter to accumulate each amount (the annual accumulation factor is 1.10) for the corresponding number of years.

```
      2000 3000 4000 5000 1000+.×1.10*DAYS÷365
17242.31
```

Since $17,242.31 is greater than $16,105.10, you will be better off to accept this investment opportunity than to leave your $10,000 in the bank.

Will the conclusion be the same if the rate of interest offered by the bank is 12%? How about 15%? The higher the rate of interest, the less important are future cash flows and the more important are current flows. At what interest rate will the accumulated value of the investment cash flows be exactly the same as the accumulated value of your money left in the bank? That is, at what interest rate will you be indifferent (ignoring the riskiness of the investment) between making the investment and leaving your money in the bank? This rate of interest is called the "internal rate of return" (IROR) of the investment cash flows. If you can realize a higher interest rate than the IROR by an alternative investment (such as leaving your money in the bank), you should not invest. However, an investment is a good investment if the IROR of its cash flows is higher than any alternative investment.

To determine the IROR of a set of cash flows, you must employ an iterative (trial and error) technique known as "successive approximations". Your task is to determine the interest rate (I) for which the following equation is true:

```
(10000×(1+I)*T0) = (2000×(1+I)*T1) + (3000×(1+I)*T2) + ...
```

where T0, T1, T2, ... are the numbers of years from the corresponding cash flow to the date of the last cash flow.

You may try different interest rates and see which produces the best results. From these observations you can choose better interest rates and try again. By repeating this procedure, you will gradually narrow in on the correct IROR.

To speed up this process, you may apply a technique known as the Newton-Raphson Method which uses the results of the previous guess to make an intelligent next guess. The guesses converge to the correct result much faster than standard interpolation procedures because the method considers the derivative (slope) of the formula.

The method is:

```
NEXTI = f(LASTI)÷f'(LASTI)
```

where:

NEXTI: the next interest rate to try;

LASTI: the previous interest rate tried;

f(I): the formula as a function of the interest rate I;

f'(I): the derivative with respect to I of the formula f(I).

Given this method, we may write a function IROR which returns the internal rate of return for a specified set of cash flows. The left argument is a vector of the dates (YYYYMMDD) of the cash flows and the right argument is a vector of the amounts of the corresponding flows, where outflows are represented as negative numbers and inflows are represented as positive numbers (or vice versa). The solution to this problem is then:

```
      DATES←19870512 19890101 19900101 19910101 19920101
            19920512
      AMTS←¯10000 2000 3000 4000 5000 1000
      DATES IROR AMTS
0.12165
```

From this, we conclude that the investment should be made if we have available no alternative investments which will generate more than 12.165%.

If an investment opportunity involves several outflows interspersed among the inflows, it is possible that the flows may not define a distinct internal rate of return. That is, several different interest rates may be used to accumulate (or discount) all the outflows to the same value as the inflows. Beware.

The IROR function is listed below. Note that it expresses the formulas as present value formulas in terms of the annual discount factor V (i.e. the reciprocal of the accumulation factor) rather than in terms of the interest rate. Also note that the iterations stop once the discount factor is determined within .0000001 or when 10 iterations have occurred. If the result is not determined in 10 iterations, the result is set to 0. Modify the function if you desire greater accuracy or more iterations or if you wish to try a starting value other than 10%.

```
                                      [WSID: INTEREST]
    ∇ INT←DATES IROR AMTS;DAYS;DIFF;I;TAMTS;TYRS;V;YRS
[1]  ⍝ Computes the internal rate of return, as an
[2]  ⍝ effective (annual) interest rate, for the stream
[3]  ⍝ of cash flows defined by the left argument vector
[4]  ⍝ of dates (in YYYYMMDD format) and the
[5]  ⍝ corresponding right argument vector of amounts.
[6]  ⍝ Positive amounts are inflows and negative amounts
[7]  ⍝ are outflows.  Requires subfunction: TODAYS.
[8]  ⍝
[9]  ⍝ Translate dates into days since Feb. 29, 0000:
[10]  DAYS←TODAYS DATES
[11] ⍝ Translate days into years (365 days per year)
[12] ⍝ since the day of the first cash flow:
[13]  YRS←(DAYS-⌊/DAYS)÷365
[14] ⍝ Precompute factors needed in formula below:
[15]  TAMTS←AMTS×YRS
[16]  TYRS←¯1+YRS
[17] ⍝ Start with an effective interest rate of 10 pct.:
[18]  V←÷1.1
[19] ⍝ Number of iterations performed so far:
[20]  I←0
[21] ⍝ Apply Newton-Raphson to get new V:
[22] LOOP:V←V-DIFF←(AMTS+.×V⋆YRS)÷TAMTS+.×V⋆TYRS
[23] ⍝ Exit if done (change less than .0000001):
[24]  →(1E¯7≥|DIFF)⍴DONE
[25] ⍝ Branch to next iteration unless 10 itns. already:
[26]  →(10>I←I+1)⍴LOOP
[27] ⍝ Else set discount factor to 1 if unknown in 10 itns.:
[28]  V←1
[29] DONE:INT←¯1+÷V
    ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you purchase a bond on February 17, 1987 for $4225. On April 1, 2003 (the maturity date), the bond will mature and you will receive $5000 (the par or redemption value). From the purchase date to the maturity date, you will receive semiannual interest payments (coupons) of $300 on April 1 and October 1 of each year. (The annual coupon rate is 12%, i.e. .12=(2×$300)÷$5000). In addition to the $4225 purchase price, you must pay the seller his portion of the upcoming (April 1) coupon. The computation of this payment (the accrued interest) is a proration based upon the number of days of the 6 month coupon period during which the bond was held by the seller. What is the internal rate of return of this investment?
~~~~

TOPIC: Bond Calculations

The internal rate of return of a bond is called its "yield to maturity". The IROR may be determined by the methods described in the previous section. That is, you may construct a vector of the 34 dates on which cash flows occur (19870217 19870401 19871001 19880401 ... 20030401) and a corresponding vector of the amounts payable on those dates. You may then use the IROR function directly.

However, since the cash flows of the bond consist of a simple annuity (the coupon payments) and a single maturity payment, a relatively simple formula exists to describe the present value of the bond's future cash flows. By taking the derivative of this forumla (tedious but not difficult) and applying the Newton-Raphson Method, the yield to maturity may be determined in a few iterations.

Suppose you need to determine the yields to maturity for an entire portfolio of bonds (say 500 bonds). If you use the former approach (the IROR function), you will need to create the date and amount vectors individually for each bond (since they may be of different lengths and may involve different dates). If you use the second approach (the formula), you may perform the successive approximation process on all the bonds at once. After a few iterations, you will have the yields for all the bonds.

The function FCYIELD ("fixed coupon yield"), listed below, uses this latter approach to compute the yield to maturity for one or more bonds. Once the yield is determined for a given bond, that bond is excluded from the successive approximation process. This function is a extremely efficient solution to a problem which is iterative by nature and is generally viewed as a "processing hog" when solved using APL.

The result of FCYIELD is a "couponly" rate (e.g. a 6 month rate for semiannual coupons). To convert the result to an effective (annual) rate, use the EFFECTIVE function presented earlier in this chapter.

One final note: this approach uses the formula for a regular annuity. However, because of leap years and months of irregular lengths, the coupon payments are not perfectly regular. The FCYIELD function (and the investment community in general) assumes that the year consists of 12 30-day months. Dates which have a day portion of 31 (e.g. 5/31/87) are treated like the 30th day of the same month (e.g. 5/30/87). Therefore, the results are not as precise as they would be if more care were taken when counting days. However, for most purposes the accuracy of the results is adequate.

```
                                          [WSID: INTEREST]
     ∇ YLD←FCYIELD PARAMS;COST;COUP;CRATE;CRY;DAYS;DIFF;F;F1;
       F2;F3;I;IND;MCOST;MDATE;MORE;MVAL;N;NCOUPS;NEW;OK;PAR;
       PDATE;RY;W;Y
[1]   ⍝ Returns couponly yield rates for fixed-coupon
[2]   ⍝ securities defined in PARAMS, one row per security.
[3]   ⍝
[4]   ⍝   PARAMS[;1]  par value
[5]   ⍝         [;2]  purchase cost excluding accrued
[6]   ⍝               interest (no coupon rec'd nor interest
[7]   ⍝               paid if purchased on a coupon date)
[8]   ⍝         [;3]  purchase date (YYYYMMDD)
[9]   ⍝         [;4]  maturity date (YYYYMMDD)
[10]  ⍝         [;5]  annual coupon rate
[11]  ⍝         [;6]  number of coupons per year
[12]  ⍝         [;7]  (optional) maturity value (par value if
[13]  ⍝               omitted)
[14]  ⍝
[15]   PAR←MVAL←PARAMS[;1]
[16]   COST←PARAMS[;2]
[17]   PDATE←PARAMS[;3]
[18]   MDATE←PARAMS[;4]
[19]   CRATE←PARAMS[;5]÷NCOUPS←PARAMS[;6]
[20]   →(6=1↓⍴PARAMS)⍴START
[21]   MVAL←PARAMS[;7]
[22]  ⍝
[23]  ⍝ Formula:
[24]  ⍝
[25]  ⍝      COST = ((1+Y)*-F)×(MVAL×(1+Y)*-W)+PAR×CRATE×1+
[26]  ⍝             (1-(1+Y)*-W)÷Y
[27]  ⍝
[28]  ⍝ Where:  Y = couponly yield rate
[29]  ⍝         F = the fraction (0<F≤1) of a coupon period
[30]  ⍝             from PDATE to the next coupon
[31]  ⍝         W = the number of whole coupon periods
[32]  ⍝             remaining from PDATE to MDATE (less 1 if
[33]  ⍝             purchased on a coupon date)
[34]  ⍝      COST = purchase cost including accrued interest
[35]  ⍝
[36]  ⍝ Convert the formula to a function in Y by moving COST
[37]  ⍝ to the right side:
[38]  ⍝
[39]  ⍝      f(Y) = (-COST)+((1+Y)*-F)×(MVAL×(1+Y)*-W)+PAR×
[40]  ⍝             CRATE×1+(1-(1+Y)*-W)÷Y
[41]  ⍝
[42]  ⍝ The problem is to determine Y for which f(Y)=0. Solve
[43]  ⍝ by the method of successive approximations, using
[44]  ⍝ different values of Y.  Start by trying Y=CRATE. Then
[45]  ⍝ use Newton-Raphson method to determine successive
[46]  ⍝ values of Y:
[47]  ⍝
[48]  ⍝   Y(N+1) = Y(N)-f(Y(N))÷f'(Y(N))
[49]  ⍝
```

```
    ∇ FCYIELD (continued)
[50] ⍝ Determine f'(Y) by taking the derivative of f(Y).
[51] ⍝ After tedious computations:
[52] ⍝
[53] ⍝   f'(Y) = ((1+Y)⋆-F)×(((1+Y)⋆¯1-W)×((W+F)×(PAR×
[54] ⍝              CRATE÷Y)-MVAL)+(PAR×CRATE×1+Y)÷Y⋆2)-
[55] ⍝              PAR×CRATE×(F÷Y)+÷Y⋆2
[56] ⍝
[57] ⍝ Let us define the following:
[58] ⍝
[59] ⍝  F1=1+Y    F2=(1+Y)⋆-F    F3=(1+Y)⋆-W    COUP=PAR×CRATE
[60] ⍝  N=F+W     RY=÷Y          CRY=COUP÷Y     MCOST=-COST
[61] ⍝
[62] ⍝ The formulas for f(Y) and f'(Y) become:
[63] ⍝
[64] ⍝    f(Y) = MCOST+F2×(MVAL×F3)+COUP+CRY×1-F3
[65] ⍝
[66] ⍝   f'(Y) = F2×((F3÷F1)×(N×CRY-MVAL)+F1×CRY×RY)
[67] ⍝              -CRY×F+RY
[68] ⍝
[69] ⍝ Compute approx days (360 days/yr) from purchase to
[70] ⍝ maturity (change 31 days to 30):
[71] START:DAYS← 360 30 1 +.× 0 100 100 ⊤MDATE-31=100|MDATE
[72]  DAYS←DAYS- 360 30 1 +.× 0 100 100 ⊤PDATE-31=100|PDATE
[73] ⍝ No. coupon periods from purchase to maturity:
[74]  N←(DAYS×NCOUPS)÷360
[75] ⍝ Fractional and whole coupons from purch to matur:
[76]  F←N-W←⌈N+¯1
[77]  COUP←PAR×CRATE
[78] ⍝ Include accrued interest (prorated) in purch cost:
[79]  COST←COST+COUP×1-F
[80]  MCOST←-COST
[81] ⍝ Start with couponly rates from approximate yield
[82] ⍝ formula:
[83]  YLD←Y←(MVAL+MCOST+N×COUP)÷(MVAL×N)+((N+1)×COST-MVAL)÷2
[84] ⍝ Indices into YLD of yields not yet known:
[85]  IND←⍳⍴YLD
[86] ⍝ Number of next iteration:
[87]  I←1
[88] ⍝
[89] LOOP:F1←1+Y
[90]  F2←F1⋆-F
[91]  F3←F1⋆-W
[92]  CRY←COUP×RY←÷Y
[93] ⍝ Apply Newton-Raphson to get new Y:
[94]  DIFF←(MCOST+F2×(MVAL×F3)+COUP+CRY×1-F3)÷F2×((F3÷F1)×(N
      ×CRY-MVAL)+F1×CRY×RY)-CRY×F+RY
[95]  NEW←Y-DIFF
[96] ⍝ Flag those found (changed less than .0000001):
[97]  OK←1E¯7≥|DIFF
[98] ⍝ Compute indices of remaining elts:
[99]  MORE←(~OK)/⍳⍴OK
[100] ⍝ Branch if no elts found:
[101]  →((⍴OK)=⍴MORE)⍴NEXT
```

```
    ∇ FCYIELD (continued)
[102] ⍝ Insert found elements:
[103]  YLD[OK/IND]←OK/NEW
[104] ⍝ Exit if no remaining elts:
[105]  →(×⍴MORE)↓END
[106] ⍝ Else, squeeze down arrays:
[107]  MVAL←MVAL[MORE]
[108]  F←F[MORE]
[109]  W←W[MORE]
[110]  N←N[MORE]
[111]  COUP←COUP[MORE]
[112]  MCOST←MCOST[MORE]
[113]  IND←IND[MORE]
[114] ⍝ Update current yields to latest values:
[115] NEXT:Y←NEW[MORE]
[116] ⍝ Branch to next iteration unless 10 itns. already:
[117]  →(10≥I←I+1)⍴LOOP
[118] ⍝ Else, set yield to 0 if unknown in 10 iterations:
[119]  YLD[IND]←0
[120] ⍝
[121] END:
    ∇
```

We will use the two approaches to solve the problem stated at the beginning of this section.

Approach 1: Using IROR

(note: purchase price includes $226.67 accrued interest)

```
      DATES←19870217,(,(10000×1986+⍳16)∘.+401 1001),20030401
      AMTS←¯4451.67,(32⍴300),5300
      DATES IROR AMTS
0.150274
```

Approach 2: Using FCYIELD

```
      Y←FCYIELD 1 6⍴5000 4225 19870217 20030401 .12 2
      2 EFFECTIVE 2×Y
0.150305
```

The slight difference between these two yield rates is the result of using an exact-days assumption (and 365 days per year) in the first approach, and a 30-days-per-month, regular annuity assumption in the second approach.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~
~~~~

PROBLEMS: (Solutions on pages 476 to 478)

1. You deposit $1000 in an 18 month Certificate of Deposit at 11% compounded daily. What will be the value of your deposit when it matures?

2. You make a deposit of $10 each week for 40 weeks into your bank's Christmas Club plan. Your bank pays 8%, converted monthly. How much will you have at the end of the 40 weeks?

3. You borrow $10,000 from your bank to buy a car. The term of the loan is 4 years and the interest rate is 14%. What will your monthly payment be? If you decide to repay the loan after 3 years, how much must you pay the bank (assuming no prepayment penalty)? How much interest will you have paid during the 3 years?

4. Your brother-in-law is opening a new restaurant and has approached you with the opportunity to invest in his venture. He is asking for an immediate outlay of $10,000 and a second outlay of $5,000 in 6 months. Starting 5 years from now, he will pay you $3,000 a year for 15 years. What is the internal rate of return of this investment (assuming all payments will be made as scheduled)?

5. When the purchase price of a bond is less than its par (face) value, the bond is said to be selling at a "discount". When its price is greater than its par value, it is selling at a "premium". In general, a bond sells at a discount when interest rates are higher than the bond's coupon rate, and sells at a premium when interest rates are lower than the bond's coupon rate. Fluctuating interest rates hence cause inverse fluctuations in the market values (purchase costs) of bonds. When interest rates are up, bond prices are down and vice versa. As the maturity date of a bond gets nearer, the fluctuations of its market value are less pronounced. On the maturity date, the market value of the bond is exactly equal to its par value, regardless of prevailing interest rates.

   When you buy a bond, its value on your books (its "book value") is the purchase cost. When the bond matures, its value on your books is the par value. These two values are not the same when you buy the bond at a discount or a premium. Since the book value of the bond changes from the purchase date to the maturity date, and since you do not want the change to appear as an abrupt change at maturity, you must "amortize" the amount of the discount or premium over time, modifying the book value of the bond accordingly.

In general, the book value of a bond on a given date is computed by using its yield-to-maturity and the present value formula in the FCYIELD function to determine the present value of the bond on the coupon dates before and after the given date and by interpolating between the two dates.

Write a function FCBOOK (fixed coupon book value) which returns the book values of a specified portfolio of bonds as of a specified date. The left argument of FCBOOK is the scalar date (YYYYMMDD) as of when the book values are to be computed. The right argument is a matrix of bond parameters with one row per bond. The 6 columns contain, respectively, par value, maturity date (YYYYMMDD), annual coupon rate, number of coupons per year, couponly yield rate (as from FCYIELD), maturity value (par value if omitted). The result is a vector of book values with one element per bond (row of the right argument).

Chapter 19

EXCEPTION HANDLING

In this chapter we discuss the concepts of exception handling in APL, and illustrate exception handling techniques on some of the popular implementations of APL.

An exception is an event which, if not handled, will cause a function to suspend. Specifically, it is an error or an attention (pressing the BREAK key). When an exception occurs and is not handled, diagnostic information displays and the user is left in immediate execution mode. In other words, the function being executed no longer has control of what will happen. For better or worse, the user must decide what action to take next.

The concept of exception handling is that facilities are provided to enable the programmer to insert code into a system which will detect an exception when it occurs and will take some action other than simple suspension.

What kinds of action is the function (i.e. the programmer) likely to take when handling an exception? There are 4 typical choices:

1. Do something and then return to immediate execution mode with the function suspended on the exception line. This is the default (unhandled) behavior where the "something" is to display diagnostic information.

2. Do something and then resume execution at the exception line. Hopefully, the "something" (e.g. allocating more disk storage) removes the cause of the exception so that the line will complete without exception this time.

3. Branch to another line of the exception function where special logic has been included to evaluate the exception and to take appropriate action. After taking such action, the function may choose to branch back to the exception line or to branch elsewhere.

4. Leave the exception function altogether by signalling an error (i.e. exception) to the environment which called the exception function. This is the behavior taken by primitive APL functions. For example, if you attempt to divide by 0, the divide primitive (÷) does not suspend within its assembler code

but rather signals an error (DOMAIN ERROR) to the function line on which divide was called:

```
DOMAIN ERROR
CALC[15]  A←B÷C
              ^
```

Likewise, it may be desirable to have the exception function (e.g. SQRT) signal an error to its calling environment:

```
DOMAIN ERROR
CALC[25]  A←SQRT B
             ^
```

The calling environment may then handle or not handle the exception as appropriate.

There are a number of different implementations of exception handling in APL. Each of these implementations takes a different approach to allow the 4 choices above. We will illustrate the 3 major exception handling implementations (APL*PLUS, SHARP APL, APL2) in this chapter.

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given a character vector named INPUT which contains a user-entered APL expression, execute the expression and return its result. If the execution of the expression (or its assignment to the result variable R) generates an error, report the error and reprompt by branching to the line labeled PROMPT.

TOPIC: Detecting the Error

In APL*PLUS, the system variable ⎕ELX (error latent expression) may be assigned any executable character vector (as with ⎕LX). The character vector is executed only if an error occurs. The niladic system function ⎕DM (diagnostic message) returns a character vector representation (with embedded newline, i.e. carriage return, characters) of the diagnostic message of the most recent exception. The default setting of ⎕ELX in a clear workspace is '⎕DM'.
~~~~

The solution to the problem using APL★PLUS:

```
[0]     ...;⎕ELX
[1]    ⎕ELX←'⎕DM'           If an error occurs on lines 2 to 14,
 :                          display the diagnostic message and
 :                          suspend.
[15]   ⎕ELX←'→ERR'          Branch to ERR if an error on line 16.
[16]   ⍎'R←',INPUT
[17]   ⎕ELX←'⎕DM'
 :
 :
[25] ERR:⎕←⎕DM              Display diagnostic message.
[26]   →PROMPT              Ask again.
```

In SHARP APL, the system variable ⎕TRAP may be assigned a delimited character vector (say, delimited by the '∇' character) or a character matrix where each partition or row specifies the action to be taken for a given class of errors. For example, the expression

```
⎕TRAP←'∇0 E →ERR'
```

says to execute (E) the expression →ERR if any (0) error occurs. The system variable ⎕ER (event report) contains a 3-row character matrix representation of the most recent exception. The first 5 characters of the first row of ⎕ER contain the "event number" (e.g. 4 for RANK ERROR). The default setting of ⎕TRAP in a clear workspace is ''.

The solution to the problem using SHARP APL:

```
[0]     ...;⎕TRAP
[1]    ⎕TRAP←''             Normal message and suspension if
 :                          error on lines 2 to 14.
 :
[15]   ⎕TRAP←'∇ 0 E →ERR'      Branch to ERR if an error on
[16]   ⍎'R←',INPUT             line 16.
[17]   ⎕TRAP←''
 :
 :
[25] ERR:⎕←5↓⎕ER[⎕IO;]     Show error message but not event
[26]   ⎕←1 0↓⎕ER            number.  Show rest of diagnostic
[27]   →PROMPT              message.  Ask again.
```

In APL2, the dyadic system function ⎕EA (execute alternate) takes an executable expression as its character vector right argument and executes it ala ⍎. If the expression can be executed without exception, the left argument of ⎕EA is not considered. However, if the execution of the right argument of ⎕EA generates an exception, the left argument is executed. The system variable ⎕EM (event message) contains a 3-row character matrix representation of the diagnostic message of the most recent exception having occurred at the current level of the state indicator.

The solution to the problem using APL2:

```
[16]  '→ERR' ⎕EA 'R←',INPUT    Branch to ERR if an error
 :                             during ⍎.
 :
[25] ERR:⎕←⎕EM         Display diagnostic message.
[26]  →PROMPT          Ask again.
```

A different APL2 solution is possible by using the monadic system function ⎕EC (execute controlled). ⎕EC is like ⎕EA in that its character vector right argument is an executable expression. However, the result of ⎕EC is a 3 item nested array which contains information about the attempt to execute the right argument. The result can be evaluated to determine whether or not an error has occurred. ⎕EC distinguishes among the various types of executable expressions and so allows messages which are more specific than those possible from ⎕EA.

See an APL2 reference manual for more complete documentation on ⎕EC. Here is the solution using ⎕EC in APL2:

```
[16]  Z←⎕EC INPUT
[17]  →ERR UNLESS Z[1]∊1 2      Branch unless result or
[18]  R←3⊃Z                     assignment.
 :
 :
[25] ERR:→(Z[1]=0 3 4 5)/ERR1,ERR2,ERR3,ERR3
[26] ERR1:⎕←⎕EM
[27]  →PROMPT
[28] ERR2:⎕←'EXPRESSION HAS NO RESULT'
[29]  →PROMPT
[30] ERR3:⎕←'BRANCHING NOT ALLOWED'
[31]  →PROMPT
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Suppose you have written a dyadic function STAT whose arguments are same-length numeric vectors and whose result is a numeric vector of various statistics. The function is self-contained, i.e. it requires no subfunctions or global variables and it assigns no global variables. Because of its syntax and self-containment, STAT behaves like a dyadic primitive function (once copied into the workspace) in all regards but one: error handling. When provided with faulty arguments, STAT suspends on one of its lines after displaying one of: RANK ERROR, LENGTH ERROR or DOMAIN ERROR. Rewrite STAT to signal these errors and others (say, WS FULL) to its calling environment rather than suspending.
~~~~

TOPIC: Signalling the Error

In APL*PLUS, the monadic system function ⎕ERROR takes a character vector error message argument and signals that message to the environment from which was called the function in which ⎕ERROR is executed. For example:

```
          ∇ TEST N
    [1]   ⎕ERROR 'OOPS'
    [2]   ∇
          TEST 5
    OOPS
          TEST 5
          ∧
```

Once the error occurs within STAT, the message may be found as the first line of the result of ⎕DM. The first line is defined as all characters up to the first newline character (represented in APL*PLUS as ⎕TCNL).

The solution to the problem using APL*PLUS:

```
    [0]    ...;⎕ELX
    [1]   ⎕ELX←'⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM'
```

In SHARP APL, the dyadic system function ⎕SIGNAL takes an optional character vector error message left argument and a corresponding event number right argument and signals the message (and event number as the first 5 characters of ⎕ER) to the environment from which was called the function in which ⎕SIGNAL is executed. For example:

```
          ∇ TEST N
    [1]   'OOPS' ⎕SIGNAL 599
    [2]   ∇
          TEST 5
    OOPS
          TEST 5
          ∧
```

The function ⎕SIGNAL may also be used monadically. Its argument is an integer event number in the range 1 to 999 (e.g. 2 for the standard APL error message SYNTAX ERROR).

The solution to the problem using SHARP APL:

```
    [0]    ...;⎕TRAP
    [1]   ⎕TRAP←'∇0 E ⎕SIGNAL⍎5ρ⎕ER'
```

or, using ⎕EC (environment condition):

```
    [0]    ...;⎕EC
    [1]   ⎕EC←1 ⍝ Disallow suspension
```

In APL2, the monadic system function ⎕ES (event simulation) takes a character vector error message argument or a 2-element integer vector event type code (e.g. 1 3 for WS FULL) and signals the message to the environment from which was called the function in which ⎕ES is executed. For example:

```
          ∇ TEST N
    [1]     ⎕ES 'OOPS'
    [2]     ∇
          TEST 5
    OOPS
          TEST 5
          ^
```

The system variable ⎕ET (event type) contains the 2-element integer vector event type code of the most recent exception having occurred at the current level of the state indicator.

The solution to the problem using APL2:

```
    [1]     '⎕ES ⎕ET' ⎕EA 'line 1 of STAT'
    [2]     '⎕ES ⎕ET' ⎕EA 'line 2 of STAT'
    [3]     '⎕ES ⎕ET' ⎕EA 'line 3 of STAT'
     :               :
     :               :
```

If self-containment were not an issue, the solution using APL2 would be to rename STAT to STAT1 and to write a new STAT function:

```
         ∇ R←A STAT B
    [1]     '⎕ES ⎕ET' ⎕EA 'R←A STAT1 B'
         ∇
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given a 70-line function of critical code named PROCESS, incorporate exception handling such that an interrupt (the BREAK key) will cause the message PLEASE BE PATIENT to display and execution to resume.

TOPIC: Detecting the Attention

In APL*PLUS, the system variable ⎕ALX (attention latent expression) may be assigned any executable character vector (as with ⎕LX or ⎕ELX). The character vector is executed only if an attention is signalled (via the BREAK key). The default setting of ⎕ALX in a
~~~~

clear workspace is '⎕DM'. That is, the default behavior of the system is to display the diagnostic message generated by the attention and to suspend.

The niladic system function ⎕LC (line counter) returns an integer vector of the numbers of the suspended or pendent lines in the state indicator. The first number corresponds to the top (most recent) level in the state indicator and the last number corresponds to the bottom level. Since the branch primitive function (→) only considers the first element of its argument, the expression →⎕LC will cause the flow of execution to proceed to the line on which the expression was executed.

The solution to the problem using APL*PLUS (⋄ is a statement separator):

```
[0]    ...;⎕ALX
[1]   ⎕ALX←'⎕←''PLEASE BE PATIENT'' ⋄ →⎕LC'
```

In SHARP APL, the event number 1000 represents any interrupt. The solution using SHARP APL:

```
[0]    ...;⎕TRAP
[1]   ⎕TRAP←'∇1000 E ⎕←''PLEASE BE PATIENT'' ⋄ →⎕LC'
```

In APL2, an attention is not considered an exception which can be handled. Therefore, this problem as stated cannot be solved using APL2. If the PLEASE BE PATIENT message is omitted from the problem, you may solve the problem in APL2 by "conditioning" the PROCESS function to not be interruptable:

```
0 0 1 0 ⎕FX ⎕CR 'PROCESS'
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: You wish to install a fully-tested production system such that if any unexpected (i.e. unhandled) error occurs, the message "AN UNEXPECTED ERROR HAS OCCURRED; CONTACT J. SMITH IMMEDIATELY" will display and the system will suspend on the exact line of the error. How will you modify the system to do so?

TOPIC: Suspending the Function

In APL*PLUS, the default behavior of the system when handling an exception is to suspend on the exception line. In fact, if ⎕ELX is
~~~~

set to '', that is all that will happen. Since this behavior would be confusing to the user (suddenly in immediate execution mode with no indication of an exception), the default setting of ⎕ELX is '⎕DM'. With this setting, the diagnostic message is returned and displayed and then the function is suspended. The only way to not suspend the function is to branch (e.g. ⎕ELX←'→ENTER') so that execution may resume or to signal an error (e.g. ⎕ELX←'⎕ERROR ''OOPS''') which may be handled by a more global ⎕ELX.

The solution to the problem using APL*PLUS is to assign ⎕ELX globally:

```
          ⎕ELX←'⎕←''AN UNEXPECTED...IMMEDIATELY'''
```

If an error occurs, ⎕ELX will be executed causing the message to display. Then execution will suspend. When Smith arrives, he will type ⎕DM to display the diagnostic message and may resume execution by typing →⎕LC after correcting the error.

If a function lacks exception handling but is a high security function, you may not want to allow the function to be left suspended after an exception. For example, a function which updates a payroll file may contain sensitive salary information in its local variables during execution. To prevent a function from suspending (i.e. to perform a → when entering immediate execution mode) in APL*PLUS, localize the system variable ⎕SA (stop action) and assign it the value 'EXIT'. The default setting of ⎕SA is ''.

In SHARP APL, if an exception is not handled by the current setting of ⎕TRAP, the diagnostic message will display and the function will suspend on the exception line. As with APL*PLUS, the only way to not suspend the function is to branch (e.g. ⎕TRAP←'∇0 E →ENTER') so that execution may resume or to signal an error (e.g. ⎕TRAP←'∇0 E ''OOPS'' ⎕SIGNAL 599') which may be handled by a more global ⎕TRAP.

The solution to the problem using SHARP APL is to assign ⎕TRAP globally:

```
          ⎕TRAP←'∇0 E ⎕←''AN UNEXPECTED...IMMEDIATELY'''
```

If an error occurs, the message will display and execution will suspend. When Smith arrives, he will type ⎕ER to display the error report and may resume execution by typing →⎕LC after correcting the error.

To prevent a high security function from suspending in SHARP APL, include the partition '∇2001 D EXIT' in the current (local) definition of ⎕TRAP. Event number 2001 represents the "immediate execution mode" event. D stands for "do".

In APL2, if an exception occurs outside the execution of the right argument to ⎕EA (or ⎕EC, a similar function), the diagnostic message will display and the function will suspend on the execution line. To keep the diagnostic message from displaying, each statement which may generate an error must be executed within the right argument of ⎕EA, or must be on a line of a function which is executed within the right argument of ⎕EA, or must be on a line of a function called by a function called by ⎕EA, and so on.

Therefore, if COVERFN is the name of a function to which all of the other functions in the system are subfunctions, you may control the display of the diagnostic message by invoking the system with an expression such as:

```
'HANDLER' ⎕EA 'COVERFN'
```

Unfortunately, if an unhandled exception occurs anywhere within COVERFN, a suspension will not occur. Rather, the levels of the state indicator associated with COVERFN will be reset and HANDLER will be executed. Hence, we can choose either to detect the error but lose the suspension or to suspend by not detecting the error. For example, the expression

```
'⎕←''AN UNEXPLAINED...IMMEDIATELY' ⎕EA 'COVERFN'
```

will cause the proper message to display. However, when Smith arrives, he will type ⎕EM to display the event message and will find that it is empty and the state indicator is empty (unless an old suspension is lying around).

To prevent a high security function from suspending in APL2, you may "condition" the function (say, PAYROLL) to not be suspendable:

```
0 1 0 0 ⎕FX ⎕CR 'PAYROLL'
```

~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~ ~~~~

PROBLEM: Given a 20-line function MAINFN which calls a multitude of subfunctions, you would like to incorporate exception handling in MAINFN such that any error (even if in a subfunction with no exception handling) causes a branch to the line labeled ERRCODE in MAINFN and any attention causes a branch to the line labeled ATTNCODE.

TOPIC: Controlling the State Indicator

In APL*PLUS, there is no direct capability for solving this problem. Suppose MAINFN calls SETUP which calls READVARS which encounters an error on its 5th line. If there is no exception handling, the READVARS function will suspend and the state indicator will look something like:
~~~~

```
          )SI
     READVARS[5] *
     SETUP[19]
     MAINFN[4]
```

A naive solution to the problem would be to include the following in MAINFN:

```
          ∇ MAINFN;....;⎕ALX;⎕ELX
     [1]    ⎕ALX←'→ATTNCODE'
     [2]    ⎕ELX←'→ERRCODE'
      :
      :
     [31] ERRCODE:etc.
      :
     [41] ATTNCODE:etc.
```

However, when the error in READVARS occurs, the value of ⎕ELX ('→ERRCODE') will be executed, causing a branch to line 31 (the value of ERRCODE unless shadowed by a different ERRCODE local to SETUP or READVARS) of READVARS, not to line 31 of MAINFN. Somehow, we must get out of (i.e. →0) both READVARS and SETUP before branching to the line labeled ERRCODE. The only way to do this without landing in immediate execution mode (which would happen if you localized ⎕SA←'EXIT' in both READVARS and SETUP) is to use ⎕ERROR.

The approach we will take is this: set ⎕ELX to check whether ⎕ELX has been localized at the current top level of the state indicator; if so (i.e. if in MAINFN), branch to ERRCODE; if not (i.e. if in SETUP or READVARS), use ⎕ERROR to reset the top level of the state indicator and to signal an error (say, the same error message) to the next level; since ⎕ERROR will trigger ⎕ELX, the process will be repeated until MAINFN is at the top of the state indicator. This process is called "propagating the error message".

The monadic system function ⎕IDLOC (identifier localization) returns a 1 row integer matrix of localization codes for the identifier whose name is provided as the character vector right argument. Each column of the result corresponds to one level of the state indicator (local to global) and the code ¯1 represents unlocalized. Therefore, if the result of 1ρ⎕IDLOC '⎕ELX' is ¯1, then ⎕ELX is not localized at the top level of the state indicator.

The initial solution to the problem using APL*PLUS:

```
          ∇ MAINFN;...;⎕ELX
     [1]    ⎕ELX←'⍎(¯1=1ρ⎕IDLOC''⎕ELX'')/''⎕ERROR(∧\⎕DM≠⎕TCNL)/
            ⎕DM''⋄→ERRCODE'
```

This handles errors. What about attentions? We will use a similar approach: set ⎕ALX to check whether ⎕ALX has been localized at the current top level of the state indicator; if so (i.e. if in MAINFN), branch to ATTNCODE; if not (i.e. if in SETUP or READVARS), use ⎕ERROR to reset the top level of the state indicator and to signal an error (say 'ATTN') to the next level; ⎕ELX will be triggered at the next level and will behave as described above except it will branch to ATTNCODE rather than ERRCODE if the error message is 'ATTN'.

The final solution to the problem using APL*PLUS:

```
      ∇ MAINFN;...;⎕ALX;⎕ELX
  [1]   ⎕ALX←'⍎(¯1=1⍴⎕IDLOC''⎕ALX'')/''⎕ERROR''''ATTN''''''
        ⋄→ATTNCODE'
  [2]   ⎕ELX←'⍎(¯1=1⍴⎕IDLOC''⎕ELX'')/''⎕ERROR(∧\⎕DM≠⎕TCNL)/
        ⎕DM''⋄→(''ATTN''∧.=4↑⎕DM)⌽ERRCODE,ATTNCODE'
```

In SHARP APL, there is a direct capability for solving this problem. The action code C (cut) can be specified within the ⎕TRAP definition to specify that the state indicator should be "cut back" (i.e. reset) to the level at which ⎕TRAP is local.

The solution to the problem using SHARP APL:

```
      ∇ MAINFN;...;⎕TRAP
  [1]   ⎕TRAP←'∇0 C →ERRCODE ∇1000 C →ATTNCODE'
```

In APL2, the only way to solve the problem is to execute each of the 20 lines of MAINFN as the right argument of ⎕EA. If an exception occurs during the execution of the line, the state indicator is automatically "cut back" so that MAINFN is at the top level; then the left argument of ⎕EA is executed. Since attentions cannot be detected as exceptions in APL2, no branch to ATTNCODE is possible.

The solution to the problem using APL2:

```
      ∇ MAINFN;...;ELX
  [1]   ELX←'→ERRCODE'
  [2]   ELX ⎕EA 'line 1 of MAINFN'
  [3]   ELX ⎕EA 'line 2 of MAINFN'
  [4]   ELX ⎕EA 'line 3 of MAINFN'
   :          :
   :          :
  [26]  ERRCODE:etc.
```

The only problem with this solution is its brute force appearance. A typical APL2 solution to the problem involves restating the problem: rename MAINFN to be MAINFN1 and write a new MAINFN which will handle any exceptions occurring within MAINFN1.

Here is the alternative APL2 solution:

```
      ∇ MAINFN
  [1]   '→ERRCODE' ⎕EA 'MAINFN1'
  [2]   →0
  [3]  ERRCODE:etc.
```

PROBLEMS: (Solutions on pages 479 to 482)

1. The following function displays the records of a file. The user is asked for the number of the record at which displaying is to begin. Then the function reads and displays every record from the specified record to the end of the file. Incorporate exception handling so that the user may press the BREAK key to halt the display and to ask again for the number of the record at which displaying is to begin. Assume that the READ function uses exception handling such that an attention within READ causes READ to signal the error message 'ATTN' to its calling environment.

```
     ∇ SHOWFILE NAME;LIM;N
[1]    LIM←TIEFILE NAME
[2]   ASK:⎕←'BEGIN WITH WHICH RECORD?'
[3]    N←⎕
[4]    →(0=N)/0
[5]   LOOP:⎕←''
[6]    ⎕←(⍕N),': ',READ N
[7]    ⎕←''
[8]    →(LIM≥N←N+1)/LOOP
     ∇
```

2. Write a dyadic function NXPROMPTE which prompts for and returns a vector of numbers. Its right argument PROMPT is a character vector which is displayed to prompt the user for input. Input is accepted beyond the prompt on the same line. The left argument is the number of numbers required. If 0, the function will allow any number of numbers. If the word END is entered, the result is the scalar 1. Any primitive APL expression may be entered and will be executed. User defined functions or variables may not be included in the response. Note that this function is identical to the NPROMPTE function developed in the chapter, Writing User-Friendly Interactive Functions, except APL expressions are allowed. For example:

```
      Q←0 NXPROMPT 'ENTER PROJECT NO.S: '
ENTER PROJECT NO.S: 10+⍳8
      Q
11 12 13 14 15 16 17 18
```

3. Write a dyadic function ERRATTN to initialize the exception handling facilities such that any subsequent error will be handled as directed by its arguments. If the left argument, ELX, is a character vector, it will be executed directly if any error should occur; if the left argument is a numeric line label, the state indicator will be reset (cut back) to the level at which ERRATTN is called and a branch to the scalar will take place. The meaning of the right argument, ALX, is the same as that of the left argument but is for attention handling rather than error handling. If either argument is empty (e.g. ''), the response to the respective exception is to display the diagnostic message and suspend. For example:

```
[1]   '' ERRATTN ''
 :
 :
[6]   'FIXFILE ◇ →⎕LC' ERRATTN L5
[7]  L4:APPEND DATA
[8]   →MORE
[9]  L5:⎕←'FILE INCOMPLETE DUE TO INTERRUPT.'
 :
 :
```

On line 1, the error and attention handling is set to its default behavior (if an exception, display diagnostic message and suspend). On line 6, a character vector error handler and a numeric line label attention handler are provided. If an error occurs (say, within the APPEND function), the FIXFILE function is executed and execution is resumed. If an attention occurs, execution is resumed on the line labeled L5 (even if the error occurs within the APPEND function).

POSTSCRIPT

The functions described and listed in this book are available on a floppy disk for the APL*PLUS PC system. To order one or more of these floppy disks, please send your check or money order to:

ADVANCED APL FUNCTIONS
ZARK INCORPORATED
53 SHENIPSIT STREET
VERNON, CT 06066

Specify the number of disks desired and enclose $15 per disk. Postage and handling charges are included. Connecticut residents, please include sales tax.

The following is a list of the workspaces and functions included on the disk.

```
      )WSID BOOLEAN
pANDMAP   pANDRED   pANDSCAN  pEQMAP    pEQSCAN   pGEMAP    pGTMAP
pLESCAN   pLTSCAN   pNEMAP    pNESCAN   pORMAP    pORRED    pORSCAN
pPLUSRED
```

```
      )WSID CASHBAL
CASH1  CASH2  CASH3  CASH4
```

```
      )WSID CNFNS
ASSIGN     CNCAT      CNEQ       CNEST      CNGRADEUP  CNIDX
CNIDXA     CNIOTA     CNLEN      CNΔM       CNΔV
```

```
      )WSID COMMENTS
UNCOMMENT  UNLAMP
```

```
      )WSID CRTIMING
TIMER        TIMEΔDEFINE
```

```
      )WSID DATES
FROMDAYS     FROMDAYS360  FROMDAYS∆    FROMMDY      FROMMDY∆
FROMQTS      FROMYD       IPDATEMDY    TODAYS       TODAYS360
TODAYS∆      TOMDY        TOMDY∆       TOQTS        TOYD

      )WSID ERROR
ERRATTNP  ERRATTNS

      )WSID FILEDOC
FILEDOC

      )WSID FLF
EMPLOYEES

      )WSID FNIDS
IDENTIFY     LOCALIZE     OBFUSCATE    RELABEL      UNDIAMOND
UNOBFUSCATE

      )WSID FNREP
CR∆VR  VR∆CR

      )WSID FNSFILE
DROPFN    FNCREATE  GETFN     PUTFN

      )WSID FORMAT
CENTER     CJUST      COLFMT     COLUMNIZE  DEB        DLB
DTB        HEADINGS   LJUST      RJUST      ROWFMT     THORN
TITLES

      )WSID INPUT
CPROMPT    CPROMPTE   ESCAPE     IF         LPROMPT    LPROMPTE
MESSAGE    NINPUT     NINPUT2    NPROMPT    NPROMPT2   NPROMPTE
NPROMPTE2  NXPROMPTE  PROPOSAL   UNLESS

      )WSID INTEREST
EFFECTIVE  FCBOOK     FCYIELD    IROR       NOMINAL    SCHEDULE
VALUE

      )WSID LOOP
LOOPI  NEXTI
```

)WSID MSF

ADDEMP	CATEMP	CINPUT	DELEMP	EMPLOYEES	IF
LISTEMP	MESSAGE	NINPUT	RCAT	RESTART	SELECT
SQZEMP	START	UNLESS			

)WSID MULTI2

ASSIGN	CATREC	CATRECWS	COMPRESS	DELREC
EXECUTE	FCREATE	FERASE	FOR	FREAD
FREPLACE	FTIE	FUNTIE	INDEX	INDEXA
INDEXWS	INDEXWSA	INITFILE	IOTA	IOTARHO
LAYERS	NREC∆RECL	SELECT	SELECTWS	SLASHIOTARHO

)WSID MULTIFLO

ASSIGN	CATREC	CATRECWS	COMPRESS	DELREC
EMPLOYEES	EXECUTE	FOR	INDEX	INDEXA
INDEXWS	INDEXWSA	INITFILE	IOTA	IOTARHO
LAYERS	SELECT	SELECTWS	SLASHIOTARHO	

)WSID MULTISA

ASSIGN	CATREC	CATRECWS	COMPRESS	DELREC
EXECUTE	FOR	INDEX	INDEXA	INDEXWS
INDEXWSA	INITFILE	IOTA	IOTARHO	LAYERS
SELECT	SELECTWS	SLASHIOTARHO		

)WSID NNFNS

ASSIGN	NNCAT	NNCATSS	NNCATSV	NNCATVS	NNCATVV	NNEST
NNIDX	NNIDXA	NNLEN	NNSUMCOL			

)WSID PRTFILE

PRINT

)WSID QDOC

QDOC

)WSID REDUCE

ANDRED	MAXRED	MINRED	ORRED	PLUSRED	∆AND	∆ANDRED
∆ANDWAY	∆MAX	∆MAXRED	∆MAXWAY	∆MIN	∆MINRED	∆MINWAY
∆OR	∆ORRED	∆ORWAY	∆PLUS	∆PLUSRED	∆PLUSWAY	

)WSID SEARCH

BY	CMIOTA	CMIOTA1	CMIOTA2	DEB	IOTA	LIOTA
LIOTA1	REPLACE	UIOTA	UIOTA1	UNQCM	UNQCV	UNQI0
UNQI1	UNQNV	∆SS				

```
      )WSID SORT
CGRADEUP   CGRADEUP1   CGRADEUP2   UPPERCASE


      )WSID TIMING
COST          TIMER          TIME∆DEFINE   TIME∆DISPLAY   TIME∆RESET
⍙


      )WSID USEDBY
USEDBY


      )WSID UTILITY
MONIOTA   REPL


      )WSID WP
WRAP    WRAPLP


      )WSID WSDOC
WSDOC
```

Chapter 1 Solutions

LIMBERING UP

1. `AMOUNT[AMOUNTι645]←845` (if exactly one occurrence of 645)

   or

   `AMOUNT[(AMOUNT=645)/ιρAMOUNT]←845` (if 0 or more occurrences)

   or

   `((AMOUNT=645)/AMOUNT)←845` (APL2)

2. `∧/(PREMS>100)∧PREMS<500`

3. `+/24=⌊0.5+WEIGHT`

4. `×/ρMAT` or `ρ,MAT`

5. `'ANSWER IS ',(⍕ANS),' YEARS'`

6. `NAMES←NAMES,[1](1↓ρNAMES)↑NAME` (origin 1)

7. A. Since a scalar has no shape, its shape is an empty vector. Therefore, the result of ρ12 is an empty vector. When the right argument of branch (→) is an empty vector, no branch takes place. Control proceeds to the next statement.

8.
```
        R←(((ρV)ρ1 0)/V)+256×((ρV)ρ0 1)/V
    or
        R←256⊥⍉⌽(N,2)ρV
    or
        R←((N,2)ρV)+.×1 256
    or
        R←+/(N,2)ρV×(ρV)ρ1 256
```

9. The reduction of an empty vector returns the identity element for the dyadic function involved in the reduction. The identity element is that value which when supplied as one of the arguments of the dyadic function will return the other argument. For example, since 0+5 is 5 and 1×5 is 5, the identity element for plus (+) is 0 and for times (×) is 1. The only argument to minimum (⌊) which will always return the other argument is positive infinity. Since positive infinity cannot be represented as a number, APL returns the next best thing, the largest possible number which can be represented on the computer. This value varies from implementation to implementation but is generally some value like 1E77 or 2E300. For non-commutative functions (like ÷), the identity element is that value which when supplied as the right argument of the dyadic function will return the left argument. For example, since 5÷1 is 5, the identity element for divide is 1.

10. The running alternating sum.

```
              -\ 3 8 6 5
        3 ¯5 1 ¯4          (3),(3+(-8)),(3+(-8)+6),(3+(-8)+6+(-5))
```

11. Display the state indicator (via ")SI") to see where the suspension occurred and branch to the line number shown on the top of the state indicator. Alternately:

```
              →⎕LC
```

12. Most implementations of APL insist that the header of a function not be changed once the function is called. It is too tricky to handle the problems which arise when you make a global variable local, or vice versa, while the function is suspended. Likewise, labels may not be added or deleted (or possibly moved to different lines) in a suspended function, since the values of

labels are assigned at the moment the function is called. Most implementations of APL will allow you to make such editing changes to a suspended function but will display a message like SI DAMAGED or SI ERROR or SI WARNING and will not allow you to resume execution at the point of the error. You will need to reset the state indicator and rerun the function from the beginning.

13. The "rank" or "ranking" of VECTOR. For example, if 6 students have these test scores,

```
            SCORES←87 99 83 85 65 100
```

the ranks of the respective students are:

```
            ⍋⍋SCORES
      4 5 2 3 1 6
```

where the 1 corresponds to the student with the lowest score and the 6 corresponds to the student with the highest score.

14.a. `(¯1↓⍴A),1↓⍴B`

b. ditto

15.a. `(⍴A),⍴B`

b. ditto

16. ⎕AI (accounting information), a niladic system function. Usually the second element of the result of ⎕AI is a measure of CPU time consumed since signon to APL.

17.
```
      ⍞←PROMPT                         ⍞←PROMPT
      R←(⍴PROMPT)↓⍞          (or:      ⎕ARBOUT ⍳0     in APL*PLUS)
                                       R←⍞
```

```
18.a.      (N,1)ρ' '               (or:  (N,0)ρ''  in APL2)

   b.      (N-1)ρ⎕TCNL             (APL*PLUS)
           (N-1)ρ⎕AV[156+⎕IO]      (SHARP APL)
           (N-1)ρ⎕TC[1+⎕IO]        (APL2)
```

Note: None of these three expressions works correctly if N=0. They generate a DOMAIN ERROR.

```
19.        ⎕EX ⎕NL 2               (APL*PLUS, SHARP APL, APL2)
           ⎕ERASE ⎕IDLIST 2        (APL*PLUS)
           6 ⎕FD 1 ⎕WS 2           (SHARP APL)
```

```
20.a.      S∆MODEL←12 14           (APL*PLUS, SHARP APL, APL2)
           12 14 ⎕STOP 'MODEL'     (APL*PLUS PC, APL*PLUS UNX)
```

b. Localize T in INTERPOLATE. T was somehow reassigned after line 11 and before line 13. Since the reassignment does not take place on line 12, it must take place within INTERPOLATE (or a subfunction of INTERPOLATE).

21. Including the header in the count of lines,

```
           1↑ρ⎕CR 'CALC'                      (APL*PLUS, APL2)
           1↑ρ2 ⎕FD 'CALC'                    (SHARP APL)
           ¯1++/⎕TCNL=⎕VR 'CALC'              (APL*PLUS)
           ¯1++/⎕AV[156+⎕IO]=1 ⎕FD 'CALC'     (SHARP APL)
```

22. Because ⎕IO=0 and some elements of V2 are not found in V1.

Chapter 2 Solutions

BRANCHING AND LOOPING

1. The expression does not work correctly in index origin 0; it modifies the random link (⎕RL); and it labels the programmer as having brain damage.

2.
```
      →(×N)⌽ZERO,POSITIVE,NEGATIVE
or
      →(NEGATIVE,ZERO,POSITIVE)[2+×N]
or
      →(¯1 0 1=×N)/NEGATIVE,ZERO,POSITIVE
```

3. a.
```
 SUM←0
 I←11
LOOP:→ENDLOOP IF I>308
 SUM←SUM+READ I
 I←I+3
 →LOOP
ENDLOOP:
```

b.
```
 SUM←0
 CMPS←8+3×⍳100
 I←1
 LAB←(100⍴LOOP),ENDLOOP
LOOP:SUM←SUM+READ CMPS[I]
 I←I+1
 →LAB[I]
ENDLOOP:
```

c.
```
 SUM←0
 I⍎ENDLOOP,100 11 3
 SUM←SUM+READ I
 ⍎I
ENDLOOP:
```

d.
```
 SUM←0
 →LOOPI ENDLOOP,100 11 3
 SUM←SUM+READ I
 →NEXTI
ENDLOOP:
```

e.
```
 SUM←0
 SUMUP¨ 8+3×⍳100
```
where:
```
     ∇ SUMUP CMP
[1]   SUM←SUM+READ CMP
     ∇
```

4.

```
⎕IO←1
RSCAN←(1+RATE)*⍳TERM
OPRIN←RSCAN×LOAN-+\PMT÷RSCAN
```

5. After performing the transformations, the result is DEPOSIT. Undoing the transformations yields the following function:

```
                                        [WSID: CASHBAL]
    ∇ BALANCE←RATE CASH4 DEPOSIT;ACCUM
[1]   ⍝ Returns stream of cash balances for deposits
[2]   ⍝ DEPOSIT and corresponding rates RATE.
[3]   ⍝ Performs:
[4]   ⍝  BALANCE[I]←DEPOSIT[I]+BALANCE[I-1]×RATE[I-1]+1
[5]    ACCUM←⌽×\1,⌽1+RATE
[6]    BALANCE←(+\DEPOSIT×ACCUM)÷ACCUM
    ∇
```

Note to actuaries: The approach taken here is to compute the future value of each deposit as of the last period, subtotal the future values, and then discount each subtotal back to the deposit date. Alternately, the approach in CASH3 is to compute the present value of each deposit as of the start of the first period, subtotal the present values, and then accumulate each subtotal back to the deposit date.

6. Notice that the WRAP function below will iterate only as many times as the number of lines generated by the longest sentence.

```
                                             [WSID: WP]
    ∇ R←WID WRAP CVEC;⎕IO;BL;BREAK;LAST;LEN;MORE;NL;START;
       TCNL
[1]   ⍝ Wraps text CVEC into lines of length WID
[2]   ⍝ or less by inserting newline characters.
[3]   ⍝ Origin 1:
[4]    ⎕IO←1
[5]   ⍝ Newline character:
[6]    TCNL←⎕TCNL ⍝ APL*PLUS
[7]   ⍝ TCNL←⎕TC[2] ⍝ APL2
[8]   ⍝ TCNL←⎕AV[157] ⍝ SHARP APL
[9]   ⍝ Flag newline characters:
[10]   NL←CVEC=TCNL
[11]  ⍝ Index before start of each sentence:
[12]   START←0,NL/⍳⍴NL
[13]  ⍝ Lengths of sentences (between newlines):
[14]   LEN←¯1+(1↓START,1+⍴CVEC)-START
```

```
    ∇ WRAP (continued)
[15] ⍝ Flag valid break points (blank followed by
[16] ⍝ nonblank):
[17]  BL←CVEC=' '
[18]  BREAK←BL>1⌽BL
[19] ⍝ Initialize result from argument:
[20]  R←CVEC
[21] ⍝ Flag sentences still to be broken:
[22] LOOP:MORE←LEN>WID
[23] ⍝ Select just those remaining:
[24]  LEN←MORE/LEN
[25] ⍝ Exit if none left:
[26]  →(0=⍴LEN)/0
[27]  START←MORE/START
[28] ⍝ Find last break point within WID chars of line:
[29]  LAST←+/∨\BREAK[START∘.+⌽⍳WID]
[30] ⍝ Advance start to new break point:
[31]  START←START+LAST
[32] ⍝ Insert newlines:
[33]  R[START]←TCNL
[34] ⍝ Decrement remaining lengths:
[35]  LEN←LEN-LAST
[36] ⍝ Repeat:
[37]  →LOOP
    ∇
```

Chapter 3 Solutions

COMPUTER EFFICIENCY CONSIDERATIONS

1.

```
                                        [WSID: TIMING]
     ∇ COST;T
[1]   ⍝ Displays dollars consumed since COST was last
[2]   ⍝ run (as recorded in global ∆AI) and since
[3]   ⍝ signon, assuming 75 cents per unit of ⎕AI[2].
[4]   ⍝ Record time consumed so far:
[5]    T←⎕AI[1+⎕IO]
[6]   ⍝ Check for global ∆AI and branch unless found:
[7]    →(×⎕NC '∆AI')↓L1
[8]   ⍝ Display consumption since last use of COST:
[9]    (10 2 ⍕0.75×T-∆AI),' DOLLARS CONSUMED'
[10]  ⍝ Display consumption since signon:
[11]  L1:(10 2 ⍕0.75×T),' DOLLARS SINCE SIGNON'
[12]  ⍝ Reassign global ∆AI:
[13]   ∆AI←T
     ∇
```

2.

```
                                        [WSID: CRTIMING]
     ∇ ⍙R⍙←⍙N⍙ TIMER ⍙C⍙;⍙A⍙;⍙B⍙;⍙F⍙;⍙G⍙
[1]   ⍝ Times the execution of the character vector ⍙C⍙
[2]   ⍝ by running it ⍙N⍙ times.  Returns a numeric scalar
[3]   ⍝ of the average CPU time consumed per run.
[4]   ⍝
[5]   ⍝ Prepare to build local functions...
[6]   ⍝ No. of columns in canonical representation:
[7]    ⍙B⍙←24⌈4+⍴,⍙C⍙
[8]    ⍙A⍙←(7,⍙B⍙)⍴' '
[9]    ⍙A⍙[⎕IO;]←⍙B⍙↑'⍙E⍙←⍙F⍙ ⍙N⍙;⍙I⍙'
[10]   ⍙A⍙[⎕IO+1;]←⍙B⍙↑'⍙E⍙←⎕AI[1+⎕IO]'
[11]   ⍙A⍙[⎕IO+2;]←⍙B⍙↑'⍙I⍙←0'
[12]   ⍙A⍙[⎕IO+3;]←⍙B⍙↑'⍙L⍙:→(⍙N⍙<⍙I⍙←⍙I⍙+1)⍴⍙Z⍙'
[13]   ⍙A⍙[⎕IO+4;]←⍙B⍙↑'⍙D⍙:',⍙C⍙
[14]   ⍙A⍙[⎕IO+5;]←⍙B⍙↑'→⍙L⍙'
[15]   ⍙A⍙[⎕IO+6;]←⍙B⍙↑'⍙Z⍙:⍙E⍙←⎕AI[1+⎕IO]-⍙E⍙'
[16]  ⍝
```

```
     ∇ TIMER (continued)
[17] ⍝ Define local fn ∆F∆ to run ∆C∆:
[18]  ∆R∆←⎕FX ∆A∆
[19] ⍝
[20] ⍝ Define local fn ∆G∆ to run nothing:
[21]  ∆A∆[⎕IO;5+⎕IO]←'G'
[22]  ∆A∆[⎕IO+4;]←∆B∆↑'∆D∆:'
[23]  ∆R∆←⎕FX ∆A∆
[24] ⍝
[25] ⍝ Run the functions (disallow negative result):
[26]  ∆R∆←0⌈(∆F∆ ∆N∆)-∆G∆ ∆N∆
[27] ⍝ Return the average:
[28]  ∆R∆←∆R∆÷∆N∆
     ∇
```

3.

```
     ∇ T←TRY SIZES;L;R;I
[1]  ⍝ Use as:  TRY 50 100   for 50 row left arg, 100 right
[2]   L←SIZES[1]
[3]   R←SIZES[2]
[4]   L←(L,12)⍴⎕AV[?(L×12)⍴256]
[5]   R←L[?R⍴1⍴⍴L]
[6]  ⍝ Run it 5 times:
[7]   T←5 TIMER 'I←L CMIOTA R'
     ∇
```

Define the numbers of rows for the left arguments:

```
L←5/10 50 100 500 1000
```

and for the right arguments:

```
R←25⍴10 50 100 500 1000
```

Then time them all:

```
     ∇ T←L DOIT R
[1]   T←(⍴L)⍴0
[2]   I←0
[3]  LOOP:→((⍴L)<I←I+1)⍴0
[4]   T[I]←TRY L[I],R[I]
[5]   →LOOP
     ∇

T1←L DOIT R
```

Change CMIOTA as instructed (to activate the looping algorithm) and run DOIT again:

```
∇ CMIOTA
(edit it)
∇

T2←L DOIT R
```

The variables L, R, T1 and T2 will be needed in a problem at the end of the Curve Fitting chapter. Record and save them:

```
)SAVE CMTIMES
```

Chapter 4 Solutions

POSITIONING CHARACTER DATA

1.
```
      R←(∧\TEXT≠NL)/TEXT
  or
      R←((TEXTιNL)-⎕IO)ρTEXT
  or
      R←(+/∧\TEXT≠NL)ρTEXT
```

2.
```
      R←(CODE='/')∧'ι'=1↓CODE,'*'

  (What is wrong with:   R←(CODE='/')∧'ι'=1⌽CODE    ?)
  (Try it on:  CODE←'ιI←B/ιρB ⍝ USES /' )

      R←CODE ⎕SS '/ι'          (APL*PLUS)
      R←'/ι'⍷CODE              (APL2)
```

3.
```
                                          [WSID: FORMAT]
        ∇ R←W CENTER C
      [1]  ⍝ Pads character vector C to width W, centering
      [2]  ⍝ it within that width.
      [3]   R←W↑((⌊(0⌈W-ρ,C)÷2)ρ' '),C
      [4]  ⍝ Alternative:
      [5]  ⍝ R←W↑(-⌊(W+ρ,C)÷2)↑C
        ∇
```

4.

```
                                          [WSID: FORMAT]
    ∇ R←D ROWFMT N
[1]   ⍝ Formats a numeric matrix N into a character matrix.
[2]   ⍝ D is an integer vector with one element per row
[3]   ⍝ of N.  The integers indicate the number of decimal
[4]   ⍝ places, for each numeric row, to be displayed in
[5]   ⍝ the character matrix result.  Each number is
[6]   ⍝ formatted in a width of <width> characters (e.g. 10),
[7]   ⍝ where <width> is an integer scalar global variable.
[8]   ⍝ Requires subfunction: COLFMT
[9]    R←D COLFMT⍉N
[10]   R←((1,width)×ρN)ρ 2 1 3 ⍉((⌽ρN),width)ρR
    ∇
```

5.

```
                                          [WSID: FORMAT]
    ∇ R←RC COLUMNIZE CMAT;C;COLS;PGS;ROWS;⎕IO
[1]   ⍝ Restructures skinny character matrix CMAT into
[2]   ⍝ a fat one.  RC is scalar or 1 or 2 element
[3]   ⍝ vector.  Last element is no. of "columns" of
[4]   ⍝ CMAT running down each page of the result.
[5]   ⍝ If RC has 1 element, result is a short
[6]   ⍝ (⌈(1↑ρCMAT)÷RC rows), fat (RC×1↓ρCMAT columns)
[7]   ⍝ matrix.  If RC has 2 elements, first element is
[8]   ⍝ number of rows per page.  Result is a 3
[9]   ⍝ dimensional character array with
[10]  ⍝ (⌈(1↑ρCMAT)÷×/RC) planes, RC[1] rows and
[11]  ⍝ (RC×1↓ρCMAT) columns.
[12]   ⎕IO←1
[13]   RC←,RC
[14]   COLS←¯1↑RC
[15]   C←1↓ρCMAT
[16]  ⍝ Branch if 3 dimensional result:
[17]   →(2=ρRC)ρL1
[18]  ⍝ 2 dimensional result:
[19]   ROWS←⌈(1↑ρCMAT)÷COLS
[20]   R←(ROWS,COLS×C)ρ 2 1 3 ⍉(COLS,ROWS,C)ρ((COLS×ROWS),C)↑
       CMAT
[21]   →0
[22]  ⍝ 3 dimensional result:
[23]  L1:ROWS←1↑RC
[24]   PGS←⌈(1↑ρCMAT)÷ROWS×COLS
[25]   R←(PGS,ROWS,COLS×C)ρ 1 3 2 4 ⍉(PGS,COLS,ROWS,C)ρ((PGS×
       COLS×ROWS),C)↑CMAT
    ∇
```

6.

```
                                        [WSID: FORMAT]
     ∇ R←WID HEADINGS CS;⎕IO;A;ARGSTART;B;BHDG;BSEG;HDGLEAD;
       LEN;NCOLS;NHDG;NROWS;NSEG;RESSTART;S;SEGLEAD;SEGROWS;T
[1]   ⍝ Creates column headings from text CS within
[2]   ⍝ field widths WID.  CS is a character vector with
[3]   ⍝ text for successive headings separated by '∩'.
[4]   ⍝ The format of WID is:  (widths of headings, not
[5]   ⍝ including spacing),(spacing between columns).
[6]   ⍝ If WID has fewer elements than CS has segments,
[7]   ⍝ its values are repeated; if it does not include
[8]   ⍝ spacing specification, 2 is used.  Empty headings
[9]   ⍝ are not underlined.  Separate lines of multi-line
[10]  ⍝ heading by '←'.
[11]  ⍝
[12]   ⎕IO←0
[13]  ⍝ 1s for hdg starts:
[14]   BHDG←CS='∩'
[15]  ⍝ 1s for segment starts:
[16]   BSEG←CS∊'∩←'
[17]  ⍝
[18]  ⍝ Flag end of hdgs (1 elt per segment):
[19]   BHDG←1⌽BSEG/BHDG
[20]  ⍝ No. segments per hdg:
[21]   T←BHDG/⍳⍴BHDG
[22]   NHDG←⍴NSEG←T-¯1↓¯1,T
[23]  ⍝ Segment lengths:
[24]   T←BSEG/⍳⍴BSEG
[25]   LEN←(1↓T,⍴BSEG)-T+1
[26]  ⍝ Spacing between hdgs (2s if omitted):
[27]   S←NHDG↓WID
[28]   S←NHDG⍴S,(0=⍴S)⍴2
[29]  ⍝ Reshape widths to conform with headings:
[30]   WID←NHDG⍴WID
[31]  ⍝ Truncated segment lengths:
[32]   LEN←LEN⌊NSEG/WID
[33]  ⍝ Leading blanks per segment, to center:
[34]   SEGLEAD←⌈((NSEG/WID)-LEN)÷2
[35]  ⍝ Col in which each hdg begins:
[36]   HDGLEAD←¯1↓0,+\WID+S
[37]  ⍝ Index of char. following delim. of each segment:
[38]   ARGSTART←1+BSEG/⍳⍴BSEG
[39]  ⍝ No. of rows and columns in result (A is no. rows
[40]  ⍝ without underlines):
[41]   NROWS←1+A←⌈/NSEG
[42]   NCOLS←(+/(NHDG+¯1)⍴S)++/WID
[43]  ⍝ No. whole rows before each segment:
[44]   T←NSEG/A-+\NSEG
[45]   SEGROWS←T+⍳⍴T
[46]  ⍝ Index in raveled result where each segment starts:
[47]   RESSTART←SEGLEAD+(NSEG/HDGLEAD)+SEGROWS×NCOLS
[48]  ⍝ Create blank, raveled result:
[49]   R←(B←NROWS×NCOLS)⍴' '
```

```
    ∇ HEADINGS (continued)
[50] ⍝ Flag nonempty headings:
[51]  T←NHDGρ0
[52]  T[(×LEN)/NSEG/ιNHDG]←1
[53] ⍝ Indices of underlines in raveled result:
[54]  A←T/WID
[55]  A←A/(T/+\¯1↓(B-NCOLS),WID+S)-¯1↓0,+\A
[56]  A←A+ιρA
[57] ⍝ Insert underlines:
[58]  R[A]←'-'
[59] ⍝ T←MONIOTA LEN:
[60]  T←T+ιρT←LEN/-¯1↓0,+\LEN
[61]  R[T+LEN/RESSTART]←CS[T+LEN/ARGSTART]
[62] ⍝ Reshape result to matrix:
[63]  R←(NROWS,NCOLS)ρR
    ∇
```

Chapter 5 Solutions

SORTING AND SEARCHING

1. If your APL implementation supports matrix right arguments to ⍋:

```
SPNUM←PNUM[⍋PNUM;]
```

   Otherwise (major-to-minor sorting done in minor-to-major order):

```
G←⍋PNUM[;3]
G←G[⍋PNUM[G;2]]
G←G[⍋PNUM[G;1]]
SPNUM←PNUM[G;]
```

   Or pack and sort:

```
SPNUM←PNUM[⍋1E3 1E7 1E4⊥⍉PNUM;]
```

   Note: Numbers are packed to 14 digit floating point numbers (the 1E4 assumes 4 digit extensions). Since the 14 digits do not exceed the 16 or 17 digits of available precision and since ⍋ works with the full precision (i.e. does not refer to ⎕CT), the result will be accurate.

2. Since dyadic ⍳ is dependent upon ⎕CT (for floating point arguments), it will "find" matches which are very close but not exact. The IOTA function will overlook such close values and will "find" only those values which match exactly (to 16 or 17 digits of precision). This will usually not be a problem since IOTA will most often be used on integer arguments or on floating point arguments whose values have not been computed. Only through such computations will "equal" values become slightly different.

```
                                         [WSID: SEARCH]
    ∇ INDS←BASE IOTA VALS;A;F;G;I;L
[1]  ⍝ Returns the indices of BASE at which the
[2]  ⍝ elements of VALS first match, i.e.
[3]  ⍝ INDS←BASE⍳VALS     (but maybe faster)
[4]  ⍝ Branch if right arg a vector:
[5]   →(1=⍴⍴VALS)⍴L1
[6]  ⍝ Handle scalar right arg:
[7]   INDS←BASE⍳VALS
[8]   →0
[9]  L1:L←(⍴BASE)[⎕IO]
[10]  A←(⍴VALS)[⎕IO]
[11] ⍝ Branch unless no elts in either arg:
[12]  →(×F←A⌊L)⍴L2
[13] ⍝ Handle empty arg:
[14]  INDS←A⍴⎕IO
[15]  →0
[16] ⍝ Branch if both args have more than 1 elt:
[17] L2:→(F≠1)⍴L4
[18] ⍝ Branch unless left arg has 1 elt:
[19]  →(L≠1)⍴L3
[20] ⍝ Handle 1 elt left arg:
[21]  INDS←⎕IO+VALS≠BASE
[22]  →0
[23] ⍝ Handle 1 elt right arg:
[24] L3:INDS←BASE⍳VALS
[25]  →0
[26] ⍝ Branch if sort alg. costs more than looping alg.:
[27] ⍝    (remove ⍝ after replacing C1,C2,C3,C4 by
[28] ⍝    computed constants):
[29] L4: ⍝→((C4+C5×L+A)>C1+A×C2+C3×L)⍴L5
[30] ⍝ Combine args. and sort (like values together):
[31]  G←⍋A←BASE,VALS
[32]  A←A[G]
[33] ⍝ Flag 1st of distinct elts by shifting and comparing:
[34]  F←A≠¯1⌽A
[35] ⍝ Insure 1st elt is 1 (in case all rows the same):
[36]  F[⎕IO]←1
[37] ⍝ Indices of 1st distinct elts:
[38]  I←F/G
[39] ⍝ Replicate for each like elt:
[40]  F[⎕IO]←⎕IO
[41]  I←I[+\F]
[42] ⍝ Unsort indices (to catenated order):
[43]  INDS←I
[44]  INDS[G]←I
[45] ⍝ Keep those corresponding to right arg:
[46]  INDS←L↓INDS
[47] ⍝ Set 'not found' inds to 'one greater':
[48]  INDS←INDS⌊L+⎕IO
[49]  →0
[50] ⍝ Use looping algorithm if more efficient:
[51] L5:INDS←BASE⍳VALS
    ∇
```

3.
```
I←(PNUM∧.=P)⍳1
```
or
```
I←(1E3 1E7 1E4⊥⍉PNUM)⍳1E3 1E7 1E4⊥P
```

4.
```
LOWER←1000 10000 20000 50000 70000 100000
R←(1 2 5 3 5 4 5)[LOWER LIOTA SALARY]
```

5.

[WSID: SEARCH]
```
    ∇ R←DEB CVEC
[1]   ⍝ Deletes extraneous (leading, trailing or
[2]   ⍝ redundant) blanks from its argument and
[3]   ⍝ returns the compressed result.
[4]   ⍝ Put extra blank on beginning and end:
[5]    CVEC←' ',CVEC,' '
[6]   ⍝ Search for 2 contiguous blanks:
[7]    R←¯1↓1↓(~CVEC ∆SS '  ')/CVEC
    ∇
```

6.
```
⎕←¯1↓((⍕NVEC),' ') REPLACE '¯1 ' BY 'N/A '
```

7.
```
⎕←(∨/0 ¯2↓(⍴ENAMES)⍴(,UPPERCASE ENAMES)∆SS 'SON')⌿ENAMES
```

(Note: the last 2 Boolean columns are dropped to avoid coincidental wrap-around matches, e.g. if row 5 ends with 'SO' and row 6 starts with 'N')

In APL2:
```
⎕←(∨/'SON'⍷UPPERCASE ENAMES)⌿ENAMES
```

Chapter 6 Solutions

SELECTING

1.
```
        ((ρV)ρ1 0)/V
    or
        V[(-⎕IO)+2×ι(ρV)÷2]
    or
        ((((ρV)÷2),2)ρV)[;⎕IO]
```

2.
```
        (2ρ⎕IO)⍉M
    or
        (,(ρM)ρ(1+1ρρM)↑1)/,M
    or
        (((-⎕IO)+ι1ρρM)⌽M)[;⎕IO]
```

3. The expressions (¯1↑V) or V[ρV] return one element vectors, not scalars. Therefore, the result of M[;¯1↑V] is a one column matrix, not a vector. The following produce correct results:

```
        ,M[;¯1↑V]
    or
        ,M[;V[(ρV)-~⎕IO]]
    or
        M[;(ι0)ρ¯1↑V]
    or
        M[;(ι0)ρ⌽V]
```

4. Approach 1:

```
        SHAPE←ρNAMES
        NAMES←,NAMES
        NAMES[(NAMES='/')/ιρNAMES]←','
        NAMES←SHAPEρNAMES
```

Approach 2:

```
A1←A2←(⎕AV∊NAMES)/⎕AV
A1[(A1='/')/⍳⍴A1]←','
NAMES←A1[A2⍳NAMES]
```

Approach 3 (APL2):

```
(('/'=,NAMES)/,NAMES)←','
```

5. A.

```
⎕IO←0
MRATES←(,RATES)[DUR+16×IAGE+100×SEX]
```

B.

```
⎕IO←0
∆DUR←15⌊DUR
∆IAGE←IAGE+DUR-∆DUR
MRATES←(,RATES)[∆DUR+16×∆IAGE+100×SEX]
```

C.

```
⎕IO←0
VRATES←,RATES
VRATES[NEWDUR+16×NEWIAGE+100×NEWSEX]←NEWRATES
RATES←(⍴RATES)⍴VRATES
```

6.

```
                                    [WSID: SEARCH]
    ∇ R←UNQNV NV;FIRST;G;SORTED
[1]   ⍝ Returns the distinct elements of the
[2]   ⍝ numeric vector NV.
[3]    SORTED←NV[G←⍋NV]
[4]    FIRST←SORTED≠¯1⌽SORTED
[5]   ⍝ Set 1st elt to 1 in case FIRST all 0s:
[6]    FIRST[⍳×⍴FIRST]←1
[7]    R←FIRST/SORTED
[8]   ⍝ FIRST[⍳×⍴FIRST]←⎕IO
[9]   ⍝ ind←G
[10]  ⍝ ind[G]←+\FIRST
    ∇
```

```
                                    [WSID: SEARCH]
    ∇ R←UNQCV CV
[1]   ⍝ Returns the distinct elements of the
[2]   ⍝ character vector CV.
[3]    R←(⎕AV∊CV)/⎕AV
[4]   ⍝ ind←R⍳CV
    ∇
```

[WSID: SEARCH]

```
    ∇ R←UNQCM CM;FIRST;G;SORTED
[1]   ⍝ Returns the distinct rows of the character
[2]   ⍝ matrix CM.
[3]    G←⎕AV⍋CM
[4]   ⍝ If dyadic ⍋ unavailable:
[5]   ⍝ G←⎕AV CGRADEUP CM
[6]    SORTED←CM[G;]
[7]    FIRST←∨/SORTED≠¯1⊖SORTED
[8]   ⍝ Set 1st elt to 1 in case FIRST all 0s:
[9]    FIRST[×⍳⍴FIRST]←1
[10]   R←FIRST⌿SORTED
[11]  ⍝ FIRST[⍳×⍴FIRST]←⎕IO
[12]  ⍝ ind←G
[13]  ⍝ ind[G]←+\FIRST
    ∇
```

[WSID: SEARCH]

```
    ∇ R←N UNQI1 IV;BIT;⎕IO
[1]   ⍝ Returns the distinct elements of the
[2]   ⍝ origin 1 index vector IV.  All elements
[3]   ⍝ of IV must be elements of ⍳N.
[4]    ⎕IO←1
[5]    BIT←N⍴0
[6]    BIT[IV]←1
[7]    R←BIT/⍳N
[8]   ⍝ ind←(BIT\⍳⍴R)[IV]
[9]   ⍝ Alternative:
[10]  ⍝   ind←(+\BIT)[IV]
    ∇
```

[WSID: SEARCH]

```
    ∇ R←N UNQI0 IV;BIT;⎕IO
[1]   ⍝ Returns the distinct elements of the
[2]   ⍝ origin 0 index vector IV.  All elements
[3]   ⍝ of IV must be elements of ⍳N.
[4]    ⎕IO←0
[5]    BIT←N⍴0
[6]    BIT[IV]←1
[7]    R←BIT/⍳N
[8]   ⍝ ind←(BIT\⍳⍴R)[IV]
[9]   ⍝ Alternative:
[10]  ⍝   ind←(+\BIT)[IV]-1
    ∇
```

Chapter 7 Solutions

FREQUENCY COUNTS, ACCUMULATIONS AND CROSS-TABULATIONS

1.

```
        R←+/'ECMPH'∘.=TZONE
  or
        I11←'ECMPH'ιTZONE            (utility function)
        R←5 11 PLUSRED 1
  or
        R←(5;'ECMPH'ιTZONE)+/1       (hypothetical)
```

2.

```
        R←('ECMPH'∘.=TZONE)+.×SALES
  or
        I11←'ECMPH'ιTZONE            (utility function)
        R←5 11 PLUSRED SALES
  or
        R←(5;'ECMPH'ιTZONE)+/SALES   (hypothetical)
```

3.

```
        A←'ECMPH'∘.=TZONE
        B←TYPE∘.='BCPS'
        FRQ←A+.∧B
        AMT←(A×(ρA)ρSALES)+.×B
        MAX←(A×(ρA)ρSALES)⌈.×B
  or
        I11←'ECMPH'ιTZONE            (utility functions)
        I12←'BCPS'ιTYPE
        FRQ←5 4 11 12 PLUSRED 1
        AMT←5 4 11 12 PLUSRED SALES
        MAX←5 4 11 12 MAXRED SALES
  or
        I11←'ECMPH'ιTZONE            (hypothetical)
        I12←'BCPS'ιTYPE
        FRQ←(5 4;I11;I12)+/1
        AMT←(5 4;I11;I12)+/SALES
        MAX←(5 4;I11;I12)⌈/SALES
```

4.

```
                                        [WSID: REDUCE]
      ∇ R←L PLUSRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;N
        ;RANK;RRI;SORTED;URRI
[1]   ⍝ No. of ways for N-way reduction:
[2]    N←⌊(ρ,L)÷2
[3]   ⍝ Which dimension to be "reduced"?
[4]    DIM←1↑((N+N)↓L),¯1↑⍳RANK←ρρARRAY
[5]   ⍝ Branch unless 0-way reduction:
[6]    →(×N)ρL1
[7]    R←+/[DIM]ARRAY
[8]    →0
[9]   ⍝ Separate left arg into its pieces:
[10]  L1:DSHAPE←NρL
[11]   CIND←NρN↓L
[12]  ⍝ Begin to compute "raveled result indices":
[13]   I←⎕IO
[14]  ⍝ Index from ⍳DSHAPE[I] to cause index error if
[15]  ⍝ invalid indices:
[16]   RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[17]  ⍝ Branch if origin is 1:
[18]   →⎕IOρLOOP1
[19]  ⍝ Continue computing RRI (N iterations for N-way
[20]  ⍝ reduction):
[21]  LOOP0:→(N≤I←I+1)ρENDLP
[22]   RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[23]   →LOOP0
[24]  LOOP1:→(N<I←I+1)ρENDLP
[25]   RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[26]   →LOOP1
[27]  ⍝ Determine unique elements of RRI:
[28]  ENDLP:GRADE←⍋RRI
[29]   SORTED←RRI[GRADE]
[30]   LAST←SORTED≠1⌽SORTED
[31]   LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[32]   URRI←LAST/SORTED
[33]  ⍝ Branch unless ARRAY a scalar (i.e. freq. count):
[34]   →(×RANK)ρL3
[35]  ⍝ Perform partitioned frequency count:
[36]   CUM←LAST/⍳ρLAST
[37]   CUM←CUM-(ρCUM)ρ(¯1+⎕IO),CUM
[38]  ⍝ Branch if ARRAY is 1 (usually):
[39]   →(1=ARRAY)ρL2
[40]  ⍝ Multiply freq. counts by scalar:
[41]   CUM←ARRAY×CUM
[42]  ⍝ Initialize result with 0's:
[43]  L2:R←(×/DSHAPE)ρ0
[44]  ⍝ Insert result of partitioned freq. count:
[45]   R[URRI]←CUM
[46]  ⍝ Reshape to desired shape:
[47]   R←DSHAPEρR
[48]   →0
```

```
    ∇ PLUSRED (continued)
     ⍝ Reorder ARRAY to conform with SORTED:
     L3:M←DIM-⎕IO
      ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
     ⍝ Perform partitioned reduction:
      CUM←LAST/[DIM]+\[DIM]ARRAY
      CUM←CUM-(ρCUM)↑0,[DIM]CUM
     ⍝ Initialize result. Fill with identity elt.
     ⍝ Ravel the DSHAPE dim.s:
      R←((-M)⌽(×/DSHAPE),1↓M⌽ρARRAY)ρ0
     ⍝ Insert result of partitioned reduction:
      ⍎'R[',(Mρ';'),'URRI',((RANK-M+1)ρ';'),']←CUM'
     ⍝ Reshape to desired shape (unravel DSHAPE dim.s):
      R←((-M)⌽DSHAPE,1↓M⌽ρARRAY)ρR
    ∇
```

[WSID: REDUCE]

```
    ∇ R←L MAXRED ARRAY;CIND;DIF;DIM;DSHAPE;GRADE;I;LAST;M;N;
      RANK;RRI;SORTED;URRI
     ⍝ No. of ways for N-way reduction:
      N←⌊(ρ,L)÷2
     ⍝ Which dimension to be "reduced"?
      DIM←1↑((N+N)↓L),¯1↑⍳RANK←ρρARRAY
     ⍝ Branch unless 0-way reduction:
      →(×N)ρL1
      R←⌈/[DIM]ARRAY
      →0
     ⍝ Separate left arg into its pieces:
     L1:DSHAPE←NρL
      CIND←NρN↓L
     ⍝ Begin to compute "raveled result indices":
      I←⎕IO
     ⍝ Index from ⍳DSHAPE[I] to cause index error if
     ⍝ invalid indices:
      RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
     ⍝ Branch if origin is 1:
      →⎕IOρLOOP1
     ⍝ Continue computing RRI (N iterations for
     ⍝ N-way reduction):
     LOOP0:→(N≤I←I+1)ρENDLP
      RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
      →LOOP0
     LOOP1:→(N<I←I+1)ρENDLP
      RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
      →LOOP1
     ⍝ Determine unique elements of RRI:
     ENDLP:GRADE←⍋RRI
      SORTED←RRI[GRADE]
      LAST←SORTED≠1⌽SORTED
      LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
      URRI←LAST/SORTED
```

```
    ∇ MAXRED (continued)
[33] ⍝ Reorder ARRAY to conform with SORTED:
[34]  M←DIM-⎕IO
[35]  ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
[36] ⍝ Perform partitioned reduction:
[37]  DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[38]  DIF←(⍋DIM=ιRANK)⍉DIF∘.×+\¯1⌽LAST
[39]  DIF←(LAST/[DIM]⌈\[DIM]ARRAY+DIF)-LAST/[DIM]DIF
[40] ⍝ Initialize result. Fill with identity elt.
[41] ⍝ Ravel the DSHAPE dim.s:
[42]  R←((-M)⌽(×/DSHAPE),1↓M⌽ρARRAY)ρ⌈/ι0
[43] ⍝ Insert result of partitioned reduction:
[44]  ⍎'R[',(Mρ';'),'URRI',((RANK-M+1)ρ';'),']←DIF'
[45] ⍝ Reshape to desired shape (unravel DSHAPE dim.s):
[46]  R←((-M)⌽DSHAPE,1↓M⌽ρARRAY)ρR
    ∇


                                        [WSID: REDUCE]
    ∇ R←L MINRED ARRAY;CIND;DIF;DIM;DSHAPE;GRADE;I;LAST;M;N;
      RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(ρ,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),¯1↑ιRANK←ρρARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)ρL1
[7]   R←⌊/[DIM]ARRAY
[8]   →0
[9]  ⍝ Separate left arg into its pieces:
[10] L1:DSHAPE←NρL
[11]  CIND←NρN↓L
[12] ⍝ Begin to compute "raveled result indices":
[13]  I←⎕IO
[14] ⍝ Index from ιDSHAPE[I] to cause index error
[15] ⍝ if invalid indices:
[16]  RRI←(ιDSHAPE[I])[⍎'I',⍕CIND[I]]
[17] ⍝ Branch if origin is 1:
[18]  →⎕IOρLOOP1
[19] ⍝ Continue computing RRI (N iterations for
[20] ⍝ N-way reduction):
[21] LOOP0:→(N≤I←I+1)ρENDLP
[22]  RRI←(ιDSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[23]  →LOOP0
[24] LOOP1:→(N<I←I+1)ρENDLP
[25]  RRI←(ιDSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[26]  →LOOP1
[27] ⍝ Determine unique elements of RRI:
[28] ENDLP:GRADE←⍋RRI
[29]  SORTED←RRI[GRADE]
[30]  LAST←SORTED≠1⌽SORTED
[31]  LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[32]  URRI←LAST/SORTED
```

```
    ∇ MINRED (continued)
[33] ⍝ Reorder ARRAY to conform with SORTED:
[34]  M←DIM-⎕IO
[35]  ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
[36] ⍝ Perform partitioned reduction:
[37]  DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[38]  DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[39]  DIF←(LAST/[DIM]⌊\[DIM]ARRAY-DIF)+LAST/[DIM]DIF
[40] ⍝ Initialize result. Fill with identity elt.
[41] ⍝ Ravel the DSHAPE dim.s:
[42]  R←((-M)⌽(×/DSHAPE),1↓M⌽⍴ARRAY)⍴⌊/⍳0
[43] ⍝ Insert result of partitioned reduction:
[44]  ⍎'R[',(M⍴';'),'URRI',((RANK-M+1)⍴';'),']←DIF'
[45] ⍝ Reshape to desired shape (unravel DSHAPE dim.s):
[46]  R←((-M)⌽DSHAPE,1↓M⌽⍴ARRAY)⍴R
    ∇
```

```
                                        [WSID: REDUCE]
    ∇ R←L ANDRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;N;
      RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(⍴,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),¯1↑⍳RANK←⍴⍴ARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)⍴L1
[7]   R←∧/[DIM]ARRAY
[8]   →0
[9]  ⍝ Separate left arg into its pieces:
[10] L1:DSHAPE←N⍴L
[11]  CIND←N⍴N↓L
[12] ⍝ Begin to compute "raveled result indices":
[13]  I←⎕IO
[14] ⍝ Index from ⍳DSHAPE[I] to cause index error
[15] ⍝ if invalid indices:
[16]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[17] ⍝ Branch if origin is 1:
[18]  →⎕IO⍴LOOP1
[19] ⍝ Continue computing RRI (N iterations for
[20] ⍝ N-way reduction):
[21] LOOP0:→(N≤I←I+1)⍴ENDLP
[22]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[23]  →LOOP0
[24] LOOP1:→(N<I←I+1)⍴ENDLP
[25]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[26]  →LOOP1
[27] ⍝ Determine unique elements of RRI:
[28] ENDLP:GRADE←⍋RRI
[29]  SORTED←RRI[GRADE]
[30]  LAST←SORTED≠1⌽SORTED
[31]  LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
[32]  URRI←LAST/SORTED
```

```
    ∇ ANDRED (continued)
[33] ⍝ Reorder ARRAY to conform with SORTED:
[34]  M←DIM-⎕IO
[35]  ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
[36] ⍝ Perform partitioned reduction (note: ∧/ ←→ ~∨/~):
[37]  CUM←LAST/[DIM]+\[DIM]~ARRAY
[38]  CUM←CUM=(ρCUM)↑0,[DIM]CUM
[39] ⍝ Initialize result. Fill with identity elt.
[40] ⍝ Ravel the DSHAPE dim.s:
[41]  R←((-M)⌽(×/DSHAPE),1↓M⌽ρARRAY)ρ1
[42] ⍝ Insert result of partitioned reduction:
[43]  ⍎'R[',(Mρ';'),'URRI',((RANK-M+1)ρ';'),']←CUM'
[44] ⍝ Reshape to desired shape (unravel DSHAPE dim.s):
[45]  R←((-M)⌽DSHAPE,1↓M⌽ρARRAY)ρR
    ∇


                                        [WSID: REDUCE]
    ∇ R←L ORRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;N;
      RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(ρ,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),¯1↑⍳RANK←ρρARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)ρL1
[7]   R←∨/[DIM]ARRAY
[8]   →0
[9]  ⍝ Separate left arg into its pieces:
[10] L1:DSHAPE←NρL
[11]  CIND←NρN↓L
[12] ⍝ Begin to compute "raveled result indices":
[13]  I←⎕IO
[14] ⍝ Index from ⍳DSHAPE[I] to cause index error
[15] ⍝ if invalid indices:
[16]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[17] ⍝ Branch if origin is 1:
[18]  →⎕IOρLOOP1
[19] ⍝ Continue computing RRI (N iterations for
[20] ⍝ N-way reduction):
[21] LOOP0:→(N≤I←I+1)ρENDLP
[22]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[23]  →LOOP0
[24] LOOP1:→(N<I←I+1)ρENDLP
[25]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[26]  →LOOP1
[27] ⍝ Determine unique elements of RRI:
[28] ENDLP:GRADE←⍋RRI
[29]  SORTED←RRI[GRADE]
[30]  LAST←SORTED≠1⌽SORTED
[31]  LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[32]  URRI←LAST/SORTED
```

```
    ∇ ORRED (continued)
[33] ⍝ Reorder ARRAY to conform with SORTED:
[34]  M←DIM-⎕IO
[35]  ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
[36] ⍝ Perform partitioned reduction:
[37]  CUM←LAST/[DIM]+\[DIM]ARRAY
[38]  CUM←CUM≠(⍴CUM)↑0,[DIM]CUM
[39] ⍝ Initialize result. Fill with identity elt.
[40] ⍝ Ravel the DSHAPE dim.s:
[41]  R←((-M)⌽(×/DSHAPE),1↓M⌽⍴ARRAY)⍴0
[42] ⍝ Insert result of partitioned reduction:
[43]  ⍎'R[',(M⍴';'),'URRI',((RANK-M+1)⍴';'),']←CUM'
[44] ⍝ Reshape to desired shape (unravel DSHAPE dim.s):
[45]  R←((-M)⌽DSHAPE,1↓M⌽⍴ARRAY)⍴R
    ∇
```

5.

```
                                             [WSID: REDUCE]
    ∇ R←L ∆PLUSRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;
      N;RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(⍴,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),⎕IO⌈¯1↑⍳RANK←⍴⍴ARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)⍴L1
[7]   ARRAY←+/[DIM]ARRAY
[8]  ⍝ Construct milky-way result:
[9]   R←((⍴⍴ARRAY),(DIM-⎕IO),N,⍴ARRAY),,ARRAY
[10]  →0
[11] ⍝ Separate left arg into its pieces:
[12] L1:DSHAPE←N⍴L
[13]  CIND←N⍴N↓L
[14] ⍝ Begin to compute "raveled result indices":
[15]  I←⎕IO
[16] ⍝ Index from ⍳DSHAPE[I] to cause index error if
[17] ⍝ invalid indices:
[18]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[19] ⍝ Branch if origin is 1:
[20]  →⎕IO⍴LOOP1
[21] ⍝ Continue computing RRI (N iterations for N-way
[22] ⍝ reduction):
[23] LOOP0:→(N≤I←I+1)⍴ENDLP
[24]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[25]  →LOOP0
[26] LOOP1:→(N<I←I+1)⍴ENDLP
[27]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[28]  →LOOP1
[29] ⍝ Determine unique elements of RRI:
[30] ENDLP:GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
```

```
    ∇ ∆PLUSRED (continued)
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[34]  URRI←LAST/SORTED
[35] ⍝ Branch unless ARRAY a scalar (i.e. freq count):
[36]  →(×RANK)ρL3
[37] ⍝ Perform partitioned frequency count:
[38]  CUM←LAST/⍳ρLAST
[39]  CUM←CUM-(ρCUM)ρ(¯1+⎕IO),CUM
[40] ⍝ Branch if ARRAY is 1 (usually):
[41]  →(1=ARRAY)ρL2
[42] ⍝ Multiply freq. counts by scalar:
[43]  CUM←ARRAY×CUM
[44] ⍝ Construct milky-way result:
[45] L2:R←((1 0 ,N,(ρCUM),DSHAPE),URRI-⎕IO),CUM
[46]  →0
[47] ⍝ Reorder ARRAY to conform with SORTED:
[48] L3:M←DIM-⎕IO
[49]  ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
[50] ⍝ Perform partitioned reduction:
[51]  CUM←LAST/[DIM]+\[DIM]ARRAY
[52]  CUM←CUM-(ρCUM)↑0,[DIM]CUM
[53] ⍝ Construct milky-way result:
[54]  R←((RANK,M,N,(ρCUM),DSHAPE),URRI-⎕IO),,CUM
    ∇
```

[WSID: REDUCE]

```
    ∇ R←L ∆MAXRED ARRAY;CIND;DIF;DIM;DSHAPE;GRADE;I;LAST;M;N
      ;RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(ρ,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),⎕IO⌈¯1↑⍳RANK←ρρARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)ρL1
[7]   ARRAY←⌈/[DIM]ARRAY
[8]  ⍝ Construct milky-way result:
[9]   R←((ρρARRAY),(DIM-⎕IO),N,ρARRAY),,ARRAY
[10]  →0
[11] ⍝ Separate left arg into its pieces:
[12] L1:DSHAPE←NρL
[13]  CIND←NρN↓L
[14] ⍝ Begin to compute "raveled result indices":
[15]  I←⎕IO
[16] ⍝ Index from ⍳DSHAPE[I] to cause index error if
[17] ⍝ invalid indices:
[18]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[19] ⍝ Branch if origin is 1:
[20]  →⎕IOρLOOP1
[21] ⍝ Continue computing RRI (N iterations for
[22] ⍝ N-way reduction):
[23] LOOP0:→(N≤I←I+1)ρENDLP
[24]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
```

```
    ∇ ∆MAXRED (continued)
[25]  →LOOP0
[26] LOOP1:→(N<I←I+1)ρENDLP
[27]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[28]  →LOOP1
[29] ⍝ Determine unique elements of RRI:
[30] ENDLP:GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[34]  URRI←LAST/SORTED
[35] ⍝ Reorder ARRAY to conform with SORTED:
[36]  M←DIM-⎕IO
[37]  ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
[38] ⍝ Perform partitioned reduction:
[39]  DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[40]  DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[41]  DIF←(LAST/[DIM]⌈\[DIM]ARRAY+DIF)-LAST/[DIM]DIF
[42] ⍝ Construct milky-way result:
[43]  R←((RANK,M,N,(ρDIF),DSHAPE),URRI-⎕IO),,DIF
    ∇
```

```
                                        [WSID: REDUCE]
    ∇ R←L ∆MINRED ARRAY;CIND;DIF;DIM;DSHAPE;GRADE;I;LAST;M;N
      ;RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(ρ,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),⎕IO⌈¯1↑⍳RANK←ρρARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)ρL1
[7]   ARRAY←⌊/[DIM]ARRAY
[8]  ⍝ Construct milky-way result:
[9]   R←((ρρARRAY),(DIM-⎕IO),N,ρARRAY),,ARRAY
[10]  →0
[11] ⍝ Separate left arg into its pieces:
[12] L1:DSHAPE←NρL
[13]  CIND←NρN↓L
[14] ⍝ Begin to compute "raveled result indices":
[15]  I←⎕IO
[16] ⍝ Index from ⍳DSHAPE[I] to cause index error if
[17] ⍝ invalid indices:
[18]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[19] ⍝ Branch if origin is 1:
[20]  →⎕IOρLOOP1
[21] ⍝ Continue computing RRI (N iterations for
[22] ⍝ N-way reduction):
[23] LOOP0:→(N≤I←I+1)ρENDLP
[24]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[25]  →LOOP0
[26] LOOP1:→(N<I←I+1)ρENDLP
[27]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[28]  →LOOP1
```

```
    ∇ ΔMINRED (continued)
[29] ⍝ Determine unique elements of RRI:
[30] ENDLP:GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
[34]  URRI←LAST/SORTED
[35] ⍝ Reorder ARRAY to conform with SORTED:
[36]  M←DIM-⎕IO
[37]  ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
[38] ⍝ Perform partitioned reduction:
[39]  DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[40]  DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[41]  DIF←(LAST/[DIM]⌊\[DIM]ARRAY-DIF)+LAST/[DIM]DIF
[42] ⍝ Construct milky-way result:
[43]  R←((RANK,M,N,(⍴DIF),DSHAPE),URRI-⎕IO),,DIF
    ∇
```

[WSID: REDUCE]

```
    ∇ R←L ΔANDRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;N
      ;RANK;RRI;SORTED;URRI
[1]  ⍝ No. of ways for N-way reduction:
[2]   N←⌊(⍴,L)÷2
[3]  ⍝ Which dimension to be "reduced"?
[4]   DIM←1↑((N+N)↓L),⎕IO⌈¯1↑⍳RANK←⍴⍴ARRAY
[5]  ⍝ Branch unless 0-way reduction:
[6]   →(×N)⍴L1
[7]   ARRAY←∧/[DIM]ARRAY
[8]  ⍝ Construct milky-way result:
[9]   R←((⍴⍴ARRAY),(DIM-⎕IO),N,⍴ARRAY),,ARRAY
[10]  →0
[11] ⍝ Separate left arg into its pieces:
[12] L1:DSHAPE←N⍴L
[13]  CIND←N⍴N↓L
[14] ⍝ Begin to compute "raveled result indices":
[15]  I←⎕IO
[16] ⍝ Index from ⍳DSHAPE[I] to cause index error if
[17] ⍝ invalid indices:
[18]  RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
[19] ⍝ Branch if origin is 1:
[20]  →⎕IO⍴LOOP1
[21] ⍝ Continue computing RRI (N iterations for
[22] ⍝ N-way reduction):
[23] LOOP0:→(N≤I←I+1)⍴ENDLP
[24]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
[25]  →LOOP0
[26] LOOP1:→(N<I←I+1)⍴ENDLP
[27]  RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
[28]  →LOOP1
[29] ⍝ Determine unique elements of RRI:
[30] ENDLP:GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
```

```
    ∇ ∆ANDRED (continued)
       LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
       URRI←LAST/SORTED
      ⍝ Reorder ARRAY to conform with SORTED:
       M←DIM-⎕IO
       ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
      ⍝ Perform partitioned reduction (note: ∧/ ←→ ~∨/~):
       CUM←LAST/[DIM]+\[DIM]~ARRAY
       CUM←CUM=(⍴CUM)↑0,[DIM]CUM
      ⍝ Construct milky-way result:
       R←((RANK,M,N,(⍴CUM),DSHAPE),URRI-⎕IO),,CUM
    ∇
```

[WSID: REDUCE]

```
    ∇ R←L ∆ORRED ARRAY;CIND;CUM;DIM;DSHAPE;GRADE;I;LAST;M;N;
       RANK;RRI;SORTED;URRI
      ⍝ No. of ways for N-way reduction:
       N←⌊(⍴,L)÷2
      ⍝ Which dimension to be "reduced"?
       DIM←1↑((N+N)↓L),⎕IO⌈¯1↑⍳RANK←⍴⍴ARRAY
      ⍝ Branch unless 0-way reduction:
       →(×N)⍴L1
       ARRAY←∨/[DIM]ARRAY
      ⍝ Construct milky-way result:
       R←((⍴⍴ARRAY),(DIM-⎕IO),N,⍴ARRAY),,ARRAY
       →0
      ⍝ Separate left arg into its pieces:
      L1:DSHAPE←N⍴L
       CIND←N⍴N↓L
      ⍝ Begin to compute "raveled result indices":
       I←⎕IO
      ⍝ Index from ⍳DSHAPE[I] to cause index error if
      ⍝ invalid indices:
       RRI←(⍳DSHAPE[I])[⍎'I',⍕CIND[I]]
      ⍝ Branch if origin is 1:
       →⎕IO⍴LOOP1
      ⍝ Continue computing RRI (N iterations for
      ⍝ N-way reduction):
      LOOP0:→(N≤I←I+1)⍴ENDLP
       RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI
       →LOOP0
      LOOP1:→(N<I←I+1)⍴ENDLP
       RRI←(⍳DSHAPE[I])[(⍎'I',⍕CIND[I])]+DSHAPE[I]×RRI+¯1
       →LOOP1
      ⍝ Determine unique elements of RRI:
      ENDLP:GRADE←⍋RRI
       SORTED←RRI[GRADE]
       LAST←SORTED≠1⌽SORTED
       LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
       URRI←LAST/SORTED
      ⍝ Reorder ARRAY to conform with SORTED:
       M←DIM-⎕IO
       ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
```

```
    ∇ ∆ORRED (continued)
  ⍝ Perform partitioned reduction:
   CUM←LAST/[DIM]+\[DIM]ARRAY
   CUM←CUM≠(⍴CUM)↑0,[DIM]CUM
  ⍝ Construct milky-way result:
   R←((RANK,M,N,(⍴CUM),DSHAPE),URRI-⎕IO),,CUM
    ∇
```

[WSID: REDUCE]

```
    ∇ R←W ∆PLUSWAY ARRAY;DIM;DS;GRADE;LAST;M;N;NDS;RANK;RRI;
      S;SORTED
  ⍝ Force W to be a vector:
   W←,W
  ⍝ Rank; dimension reduced (origin 0); no. ways:
   RANK←ARRAY[⎕IO]
   DIM←⎕IO+M←ARRAY[⎕IO+1]
   N←ARRAY[⎕IO+2]
  ⍝ Shape of reduced array; new/old resulting shape
  ⍝ of reduced dimension:
   S←ARRAY[3+⍳RANK]
   DS←ARRAY[(3+RANK)+⍳N]
  ⍝ New resulting shape of reduced dimension:
   NDS←DS[W]
  ⍝ Branch unless 0-way reduction:
   →(×N)⍴L0
   R←S⍴(3+RANK)↓ARRAY
   →0
  ⍝ Result indices; reduced array:
  L0:RRI←ARRAY[(3+RANK+N)+⍳S[DIM]]
   ARRAY←S⍴(3+RANK+N+S[DIM])↓ARRAY
  ⍝ Treat special if W is empty:
   →(0≠⍴W)⍴L1
   R←+/[DIM]ARRAY
   →0
  ⍝ Treat special if W is ⍳⍴DS (i.e. all dimensions):
  L1:→((⍴W)≠⍴DS)⍴L2
   →(W∧.=⍳⍴DS)⍴L3
  ⍝ New result indices:
  L2:RRI←NDS⊥(DS⊤RRI)[W;]
  ⍝ Determine unique elements of RRI:
   GRADE←⍋RRI
   SORTED←RRI[GRADE]
   LAST←SORTED≠1⌽SORTED
   LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
   RRI←LAST/SORTED
  ⍝ Reorder ARRAY to conform with SORTED:
   ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
  ⍝ Perform partitioned reduction:
   ARRAY←LAST/[DIM]+\[DIM]ARRAY
   ARRAY←ARRAY-(⍴ARRAY)↑0,[DIM]ARRAY
  ⍝ Initialize result.  Fill with identity elt.
  ⍝ Ravel the DSHAPE dim.s:
  L3:R←((-M)⌽(×/NDS),1↓M⌽⍴ARRAY)⍴0
```

```
    ∇ ∆PLUSWAY (continued)
  ⍝ Insert result of partitioned reduction:
   ⍎'R[',(M⍴';'),'⎕IO+RRI',((RANK-M+1)⍴';'),']←ARRAY'
  ⍝ Reshape to desired shape (unravel NDS dim.s):
   R←((-M)⌽NDS,1↓M⌽⍴ARRAY)⍴R
    ∇
```

[WSID: REDUCE]

```
    ∇ R←W ∆MAXWAY ARRAY;DIF;DIM;DS;GRADE;LAST;M;N;NDS;RANK;
      RRI;S;SORTED
  ⍝ Force W to be a vector:
   W←,W
  ⍝ Rank; dimension reduced (origin 0); no. ways:
   RANK←ARRAY[⎕IO]
   DIM←⎕IO+M←ARRAY[⎕IO+1]
   N←ARRAY[⎕IO+2]
  ⍝ Shape of reduced array; new/old resulting shape
  ⍝ of reduced dimension:
   S←ARRAY[3+⍳RANK]
   DS←ARRAY[(3+RANK)+⍳N]
  ⍝ New resulting shape of reduced dimension:
   NDS←DS[W]
  ⍝ Branch unless 0-way reduction:
   →(×N)⍴L0
   R←S⍴(3+RANK)↓ARRAY
   →0
  ⍝ Result indices; reduced array:
  L0:RRI←ARRAY[(3+RANK+N)+⍳S[DIM]]
   ARRAY←S⍴(3+RANK+N+S[DIM])↓ARRAY
  ⍝ Treat special if W is empty:
   →(0≠⍴W)⍴L1
   R←⌈/[DIM]ARRAY
   →0
  ⍝ Treat special if W is ⍳⍴DS (i.e. all dimensions):
  L1:→((⍴W)≠⍴DS)⍴L2
   →(W∧.=⍳⍴DS)⍴L3
  ⍝ New result indices:
  L2:RRI←NDS⊥(DS⊤RRI)[W;]
  ⍝ Determine unique elements of RRI:
   GRADE←⍋RRI
   SORTED←RRI[GRADE]
   LAST←SORTED≠1⌽SORTED
   LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
   RRI←LAST/SORTED
  ⍝ Reorder ARRAY to conform with SORTED:
   ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
  ⍝ Perform partitioned reduction:
   DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
   DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
   ARRAY←(LAST/[DIM]⌈\[DIM]ARRAY+DIF)-LAST/[DIM]DIF
  ⍝ Initialize result.  Fill with identity elt.
  ⍝ Ravel the DSHAPE dim.s:
  L3:R←((-M)⌽(×/NDS),1↓M⌽⍴ARRAY)⍴⌈/⍳0
```

```
    ∇ ∆MAXWAY (continued)
[44] ⍝ Insert result of partitioned reduction:
[45]  ⍎'R[',(Mρ';'),'⎕IO+RRI',((RANK-M+1)ρ';'),']←ARRAY'
[46] ⍝ Reshape to desired shape (unravel NDS dim.s):
[47]  R←((-M)⌽NDS,1↓M⌽ρARRAY)ρR
    ∇
```

[WSID: REDUCE]

```
    ∇ R←W ∆MINWAY ARRAY;DIF;DIM;DS;GRADE;LAST;M;N;NDS;RANK;
      RRI;S;SORTED
[1]  ⍝ Force W to be a vector:
[2]   W←,W
[3]  ⍝ Rank; dimension reduced (origin 0); no. ways:
[4]   RANK←ARRAY[⎕IO]
[5]   DIM←⎕IO+M←ARRAY[⎕IO+1]
[6]   N←ARRAY[⎕IO+2]
[7]  ⍝ Shape of reduced array; new/old resulting shape
[8]  ⍝ of reduced dimension:
[9]   S←ARRAY[3+⍳RANK]
[10]  DS←ARRAY[(3+RANK)+⍳N]
[11] ⍝ New resulting shape of reduced dimension:
[12]  NDS←DS[W]
[13] ⍝ Branch unless 0-way reduction:
[14]  →(×N)ρL0
[15]  R←Sρ(3+RANK)↓ARRAY
[16]  →0
[17] ⍝ Result indices; reduced array:
[18] L0:RRI←ARRAY[(3+RANK+N)+⍳S[DIM]]
[19]  ARRAY←Sρ(3+RANK+N+S[DIM])↓ARRAY
[20] ⍝ Treat special if W is empty:
[21]  →(0≠ρW)ρL1
[22]  R←⌊/[DIM]ARRAY
[23]  →0
[24] ⍝ Treat special if W is ⍳ρDS (i.e. all dimensions):
[25] L1:→((ρW)≠ρDS)ρL2
[26]  →(W∧.=⍳ρDS)ρL3
[27] ⍝ New result indices:
[28] L2:RRI←NDS⊥(DS⊤RRI)[W;]
[29] ⍝ Determine unique elements of RRI:
[30]  GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
[34]  RRI←LAST/SORTED
[35] ⍝ Reorder ARRAY to conform with SORTED:
[36]  ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
[37] ⍝ Perform partitioned reduction:
[38]  DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[39]  DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[40]  ARRAY←(LAST/[DIM]⌊\[DIM]ARRAY-DIF)+LAST/[DIM]DIF
[41] ⍝ Initialize result.  Fill with identity elt.
[42] ⍝ Ravel the DSHAPE dim.s:
[43] L3:R←((-M)⌽(×/NDS),1↓M⌽ρARRAY)ρ⌊/⍳0
```

```
    ∇ ΔMINWAY (continued)
  ⍝ Insert result of partitioned reduction:
   ⍎'R[',(Mρ';'),'⎕IO+RRI',((RANK-M+1)ρ';'),']←ARRAY'
  ⍝ Reshape to desired shape (unravel NDS dim.s):
   R←((-M)⌽NDS,1↓M⌽ρARRAY)ρR
    ∇
```

[WSID: REDUCE]

```
    ∇ R←W ΔANDWAY ARRAY;DIM;DS;GRADE;LAST;M;N;NDS;RANK;RRI;S
      ;SORTED
  ⍝ Force W to be a vector:
   W←,W
  ⍝ Rank; dimension reduced (origin 0); no. ways:
   RANK←ARRAY[⎕IO]
   DIM←⎕IO+M←ARRAY[⎕IO+1]
   N←ARRAY[⎕IO+2]
  ⍝ Shape of reduced array; new/old resulting shape
  ⍝ of reduced dimension:
   S←ARRAY[3+⍳RANK]
   DS←ARRAY[(3+RANK)+⍳N]
  ⍝ New resulting shape of reduced dimension:
   NDS←DS[W]
  ⍝ Branch unless 0-way reduction:
   →(×N)ρL0
   R←Sρ(3+RANK)↓ARRAY
   →0
  ⍝ Result indices; reduced array:
  L0:RRI←ARRAY[(3+RANK+N)+⍳S[DIM]]
   ARRAY←Sρ(3+RANK+N+S[DIM])↓ARRAY
  ⍝ Treat special if W is empty:
   →(0≠ρW)ρL1
   R←∧/[DIM]ARRAY
   →0
  ⍝ Treat special if W is ⍳ρDS (i.e. all dimensions):
  L1:→((ρW)≠ρDS)ρL2
   →(W∧.=⍳ρDS)ρL3
  ⍝ New result indices:
  L2:RRI←NDS⊥(DS⊤RRI)[W;]
  ⍝ Determine unique elements of RRI:
   GRADE←⍋RRI
   SORTED←RRI[GRADE]
   LAST←SORTED≠1⌽SORTED
   LAST[(×ρLAST)ρ(ρLAST)-~⎕IO]←1
   RRI←LAST/SORTED
  ⍝ Reorder ARRAY to conform with SORTED:
   ARRAY←⍎'ARRAY[',(Mρ';'),'GRADE',((RANK-M+1)ρ';'),']'
  ⍝ Perform partitioned reduction:
   ARRAY←LAST/[DIM]+\[DIM]~ARRAY
   ARRAY←ARRAY=(ρARRAY)↑0,[DIM]ARRAY
  ⍝ Initialize result.  Fill with identity elt.
  ⍝ Ravel the DSHAPE dim.s:
  L3:R←((-M)⌽(×/NDS),1↓M⌽ρARRAY)ρ1
```

```
    ∇ ∆ANDWAY (continued)
[43] ⍝ Insert result of partitioned reduction:
[44]  ⍎'R[',(M⍴';'),'⎕IO+RRI',((RANK-M+1)⍴';'),']←ARRAY'
[45] ⍝ Reshape to desired shape (unravel NDS dim.s):
[46]  R←((-M)⌽NDS,1↓M⌽⍴ARRAY)⍴R
    ∇


                                        [WSID: REDUCE]
    ∇ R←W ∆ORWAY ARRAY;DIM;DS;GRADE;LAST;M;N;NDS;RANK;RRI;S;
      SORTED
[1]  ⍝ Force W to be a vector:
[2]   W←,W
[3]  ⍝ Rank; dimension reduced (origin 0); no. ways:
[4]   RANK←ARRAY[⎕IO]
[5]   DIM←⎕IO+M←ARRAY[⎕IO+1]
[6]   N←ARRAY[⎕IO+2]
[7]  ⍝ Shape of reduced array; new/old resulting shape
[8]  ⍝ of reduced dimension:
[9]   S←ARRAY[3+⍳RANK]
[10]  DS←ARRAY[(3+RANK)+⍳N]
[11] ⍝ New resulting shape of reduced dimension:
[12]  NDS←DS[W]
[13] ⍝ Branch unless 0-way reduction:
[14]  →(×N)⍴L0
[15]  R←S⍴(3+RANK)↓ARRAY
[16]  →0
[17] ⍝ Result indices; reduced array:
[18] L0:RRI←ARRAY[(3+RANK+N)+⍳S[DIM]]
[19]  ARRAY←S⍴(3+RANK+N+S[DIM])↓ARRAY
[20] ⍝ Treat special if W is empty:
[21]  →(0≠⍴W)⍴L1
[22]  R←∨/[DIM]ARRAY
[23]  →0
[24] ⍝ Treat special if W is ⍳⍴DS (i.e. all dimensions):
[25] L1:→((⍴W)≠⍴DS)⍴L2
[26]  →(W∧.=⍳⍴DS)⍴L3
[27] ⍝ New result indices:
[28] L2:RRI←NDS⊥(DS⊤RRI)[W;]
[29] ⍝ Determine unique elements of RRI:
[30]  GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×⍴LAST)⍴(⍴LAST)-~⎕IO]←1
[34]  RRI←LAST/SORTED
[35] ⍝ Reorder ARRAY to conform with SORTED:
[36]  ARRAY←⍎'ARRAY[',(M⍴';'),'GRADE',((RANK-M+1)⍴';'),']'
[37] ⍝ Perform partitioned reduction:
[38]  ARRAY←LAST/[DIM]+\[DIM]ARRAY
[39]  ARRAY←ARRAY≠(⍴ARRAY)↑0,[DIM]ARRAY
[40] ⍝ Initialize result.  Fill with identity elt.
[41] ⍝ Ravel the DSHAPE dim.s:
[42] L3:R←((-M)⌽(×/NDS),1↓M⌽⍴ARRAY)⍴0
```

```
    ∇ ∆ORWAY (continued)
[43] ⍝ Insert result of partitioned reduction:
[44]  ⍎'R[',(Mρ';'),'⎕IO+RRI',((RANK-M+1)ρ';'),']←ARRAY'
[45] ⍝ Reshape to desired shape (unravel NDS dim.s):
[46]  R←((-M)⌽NDS,1↓M⌽ρARRAY)ρR
    ∇
```

```
                                            [WSID: REDUCE]
    ∇ R←A ∆PLUS B;ARRAY;CUM;DIM;DS;GRADE;LAST;N;RANK;RRI;S;
      SORTED;URRI;⎕IO
[1]  ⍝ Adds together two compressed Milky-Way arrays,
[2]  ⍝ returning a third such array.
[3]  ⍝
[4]  ⍝ Return left argument if right is empty:
[5]   →(0∊ρB)↓L1
[6]   R←A
[7]   →0
[8]  ⍝ Return right if left is empty:
[9]  L1:→(0∊ρA)↓L2
[10]  R←B
[11]  →0
[12] ⍝ Extract components from left argument:
[13] L2:⎕IO←0
[14]  RANK←A[0]
[15]  DIM←A[1]
[16]  N←A[2]
[17] ⍝ Branch unless a 0-way reduction:
[18]  →(×N)ρL3
[19]  R←((3+RANK)ρA),((3+RANK)↓A)+(3+RANK)↓B
[20]  →0
[21] L3:S←A[3+⍳RANK]
[22]  DS←A[(3+RANK)+⍳N]
[23]  RRI←A[(3+RANK+N)+⍳S[DIM]]
[24]  ARRAY←Sρ(3+RANK+N+S[DIM])↓A
[25] ⍝ Include components from right argument:
[26]  S←B[3+⍳RANK]
[27]  RRI←RRI,B[(3+RANK+N)+⍳S[DIM]]
[28]  ARRAY←ARRAY,[DIM]Sρ(3+RANK+N+S[DIM])↓B
[29] ⍝ Use same logic as in ∆PLUSRED:
[30]  GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ¯1+ρLAST]←1
[34]  URRI←LAST/SORTED
[35]  ARRAY←⍎'ARRAY[',(DIMρ';'),'GRADE',((RANK-DIM+1)ρ';'),'
      ]'
[36]  CUM←LAST/[DIM]+\[DIM]ARRAY
[37]  CUM←CUM-(ρCUM)↑0,[DIM]CUM
[38]  R←((RANK,DIM,N,(ρCUM),DS),URRI),,CUM
    ∇
```

```
                                              [WSID: REDUCE]
     ∇ R←A ∆MAX B;ARRAY;DIF;DIM;DS;GRADE;LAST;N;RANK;RRI;S;
       SORTED;URRI;⎕IO
[1]   ⍝ Adds together two compressed Milky-Way arrays,
[2]   ⍝ returning a third such array.
[3]   ⍝
[4]   ⍝ Return left argument if right is empty:
[5]    →(0∊⍴B)↓L1
[6]    R←A
[7]    →0
[8]   ⍝ Return right if left is empty:
[9]   L1:→(0∊⍴A)↓L2
[10]   R←B
[11]   →0
[12]  ⍝ Extract components from left argument:
[13]  L2:⎕IO←0
[14]   RANK←A[0]
[15]   DIM←A[1]
[16]   N←A[2]
[17]  ⍝ Branch unless a 0-way reduction:
[18]   →(×N)⍴L3
[19]   R←((3+RANK)⍴A),((3+RANK)↓A)⌈(3+RANK)↓B
[20]   →0
[21]  L3:S←A[3+⍳RANK]
[22]   DS←A[(3+RANK)+⍳N]
[23]   RRI←A[(3+RANK+N)+⍳S[DIM]]
[24]   ARRAY←S⍴(3+RANK+N+S[DIM])↓A
[25]  ⍝ Include components from right argument:
[26]   S←B[3+⍳RANK]
[27]   RRI←RRI,B[(3+RANK+N)+⍳S[DIM]]
[28]   ARRAY←ARRAY,[DIM]S⍴(3+RANK+N+S[DIM])↓B
[29]  ⍝ Use same logic as in ∆PLUSRED:
[30]   GRADE←⍋RRI
[31]   SORTED←RRI[GRADE]
[32]   LAST←SORTED≠1⌽SORTED
[33]   LAST[(×⍴LAST)⍴¯1+⍴LAST]←1
[34]   URRI←LAST/SORTED
[35]   ARRAY←⍎'ARRAY[',(DIM⍴';'),'GRADE',((RANK-DIM+1)⍴';'),'
       ]'
[36]   DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[37]   DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[38]   DIF←(LAST/[DIM]⌈\[DIM]ARRAY+DIF)-LAST/[DIM]DIF
[39]   R←((RANK,DIM,N,(⍴DIF),DS),URRI),,DIF
     ∇


                                              [WSID: REDUCE]
     ∇ R←A ∆MIN B;ARRAY;DIF;DIM;DS;GRADE;LAST;N;RANK;RRI;S;
       SORTED;URRI;⎕IO
[1]   ⍝ Adds together two compressed Milky-Way arrays,
[2]   ⍝ returning a third such array.
[3]   ⍝
[4]   ⍝ Return left argument if right is empty:
[5]    →(0∊⍴B)↓L1
```

```
    ∇ ∆MIN (continued)
[6]    R←A
[7]    →0
[8]   ⍝ Return right if left is empty:
[9]   L1:→(0∊⍴A)↓L2
[10]   R←B
[11]   →0
[12]  ⍝ Extract components from left argument:
[13]  L2:⎕IO←0
[14]   RANK←A[0]
[15]   DIM←A[1]
[16]   N←A[2]
[17]  ⍝ Branch unless a 0-way reduction:
[18]   →(×N)⍴L3
[19]   R←((3+RANK)⍴A),((3+RANK)↓A)⌊(3+RANK)↓B
[20]   →0
[21]  L3:S←A[3+⍳RANK]
[22]   DS←A[(3+RANK)+⍳N]
[23]   RRI←A[(3+RANK+N)+⍳S[DIM]]
[24]   ARRAY←S⍴(3+RANK+N+S[DIM])↓A
[25]  ⍝ Include components from right argument:
[26]   S←B[3+⍳RANK]
[27]   RRI←RRI,B[(3+RANK+N)+⍳S[DIM]]
[28]   ARRAY←ARRAY,[DIM]S⍴(3+RANK+N+S[DIM])↓B
[29]  ⍝ Use same logic as in ∆PLUSRED:
[30]   GRADE←⍋RRI
[31]   SORTED←RRI[GRADE]
[32]   LAST←SORTED≠1⌽SORTED
[33]   LAST[(×⍴LAST)⍴¯1+⍴LAST]←1
[34]   URRI←LAST/SORTED
[35]   ARRAY←⍎'ARRAY[',(DIM⍴';'),'GRADE',((RANK-DIM+1)⍴';'),'
       ]'
[36]   DIF←(⌈/[DIM]ARRAY)-⌊/[DIM]ARRAY
[37]   DIF←(⍋DIM=⍳RANK)⍉DIF∘.×+\¯1⌽LAST
[38]   DIF←(LAST/[DIM]⌊\[DIM]ARRAY-DIF)+LAST/[DIM]DIF
[39]   R←((RANK,DIM,N,(⍴DIF),DS),URRI),,DIF
    ∇
```

[WSID: REDUCE]

```
    ∇ R←A ∆AND B;ARRAY;CUM;DIM;DS;GRADE;LAST;N;RANK;RRI;S;
      SORTED;URRI;⎕IO
[1]   ⍝ Adds together two compressed Milky-Way arrays,
[2]   ⍝ returning a third such array.
[3]   ⍝
[4]   ⍝ Return left argument if right is empty:
[5]    →(0∊⍴B)↓L1
[6]    R←A
[7]    →0
[8]   ⍝ Return right if left is empty:
[9]   L1:→(0∊⍴A)↓L2
[10]   R←B
[11]   →0
```

```
    ∇ ∆AND (continued)
[12] ⍝ Extract components from left argument:
[13] L2:⎕IO←0
[14]  RANK←A[0]
[15]  DIM←A[1]
[16]  N←A[2]
[17] ⍝ Branch unless a 0-way reduction:
[18]  →(×N)ρL3
[19]  R←((3+RANK)ρA),((3+RANK)↓A)∧(3+RANK)↓B
[20]  →0
[21] L3:S←A[3+⍳RANK]
[22]  DS←A[(3+RANK)+⍳N]
[23]  RRI←A[(3+RANK+N)+⍳S[DIM]]
[24]  ARRAY←Sρ(3+RANK+N+S[DIM])↓A
[25] ⍝ Include components from right argument:
[26]  S←B[3+⍳RANK]
[27]  RRI←RRI,B[(3+RANK+N)+⍳S[DIM]]
[28]  ARRAY←ARRAY,[DIM]Sρ(3+RANK+N+S[DIM])↓B
[29] ⍝ Use same logic as in ∆PLUSRED:
[30]  GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ¯1+ρLAST]←1
[34]  URRI←LAST/SORTED
[35]  ARRAY←⍎'ARRAY[',(DIMρ';'),'GRADE',((RANK-DIM+1)ρ';'),'
      ]'
[36]  CUM←LAST/[DIM]+\[DIM]~ARRAY
[37]  CUM←CUM=(ρCUM)↑0,[DIM]CUM
[38]  R←((RANK,DIM,N,(ρCUM),DS),URRI),,CUM
    ∇
```

[WSID: REDUCE]

```
    ∇ R←A ∆OR B;ARRAY;CUM;DIM;DS;GRADE;LAST;N;RANK;RRI;S;
      SORTED;URRI;⎕IO
[1]  ⍝ Adds together two compressed Milky-Way arrays,
[2]  ⍝ returning a third such array.
[3]  ⍝
[4]  ⍝ Return left argument if right is empty:
[5]   →(0∊ρB)↓L1
[6]   R←A
[7]   →0
[8]  ⍝ Return right if left is empty:
[9]  L1:→(0∊ρA)↓L2
[10]  R←B
[11]  →0
[12] ⍝ Extract components from left argument:
[13] L2:⎕IO←0
[14]  RANK←A[0]
[15]  DIM←A[1]
[16]  N←A[2]
[17] ⍝ Branch unless a 0-way reduction:
[18]  →(×N)ρL3
[19]  R←((3+RANK)ρA),((3+RANK)↓A)∨(3+RANK)↓B
```

```
    ∇ ∆OR (continued)
[20]  →0
[21] L3:S←A[3+⍳RANK]
[22]  DS←A[(3+RANK)+⍳N]
[23]  RRI←A[(3+RANK+N)+⍳S[DIM]]
[24]  ARRAY←Sρ(3+RANK+N+S[DIM])↓A
[25] ⍝ Include components from right argument:
[26]  S←B[3+⍳RANK]
[27]  RRI←RRI,B[(3+RANK+N)+⍳S[DIM]]
[28]  ARRAY←ARRAY,[DIM]Sρ(3+RANK+N+S[DIM])↓B
[29] ⍝ Use same logic as in ∆PLUSRED:
[30]  GRADE←⍋RRI
[31]  SORTED←RRI[GRADE]
[32]  LAST←SORTED≠1⌽SORTED
[33]  LAST[(×ρLAST)ρ¯1+ρLAST]←1
[34]  URRI←LAST/SORTED
[35]  ARRAY←⍎'ARRAY[',(DIMρ';'),'GRADE',((RANK-DIM+1)ρ';'),'
      ]'
[36]  CUM←LAST/[DIM]+\[DIM]ARRAY
[37]  CUM←CUM≠(ρCUM)↑0,[DIM]CUM
[38]  R←((RANK,DIM,N,(ρCUM),DS),URRI),,CUM
    ∇
```

6.

```
        ⎕IO←1
        I1←'ECMPH'⍳TZONE                    (5 classes)
        I2←'BCPS'⍳TYPE                      (4 classes)
        I3←+/SALES∘.≥0 1E6 5E6              (3 classes)
  (or:  I3←0 1E6 5E6 LIOTA SALES    )
             (LIOTA is defined in Sorting and Searching chapter)
        I4←ALLSTATES CMIOTA STATES          (50 classes)
             (CMIOTA is defined in Sorting and Searching chapter)
        I5←301 304 310 322 329⍳MGR          (6 classes)


        SUM←(5 4 3 50 6,1 2 3 4 5)∆PLUSRED 1,[1]SALES,[.5]FIT

       ⍝ Returns a 3 (frequency, SALES, FIT) by 5 by 4 by 3 by 50
       ⍝ by 6 result as a compressed Milky-Way result.
```

1.

```
        (3 ∆PLUSWAY SUM)[1;]
    or
        3 3 PLUSRED 1
```

2.

```
        4 ∆PLUSWAY SUM
    or
        50 4 PLUSRED 1,[1]SALES,[.5]FIT
```

```
3.   (5 3 2 ∆PLUSWAY SUM)[2 3;;;]
  or
     6 3 4 5 3 2 PLUSRED SALES,[.5]FIT

4.   (4 2 ∆PLUSWAY SUM)[1;;]
  or
     50 4 4 2 PLUSRED 1

5.   (2 3 ∆PLUSWAY SUM)[3;;]
  or
     4 3 2 3 PLUSRED FIT
```

Chapter 8 Solutions

WRITING USER-FRIENDLY INTERACTIVE FUNCTIONS

1.

```
                                                    [WSID: INPUT]
    ∇ R←CHOICES LPROMPTE PROMPT
[1]   ⍝ Prompts for a single letter.  PROMPT is the
[2]   ⍝ character vector prompt.  CHOICES is a character
[3]   ⍝ vector of allowable single characters to be
[4]   ⍝ entered.  R is a one element vector index into
[5]   ⍝ CHOICES of the character entered or is a numeric
[6]   ⍝ scalar escape code if an escape word is typed.
[7]   ⍝ Requires: CPROMPTE.
[8]   L1:R←CPROMPTE PROMPT
[9]   ⍝ Exit if scalar escape code:
[10]   →(⍴⍴R)↓0
[11]   →(1=⍴R)⍴L2
[12]   ⎕←'** ENTER ONE CHARACTER ONLY'
[13]   →L1
[14]  ⍝ Convert R to index:
[15]  L2:R←CHOICES⍳R
[16]   →(R<⎕IO+⍴CHOICES)⍴0
[17]   ⎕←'** INVALID CHOICE.  ENTER ONE OF: '
[18]   ⎕←1↓,',',[⎕IO+0.5]CHOICES
[19]   →L1
    ∇
```

2. Insert the expression

```
R[(R='-')/⍳⍴R]←'¯'
```

in the functions NINPUT and NPROMPTE just before using ⎕VI on R. Insert the APL2 expression

```
((R='-')/R)←'¯'
```

in the functions NINPUT2 and NPROMPTE2 just before checking R for numeric characters.

3.

```
                                        [WSID: INPUT]
    ∇ PROPOSAL;AGES;NAME;NKIDS;R
[1]   ⍝ Illustration of input utility functions.
[2]   L1:→0 ESCAPE NAME←CPROMPTE 'NAME: '
[3]   L2:→0 ESCAPE NKIDS←1 NPROMPTE 'NUMBER OF KIDS: '
[4]    →L2 IF(~NKIDS∊0,⍳20)MESSAGE '** 0 TO 20 KIDS ONLY'
[5]    →L4 UNLESS×NKIDS
[6]   L3:→0 ESCAPE AGES←NKIDS NPROMPTE 'AGES OF KIDS: '
[7]    →L3 IF((AGES∨.≠⌈AGES)∨(AGES∨.<0)∨AGES∨.>99)MESSAGE '**
        0 TO 99 AGES ONLY'
[8]   ⍝ Allow alignment of paper:
[9]   L4:→0 ESCAPE CPROMPTE 'PRESS ENTER WHEN READY...'
[10]  ⍝ 3 blank lines:
[11]   ⎕← 3 1 ⍴' '
[12]   ⎕←'Dear ',NAME,':'
[13]  ⍝ 1 blank line:
[14]   ⎕←''
[15]   ⍞←'As a proud parent of ',(⍕NKIDS),' kid'
[16]   ⎕←((1≠NKIDS)⍴'s'),' (whose'
[17]   ⍞←'average age is ',⍕(+/AGES)÷NKIDS
[18]   ⎕←'), you need insurance.'
[19]   ⎕← 3 1 ⍴' '
[20]  ⍝ Allow alignment of paper:
[21]   →0 ESCAPE CPROMPTE ''
[22]   →0 ESCAPE R←'YN' LPROMPTE 'GENERATE ANOTHER PROPOSAL?
       '
[23]  ⍝ Do another if response is Y:
[24]   →L1 IF R=1
    ∇
```

Chapter 9 Solutions

MANIPULATING DATES

1.
```
1-1|((TODAYS360 MDATES)-TODAYS360 PDATES)÷180
```
or
```
1|((TODAYS360 PDATES)-TODAYS360 MDATES)÷180
```

These two expressions return different results if the purchase date occurs on a coupon date (1 and 0 respectively). The first expression is correct if the buyer receives the coupon and the second is correct if the seller receives the coupon.

2.
```
(1000×1.001*-/TODAYS RDATE,BDATE)-1000
```

3.
```
                                              [WSID: DATES]
    ∇ YYYYDDD←TOYD YYYYMMDD;DD;LEAP;MM;MMDD;YYYY;⎕IO
[1]   ⍝ Converts dates (YYYYMMDD) to Julian dates (YYYYDDD
[2]   ⍝ where DDD is number of days since prior Dec. 31).
[3]    ⎕IO←1
[4]    DD←100|YYYYMMDD
[5]    MMDD←10000|YYYYMMDD
[6]    MM←(MMDD-DD)÷100
[7]   ⍝ Year and year times 1000 (e.g. 1986000):
[8]    YYYY←(YYYYMMDD-MMDD)÷10000
[9]    YYYYDDD←1000×YYYY
[10]  ⍝ Add in days from start of month, and from start
[11]  ⍝ of year to start of month:
[12]   YYYYDDD←YYYYDDD+DD+(0 31 59 90 120 151 181 212 243 273
        304 334)[MM]
[13]  ⍝ Determine whether a leap year:
[14]   LEAP←(0=4|YYYY)∧(0≠100|YYYY)∨0=400|YYYY
[15]  ⍝ Add in leap day if month is March or later:
[16]   YYYYDDD←YYYYDDD+LEAP∧MM≥3
    ∇
```

```
                                         [WSID: DATES]
    ∇ YYYYMMDD←FROMYD YYYYDDD;DD;DDD;FEB29;LEAP;MM;YYYY;⎕IO
[1]  ⍝ Convert Julian dates (YYYYDDD where DDD is number of
[2]  ⍝ days since prior Dec. 31) to YYYYMMDD dates.
[3]   ⎕IO←1
[4]   DDD←1000|YYYYDDD
[5]  ⍝ Year and year times 10000 (e.g. 19860000):
[6]   YYYY←(YYYYDDD-DDD)÷1000
[7]   YYYYMMDD←10000×YYYY
[8]  ⍝ Determine whether a leap year:
[9]   LEAP←(0=4|YYYY)∧(0≠100|YYYY)∨0=400|YYYY
[10] ⍝ Is day a leap day (i.e. Feb. 29)?
[11]  FEB29←LEAP∧DDD=60
[12] ⍝ Subtract leap day if Feb. 29 or later to determine
      month:
[13]  DDD←DDD-LEAP∧DDD≥60
[14]  MM←(31 28 31 30 31 30 31 31 30 31 30 31 /⍳12)[DDD]
[15] ⍝ Days since start of month:
[16]  DD←DDD-(0 31 59 90 120 151 181 212 243 273 304 334)[MM
      ]
[17] ⍝ Add back one day if Feb. 29:
[18]  DD←DD+FEB29
[19]  YYYYMMDD←YYYYMMDD+DD+100×MM
    ∇
```

4.

```
WKDAYS←7 9⍴'MONDAY   TUESDAY  WEDNESDAYTHURS...SUNDAY   '
WKDAY←WKDAYS[⎕IO+7|1+TODAYS FROMQTS 3⍴⎕TS;]
(WKDAY≠' ')/WKDAY
```

Chapter 10 Solutions

WRITING REPORTS

1.
```
H1←15 15 3 HEADINGS '∩LAST YEAR∩THIS YEAR'
HDG←'∩AVG.←SALE∩TOTAL←SALES∩AVG.←SALE∩TOTAL←SALES'
HDG←HDG,'∩GROWTH←IN←TOTAL←SALES'
HDG←6 7 6 7 7 2 3 2 3 HEADINGS HDG
HDG[1 2;⍳1↓⍴H1]←H1
```

2.
```
⍝ Format the date:
FDATE←,'G<Z9/99/99>' ⎕FMT DATE ⍝ APL*PLUS or SHARP APL
FDATE←'55/55/50'⍕DATE ⍝ APL2
FDATE←1 1 0 1 1 0 1 1\6 0⍕DATE ⍝ Other APL systems
FDATE[3 6]←'/' ⍝ Other APL systems

TITLE←'⊃PAGE ',(⍕PNO),'∩∩FINANCIAL SUMMARY∩',FDATE
TITLE←TITLE,'∩WESTERN REGION'
TITLE←65 TITLES TITLE
```

3. Approach 1 (using the newline character):

```
TCNL←⎕TCNL ⍝ APL*PLUS
TCNL←⎕AV[156+⎕IO] ⍝ SHARP APL
TCNL←⎕TC[1+⎕IO] ⍝ APL2
TCNL←⎕AV[???] ⍝ Other APL systems

CMAT←('4(I5,X2),<',TCNL,'>,4F7.1') ⎕FMT 3 8⍴NMAT ⍝ APL*PLUS,
                                                  SHARP APL

CMAT←((28⍴'55550  '),TCNL,28⍴' 5550.0')⍕NMAT ⍝ APL2

CMAT←((8⍴7 0),8⍴7 1)⍕3 8⍴NMAT ⍝ Other APL systems
CMAT←(0 2↓3 28↑CMAT),TCNL,3 ¯28↑CMAT ⍝ Other APL systems
```

Approach 2 (using dyadic transpose):

```
CMAT←(6⍴2 0)⌽6 28⍴2 1 3⍉4 6 7⍴(12⍴7 0 7 1)⍕⍉NMAT
```

Chapter 13 Solutions

WORKSPACE DESIGN AND DOCUMENTATION

1. The following are good candidates for "visual representation manipulation" functions. Their listings are not presented here but are available on disk. See the Postscript at the end of the book.

Syntax: NEWVR←UNLAMP OLDVR

The UNLAMP function removes all comments from the visual representation. Both end-of-line and full-line comments are removed completely, including the comment symbol (⍝). The function lines are renumbered as needed to allow for deleted full-line comments. UNLAMP is different from UNCOMMENT in that UNCOMMENT does not delete the comment symbol on full-line comments and so has no line-renumbering.

Syntax: NEWVR←OBFUSCATE OLDVR

The OBFUSCATE function modifies all local identifiers (labels, result variable, argument variables, localized variables) by prefixing their names by the characters '∆∆'. In this way, the identifiers of the function are obfuscated so that chances of a name conflict are minimized when the function uses execute (⍎) to access variables or functions whose definitions are global to the function being obfuscated. WSDOC is an example of an obfuscated function.

Syntax: NEWVR←UNOBFUSCATE OLDVR

The UNOBFUSCATE function deletes all occurrences of the characters '∆∆', thereby undoing the effects of OBFUSCATE. Since UNOBFUSCATE does not limit its seach to just local identifiers, it will delete even those occurrences of '∆∆' which were not inserted by OBFUSCATE (such as in character constants, in comments or in the middle of identifier names).

Syntax: NEWVR←UNDIAMOND OLDVR

The UNDIAMOND function breaks all multi-statement lines (⋄ delimited) into single-statement lines, renumbering lines as needed. For example:

```
      ∇ TEST                                    ∇ TEST
[1]     A←0 ⋄ B←A+2 ⋄ C←A+B         →      [1]    A←0
[2]    ⍝ Proceed:                   →      [2]    B←A+2
[3]     CALC                        →      [3]    C←A+B
      ∇                                    [4]   ⍝ Proceed:
                                           [5]    CALC
                                                ∇
```

The UNDIAMOND function may be used when moving a function from an APL environment which supports statement separators (⋄) to one which does not; or you may find single-statement lines easier to read than multi-statement lines.

2.

```
                                         [WSID: WSDOC]
      ∇ WSDOC ∆∆PAGE;∆∆B;∆∆BOTTOM;∆∆C;∆∆D;∆∆DATA;∆∆DONE;∆∆F;
        ∆∆FIRST;∆∆FNS;∆∆FOOT;∆∆HEIGHT;∆∆I;∆∆IND;∆∆LAST;∆∆LEFT;
        ∆∆LEN;∆∆LIM;∆∆LINES;∆∆MARGIN;∆∆N;∆∆NAME;∆∆NL;
        ∆∆NONDISPLAY;∆∆P;∆∆PNO;∆∆QUOTE;∆∆R;∆∆S;∆∆T;∆∆TCNL;
        ∆∆TITLE;∆∆TOP;∆∆TXT;∆∆VARS;∆∆VR;∆∆W;∆∆WIDTH
[1]    ⍝ Displays paged WS documentation. All output
[2]    ⍝ is via ⎕← so replace all ⎕← by custom fn
[3]    ⍝ (e.g. PRINT) to redirect output.  PAGE: rows,
[4]    ⍝ columns, margins (top, bottom, left, right).
[5]     ∆∆TOP←∆∆PAGE[2+⎕IO]
[6]     ∆∆BOTTOM←∆∆PAGE[3+⎕IO]
[7]     ∆∆HEIGHT←∆∆PAGE[⎕IO]-∆∆TOP+∆∆BOTTOM
[8]     ∆∆LEFT←∆∆PAGE[4+⎕IO]
[9]     ∆∆WIDTH←∆∆PAGE[1+⎕IO]-∆∆PAGE[5+⎕IO]
[10]   ⍝ Construct newline character:
[11]    ∆∆TCNL←⎕TCNL ⍝ APL*PLUS
[12]   ⍝ TCNL←⎕TC[1+⎕IO] ⍝ APL2
[13]   ⍝ TCNL←⎕AV[156+⎕IO] ⍝ SHARP APL
[14]   ⍝ Format today's date:
[15]    ∆∆D←⍕⎕TS[1 2 0 +⎕IO]
[16]    ∆∆D[(∆∆D=' ')/⍳⍴∆∆D]←'/'
[17]   ⍝ Format the time:
[18]    ∆∆T←(⍕⎕TS[3+⎕IO]),':',¯2↑'0',⍕⎕TS[4+⎕IO]
[19]   ⍝ Format the WSID:
[20]    ∆∆TITLE←⎕WSID ⍝ APL*PLUS
[21]   ⍝ TITLE←2 ⎕WS 1 ⍝ SHARP APL
[22]   ⍝ S←100 ⎕SVO 'C' ⍝ APL2 (TSO)
[23]   ⍝ S←0 0 1 1 ⎕SVC 'C' ⍝ APL2 (TSO)
[24]   ⍝ C←')WSID' ⍝ APL2 (TSO)
[25]   ⍝ TITLE←C[⎕IO;]~' ' ⍝ APL2 (TSO)
```

```
    ∇ WSDOC (continued)
[26] ⍝ S←⎕SVR 'C' ⍝ APL2 (TSO)
[27] ⍝ Provide WSID as left arg otherwise:
[28] ⍝ TITLE←WSID ⍝ APL2 (CMS)
[29] ⍝ Delete leading/trailing blanks:
[30]  ΔΔTITLE←(-+/∧\⌽ΔΔTITLE=' ')↓(+/∧\ΔΔTITLE=' ')↓ΔΔTITLE
[31] ⍝ Format page title:
[32]  ΔΔTITLE←(ΔΔTOPρΔΔTCNL),ΔΔWIDTH↑(ΔΔLEFTρ' '),ΔΔTITLE,'
      * ',ΔΔD,' ',ΔΔT
[33] ⍝ Insert page number:
[34]  ΔΔPNO←1
[35]  ΔΔT←'PAGE 1'
[36]  ΔΔTITLE[(-ρΔΔT)↑ιρΔΔTITLE]←ΔΔT
[37] ⍝ Build first page:
[38]  ΔΔTXT←ΔΔTITLE,ΔΔTCNL
[39] ⍝ Keep track of lines used so far (below
[40] ⍝ top margin) in TXT:
[41]  ΔΔLINES←2
[42] ⍝ Build nondefault environment:
[43]  ΔΔT←ΔΔTCNL,(ΔΔLEFTρ' '),'NONDEFAULT WORKSPACE
      ENVIRONMENT:',ΔΔTCNL
[44]  ΔΔMARGIN←(ΔΔLEFT+3)ρ' '
[45]  ΔΔQUOTE←''''
[46] ⍝ Define chars which don't display normally:
[47]  ΔΔNONDISPLAY←⎕TCNL,⎕TCLF,⎕TCBS,⎕TCBEL,⎕TCDEL,⎕TCNUL,
      ⎕TCESC,⎕TCFF ⍝ APL*PLUS
[48] ⍝ ΔΔNONDISPLAY←⎕AV[⎕IO+0 1 156 158 159] ⍝ SHARP APL
[49] ⍝ ΔΔNONDISPLAY←⎕TC ⍝ APL2
[50] ⍝ Format nondefaults:
[51] ⍝ Branch if default ⎕LX:
[52]  →(ρΔΔDATA←,⎕LX)↓ΔΔL2
[53] ⍝ Replace nondisplayable chars by ⊟:
[54]  ΔΔDATA[(ΔΔDATA∊ΔΔNONDISPLAY)/ιρΔΔDATA]←'⊟'
[55] ⍝ Double up quote chars:
[56]  ΔΔDATA←ΔΔQUOTE,((1+ΔΔDATA=ΔΔQUOTE)/ΔΔDATA),ΔΔQUOTE
[57] ⍝ Width available (after '   ⎕LX←'):
[58]  ΔΔW←ΔΔWIDTH-ΔΔLEFT+7
[59] ⍝ Branch if data will fit on a single line:
[60]  →(ΔΔW≥ρΔΔDATA)/ΔΔL1
[61] ⍝ Else truncate and show '...':
[62]  ΔΔDATA←((ΔΔW+¯3)ρΔΔDATA),'...'
[63] ΔΔL1:ΔΔT←ΔΔT,ΔΔTCNL,ΔΔMARGIN,'⎕LX←',ΔΔDATA
[64] ΔΔL2:ΔΔT←ΔΔT,(1≠⎕IO)/ΔΔTCNL,ΔΔMARGIN,'⎕IO←',⍕⎕IO
[65]  ΔΔT←ΔΔT,(10≠⎕PP)/ΔΔTCNL,ΔΔMARGIN,'⎕PP←',⍕⎕PP
[66]  ΔΔT←ΔΔT,(16807≠⎕RL)/ΔΔTCNL,ΔΔMARGIN,'⎕RL←',⍕⎕RL
[67] ⍝ Perform precise comparison for ⎕CT:
[68]  ΔΔC←⎕CT
[69]  ⎕CT←0
[70]  ΔΔT←ΔΔT,(ΔΔC≠2*¯46)/ΔΔTCNL,ΔΔMARGIN,'⎕CT←',⍕ΔΔC
[71] ⍝ Use 1E¯13 instead of 2*¯46 for APL2
[72]  ⎕CT←ΔΔC
[73] ⍝ Other nondefault workspace environment
[74] ⍝ parameters which may be included are: state
[75] ⍝ indicator, workspace size, workspace available,
```

```
    ∇ WSDOC (continued)
[76] ⍝ symbols reserved and used, error latent
[77] ⍝ expression (trap definition), attention latent
[78] ⍝ expression, etc.
[79] ⍝
[80] ⍝ Include blank line after any nondefaults:
[81]  ΔΔT←ΔΔT,ΔΔTCNL
[82] ⍝ Attach nondefaults, if any, to page:
[83]  ΔΔN←+/ΔΔT=ΔΔTCNL
[84]  ΔΔN←ΔΔN×ΔΔN>3
[85]  ΔΔTXT←ΔΔTXT,(×ΔΔN)/ΔΔT
[86]  ΔΔLINES←ΔΔLINES+ΔΔN
[87] ⍝
[88] ⍝ Variables:
[89]  ΔΔVARS←⎕NL 2
[90] ⍝ Squeeze out local (WSDOC) variables:
[91]  ΔΔVARS←(ΔΔVARS[; 0 1 +⎕IO]∨.≠'Δ')⌿ΔΔVARS
[92]  ΔΔMARGIN←ΔΔLEFT⍴' '
[93] ⍝ Branch if no variables:
[94]  →(1↑⍴ΔΔVARS)↓ΔΔL7
[95]  ΔΔTXT←ΔΔTXT,ΔΔTCNL,ΔΔMARGIN,'GLOBAL WORKSPACE
      VARIABLES:',ΔΔTCNL
[96]  ΔΔLINES←ΔΔLINES+2
[97] ⍝ Sort variable names if not already:
[98] ⍝ VARS←VARS[⎕AV⍋VARS;]
[99] ⍝ Right justify variable names:
[100]  ΔΔT←ΔΔVARS+.=' '
[101]  ΔΔVARS←(-ΔΔT)⌽ΔΔVARS
[102] ⍝ Drop leading all blank columns (possible by
[103] ⍝ deleting  vars):
[104]  ΔΔVARS←(0,⌊/ΔΔT)↓ΔΔVARS
[105] ⍝ Indent variable names for left margin plus 3:
[106]  ΔΔVARS←(((1↑⍴ΔΔVARS),ΔΔLEFT+3)⍴' '),ΔΔVARS
[107] ⍝ Include heading:
[108]  ΔΔT←ΔΔTCNL,((-1↓⍴ΔΔVARS)↑'NAME'),' ←    SHAPE ⍴ VALUE'
      ,ΔΔTCNL
[109]  ΔΔTXT←ΔΔTXT,ΔΔT,((-1↓⍴ΔΔVARS)↑'----'),'     -----    -
      ----'
[110]  ΔΔLINES←ΔΔLINES+2
[111] ⍝ Loop by variable:
[112]  ΔΔI←⎕IO+¯1
[113]  ΔΔLIM←(1↑⍴ΔΔVARS)-~⎕IO
[114] ΔΔL3:→(ΔΔLIM<ΔΔI←ΔΔI+1)/ΔΔL6
[115] ⍝ Format shape:
[116]  ΔΔS←(⍕⍴ΔΔDATA←⍎ΔΔNAME←ΔΔVARS[ΔΔI;]),' ⍴ '
[117] ⍝ Omit shape and ⍴ if a scalar; pad to line up
[118] ⍝ with ⍴'s:
[119]  ΔΔS←(-11⌈⍴ΔΔS)↑(3<⍴ΔΔS)/ΔΔS
[120] ⍝ Combine name and shape; compute remaining width:
[121]  ΔΔS←ΔΔNAME,' ←',ΔΔS
[122]  ΔΔW←ΔΔWIDTH-⍴ΔΔS
[123] ⍝ Branch if a numeric variable:
[124]  →(0=1↑0⍴ΔΔDATA)⍴ΔΔL4
```

```
    ∇ WSDOC (continued)
 ⍝ Else data is character; consider only up to
 ⍝ W chars:
  ∆∆DATA←(∆∆W⌊×/⍴∆∆DATA)⍴∆∆DATA
 ⍝ Replace nondisplayable chars by ⌹:
  ∆∆DATA[(∆∆DATA∊∆∆NONDISPLAY)/⍳⍴∆∆DATA]←'⌹'
 ⍝ Double up quote chars:
  ∆∆DATA←∆∆QUOTE,((1+∆∆DATA=∆∆QUOTE)/∆∆DATA),∆∆QUOTE
 ⍝ Branch if data will fit on a single line:
  →(∆∆W≥⍴∆∆DATA)/∆∆L5
 ⍝ Else truncate and show '...':
  ∆∆DATA←((∆∆W+¯3)⍴∆∆DATA),'...'
  →∆∆L5
 ⍝ If data is numeric, consider only up to
 ⍝ 1+W÷2 elements:
 ∆∆L4:∆∆DATA←((1+⌈∆∆W÷2)⌊×/⍴∆∆DATA)⍴∆∆DATA
 ⍝ Format numbers to 10 digits (CLEAR WS default):
  ∆∆P←⎕PP
  ⎕PP←10
  ∆∆DATA←⍕∆∆DATA
  ⎕PP←∆∆P
 ⍝ Format value from empty array as ⍳0:
  ∆∆DATA←∆∆DATA,(0=⍴∆∆DATA)/'⍳0'
 ⍝ Branch if data will fit on a single line:
  →(∆∆W≥⍴∆∆DATA)/∆∆L5
 ⍝ Else truncate at last space within width and
 ⍝ show '...':
  ∆∆DATA←((+/∨\' '=⌽(∆∆W+¯3)⍴∆∆DATA)⍴∆∆DATA),'...'
 ⍝
 ⍝ Append variable definition to page:
 ∆∆L5:∆∆TXT←∆∆TXT,∆∆TCNL,∆∆S,∆∆DATA
  ∆∆LINES←∆∆LINES+1
 ⍝ Branch for more vars unless bottom of page:
  →(∆∆LINES<∆∆HEIGHT)/∆∆L3
 ⍝ Display page:
  ⎕←∆∆TXT,(∆∆BOTTOM+∆∆HEIGHT-∆∆LINES)⍴∆∆TCNL
 ⍝ Format new page:
  ∆∆TXT←∆∆TITLE,∆∆TCNL
  ∆∆PNO←∆∆PNO+1
  ∆∆T←'PAGE ',⍕∆∆PNO
  ∆∆TXT[(-⍴∆∆T)↑⍳⍴∆∆TITLE]←∆∆T
  ∆∆LINES←2
 ⍝ Branch if no more vars:
  →(∆∆I≥∆∆LIM)/∆∆L7
 ⍝ Else insert new heading:
  ∆∆TXT←∆∆TXT,∆∆TCNL,∆∆MARGIN,'GLOBAL WORKSPACE
 VARIABLES (CONT.):',∆∆TCNL
  ∆∆LINES←∆∆LINES+2
  ∆∆T←∆∆TCNL,((-1↓⍴∆∆VARS)↑'NAME'),' ←   SHAPE ⍴ VALUE'
 ,∆∆TCNL
  ∆∆TXT←∆∆TXT,∆∆T,((-1↓⍴∆∆VARS)↑'----'),'     -----   -
 ----'
  ∆∆LINES←∆∆LINES+2
```

```
    ∇ WSDOC (continued)
[174] ⍝ Branch for more vars:
[175]  →ΔΔL3
[176] ⍝
[177] ⍝ No more vars (include blank line):
[178] ΔΔL6:ΔΔTXT←ΔΔTXT,ΔΔTCNL
[179]  ΔΔLINES←ΔΔLINES+1
[180] ⍝
[181] ⍝ Functions:
[182] ΔΔL7:ΔΔFNS←⎕NL 3
[183] ⍝ Squeeze out this (WSDOC) function:
[184]  ΔΔFNS←(ΔΔFNS∨.≠(1↓ρΔΔFNS)↑'WSDOC')⌿ΔΔFNS
[185] ⍝ Use next, not prior, line if CRΔVR is used below:
[186] ⍝ FNS←(∧/FNS∨.≠⍉(2,1↓ρFNS)↑2 5ρ'WSDOCCRΔVR')⌿FNS
[187] ⍝
[188] ⍝ Branch if some fns:
[189]  →(×ΔΔLIM←1↑ρΔΔFNS)/ΔΔL8
[190] ⍝ Exit if nothing on page:
[191]  →(ΔΔLINES=2)/0
[192] ⍝ Else display page and exit:
[193]  ⎕←ΔΔTXT,(ΔΔBOTTOM+ΔΔHEIGHT-ΔΔLINES)ρΔΔTCNL
[194]  →0
[195] ⍝ Branch if at least 4 lines left on page:
[196] ΔΔL8:→(ΔΔLINES≤ΔΔHEIGHT+¯4)/ΔΔL9
[197] ⍝ Else display page:
[198]  ⎕←ΔΔTXT,(ΔΔBOTTOM+ΔΔHEIGHT-ΔΔLINES)ρΔΔTCNL
[199] ⍝ Format new page:
[200]  ΔΔTXT←ΔΔTITLE,ΔΔTCNL
[201]  ΔΔPNO←ΔΔPNO+1
[202]  ΔΔT←'PAGE ',⍕ΔΔPNO
[203]  ΔΔTXT[(-ρΔΔT)↑⍳ρΔΔTITLE]←ΔΔT
[204]  ΔΔLINES←2
[205] ⍝
[206] ΔΔL9:ΔΔTXT←ΔΔTXT,ΔΔTCNL,ΔΔMARGIN,'FUNCTIONS:',ΔΔTCNL
[207]  ΔΔLINES←ΔΔLINES+2
[208] ⍝ Sort fn names if not already:
[209] ⍝ FNS←FNS[⎕AV⍋FNS;]
[210] ⍝
[211] ⍝ Pad fn names with 3 leading blank columns:
[212]  ΔΔFNS←(0 ¯3 -ρΔΔFNS)↑ΔΔFNS
[213] ⍝ How many "columns" of fn names will fit across pg?
[214]  ΔΔW←1↓ρΔΔFNS
[215]  ΔΔN←⌊ΔΔWIDTH÷ΔΔW
[216] ⍝ How many rows will this take?
[217]  ΔΔR←⌈ΔΔLIM÷ΔΔN
[218] ⍝ Pad bottom of fn list for subsequent reshape:
[219]  ΔΔFNS←((ΔΔR×ΔΔN),ΔΔW)↑ΔΔFNS
[220] ⍝ Shuffle fn list to get names in desired order:
[221]  ΔΔF←((ΔΔR,ΔΔLEFT)ρ' '),(ΔΔR,ΔΔN×ΔΔW)ρ(⎕IO+ 1 0 2)⍉(
      ΔΔN,ΔΔR,ΔΔW)ρΔΔFNS
[222] ⍝ How many will fit on this page?
[223] ΔΔL10:ΔΔN←(1ρρΔΔF)⌊ΔΔHEIGHT-ΔΔLINES
[224] ⍝ Stick them on:
[225]  ΔΔTXT←ΔΔTXT,,ΔΔTCNL,(ΔΔN,1↓ρΔΔF)↑ΔΔF
```

```
     ∇ WSDOC (continued)
[226]  ΔΔLINES←ΔΔLINES+ΔΔN
[227] ⍝ Drop them off:
[228]  ΔΔF←(ΔΔN,0)↓ΔΔF
[229] ⍝ Branch if none left:
[230]  →(1↑ρΔΔF)↓ΔΔL11
[231] ⍝ Display page:
[232]  ⎕←ΔΔTXT,ΔΔBOTTOMρΔΔTCNL
[233] ⍝ Format new page:
[234]  ΔΔTXT←ΔΔTITLE,ΔΔTCNL
[235]  ΔΔPNO←ΔΔPNO+1
[236]  ΔΔT←'PAGE ',⍕ΔΔPNO
[237]  ΔΔTXT[(-ρΔΔT)↑ιρΔΔTITLE]←ΔΔT
[238]  ΔΔLINES←2
[239]  ΔΔTXT←ΔΔTXT,ΔΔTCNL,ΔΔMARGIN,'FUNCTIONS (CONT.):',
      ΔΔTCNL
[240]  ΔΔLINES←ΔΔLINES+2
[241] ⍝ Put remaining fns on this page:
[242]  →ΔΔL10
[243] ⍝
[244] ⍝ Include 5 blank lines after fn list:
[245] ΔΔL11:ΔΔN←5⌊ΔΔHEIGHT-ΔΔLINES
[246]  ΔΔTXT←ΔΔTXT,ΔΔNρΔΔTCNL
[247]  ΔΔLINES←ΔΔLINES+ΔΔN
[248] ⍝
[249] ⍝ Loop by function (keep track of 1st one for
[250] ⍝ footnote):
[251]  ΔΔFIRST←ΔΔFNS[⎕IO;]
[252]  ΔΔFIRST←(ΔΔFIRST≠' ')/ΔΔFIRST
[253]  ΔΔLAST←''
[254]  ΔΔI←⎕IO+¯1
[255]  ΔΔLIM←ΔΔLIM-~⎕IO
[256] ΔΔL12:→(ΔΔDONE←ΔΔLIM<ΔΔI←ΔΔI+1)/ΔΔL18
[257]  ΔΔNAME←ΔΔFNS[ΔΔI;]
[258]  ΔΔNAME←(ΔΔNAME≠' ')/ΔΔNAME
[259]  ΔΔVR←⎕VR ΔΔNAME ⍝ APL*PLUS
[260] ⍝ VR←1 ⎕FD NAME ⍝ SHARP APL
[261] ⍝ VR←CRΔVR ⎕CR NAME ⍝ APL2
[262] ⍝ Get next fn if this one locked:
[263]  →(ρΔΔVR)↓ΔΔL12
[264] ⍝ Find newline characters:
[265]  ΔΔNL←(ΔΔVR=ΔΔTCNL)/ιρΔΔVR
[266] ⍝ Lengths of lines:
[267]  ΔΔLEN←¯1+ΔΔNL-¯1↓(⎕IO+¯1),ΔΔNL
[268] ⍝ Branch if no lines too long:
[269]  →(∨/ΔΔT←ΔΔLEN>ΔΔWIDTH-ΔΔLEFT)↓ΔΔL15
[270] ⍝ Retain starting indices and lengths of too
[271] ⍝ long lines:
[272]  ΔΔNL←ΔΔT/¯1↓(⎕IO+¯1),ΔΔNL
[273]  ΔΔLEN←ΔΔT/ΔΔLEN
[274] ⍝ Flag chars which shouldn't be broken when
[275] ⍝ contiguous:
[276]  ΔΔB←ΔΔVR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZΔabcdefghijklmnop
      qrstuvwxyz⍙0123456789.¯⎕'
```

```
    ∇ WSDOC (continued)
[277] ⍝ Break points exist everywhere but where these
[278] ⍝ chars occur and are followed by another one of
[279] ⍝ these chars:
[280]  ∆∆B←∆∆B⍲1⌽∆∆B
[281] ⍝ Indices of break chars found so far:
[282]  ∆∆IND←⍳0
[283] ⍝ Relative indices of first WIDTH chars
[284] ⍝ following known breaks:
[285]  ∆∆R←(~⎕IO)+⌽⍳∆∆WIDTH-∆∆LEFT
[286] ⍝ Displacement to next break points:
[287] ∆∆L13:∆∆D←+/∨\∆∆B[∆∆NL∘.+∆∆R]
[288] ⍝ Use full length if break point under 1/2 line:
[289]  ∆∆D[(∆∆D<(⍴∆∆R)÷2)/⍳⍴∆∆D]←⍴∆∆R
[290] ⍝ Update vars for new break points:
[291]  ∆∆NL←∆∆NL+∆∆D
[292]  ∆∆IND←∆∆IND,∆∆NL
[293]  ∆∆LEN←∆∆LEN-∆∆D
[294] ⍝ Any lines still too long (addl 6 space indent)?
[295] ⍝ Branch if not:
[296]  ∆∆R←∆∆WIDTH-∆∆LEFT+6
[297]  ∆∆T←∆∆LEN>∆∆R
[298]  ∆∆LEN←∆∆T/∆∆LEN
[299]  →(⍴∆∆LEN)↓∆∆L14
[300]  ∆∆NL←∆∆T/∆∆NL
[301]  ∆∆R←(~⎕IO)+⌽⍳∆∆R
[302] ⍝ Repeat:
[303]  →∆∆L13
[304] ⍝ Build replication vector to insert line breaks:
[305] ∆∆L14:∆∆R←(⍴∆∆VR)⍴1
[306] ⍝ Allow 8 positions for char, newline, 6 spaces:
[307]  ∆∆R[∆∆IND]←8
[308] ⍝ Expand visual representation:
[309]  ∆∆VR←∆∆R/∆∆VR
[310] ⍝ Adjust indices for new VR:
[311]  ∆∆IND←∆∆IND[⍋∆∆IND]+7×(⍳⍴∆∆IND)-⎕IO
[312] ⍝ Insert break characters:
[313]  ∆∆VR[∆∆IND∘.+(~⎕IO)+⍳7]←((⍴∆∆IND),7)⍴∆∆TCNL,6⍴' '
[314] ⍝ Redefine NL:
[315]  ∆∆NL←(∆∆VR=∆∆TCNL)/⍳⍴∆∆VR
[316] ⍝ Branch if no left margin:
[317] ∆∆L15:→∆∆LEFT↓∆∆L16
[318] ⍝ Build replication vector to insert left margin:
[319]  ∆∆R←(⍴∆∆VR)⍴1
[320] ⍝ Allow positions for margin:
[321]  ∆∆R[¯1↓∆∆NL]←∆∆LEFT+1
[322] ⍝ Expand visual representation:
[323]  ∆∆VR←∆∆MARGIN,∆∆R/∆∆VR
[324] ⍝ Adjust newline indices for new VR:
[325]  ∆∆NL←∆∆NL+∆∆LEFT×(⍳⍴∆∆NL)+~⎕IO
[326] ⍝ Insert left margin:
[327]  ∆∆VR[(¯1↓∆∆NL)∘.+(~⎕IO)+⍳∆∆LEFT]←' '
[328] ⍝ Branch if fn won't fit on current page:
[329] ∆∆L16:→(∆∆HEIGHT<1+∆∆LINES+⍴∆∆NL)/∆∆L18
```

```
    ∇ WSDOC (continued)
[330] ΔΔL17:ΔΔTXT←ΔΔTXT,ΔΔTCNL,ΔΔVR
[331]  ΔΔLINES←ΔΔLINES+1+ρΔΔNL
[332]  ΔΔLAST←ΔΔNAME
[333] ⍝
[334] ⍝ Incl. 2 blank lines after each fn (1 in VR
[335] ⍝ already):
[336]  ΔΔN←1⌊ΔΔHEIGHT-ΔΔLINES
[337]  ΔΔTXT←ΔΔTXT,ΔΔNρΔΔTCNL
[338]  ΔΔLINES←ΔΔLINES+ΔΔN
[339] ⍝ Get next fn:
[340]  →ΔΔL12
[341] ⍝ Prepare footnote:
[342] ΔΔL18:ΔΔFOOT←ΔΔFIRST,(((ρΔΔFIRST)≠ρΔΔLAST)∨ΔΔFIRST∨.≠(
      ρΔΔFIRST)↑ΔΔLAST)/' → ',ΔΔLAST
[343]  ΔΔFOOT←(×ρΔΔLAST)/(-ΔΔWIDTH)↑ΔΔFOOT
[344] ⍝ Display page:
[345]  ⎕←ΔΔTXT,((ΔΔHEIGHT-ΔΔLINES)ρΔΔTCNL),ΔΔFOOT,ΔΔBOTTOMρ
      ΔΔTCNL
[346] ⍝ Exit if no functions left:
[347]  →ΔΔDONE/0
[348] ⍝ Format new page:
[349]  ΔΔFIRST←ΔΔNAME
[350]  ΔΔTXT←ΔΔTITLE,ΔΔTCNL
[351]  ΔΔPNO←ΔΔPNO+1
[352]  ΔΔT←'PAGE ',⍕ΔΔPNO
[353]  ΔΔTXT[(-ρΔΔT)↑⍳ρΔΔTITLE]←ΔΔT
[354]  ΔΔLINES←2
[355] ⍝ Branch if fn will fit on the new page (with
[356] ⍝ footnote):
[357]  →(ΔΔHEIGHT≥1+ΔΔLINES+ρΔΔNL)/ΔΔL17
[358] ⍝ Flag newlines which end entire (not broken)
[359] ⍝ fn lines:
[360]  ΔΔB←(ΔΔVR[(1+ΔΔLEFT)+¯1↓ΔΔNL]='['),1
[361] ⍝ Compute no. of newlines to take for current pg:
[362]  ΔΔN←+/∨\⌽(ΔΔHEIGHT-2+ΔΔLINES)ρΔΔB
[363]  ΔΔLINES←ΔΔLINES+ΔΔN+1
[364]  ΔΔT←ΔΔN↓ΔΔNL
[365] ⍝ Compute no. of chars to take for current page:
[366]  ΔΔN←(~⎕IO)+ΔΔNL[ΔΔN-~⎕IO]
[367]  ΔΔNL←ΔΔT-ΔΔN
[368] ⍝ Include these chars on page and squeeze from VR:
[369]  ΔΔTXT←ΔΔTXT,ΔΔTCNL,ΔΔNρΔΔVR
[370]  ΔΔVR←ΔΔN↓ΔΔVR
[371]  ΔΔLAST←ΔΔNAME
[372] ⍝ Include fn name at top of remainder of VR:
[373]  ΔΔT←(ΔΔLEFTρ' '),'    ∇ ',ΔΔNAME,' (CONT.)',ΔΔTCNL
[374]  ΔΔVR←ΔΔT,ΔΔVR
[375]  ΔΔNL←(ΔΔT⍳ΔΔTCNL),ΔΔNL+ρΔΔT
[376] ⍝ Branch to display page:
[377]  →ΔΔL18
    ∇
```

3.

[WSID: USEDBY]

```
    ∇ USEDBY ΔΔFNS;⎕IO;ΔΔALP;ΔΔAN;ΔΔARGS;ΔΔB;ΔΔBL;ΔΔC;ΔΔCOLS
      ;ΔΔGDSP;ΔΔGLOBAL;ΔΔGNUM;ΔΔHDR;ΔΔI;ΔΔIDLEN;ΔΔIDS;
      ΔΔIDSTART;ΔΔIDTYPES;ΔΔIND;ΔΔINDENT;ΔΔINDEX;ΔΔJ;ΔΔLAB;
      ΔΔLCGLOBAL;ΔΔLCPARSED;ΔΔLDSP;ΔΔLEN;ΔΔLIM;ΔΔLNUM;ΔΔLOC;
      ΔΔLOCAL;ΔΔLTYPE;ΔΔN;ΔΔNAME;ΔΔNAMES;ΔΔNC;ΔΔNCCON;ΔΔNCMT
      ;ΔΔNID;ΔΔNL;ΔΔNQ;ΔΔNUM;ΔΔPAN;ΔΔPARSE;ΔΔPARSED;ΔΔPBL;
      ΔΔPID;ΔΔPLAB;ΔΔPNUM;ΔΔPSID;ΔΔR;ΔΔRESULT;ΔΔRVAR;ΔΔS;
      ΔΔSTART;ΔΔT;ΔΔTCNL;ΔΔVR
[1]   ⍝ Displays chart of fns and vars called by the fns
[2]   ⍝ specified in the character matrix or vector
[3]   ⍝ (delimited by spaces) argument.  Requires subfns:
[4]   ⍝ CMIOTA; (and CRΔVR if not APL*PLUS or SHARP APL).
[5]   ⍝ Use origin 1:
[6]    ⎕IO←1
[7]   ⍝ Indent per level:
[8]    ΔΔINDENT←3
[9]   ⍝ Branch unless fns a matrix:
[10]   →(2≠ρρΔΔFNS)ρΔΔL1
[11]  ⍝ Left justify char mat:
[12]   ΔΔFNS←(+/∧\ΔΔFNS=' ')⌽ΔΔFNS
[13]   →ΔΔL2
[14]  ⍝ Flag blank before and last char of each name:
[15]  ΔΔL1:ΔΔFNS←' ',ΔΔFNS
[16]   ΔΔB←ΔΔFNS=' '
[17]   ΔΔT←(ΔΔB≠1⌽ΔΔB)/⍳ρΔΔB
[18]   ΔΔT←(((ρΔΔT)÷2),2)ρΔΔT
[19]  ⍝ Lengths and starts of each name:
[20]   ΔΔLEN←-/⌽ΔΔT
[21]   ΔΔSTART←ΔΔT[;1]
[22]  ⍝ Number of rows and cols in desired matrix:
[23]   ΔΔR←ρΔΔLEN
[24]   ΔΔC←0⌈⌈/ΔΔLEN
[25]  ⍝ Blank, raveled array:
[26]   ΔΔT←(ΔΔR×ΔΔC)ρ' '
[27]  ⍝ Compute indices: (⍳LEN[1]),(⍳LEN[2]),...
[28]   ΔΔI←ΔΔLEN/-¯1↓0,+\ΔΔLEN
[29]   ΔΔI←ΔΔI+⍳ρΔΔI
[30]  ⍝ Insert fn names into raveled matrix:
[31]   ΔΔT[ΔΔI+ΔΔLEN/ΔΔC×¯1+⍳ρΔΔLEN]←ΔΔFNS[ΔΔI+ΔΔLEN/ΔΔSTART]
[32]  ⍝ Reshape to matrix:
[33]   ΔΔFNS←(ΔΔR,ΔΔC)ρΔΔT
[34]  ⍝ Exit if no fns specified:
[35]  ΔΔL2:ΔΔR←1ρρΔΔFNS
[36]   →ΔΔR↓0
[37]  ⍝ Construct newline character:
[38]   ΔΔTCNL←⎕TCNL ⍝ APL*PLUS
[39]  ⍝ ΔΔTCNL←⎕TC[1+⎕IO] ⍝ APL2
[40]  ⍝ ΔΔTCNL←⎕AV[156+⎕IO] ⍝ SHARP APL
[41]  ⍝ Initialize tracking variables...
[42]  ⍝ Character matrix of all distinct identifiers
[43]  ⍝ found so far:
[44]   ΔΔNAMES←((ΔΔFNS CMIOTA ΔΔFNS)=⍳ΔΔR)⌿ΔΔFNS
```

```
    ∇ USEDBY (continued)
[45] ⍝ Index (into NAMES) vector of fns analyzed
[46] ⍝ so far (¯1=initial call):
[47]  ∆∆PARSED←,¯1
[48] ⍝ Index (into NAMES) vector of global names for
[49] ⍝ fn in PARSED:
[50]  ∆∆GLOBAL←∆∆NAMES CMIOTA ∆∆FNS
[51] ⍝ No. of globals in GLOBAL for fn in PARSED:
[52]  ∆∆GNUM←∆∆R
[53] ⍝ No. of globals in GLOBAL before those for fn
[54] ⍝ in PARSED:
[55]  ∆∆GDSP←,0
[56] ⍝ Index (into NAMES) vector of local names for
[57] ⍝ fn in PARSED:
[58]  ∆∆LOCAL←⍳0
[59] ⍝ No. of locals in LOCAL for fn in PARSED:
[60]  ∆∆LNUM←,0
[61] ⍝ No. of locals in LOCAL before those for fn
[62] ⍝ in PARSED:
[63]  ∆∆LDSP←,0
[64] ⍝ Integer vector of local var types for each elt
[65] ⍝ of LOCAL (1:label; 2:result; 3:argument; 4:local):
[66]  ∆∆LTYPE←⍳0
[67] ⍝ Index into PARSED of object whose globals are
[68] ⍝ currently being evaluated:
[69]  ∆∆LCPARSED←,1
[70] ⍝ Index into partition of GLOBAL (for the object
[71] ⍝ whose globals are currently being evaluated) of
[72] ⍝ the object being evaluated:
[73]  ∆∆LCGLOBAL←,1
[74] ⍝
[75] ⍝ Determine index into NAMES of object being evaluated:
[76] ∆∆LOOP:∆∆INDEX←∆∆GLOBAL[∆∆GDSP[∆∆LCPARSED[1]]+
      ∆∆LCGLOBAL[1]]
[77] ⍝ What's its name?
[78]  ∆∆NAME←∆∆NAMES[∆∆INDEX;]
[79]  ∆∆NAME←(∆∆NAME≠' ')/∆∆NAME
[80] ⍝ Loop on elts of LCPARSED from local to global
[81] ⍝ (I=2,3,...) (to check whether object is locally
[82] ⍝ defined at higher level):
[83]  ∆∆I←1
[84]  ∆∆LIM←¯1+⍴∆∆LCPARSED
[85] ∆∆LP1:→(∆∆LIM<∆∆I←∆∆I+1)⍴∆∆L3
[86]  ∆∆IND←∆∆LCPARSED[∆∆I]
[87] ⍝ List of local objects at this level:
[88]  ∆∆T←∆∆LOCAL[∆∆LDSP[∆∆IND]+⍳∆∆LNUM[∆∆IND]]
[89] ⍝ Search for this object:
[90]  ∆∆J←∆∆T⍳∆∆INDEX
[91] ⍝ Move up a level if not found:
[92]  →(∆∆J>⍴∆∆T)⍴∆∆LP1
[93] ⍝ If found, determine type of local:
[94]  ∆∆T←∆∆LTYPE[∆∆LDSP[∆∆IND]+∆∆J]
[95] ⍝ Display object name, fn where local, type:
[96]  ∆∆R←∆∆NAMES[∆∆PARSED[∆∆IND];]
```

```
    ∇ USEDBY (continued)
[97]  ∆∆R←(∆∆R≠' ')/∆∆R
[98]  ∆∆T←(4 8 ρ'label   result  argumentlocal   ')[∆∆T;]
[99]  ∆∆T←(∆∆T≠' ')/∆∆T
[100]  ⎕←((∆∆INDENT×¯1+ρ∆∆LCPARSED)ρ' '),∆∆NAME,' (',∆∆R,' -
       ',∆∆T,')'
[101]  →∆∆L10
[102] ⍝
[103] ⍝ Branch if object is a function:
[104] ∆∆L3:→(3=⎕NC ∆∆NAME)ρ∆∆L4
[105] ⍝ Display object name and global indicator:
[106]  ⎕←((∆∆INDENT×¯1+ρ∆∆LCPARSED)ρ' '),∆∆NAME,' (global)'
[107]  →∆∆L10
[108] ⍝
[109] ⍝ Display object name:
[110] ∆∆L4:⎕←((∆∆INDENT×¯1+ρ∆∆LCPARSED)ρ' '),∆∆NAME
[111] ⍝ Branch if fn has already been parsed:
[112]  →((ρ∆∆PARSED)≥∆∆J←∆∆PARSED⍳∆∆INDEX)ρ∆∆L9
[113]  ∆∆PARSED←∆∆PARSED,∆∆INDEX
[114] ⍝
[115] ⍝ Analyze fn for global, local identifiers...
[116] ⍝ Construct visual representation:
[117]  ∆∆VR←⎕VR ∆∆NAME ⍝ APL*PLUS
[118] ⍝ ∆∆VR←1 ⎕FD ∆∆NAME ⍝ SHARP APL
[119] ⍝ ∆∆VR←CR∆VR ⎕CR ∆∆NAME ⍝ Other APL systems
[120] ⍝ Update selected vars:
[121]  ∆∆GDSP←∆∆GDSP,ρ∆∆GLOBAL
[122]  ∆∆LDSP←∆∆LDSP,ρ∆∆LOCAL
[123] ⍝ Branch unless fn locked:
[124]  →(×ρ∆∆VR)ρ∆∆L5
[125] ⍝ No identifiers found if locked:
[126]  ∆∆GNUM←∆∆GNUM,0
[127]  ∆∆LNUM←∆∆LNUM,0
[128]  →∆∆L9
[129] ⍝ Use origin 0 for fn parsing:
[130] ∆∆L5:⎕IO←0
[131] ⍝ Where does header end?
[132]  ∆∆T←∆∆VR⍳∆∆TCNL
[133] ⍝ Grab header of fn (less newline):
[134]  ∆∆HDR←∆∆Tρ∆∆VR
[135] ⍝ Drop header from vis rep:
[136]  ∆∆VR←∆∆T↓∆∆VR
[137] ⍝ Where does fn syntax end?
[138]  ∆∆T←∆∆HDR⍳';'
[139] ⍝ Localized vars:
[140]  ∆∆LOC←∆∆T↓∆∆HDR
[141] ⍝ Drop local vars:
[142]  ∆∆HDR←∆∆Tρ∆∆HDR
[143] ⍝ Drop leading junk:
[144]  ∆∆HDR←(∨\~∆∆HDR∊' ∇')/∆∆HDR
[145] ⍝ Is there an explicit result?
[146]  ∆∆T←(ρ∆∆HDR)>∆∆IND←∆∆HDR⍳'←'
[147] ⍝ Explicit result (if any):
[148]  ∆∆RESULT←(∆∆T×∆∆IND)ρ∆∆HDR
```

```
    ∇ USEDBY (continued)
[149] ⍝ Remove result from header:
[150]  ΔΔHDR←' ',((×⍴ΔΔRESULT)+⍴ΔΔRESULT)↓ΔΔHDR
[151] ⍝ Starting indices of fn name, args:
[152]  ΔΔSTART←1+(ΔΔHDR=' ')/⍳⍴ΔΔHDR
[153] ⍝ Lengths of names:
[154]  ΔΔLEN←(1↓ΔΔSTART,1+⍴ΔΔHDR)-1+ΔΔSTART
[155] ⍝ No. of args:
[156]  ΔΔT←¯1+⍴ΔΔLEN
[157] ⍝ Indices of args (if any):
[158]  ΔΔN←⍴ΔΔIND←((ΔΔT=2)⍴0),(×ΔΔT)⍴ΔΔT
[159] ⍝ Don't consider fn name:
[160]  ΔΔSTART←ΔΔSTART[ΔΔIND]
[161]  ΔΔLEN←ΔΔLEN[ΔΔIND]
[162] ⍝ Length of longest arg:
[163]  ΔΔCOLS←⌈/0,ΔΔLEN
[164] ⍝ Raveled, blank mat of args:
[165]  ΔΔARGS←(ΔΔN×ΔΔCOLS)⍴' '
[166] ⍝ Compute indices: (⍳LEN[1]),(⍳LEN[2]),...
[167]  ΔΔI←ΔΔLEN/-¯1↓0,+\ΔΔLEN
[168]  ΔΔI←ΔΔI+⍳⍴ΔΔI
[169] ⍝ Insert names into raveled matrix:
[170]  ΔΔARGS[ΔΔI+ΔΔLEN/ΔΔCOLS×⍳ΔΔN]←ΔΔHDR[ΔΔI+ΔΔLEN/ΔΔSTART
      ]
[171] ⍝ Reshape to mat of args:
[172]  ΔΔARGS←(ΔΔN,ΔΔCOLS)⍴ΔΔARGS
[173] ⍝ Flag newline chars:
[174]  ΔΔNL←ΔΔVR=ΔΔTCNL
[175] ⍝ Flag nonquotes:
[176]  ΔΔNQ←ΔΔVR≠''''
[177] ⍝ Map of chars not in quote pairs (i.e. char
[178] ⍝ constants) within each fn line (i.e. an open
[179] ⍝ quote is closed at the end of the fn line).
[180] ⍝ Leading quotes are flagged 0; closing quotes are
[181] ⍝ flagged 1 (including 1st of double-quote pairs):
[182]  ΔΔNCCON←=\ΔΔNQ≠ΔΔNL\ΔΔT≠¯1↓0,ΔΔT←~ΔΔNL/=\ΔΔNQ
[183]  ΔΔNQ←0
[184] ⍝ Flag non-⍝ chars (includes ⍝s in quotes):
[185]  ΔΔNC←ΔΔNCCON⍲ΔΔVR='⍝'
[186] ⍝ Flag newlines or ⍝s (except ⍝s in quotes):
[187]  ΔΔS←ΔΔNL≥ΔΔNC
[188] ⍝ Map of chars which do not follow a ⍝ (ignoring ⍝s
[189] ⍝ within quotes) within each fn line. ⍝s are
[190] ⍝ flagged 0:
[191]  ΔΔNCMT←~≠\ΔΔS\ΔΔT≠¯1↓0,ΔΔT←~ΔΔS/ΔΔNC
[192]  ΔΔS←ΔΔNC←0
[193] ⍝ Map of chars which are not included within ⍝s or '':
[194]  ΔΔPARSE←ΔΔNCMT∧ΔΔNCCON
[195]  ΔΔNCCON←ΔΔNCMT←0
[196] ⍝ Flag digits and letters:
[197]  ΔΔNUM←ΔΔPARSE∧ΔΔVR∊'0123456789'
[198]  ΔΔALP←ΔΔPARSE∧ΔΔVR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZΔabcdef
      ghijklmnopqrstuvwxyz⍙'
```

```
     ∇ USEDBY (continued)
[199] ⍝ Flag blanks:
[200]  ∆∆BL←∆∆PARSE∧∆∆VR=' '
[201]  ∆∆PARSE←0
[202] ⍝ Flag alphanumeric chars:
[203]  ∆∆AN←∆∆NUM∨∆∆ALP
[204] ⍝ Pole vec of contiguous digits:
[205]  ∆∆PNUM←∆∆NUM≠¯1↓0,∆∆NUM
[206]  ∆∆NUM←0
[207] ⍝ Pole vec of contiguous digits/letters:
[208]  ∆∆PAN←∆∆AN≠¯1↓0,∆∆AN
[209]  ∆∆AN←0
[210] ⍝ Pole vec of identifiers:
[211]  ∆∆PID←∆∆PAN\∆∆T∨¯1⌽∆∆T←∆∆PAN/∆∆ALP
[212]  ∆∆ALP←∆∆PAN←0
[213] ⍝ Flag '⎕' before identifiers (⎕names):
[214]  ∆∆T←∆∆T\'⎕'=(∆∆T←1⌽∆∆PID)/∆∆VR
[215] ⍝ Shift leading poles of ⎕names to include ⎕:
[216]  ∆∆PID←∆∆T∨∆∆PID>¯1⌽∆∆T
[217]  ∆∆T←0
[218] ⍝ Flag char following ] after line no.:
[219]  ∆∆START←¯1⌽∆∆PNUM\¯1⌽∆∆PNUM/¯2⌽∆∆NL
[220]  ∆∆NL←∆∆PNUM←0
[221] ⍝ Pole vec of contiguous blanks:
[222]  ∆∆PBL←∆∆BL≠¯1↓0,∆∆BL
[223]  ∆∆BL←0
[224] ⍝ Flag 1st nonblank char in each line:
[225]  ∆∆START←(∆∆START>∆∆PBL)∨∆∆PBL\¯1⌽∆∆PBL/∆∆START
[226]  ∆∆PBL←0
[227] ⍝ Pole vec of identifiers at start of line:
[228]  ∆∆PSID←∆∆PID\∆∆T∨¯1⌽∆∆T←∆∆PID/∆∆START
[229] ⍝ Pole vec of labels:
[230]  ∆∆START←0
[231]  ∆∆PLAB←∆∆PSID\∆∆T∨1⌽∆∆T←':'=∆∆PSID/∆∆VR
[232]  ∆∆PSID←0
[233] ⍝ Start and end (+1) indices of identifiers:
[234]  ∆∆IND←∆∆PID/⍳⍴∆∆PID
[235] ⍝ No. of identifiers:
[236]  ∆∆NID←(⍴∆∆IND)÷2
[237]  ∆∆IND←(∆∆NID,2)⍴∆∆IND
[238] ⍝ Start indices of identifiers:
[239]  ∆∆IDSTART←∆∆IND[;0]
[240] ⍝ Lengths of identifiers:
[241]  ∆∆IDLEN←∆∆IND[;1]-∆∆IDSTART
[242] ⍝ Starting indices of local vars:
[243]  ∆∆START←1+(∆∆LOC=';')/⍳⍴∆∆LOC
[244] ⍝ Lengths of local vars:
[245]  ∆∆LEN←(1↓∆∆START,1+⍴∆∆LOC)-1+∆∆START
[246] ⍝ Length of longest ident.:
[247]  ∆∆COLS←(⌈/∆∆LEN)⌈(⌈/∆∆IDLEN)⌈(⍴∆∆RESULT)⌈1↓⍴∆∆ARGS
[248] ⍝ Pad arg names to conform:
[249]  ∆∆ARGS←((1⍴⍴∆∆ARGS),∆∆COLS)↑∆∆ARGS
[250] ⍝ 0 row mat if no result:
[251]  ∆∆RESULT←((×⍴∆∆RESULT),∆∆COLS)⍴∆∆COLS↑∆∆RESULT
```

```
    ∇ USEDBY (continued)
[252] ⍝ Raveled blank mat of local vars:
[253]  ∆∆T←(∆∆COLS×∆∆N←⍴∆∆START)⍴' '
[254] ⍝ Compute indices: (⍳LEN[1]),(⍳LEN[2]),...
[255]  ∆∆I←∆∆LEN/-¯1↓0,+\∆∆LEN
[256]  ∆∆I←∆∆I+⍳⍴∆∆I
[257] ⍝ Insert names into raveled matrix:
[258]  ∆∆T[∆∆I+∆∆LEN/∆∆COLS×⍳∆∆N]←∆∆LOC[∆∆I+∆∆LEN/∆∆START]
[259] ⍝ Reshape to mat of local vars:
[260]  ∆∆LOC←(∆∆N,∆∆COLS)⍴∆∆T
[261] ⍝ Raveled blank mat of identifiers:
[262]  ∆∆T←(∆∆COLS×∆∆N←⍴∆∆IDSTART)⍴' '
[263] ⍝ Compute indices: (⍳IDLEN[1]),(⍳IDLEN[2]),...
[264]  ∆∆I←∆∆IDLEN/-¯1↓0,+\∆∆IDLEN
[265]  ∆∆I←∆∆I+⍳⍴∆∆I
[266] ⍝ Insert names into raveled matrix:
[267]  ∆∆T[∆∆I+∆∆IDLEN/∆∆COLS×⍳∆∆N]←∆∆VR[∆∆I+∆∆IDLEN/
      ∆∆IDSTART]
[268] ⍝ Reshape to mat of identifiers:
[269]  ∆∆IDS←(∆∆N,∆∆COLS)⍴∆∆T
[270] ⍝ Mat of label names:
[271]  ∆∆LAB←(∆∆S←((∆∆N,2)⍴∆∆PID/∆∆PLAB)[;0])⌿∆∆IDS
[272]  ∆∆PID←∆∆PLAB←0
[273] ⍝ Mat of referenced vars less labels:
[274]  ∆∆RVAR←(~∆∆S)⌿∆∆IDS
[275] ⍝ Combine different types of vars and a vector
[276] ⍝ of their types:
[277]  ∆∆IDS←(∆∆LAB,[0]∆∆RESULT,[0]∆∆ARGS,[0]∆∆LOC),[0]
      ∆∆RVAR
[278]  ∆∆IDTYPES←((1⍴⍴∆∆LAB),(1⍴⍴∆∆RESULT),(1⍴⍴∆∆ARGS),(1⍴⍴
      ∆∆LOC),1⍴⍴∆∆RVAR)/ 1 2 3 4 5
[279] ⍝ Select just the first distinct name (and type):
[280]  ∆∆T←((∆∆IDS CMIOTA ∆∆IDS)=⍳⍴∆∆IDTYPES)/⍳⍴∆∆IDTYPES
[281]  ∆∆IDS←∆∆IDS[∆∆T;]
[282]  ∆∆IDTYPES←∆∆IDTYPES[∆∆T]
[283] ⍝ Branch if no columns in IDS (i.e. no identifiers):
[284]  →(1↓⍴∆∆IDS)↓∆∆L5A
[285] ⍝ Retain ⎕-names only for system variables (add other
[286] ⍝ system vars on your APL system):
[287]  ∆∆S← 5 4 ⍴'⎕IO ⎕PP ⎕RL ⎕CT ⎕LX '
[288]  ∆∆T←'⎕'=∆∆IDS[;0]
[289]  ∆∆I←∆∆T/⍳⍴∆∆T
[290]  ∆∆R←(1⍴⍴∆∆S)≤∆∆S CMIOTA((⍴∆∆I),4)↑∆∆IDS[∆∆I;]
[291] ⍝ Branch if all ⎕-names are in this list:
[292]  →(∨/∆∆R)↓∆∆L5A
[293]  ∆∆T←~∆∆T\∆∆R
[294] ⍝ Squeeze out ⎕-names which are system fns (e.g. ⎕EX):
[295]  ∆∆IDS←∆∆T⌿∆∆IDS
[296]  ∆∆IDTYPES←∆∆T/∆∆IDTYPES
[297] ⍝ Return to origin 1:
[298] ∆∆L5A:⎕IO←1
[299] ⍝ Insert new names into NAMES and convert to indices:
[300]  →(×(∆∆S←1↓⍴∆∆NAMES)-∆∆T←1↓⍴∆∆IDS)⌽∆∆L8,∆∆L6,∆∆L7
```

```
    ∇ USEDBY (continued)
[301] ⍝ NAMES has more columns than IDS:
[302] ΔΔL6:ΔΔIDS←((1ρρΔΔIDS),ΔΔS)↑ΔΔIDS
[303]  →ΔΔL8
[304] ⍝ IDS has more columns than NAMES:
[305] ΔΔL7:ΔΔNAMES←((1ρρΔΔNAMES),ΔΔT)↑ΔΔNAMES
[306] ⍝
[307] ΔΔL8:ΔΔIND←ΔΔNAMES CMIOTA ΔΔIDS
[308] ⍝ Flag those not found:
[309]  ΔΔT←ΔΔIND>ΔΔR←1ρρΔΔNAMES
[310] ⍝ Add to NAMES and compute indices:
[311]  ΔΔNAMES←ΔΔNAMES,[1]ΔΔT⌿ΔΔIDS
[312]  ΔΔS←ΔΔT/⍳ρΔΔT
[313]  ΔΔIND[ΔΔS]←ΔΔR+⍳ρΔΔS
[314]  ΔΔIDS←ΔΔIND
[315] ⍝ How many identifiers are locals?
[316]  ΔΔN←¯1+ΔΔIDTYPES⍳5
[317]  ΔΔLNUM←ΔΔLNUM,ΔΔN
[318]  ΔΔLOCAL←ΔΔLOCAL,ΔΔNρΔΔIDS
[319]  ΔΔLTYPE←ΔΔLTYPE,ΔΔNρΔΔIDTYPES
[320] ⍝ How many identifiers are globals:
[321]  ΔΔIDS←ΔΔN↓ΔΔIDS
[322]  ΔΔGNUM←ΔΔGNUM,ρΔΔIDS
[323]  ΔΔGLOBAL←ΔΔGLOBAL,ΔΔIDS
[324] ⍝
[325] ⍝ Branch unless this object has globals:
[326] ΔΔL9:→(×ΔΔGNUM[ΔΔJ])↓ΔΔL10
[327] ⍝ Add a level to the "state indicator":
[328]  ΔΔLCPARSED←ΔΔJ,ΔΔLCPARSED
[329]  ΔΔLCGLOBAL←1,ΔΔLCGLOBAL
[330]  →ΔΔLOOP
[331] ⍝
[332] ⍝ Increment "state indicator" line:
[333] ΔΔL10:ΔΔT←ΔΔLCGLOBAL[1]←1+ΔΔLCGLOBAL[1]
[334] ⍝ Resume if more globals at this level:
[335]  →(ΔΔT≤ΔΔGNUM[ΔΔLCPARSED[1]])ρΔΔLOOP
[336] ⍝ Else drop a level from the "state indicator":
[337]  ΔΔLCPARSED←1↓ΔΔLCPARSED
[338]  ΔΔLCGLOBAL←1↓ΔΔLCGLOBAL
[339] ⍝ Continue if any levels left:
[340]  →(×ρΔΔLCPARSED)ρΔΔL10
    ∇
```

Chapter 14 Solutions

FILE DESIGN AND UTILITIES

1. Here is a possible file structure for the "functions file":

FILE NAME: FNS TIE NUMBER: up to you

DESCRIPTION: Contains representations of functions

COMP. NO.	VARIABLE	DESCRIPTION
1	DIR	Character matrix directory of the names (one per row) of the functions whose representations are stored on file. The matrix has as many columns as the longest function name has elements. The shorter names are left justified (padded to the right with spaces).
1+I	VR	Character vector visual representation of the function whose name is DIR[I;] (or canonical representation if your APL implementation of APL does not support visual representations).

```
                                    [WSID: FNSFILE]
     ∇ FNAME FNCREATE TIE;DIR;T
[1]   ⍝ Initiltizes an empty functions file named FNAME,
[2]   ⍝ tied to TIE.
[3]    FNAME ⎕FCREATE TIE ⍝ APL*PLUS
[4]   ⍝ FNAME ⎕CREATE TIE ⍝ SHARP APL
[5]   ⍝ Construct empty directory:
[6]    DIR← 0 0 ⍴''
[7]   ⍝ Append as 1st component:
[8]    T←DIR ⎕FAPPEND TIE ⍝ APL*PLUS if ⎕FAPPEND has result
[9]   ⍝ DIR ⎕FAPPEND TIE ⍝ APL*PLUS if ⎕FAPPEND has no result
[10]  ⍝ DIR ⎕APPEND TIE ⍝ SHARP APL
     ∇
```

[WSID: FNSFILE]

```
    ∇ ΔΔR←ΔΔTIE PUTFN ΔΔNAME;⎕IO;ΔΔDIR;ΔΔIND;ΔΔT;ΔΔVR
  ⍝ Appends or replaces to the functions file tied
  ⍝ to ΔΔTIE the visual representation (or canonical
  ⍝ representation) of the function named ΔΔNAME.
  ⍝ Note: will not file functions whose names are
  ⍝ local to PUTFN.  Returns ΔΔNAME if successful,
  ⍝ else empty vector.
  ⍝ Origin 1:
   ⎕IO←1
  ⍝ Construct visual representation:
   ΔΔVR←⎕VR ΔΔNAME ⍝ APL*PLUS
  ⍝ ΔΔVR←1 ⎕FD ΔΔNAME ⍝ SHARP APL
  ⍝ ΔΔVR←⎕CR ΔΔNAME ⍝ On other systems
  ⍝ Return empty vector if function locked:
   ΔΔR←''
   →(×/⍴ΔΔVR)↓0
  ⍝ Read directory of function names:
   ΔΔDIR←⎕FREAD ΔΔTIE,1 ⍝ APL*PLUS
  ⍝ ΔΔDIR←⎕READ ΔΔTIE,1 ⍝ SHARP APL
  ⍝ Delete any blanks in name:
   ΔΔR←ΔΔNAME←(ΔΔNAME≠' ')/ΔΔNAME
  ⍝ Search directory for function name (branch if
  ⍝ not found):
   →((1↓⍴ΔΔDIR)<⍴ΔΔNAME)⍴ΔΔL1
   ΔΔIND←(ΔΔDIR∧.=(1↓⍴ΔΔDIR)↑ΔΔNAME)⍳1
   →(ΔΔIND>1⍴⍴ΔΔDIR)⍴ΔΔL1
  ⍝ Replace existing visual representation with
  ⍝ new one:
   ΔΔVR ⎕FREPLACE ΔΔTIE,1+ΔΔIND ⍝ APL*PLUS
  ⍝ ΔΔVR ⎕REPLACE ΔΔTIE,1+ΔΔIND ⍝ SHARP APL
   →0
  ⍝ Add name to directory.  Branch unless ΔΔNAME
  ⍝ too long:
  ΔΔL1:→((⍴ΔΔNAME)≤1↓⍴ΔΔDIR)⍴ΔΔL2
  ⍝ Pad columns in directory to length of ΔΔNAME:
   ΔΔDIR←((1⍴⍴ΔΔDIR),⍴ΔΔNAME)↑ΔΔDIR
  ΔΔL2:ΔΔDIR←ΔΔDIR,[1](1↓⍴ΔΔDIR)↑ΔΔNAME
  ⍝ Replace directory:
   ΔΔDIR ⎕FREPLACE ΔΔTIE,1 ⍝ APL*PLUS
  ⍝ ΔΔDIR ⎕REPLACE ΔΔTIE,1 ⍝ SHARP APL
  ⍝ Append function representation:
   ΔΔT←ΔΔVR ⎕FAPPEND ΔΔTIE ⍝ APL*PLUS if ⎕FAPPEND has result
  ⍝ ΔΔVR ⎕FAPPEND ΔΔTIE ⍝ APL*PLUS if ⎕FAPPEND has no result
  ⍝ ΔΔVR ⎕APPEND ΔΔTIE ⍝ SHARP APL
    ∇
```

```
                                      [WSID: FNSFILE]
    ∇ ∆∆R←∆∆TIE GETFN ∆∆NAME;⎕IO;∆∆DIR;∆∆IND;∆∆VR
[1]   ⍝ Reads from the functions file tied to ∆∆TIE the
[2]   ⍝ visual representation (or canonical representation)
[3]   ⍝ of the function named ∆∆NAME and defines that
[4]   ⍝ function in the active workspace.  Note: will not
[5]   ⍝ define functions whose names are local to GETFN.
[6]   ⍝ Returns ∆∆NAME if successful, empty vector if
[7]   ⍝ function not found, numeric code if function not
[8]   ⍝ definable.
[9]   ⍝ Origin 1:
[10]   ⎕IO←1
[11]  ⍝ Read directory of function names:
[12]   ∆∆DIR←⎕FREAD ∆∆TIE,1 ⍝ APL*PLUS
[13]  ⍝ ∆∆DIR←⎕READ ∆∆TIE,1 ⍝ SHARP APL
[14]  ⍝ Delete any blanks in name:
[15]   ∆∆NAME←(∆∆NAME≠' ')/∆∆NAME
[16]  ⍝ Search directory for function name (exit if not
[17]  ⍝ found):
[18]   ∆∆R←''
[19]   →((1↓ρ∆∆DIR)<ρ∆∆NAME)ρ0
[20]   ∆∆IND←(∆∆DIR∧.=(1↓ρ∆∆DIR)↑∆∆NAME)⍳1
[21]   →(∆∆IND>1ρρ∆∆DIR)ρ0
[22]  ⍝ Read visual (or canonical) representation from
[23]  ⍝ file:
[24]   ∆∆VR←⎕FREAD ∆∆TIE,1+∆∆IND ⍝ APL*PLUS
[25]  ⍝ ∆∆VR←⎕READ ∆∆TIE,1+∆∆IND ⍝ SHARP APL
[26]  ⍝ Define function in workspace (result is function
[27]  ⍝ name or numeric code indicating WS FULL, SYMBOL
[28]  ⍝ TABLE FULL,...):
[29]   ∆∆R←⎕DEF ∆∆VR ⍝ APL*PLUS
[30]  ⍝ ∆∆R←3 ⎕FD ∆∆VR ⍝ SHARP APL
[31]  ⍝ ∆∆R←⎕FX ∆∆VR ⍝ On other systems
    ∇
```

```
                                      [WSID: FNSFILE]
    ∇ R←TIE DROPFN NAME;⎕IO;DIR;IND;LAST
[1]   ⍝ Removes from the functions file tied to TIE
[2]   ⍝ the visual representation (or canonical
[3]   ⍝ representation) of the function named NAME.
[4]   ⍝ Returns NAME if successful, empty vector if
[5]   ⍝ function not found.
[6]   ⍝ Origin 1:
[7]    ⎕IO←1
[8]   ⍝ Read directory of function names:
[9]    DIR←⎕FREAD TIE,1 ⍝ APL*PLUS
[10]  ⍝ DIR←⎕READ TIE,1 ⍝ SHARP APL
[11]  ⍝ Delete any blanks in name:
[12]   NAME←(NAME≠' ')/NAME
[13]  ⍝ Search directory for fn name (exit if not found):
[14]   R←''
[15]   →((1↓ρDIR)<ρNAME)ρ0
[16]   IND←(DIR∧.=(1↓ρDIR)↑NAME)⍳1
```

```
    ∇ DROPFN (continued)
[17]   →(IND>1⍴⍴DIR)⍴0
[18] ⍝ Replace this name with last one in directory:
[19]   LAST←(⍴DIR)[1]
[20]   DIR[IND;]←DIR[LAST;]
[21]   DIR← ¯1 0 ↓DIR
[22] ⍝ Replace this vis. rep. with last one on file:
[23]   (⎕FREAD TIE,1+LAST)⎕FREPLACE TIE,1+IND ⍝ APL*PLUS
[24]   ⎕FDROP TIE,¯1
[25] ⍝ (⎕READ TIE,1+LAST)⎕REPLACE TIE,1+IND ⍝ SHARP APL
[26] ⍝ ⎕DROP TIE,¯1
[27] ⍝ Replace directory:
[28]   DIR ⎕FREPLACE TIE,1 ⍝ APL*PLUS
[29] ⍝ DIR ⎕REPLACE TIE,1 ⍝ SHARP APL
    ∇
```

2.

```
                                        [WSID: FILEDOC]
    ∇ FILE FILEDOC PAGE;⎕IO;⎕PP;BOTTOM;CMPS;D;DATA;DONE;
      FIRST;FOOT;HEIGHT;I;LAST;LEFT;LIM;LINES;MARGIN;N;NEW;
      NONDISPLAY;OLDCMP;PNO;QUOTE;S;START;T;TCNL;TITLE;TOP;
      TXT;W;WIDTH
[1]  ⍝ Displays paged file documentation. All output is
[2]  ⍝ via ⎕← so replace all ⎕← by custom fn (e.g. PRINT)
[3]  ⍝ to redirect output.  PAGE: rows, columns, margins
[4]  ⍝ (top, bottom, left, right)  FILE: tie no. if
[5]  ⍝ APL*PLUS or SHARP APL; otherwise a file name.
[6]  ⍝ Use origin 1:
[7]    ⎕IO←1
[8]  ⍝ Format no.s to 10 digits (CLEAR WS default):
[9]    ⎕PP←10
[10] ⍝ Activate file if IBM's workspace 2 VAPLFILE:
[11] ⍝ USE FILE
[12]   TOP←PAGE[3]
[13]   BOTTOM←PAGE[4]
[14]   HEIGHT←PAGE[1]-TOP+BOTTOM
[15]   LEFT←PAGE[5]
[16]   WIDTH←PAGE[2]-PAGE[6]
[17] ⍝ Construct newline character:
[18]   TCNL←⎕TCNL ⍝ APL*PLUS
[19] ⍝ TCNL←⎕TC[2] ⍝ APL2
[20] ⍝ TCNL←⎕AV[157] ⍝ SHARP APL
[21] ⍝ Format today's date:
[22]   D←⍕⎕TS[2 3 1]
[23]   D[(D=' ')/⍳⍴D]←'/'
[24] ⍝ Format the time:
[25]   T←(⍕⎕TS[4]),':',¯2↑'0',⍕⎕TS[5]
[26] ⍝ Format the file name:
[27]   TITLE←,⎕FNAMES[⎕FNUMS⍳FILE;] ⍝ APL*PLUS
[28] ⍝ TITLE←,⎕NAMES[⎕NUMS⍳FILE;] ⍝ SHARP APL
[29] ⍝ TITLE←FILE ⍝ Otherwise
```

```
    ∇ FILEDOC (continued)
[30] ⍝ Delete leading/trailing blanks:
[31]  TITLE←(-+/∧\⌽TITLE=' ')↓(+/∧\TITLE=' ')↓TITLE
[32] ⍝ Number of file components and first one:
[33]  S←⎕FSIZE FILE ⍝ APL*PLUS
[34] ⍝ S←⎕SIZE FILE ⍝ SHARP APL
[35] ⍝ S←1,1+rHO FILE ⍝ 2 VAPLFILE
[36]  START←S[1]
[37]  LIM←S[2]
[38]  N←LIM-START
[39] ⍝ If 2 VAPLFILE:
[40] ⍝ CMPS←EXITt⍳S←RHo FILE
[41] ⍝ LIM←N←⍴CMPS←CMPS/⍳⍴CMPS
[42] ⍝ Exit if none:
[43]  →N↓0
[44] ⍝ Format page title:
[45]  TITLE←TITLE,' (',(⍕N),' COMPONENT',((1≠N)⍴'S'),') * ',
      D,' ',T
[46] ⍝ If 2 VAPLFILE:
[47] ⍝ TITLE←TITLE,' (',(⍕N),' COMPONENT',((1≠N)⍴'S'),' OF '
      ,(⍕S),') * ',D,' ',T
[48]  TITLE←(TOP⍴TCNL),WIDTH↑(LEFT⍴' '),'FILE: ',TITLE
[49] ⍝ Insert page number:
[50]  PNO←1
[51]  T←'PAGE 1'
[52]  TITLE[(-⍴T)↑⍳⍴TITLE]←T
[53] ⍝ Build first page:
[54]  TXT←TITLE,TCNL
[55] ⍝ Keep track of lines used so far (below top
[56] ⍝ margin) in TXT:
[57]  LINES←2
[58]  QUOTE←''''
[59] ⍝ Define characters which don't display normally:
[60]  NONDISPLAY←⎕TCNL,⎕TCLF,⎕TCBS,⎕TCBEL,⎕TCDEL,⎕TCNUL,
      ⎕TCESC,⎕TCFF ⍝ APL*PLUS
[61] ⍝ NONDISPLAY←⎕AV[1 2 157 159 160] ⍝ SHARP APL
[62] ⍝ NONDISPLAY←⎕TC ⍝ APL2
[63]  MARGIN←LEFT⍴' '
[64] ⍝ Include heading:
[65]  T←TCNL,MARGIN,'COMPONENT     SHAPE ⍴ VALUE',TCNL
[66]  TXT←TXT,T,MARGIN,'---------     -----   -----'
[67]  LINES←LINES+2
[68] ⍝ Loop by component:
[69]  OLDCMP←0
[70]  FIRST←START
[71]  I←START+¯1
[72] ⍝ If 2 VAPLFILE:
[73] ⍝ FIRST←CMPS[1]
[74] ⍝ I←0
[75] LOOP:→(DONE←LIM≤I←I+1)/L2
[76] ⍝ LOOP:→(DONE←LIM<I←I+1)/L2 ⍝ 2 VAPLFILE
[77] ⍝ Read data from file:
[78]  NEW←⎕FREAD FILE,I ⍝ APL*PLUS
[79] ⍝ NEW←⎕READ FILE,I ⍝ SHARP APL
```

```
    ∇ FILEDOC (continued)
[80] ⍝ NEW←GET CMPS[I] ⍝ 2 VAPLFILE
[81] ⍝
[82] ⍝ Branch unless this is the first component:
[83]  →(×OLDCMP)/L1
[84] ⍝ Last distinct object read from file:
[85]  DATA←NEW
[86] ⍝ Earliest such component of that object:
[87]  OLDCMP←I
[88] ⍝ OLDCMP←CMPS[I⌊LIM] ⍝ 2 VAPLFILE
[89]  →LOOP
[90] ⍝ Get next object if this one is identical (to
[91] ⍝ last one):
[92] L1:→((ρρDATA)≠ρρNEW)/L2
[93] ⍝ If 2 VAPLFILE:
[94] ⍝ L1:→(CMPS[I]≠1+CMPS[I+¯1])/L2
[95] ⍝ →((ρρDATA)≠ρρNEW)/L2
[96]  →((ρDATA)∨.≠ρNEW)/L2
[97]  →(∧/,DATA=NEW)/LOOP
[98] ⍝ Format DATA (components OLDCMP to I-1)...
[99] ⍝ Format shape:
[100] L2:S←(⍕ρDATA),' ρ '
[101] ⍝ Omit shape and ρ if a scalar; pad to line up
[102] ⍝ with ρ's:
[103]  S←(-11⌈ρS)↑(3<ρS)/S
[104] ⍝ Combine component no.(s) and shape; compute
[105] ⍝ remaining width:
[106]  T←OLDCMP≠I+¯1
[107] ⍝ T←OLDCMP≠CMPS[I+¯1] ⍝ 2 VAPLFILE
[108]  T←(⍕OLDCMP),T/'-',⍕I+¯1
[109] ⍝ T←(⍕OLDCMP),T/'-',⍕CMPS[I+¯1] ⍝ 2 VAPLFILE
[110]  T←(9⌈ρT)↑(-5⌈ρT)↑T
[111]  S←MARGIN,T,' ←',S
[112]  W←WIDTH-ρS
[113] ⍝ Branch if a numeric variable:
[114]  →(0=1↑0ρDATA)ρL3
[115] ⍝ Else data is char; consider only up to W chars:
[116]  DATA←(W⌊×/ρDATA)ρDATA
[117] ⍝ Replace nondisplayable chars by ⍰:
[118]  DATA[(DATA∊NONDISPLAY)/⍳ρDATA]←'⍰'
[119] ⍝ Double up quote chars:
[120]  DATA←QUOTE,((1+DATA=QUOTE)/DATA),QUOTE
[121] ⍝ Branch if data will fit on a single line:
[122]  →(W≥ρDATA)/L4
[123] ⍝ Else truncate and show '...':
[124]  DATA←((W+¯3)ρDATA),'...'
[125]  →L4
[126] ⍝ If data is numeric, consider only up to
[127] ⍝ 1+W÷2 elements:
[128] L3:DATA←((1+⌈W÷2)⌊×/ρDATA)ρDATA
[129]  DATA←⍕DATA
[130] ⍝ Format value from empty array as ⍳0:
[131]  DATA←DATA,(0=ρDATA)/'⍳0'
```

```
    ∇ FILEDOC (continued)
[132] ⍝ Branch if data will fit on a single line:
[133]  →(W≥⍴DATA)/L4
[134] ⍝ Else trunc. at last space within wid and show '...':
[135]  DATA←((+/∨\' '=⌽(W+¯3)⍴DATA)⍴DATA),'...'
[136] ⍝
[137] ⍝ Append variable definition to page:
[138] L4:TXT←TXT,TCNL,S,DATA
[139]  LINES←LINES+1
[140]  DATA←NEW
[141]  OLDCMP←I
[142] ⍝ OLDCMP←CMPS[I] ⍝ 2 VAPLFILE
[143]  LAST←I+¯1
[144] ⍝ LAST←CMPS[I+¯1] ⍝ 2 VAPLFILE
[145] ⍝ Branch for more unless bottom of page (2 lines
[146] ⍝ for footnote) or end of file:
[147]  →(DONE<LINES<HEIGHT+¯2)/LOOP
[148] ⍝ Construct footnote:
[149] L5:T←FIRST≠LAST
[150]  FOOT←(-WIDTH)↑'COMPONENT',(T⍴'S'),' ',(⍕FIRST),T/' TO
       ',⍕LAST
[151] ⍝ Display page:
[152]  ⎕←TXT,((HEIGHT-LINES)⍴TCNL),FOOT,BOTTOM⍴TCNL
[153]  FIRST←LAST←I
[154] ⍝ FIRST←LAST←CMPS[I⌊LIM] ⍝ 2 VAPLFILE
[155] ⍝ Exit if no more components:
[156]  →DONE/0
[157] ⍝ Format new page:
[158]  TXT←TITLE,TCNL
[159]  PNO←PNO+1
[160]  T←'PAGE ',⍕PNO
[161]  TXT[(-⍴T)↑⍳⍴TITLE]←T
[162]  LINES←2
[163] ⍝ Insert new heading:
[164]  T←TCNL,MARGIN,'COMPONENT     SHAPE ⍴ VALUE',TCNL
[165]  TXT←TXT,T,MARGIN,'---------     -----   -----'
[166]  LINES←LINES+2
[167] ⍝ Branch for more components:
[168]  →LOOP
    ∇
```

3.

```
                                        [WSID: MULTIFLO]
      ∇ EMPLOYEES;F1;F2;F3;G;GOOD;IND;NUM;P;R
[1]   ⍝ ILLUSTRATION OF FILE UTILITY FUNCTIONS.
[2]   ⍝ Assumes employee information is on file.  The file
[3]   ⍝ is identified by the global variable FP (file
[4]   ⍝ parameters).  Fields of file are:
[5]   ⍝   1.  Employee number
[6]   ⍝   2.  Employee name
[7]   ⍝   3.  Employee age
[8]   ⍝ Uses subfns: IOTA,CATRECWS,DELREC,SELECTWS.
[9]   ⍝ Ask for choice on same line:
[10]  CHOOSE:⍞←P←'ADD, DELETE, LIST OR END: '
[11]   R←(⍴P)↓⍞
[12]  ⍝ Branch based on 1st char of response:
[13]   →('ADLE'=1↑R)/ADD,DELETE,LIST,END
[14]   ⎕←'** INVALID CHOICE. CHOOSE FROM: ADLE'
[15]   →CHOOSE
[16]  ⍝
[17]  ⍝
[18]  ADD:⎕←'EMPLOYEE NUMBER (OR 0 IF DONE)'
[19]   F1←,⎕
[20]  ⍝ Continue if exactly 1 number entered:
[21]   →(1=⍴F1)/A1
[22]   ⎕←'** ENTER 1 NUMBER'
[23]   →ADD
[24]  ⍝ Branch to choice question if 0 entered:
[25]  A1:→(0=F1)/CHOOSE
[26]  ⍝ Continue unless employee number already exists:
[27]   →(¯1=1⍴(FP,1)IOTA F1)⍴A2
[28]   ⎕←'** EMPLOYEE ',(⍕F1),' ALREADY IN LIST'
[29]   →ADD
[30]  A2:⍞←P←'EMPLOYEE NAME (MAX 25 CHARACTERS): '
[31]  ⍝ Ask for name at end of same line:
[32]   F2←(⍴P)↓⍞
[33]  ⍝ Continue unless name too long:
[34]   →(25≥⍴F2)/A3
[35]   ⎕←'** NAME TOO LONG'
[36]   →A2
[37]  A3:⎕←'EMPLOYEE AGE'
[38]   F3←,⎕
[39]  ⍝ Continue if exactly 1 number entered:
[40]   →(1=⍴F3)/A4
[41]   ⎕←'** ENTER 1 NUMBER'
[42]   →A3
[43]  ⍝ Continue if a valid age:
[44]  A4:→((F3=⌈F3)∧(F3≥17)∧F3≤99)/A5
[45]   ⎕←'** AGE MUST BE INTEGER FROM 17 TO 99'
[46]   →A3
[47]  ⍝ Pad name to length 25:
[48]  A5:F2←25↑F2
[49]  ⍝ Catenate new values and ask for more:
[50]   FP←FP CATRECWS 1
[51]   →ADD
```

```
    ∇ EMPLOYEES (continued)
      ⍝
      ⍝
      DELETE:⎕←'ENTER EMPLOYEE NUMBERS TO DELETE'
      ⍝ Ravel to insure a vector, not scalar:
       NUM←,⎕
      ⍝ Continue if all valid numbers:
       IND←(FP,1)IOTA NUM
       →(∧/GOOD←¯1≠(⍴NUM)⍴IND)/D1
       ⎕←'** NOT FOUND: ',⍕(~GOOD)/NUM
       →DELETE
      ⍝ Squeeze out deleted employees:
      D1:FP←FP DELREC IND
       →CHOOSE
      ⍝
      ⍝
      LIST:⎕←'NUMBER   AGE   NAME'
       ⎕←''
      ⍝ Prepare to sort employees by number:
       '' SELECTWS FP, 1 2 3
       G←⍋F1
      ⍝ Sort and display:
       ⎕←(5 0 7 0 ⍕F1[G],[1.5]F3[G]),(((⍴F1),3)⍴' '),F2[G;]
       ⎕←''
       →CHOOSE
      ⍝
      ⍝
      END:
    ∇
```

4.

```
                                          [WSID: MULTIFLO]
    ∇ FP INITFILE FT;BLK;DEL;DISP;INCR;L;LAY;R;TIE;W;⎕IO
      ⍝ Initializes file.  Assumes the file already
      ⍝ exists, contains no components and is tied to FP[2].
       ⎕IO←1
      ⍝ Check validity of arguments:
       ⎕ERROR((1≠⍴⍴FP)∨2≠⍴⍴FT)/'RANK ERROR'
       ⎕ERROR((2≠1⍴⍴FT)∨11≠⍴FP)/'FP LENGTH ERROR'
       INCR←FP[3]
       ⎕ERROR((INCR≠1↓⍴FT)∨FP[8]≠+/×FT[1;])/'FT LENGTH ERROR'
       ⎕ERROR(FP∨.≠⌈FP)/'FP DOMAIN ERROR'
       ⎕ERROR((∧/((FT[1;]>0)/FT[2;])∊⍳4)≤(FT[1;]∨.<0)∨FT[1;]∨
       .≠⌈FT[1;])/'FT DOMAIN ERROR'
       TIE←FP[2]
       ⎕ERROR(~TIE∊⎕FNUMS)/'FILE NOT TIED'
       DISP←FP[4]
       ⎕ERROR(DISP<10)/'DISPLACEMENT ERROR'
       BLK←FP[5]
       ⎕ERROR(BLK≤0)/'BLOCK SIZE ERROR'
       ⎕ERROR(FP[6 7 9 10]∨.≠0)/'USE 0 FOR FP[6 7 9 10]'
```

```
    ∇ INITFILE (continued)
     DEL←FP[11]
     →(×DEL)↓L1
     ⎕ERROR(~(|DEL)∊⍳INCR)/'NONEXISTENT DELETION FIELD
     NUMBER'
     ⎕ERROR(1∨.≠FT[;|DEL])/'USE FT=1 1 FOR DELETION FIELD'
    L1:L←⍳0
     LAY←FP[1]
     →(×LAY)↓L2
     ⎕ERROR(~LAY∊⍳INCR)/'NONEXISTENT LAYER FIELD NUMBER'
     ⎕ERROR(0=W←FT[1;LAY])/'LAYER FIELD INACTIVE'
     ⎕ERROR(LAY=|DEL)/'LAYER AND DELETION FIELDS ARE THE
     SAME'
     L←(0,(W≠1)⍴W)⍴0
     →(2≠FT[2;LAY])⍴L2
     L←(0,(W≠1)⍴W)⍴''
    L2:R←'' ⎕FAPPEND TIE
     R←(0 0 ⍴'')⎕FAPPEND TIE
     R←(0 0 ⍴'')⎕FAPPEND TIE
     R←FT ⎕FAPPEND TIE
     R←(⍳0)⎕FAPPEND TIE
     R←(⍳0)⎕FAPPEND TIE
     R←FP ⎕FAPPEND TIE
     R←(⍳0)⎕FAPPEND TIE
     R←L ⎕FAPPEND TIE
     R←(⍳0)⎕FAPPEND TIE
     →(DISP≤10)⍴0
     R←⍳0
    LOOP:→(DISP>R ⎕FAPPEND TIE)⍴LOOP
    ∇
```

[WSID: MULTIFLO]

```
    ∇ NFP←FP CATREC MAT;ARPS;BIT;BLK;C;CMP;DATA;DCMP;DEL;
      DISP;F;FLD;FREC;FT;FILL;GOOD;I;INCR;INDS;LAYER;LC;LEAD
      ;LV;M;MIN;N;NFLD;NREC;RMAT;RPS;S;SDISP;SETS;T;TIE;VEC;
      W;WID;⎕IO
    ⍝ Catenates rows of MAT to file.
     ⎕ERROR((2<⍴⍴MAT)∨1≠⍴⍴FP)/'RANK ERROR'
     ⎕ERROR(11≠⍴FP)/'LENGTH ERROR'
     NFP←FP
    ⍝ Convert scalar or vector to matrix:
     →(2=⍴⍴MAT)⍴L1
     MAT←(¯2↑ 1 1 ,⍴MAT)⍴MAT
    ⍝ Exit if no records to catenate:
    L1:→(×1⍴⍴MAT)↓0
     ⎕IO←1
     TIE←FP[2]
     INCR←FP[3]
     DISP←FP[4]
     BLK←FP[5]
     DEL←|FP[11]
     FILL←FP[11]<0
     FT←⎕FREAD TIE,4
```

```
    ∇ CATREC (continued)
[18] ⍝ Widths (no. cols.) of fields:
[19]  WID←FT[1;]
[20] ⍝ Indices of active fields:
[21]  FLD←(×WID)/⍳INCR
[22] ⍝ Exclude deletion field:
[23]  NFLD←⍴FLD←(FLD≠DEL)/FLD
[24]  ⎕ERROR((1≠×/⍴MAT)∧(1↓⍴MAT)≠+/WID[FLD])/'LENGTH ERROR'
[25]  ⎕ERROR((2≠FT[2;FLD])∨.≠0=1↑0⍴MAT)/'DOMAIN ERROR'
[26] ⍝ Extend singleton across all columns:
[27]  →(1≠×/⍴MAT)⍴L2
[28]  MAT←(1,+/WID[FLD])⍴MAT
[29] L2:ARPS←RPS←⎕FREAD TIE,8
[30] ⍝ Branch unless ARPS should be read:
[31]  →(=/FP[7 10])⍴L3
[32]  ARPS←⎕FREAD TIE,10
[33] ⍝ Branch unless layered file:
[34] L3:→(×FP[1])↓L7
[35] ⍝ Read layer values:
[36]  LV←⎕FREAD TIE,9
[37] ⍝ Compute the col inds of MAT with layer values:
[38]  LC←(0,+\WID[FLD])[FLD⍳FP[1]]+⍳WID[FP[1]]
[39] ⍝ Convert to scalar if vector fld:
[40]  VEC←1=⍴LC
[41]  LC←(VEC↓⍴LC)⍴LC
[42] ⍝ Use layer value of 1st row of MAT:
[43] L4:LAYER←MAT[1;LC]
[44] ⍝ Branch if a vector layer field:
[45]  →VEC⍴L5
[46] ⍝ Flag rows of MAT in this layer:
[47]  GOOD←MAT[;LC]∧.=LAYER
[48] ⍝ ...and sets with this layer value:
[49]  SETS←LV∧.=LAYER
[50]  →L6
[51] L5:GOOD←MAT[;LC]=LAYER
[52]  SETS←LV=LAYER
[53] ⍝ Put remaining rows in RMAT:
[54] L6:RMAT←(~GOOD)⌿MAT
[55]  MAT←GOOD⌿MAT
[56] ⍝ Consider only non-full sets in this layer
[57] ⍝ or empty sets in any layer:
[58]  SETS←(ARPS=0)∨SETS∧ARPS≠BLK
[59]  →L8
[60] L7:SETS←ARPS≠BLK
[61] ⍝ Convert to indices:
[62] L8:SETS←SETS/⍳⍴SETS
[63]  NREC←1⍴⍴MAT
[64] ⍝ No. records filed so far:
[65]  FREC←0
[66] ⍝ Branch if no slots available in existing sets:
[67] L9:→(BLK≤MIN←⌊/|ARPS[SETS])⍴L25
[68] ⍝ Branch if more than 1 set needed:
[69]  T←NREC-FREC
[70]  →(T>BLK-MIN)⍴L10
```

```
    ∇ CATREC (continued)
[71] ⍝ Choose fullest set which will hold all recs:
[72]  S←BLK-|ARPS[SETS]
[73]  S←SETS[S⍳⌊/(S≥T)/S]
[74]  →L11
[75] ⍝ Choose set with most empty slots:
[76] L10:S←SETS[(|ARPS[SETS])⍳MIN]
[77] ⍝ No. records to be inserted within the set:
[78] L11:M←T⌊RPS[S]-|ARPS[S]
[79] ⍝ No. records to be catenated within the set:
[80]  N←(T-M)⌊BLK-RPS[S]
[81] ⍝ Displacement (no. components) before this set:
[82]  SDISP←DISP+INCR×S+¯1
[83] ⍝ Branch if no deletion field:
[84]  →(×DEL)↓L12
[85] ⍝ Read deletion field for set S:
[86]  BIT←⎕FREAD DCMP←TIE,DEL+SDISP
[87] ⍝ Branch if no records to be inserted:
[88]  →(×M)↓L12
[89] ⍝ Indices of available insertion slots:
[90]  INDS←M⍴(~BIT)/⍳⍴BIT
[91] ⍝ Columns of MAT filed so far:
[92] L12:C←0
[93] ⍝ Next field index:
[94]  I←1
[95] ⍝ Loop by active field:
[96] L13:→(I>NFLD)⍴L19
[97] ⍝ Field number:
[98]  F←FLD[I]
[99] ⍝ Field width (no. columns):
[100]  W←WID[F]
[101] ⍝ Read field F for set S:
[102]  DATA←⎕FREAD CMP←TIE,F+SDISP
[103] ⍝ Branch if a matrix field:
[104]  →(W>1)⍴L15
[105] ⍝ Column of MAT if a vector field:
[106]  C←C+1
[107] ⍝ Branch if no records to insert:
[108]  →(×M)↓L14
[109]  DATA[INDS]←MAT[FREC+⍳M;C]
[110] ⍝ Branch if no records to catenate:
[111]  →(×N)↓L18
[112] L14:DATA←DATA,MAT[(FREC+M)+⍳N;C]
[113]  →L18
[114] ⍝ Branch if no records to insert:
[115] L15:→(×M)↓L16
[116]  DATA[INDS;]←MAT[FREC+⍳M;C+⍳W]
[117] ⍝ Branch if no records to catenate:
[118]  →(×N)↓L17
[119] L16:DATA←DATA,[1]MAT[(FREC+M)+⍳N;C+⍳W]
[120] L17:C←C+W
[121] L18:DATA ⎕FREPLACE CMP
[122]  I←I+1
[123]  →L13
```

```
    ∇ CATREC (continued)
[124] ⍝ Branch if no deletion field:
[125] L19:LEAD←1
[126]  →(×DEL)↓L22
[127] ⍝ Branch if no records to insert:
[128]  →(×M)↓L20
[129] ⍝ Turn active record bits on:
[130]  BIT[INDS]←1
[131]  LEAD←∧/BIT=∧\BIT
[132] ⍝ Branch if no records to catenate:
[133]  →(×N)↓L21
[134] L20:BIT←BIT,N⍴1
[135] L21:BIT ⎕FREPLACE DCMP
[136] ⍝ Increment FREC by no. records added to this set:
[137] L22:FREC←FREC+M+N
[138]  RPS[S]←RPS[S]+N
[139]  FP[9]←FP[9]+T←0=ARPS[S]
[140] ⍝ Replace layer value if file layered and
[141] ⍝ set initially empty:
[142]  →(T∧×FP[1])↓L24
[143] ⍝ Branch if a vector layer field:
[144]  →VEC⍴L23
[145]  LV[S;]←LAYER
[146]  →L24
[147] L23:LV[S]←LAYER
[148] L24:ARPS[S]←(¯1 1)[1+LEAD]×M+N+|ARPS[S]
[149]  FP[7]←FP[7]+N
[150] ⍝ Exit if all of MAT filed:
[151]  →(NREC=FREC)⍴L34
[152]  →L9
[153] ⍝ Add new set; no. fields to be appended:
[154] L25:NFLD←FP[3]
[155] ⍝ No. records to be appended in next set:
[156]  N←BLK⌊NREC-FREC
[157] ⍝ Columns of MAT filed so far:
[158]  C←0
[159] ⍝ Next field number:
[160]  F←1
[161] ⍝ Loop by field:
[162] L26:→(F>NFLD)⍴L32
[163]  W←WID[F]
[164] ⍝ Branch unless a latent field:
[165]  →(×W)⍴L27
[166]  DATA←⍳0
[167]  →L31
[168] ⍝ Branch if a matrix field:
[169] L27:→(W>1)⍴L29
[170] ⍝ Branch unless it's the deletion field:
[171]  →(DEL≠F)⍴L28
[172]  DATA←N⍴1
[173]  →L30
[174] ⍝ Column of MAT if a vector field:
[175] L28:C←C+1
[176]  DATA←MAT[FREC+⍳N;C]
```

```
    ∇ CATREC (continued)
     →L30
    L29:DATA←MAT[FREC+⍳N;C+⍳W]
     C←C+W
    ⍝ Branch unless set must be padded to BLK records:
    L30:→FILL↓L31
     DATA←(BLK,1↓⍴DATA)↑DATA
    L31:T←DATA ⎕FAPPEND TIE
     F←F+1
     →L26
    ⍝ Increment FREC by no. records added to this set:
    L32:FREC←FREC+N
     ARPS←ARPS,N
     RPS←RPS,T←N⌈BLK×FILL
     FP[7]←FP[7]+T
     FP[6]←FP[6]+1
     FP[9]←FP[9]+1
    ⍝ Catenate layer value if file layered:
     →(×FP[1])↓L33
     LV←LV,[1]LAYER
    ⍝ Continue unless all of MAT filed:
    L33:→(NREC≠FREC)⍴L25
    L34:FP[10]←FP[10]+NREC
    ⍝ Branch unless layered:
     →(×FP[1])↓L36
    ⍝ Branch if no data left to file:
     →(×1⍴⍴RMAT)↓L35
    ⍝ Put remaining rows in MAT, continue:
     MAT←RMAT
     →L4
    L35:LV ⎕FREPLACE TIE,9
    L36:RPS ⎕FREPLACE TIE,8
     →(×DEL)↓L37
     ARPS ⎕FREPLACE TIE,10
    L37:FP ⎕FREPLACE TIE,7
     NFP←FP
    ∇
```

[WSID: MULTIFLO]

```
    ∇ NFP←FP CATRECWS NREC;ARPS;BIT;BLK;CMP;DATA;DCMP;DEL;
      DISP;F;FILED;FLD;FREC;FT;FILL;GOOD;I;INCR;INDS;LAYER;
      LEAD;LV;M;MIN;N;NFLD;NR;RANK;RPS;S;SDISP;SETS;SHAPE;T;
      TIE;VAR;VEC;W;WID;⎕IO
    ⍝ Catenates elements/rows of F1,F2,F3,... to file.
     ⎕ERROR((1≠⍴⍴FP)∨1∨.≠⍴NREC)/'RANK ERROR'
     NREC←1⍴NREC
     ⎕ERROR(11≠⍴FP)/'LENGTH ERROR'
     NFP←FP
     ⎕IO←1
     TIE←FP[2]
     INCR←FP[3]
     DISP←FP[4]
     BLK←FP[5]
```

```
    ∇ CATRECWS (continued)
[11]  DEL←|FP[11]
[12]  FILL←FP[11]<0
[13]  FT←⎕FREAD TIE,4
[14] ⍝ Widths (no. cols.) of fields:
[15]  WID←FT[1;]
[16] ⍝ Indices of active fields:
[17]  FLD←(×WID)/⍳INCR
[18] ⍝ Exclude deletion field:
[19]  NFLD←⍴FLD←(FLD≠DEL)/FLD
[20] ⍝ Index of next active field:
[21]  I←1
[22] ⍝ Loop by active field to verify field vars
[23] ⍝ (bypass this loop to make the fn faster and
[24] ⍝ to live dangerously):
[25] L1:→(I>NFLD)⍴L2
[26] ⍝ Field number:
[27]  F←FLD[I]
[28] ⍝ Field width (no. columns):
[29]  W←WID[F]
[30] ⍝ Look at field variable:
[31]  RANK←⍴SHAPE←⍴VAR←⍎'F',⍕F
[32]  ⎕ERROR((2≠FT[2;F])≠0=1↑0⍴VAR)/'DOMAIN ERROR'
[33] ⍝ Continue if singleton data:
[34]  I←I+1
[35]  →(1∧.=SHAPE)⍴L1
[36]  ⎕ERROR(RANK>2)/'RANK ERROR'
[37]  ⎕ERROR(((RANK=1)∧(W=1↑SHAPE)∨(W=1)∧NREC=1↑SHAPE)∨(RANK
      =2)∧(SHAPE∧.=NREC,W)∨SHAPE∧.=1,W)/'LENGTH ERROR'
[38]  →L1
[39] ⍝ Exit if no records to catenate:
[40] L2:→(×NREC)↓0
[41]  ARPS←RPS←⎕FREAD TIE,8
[42] ⍝ Branch unless ARPS should be read:
[43]  →(=/FP[7 10])⍴L3
[44]  ARPS←⎕FREAD TIE,10
[45] ⍝ Branch unless layered file:
[46] L3:→(×FP[1])↓L7
[47] ⍝ Read layer values:
[48]  LV←⎕FREAD TIE,9
[49] ⍝ Flag records filed so far:
[50]  FILED←NREC⍴0
[51] ⍝ Look at layer field:
[52] L4:VAR←⍎'F',⍕FP[1]
[53]  W←WID[FP[1]]
[54]  VAR←(NREC,(W>1)⍴W)⍴VAR
[55] ⍝ Branch if vector field:
[56]  VEC←W=1
[57]  →VEC⍴L5
[58] ⍝ Flag records with layer of 1st unfiled rec:
[59]  LAYER←VAR[FILED⍳0;]
[60]  GOOD←VAR∧.=LAYER
[61] ⍝ ...and sets with this record:
[62]  SETS←LV∧.=LAYER
```

```
    ∇ CATRECWS (continued)
[63]   →L6
[64]  L5:LAYER←VAR[FILEDι0]
[65]   GOOD←VAR=LAYER
[66]   SETS←LV=LAYER
[67]  ⍝ Convert to indices:
[68]  L6:GOOD←GOOD/ιρGOOD
[69]   NR←ρGOOD
[70]  ⍝ Consider only non-full sets in this layer or
[71]  ⍝ empty sets in any layer:
[72]   SETS←(ARPS=0)∨SETS∧ARPS≠BLK
[73]   →L8
[74]  L7:GOOD←ιNR←NREC
[75]   SETS←ARPS≠BLK
[76]  ⍝ Convert to indices:
[77]  L8:SETS←SETS/ιρSETS
[78]  ⍝ No. records filed so far:
[79]   FREC←0
[80]  ⍝ Branch if no slots available in existing sets:
[81]  L9:→(BLK≤MIN←⌊/|ARPS[SETS])ρL24
[82]  ⍝ Branch if more than 1 set needed:
[83]   T←NR-FREC
[84]   →(T>BLK-MIN)ρL10
[85]  ⍝ Choose fullest set which will hold all recs:
[86]   S←BLK-|ARPS[SETS]
[87]   S←SETS[Sι⌊/(S≥T)/S]
[88]   →L11
[89]  ⍝ Choose set with most empty slots:
[90]  L10:S←SETS[(|ARPS[SETS])ιMIN]
[91]  ⍝ No. records to be inserted within the set:
[92]  L11:M←T⌊RPS[S]-|ARPS[S]
[93]  ⍝ No. records to be catenated within the set:
[94]   N←(T-M)⌊BLK-RPS[S]
[95]  ⍝ Displacement (no. components) before this set:
[96]   SDISP←DISP+INCR×S+¯1
[97]  ⍝ Branch if no deletion field:
[98]   →(×DEL)↓L12
[99]  ⍝ Read deletion field for set S:
[100]   BIT←⎕FREAD DCMP←TIE,DEL+SDISP
[101] ⍝ Branch if no records to be inserted:
[102]   →(×M)↓L12
[103] ⍝ Indices of available insertion slots:
[104]   INDS←Mρ(~BIT)/ιρBIT
[105] ⍝ Next field index:
[106] L12:I←1
[107] ⍝ Loop by active field:
[108] L13:→(I>NFLD)ρL18
[109] ⍝ Field number:
[110]   F←FLD[I]
[111] ⍝ Field width (no. columns):
[112]   W←WID[F]
[113] ⍝ Look at field variable:
[114]   VAR←⍎'F',⍕F
```

```
    ∇ CATRECWS (continued)
[115] ⍝ Reshape singletons, vectors and 1-row matrices
[116] ⍝ to proper rank, shape:
[117]  VAR←(NREC,(W>1)⍴W)⍴VAR
[118] ⍝ Read field F for set S:
[119]  DATA←⎕FREAD CMP←TIE,F+SDISP
[120] ⍝ Branch if a matrix field:
[121]  →(W>1)⍴L15
[122] ⍝ Branch if no records to insert:
[123]  →(×M)↓L14
[124]  DATA[INDS]←VAR[GOOD[FREC+⍳M]]
[125] ⍝ Branch if no records to catenate:
[126]  →(×N)↓L17
[127] L14:DATA←DATA,VAR[GOOD[(FREC+M)+⍳N]]
[128]  →L17
[129] ⍝ Branch if no records to insert:
[130] L15:→(×M)↓L16
[131]  DATA[INDS;]←VAR[GOOD[FREC+⍳M];]
[132] ⍝ Branch if no records to catenate:
[133]  →(×N)↓L17
[134] L16:DATA←DATA,[1]VAR[GOOD[(FREC+M)+⍳N];]
[135] L17:DATA ⎕FREPLACE CMP
[136]  I←I+1
[137]  →L13
[138] ⍝ Branch if no deletion field:
[139] L18:LEAD←1
[140]  →(×DEL)↓L21
[141] ⍝ Branch if no records to insert:
[142]  →(×M)↓L19
[143] ⍝ Turn active record bits on:
[144]  BIT[INDS]←1
[145]  LEAD←∧/BIT=∧\BIT
[146] ⍝ Branch if no records to catenate:
[147]  →(×N)↓L20
[148] L19:BIT←BIT,N⍴1
[149] L20:BIT ⎕FREPLACE DCMP
[150] ⍝ Increment FREC by no. records added to this set:
[151] L21:FREC←FREC+M+N
[152]  RPS[S]←RPS[S]+N
[153]  FP[9]←FP[9]+T←0=ARPS[S]
[154] ⍝ Replace layer value if file layered and
[155] ⍝ set initially empty:
[156]  →(T∧×FP[1])↓L23
[157] ⍝ Branch if a vector layer field:
[158]  →VEC⍴L22
[159]  LV[S;]←LAYER
[160]  →L23
[161] L22:LV[S]←LAYER
[162] L23:ARPS[S]←(¯1 1)[1+LEAD]×M+N+|ARPS[S]
[163]  FP[7]←FP[7]+N
[164] ⍝ Exit if all of data in field vars. filed:
[165]  →(NR=FREC)⍴L33
[166]  →L9
```

```
    ∇ CATRECWS (continued)
[167] ⍝ No. records to be appended in next set:
[168] L24:N←BLK⌊NR-FREC
[169] ⍝ Next field number:
[170]  F←1
[171] ⍝ Loop by field:
[172] L25:→(F>INCR)⍴L31
[173]  W←WID[F]
[174] ⍝ Branch unless a latent field:
[175]  →(×W)⍴L26
[176]  DATA←⍳0
[177]  →L30
[178] ⍝ Branch unless it's the deletion field:
[179] L26:→(DEL≠F)⍴L27
[180]  DATA←N⍴1
[181]  →L29
[182] ⍝ Look at field variable:
[183] L27:VAR←⍎'F',⍕F
[184] ⍝ Reshape singletons, vectors and 1-row matrices
[185] ⍝ to proper rank, shape:
[186]  VAR←(NREC,(W>1)⍴W)⍴VAR
[187] ⍝ Branch if a matrix field:
[188]  →(W>1)⍴L28
[189]  DATA←VAR[GOOD[FREC+⍳N]]
[190]  →L29
[191] L28:DATA←VAR[GOOD[FREC+⍳N];]
[192] ⍝ Branch unless set must be padded to BLK records:
[193] L29:→FILL↓L30
[194]  DATA←(BLK,1↓⍴DATA)↑DATA
[195] L30:T←DATA ⎕FAPPEND TIE
[196]  F←F+1
[197]  →L25
[198] ⍝ Increment FREC by no. records added to this set:
[199] L31:FREC←FREC+N
[200]  ARPS←ARPS,N
[201]  RPS←RPS,T←N⌈BLK×FILL
[202]  FP[7]←FP[7]+T
[203]  FP[6]←FP[6]+1
[204]  FP[9]←FP[9]+1
[205] ⍝ Catenate layer value if file layered:
[206]  →(×FP[1])↓L32
[207]  LV←LV,[1]LAYER
[208] ⍝ Continue unless all of data in field vars. filed:
[209] L32:→(NR≠FREC)⍴L24
[210] ⍝ Branch unless layered:
[211] L33:→(×FP[1])↓L34
[212] ⍝ Branch if no data left to file:
[213]  FILED[GOOD]←1
[214]  →(∧/FILED)↓L4
[215]  LV ⎕FREPLACE TIE,9
[216] L34:RPS ⎕FREPLACE TIE,8
[217]  →(×DEL)↓L35
[218]  ARPS ⎕FREPLACE TIE,10
[219] L35:FP[10]←FP[10]+NREC
```

```
    ∇ CATRECWS (continued)
[220]  FP ⎕FREPLACE TIE,7
[221]  NFP←FP
[222] ⍝ Erase global field variables:
[223]  F←⍕(NFLD,1)⍴FLD
[224]  F←'F',(+/' '=F)⌽F
[225]  F←⎕EX F
    ∇
```

```
                                        [WSID: MULTIFLO]
    ∇ INDS←FP IOTA VALS;ARPS;BIT;DATA;DEL;DISP;FLD;FOUND;FT;
      INCR;IND;LV;NDEL;NSET;NUM;RPS;S;SDISP;SET;SETS;SHAPE;T
      ;TIE;VIND;WID;⎕IO
[1]   ⍝ Searches through field FP[12] for VALS and
[2]   ⍝ returns 2-row matrix of indices.  First row
[3]   ⍝ is set number (origin 1); second row is index
[4]   ⍝ (origin 1) within set.  Result contains ¯1s
[5]   ⍝ where corresponding value is not found.
[6]   ⍝ Shape of result is 2,⍴VALS.  Requires CMIOTA
[7]   ⍝ function if a character matrix field.
[8]    ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
[9]    ⎕ERROR(12≠⍴FP)/'LENGTH ERROR'
[10]   ⎕IO←1
[11]   INCR←FP[3]
[12]   DEL←|FP[11]
[13]   FLD←FP[12]
[14]   ⎕ERROR((FLD∊⍳INCR)⍲FLD≠DEL)/'INVALID FIELD NUMBER'
[15]   TIE←FP[2]
[16]   FT←⎕FREAD TIE,4
[17]  ⍝ Width (no. columns) of specified field:
[18]   WID←FT[1;FLD]
[19]   ⎕ERROR(0=WID)/'INACTIVE FIELD'
[20]   ⎕ERROR((WID≠1)∧(×⍴⍴VALS)∧WID≠¯1↑⍴VALS)/'LENGTH ERROR'
[21]  ⍝ Numeric field?
[22]   NUM←2≠FT[2;FLD]
[23]   ⎕ERROR(NUM≠0=1↑0⍴VALS)/'DOMAIN ERROR'
[24]  ⍝ Branch unless VALS is a scalar for a mat fld:
[25]   →((WID=1)∨×⍴⍴VALS)⍴L1
[26]  ⍝ Treat a scalar as a vector if a matrix field:
[27]   VALS←WID⍴VALS
[28]  ⍝ Determine shape of result; then 'ravel' VALS:
[29]  L1:SHAPE←(-WID≠1)↓⍴VALS
[30]   VALS←((×/SHAPE),(WID≠1)⍴WID)⍴VALS
[31]  ⍝ Initialize 'raveled' result:
[32]   INDS←(2,×/SHAPE)⍴¯1
[33]  ⍝ Exit if VALS is empty or if no records on file:
[34]   →((0∊SHAPE)∨0=FP[10])⍴L14
[35]   DISP←FP[4]
[36]   ARPS←RPS←⎕FREAD TIE,8
[37]  ⍝ Branch unless ARPS should be read:
[38]   →(=/FP[7 10])⍴L2
[39]   ARPS←⎕FREAD TIE,10
```

```
    ∇ IOTA (continued)
[40] ⍝ Consider only nonempty sets:
[41] L2:SETS←0≠ARPS
[42] ⍝ ...and sets specified in <layers>:
[43]  →(×FP[1])↓L6
[44]  →(×⎕NC 'layers')↓L6
[45] ⍝ Read layer values:
[46]  LV←⎕FREAD TIE,9
[47] ⍝ Branch if matrix layers field:
[48]  →(2=ρρLV)ρL3
[49]  SETS←SETS∧LV∊layers
[50]  →L6
[51] ⍝ Convert <layers> to matrix if not already:
[52] L3:⎕ERROR(FT[1;FP[1]]≠¯1↑ρlayers)/'LENGTH ERROR'
[53]  →(2=ρρlayers)ρL4
[54]  layers←((×/¯1↓ρlayers),¯1↑ρlayers)ρlayers
[55] ⍝ Branch if numeric matrix field:
[56] L4:→(0=1↑0ρLV)ρL5
[57]  SETS←SETS∧(layers CMIOTA LV)≤1ρρlayers
[58]  →L6
[59] L5:SETS←SETS∧∨/LV∧.=⍉layers
[60] ⍝ Convert to indices; erase <layers>:
[61] L6:SETS←SETS/ιρSETS
[62]  NSET←ρSETS
[63]  LV←⎕EX 'layers'
[64] ⍝ Indices into INDS of values not yet found:
[65]  VIND←ι1ρρVALS
[66] ⍝ Last set index:
[67]  S←0
[68] ⍝ Loop by nonempty set:
[69] L7:→(S≥NSET)ρL14
[70] ⍝ Current set index and number:
[71]  S←S+1
[72]  SET←SETS[S]
[73] ⍝ Displacement (no. components) before this set:
[74]  SDISP←DISP+INCR×SET+¯1
[75] ⍝ Read field FLD for set SET:
[76]  DATA←⎕FREAD TIE,FLD+SDISP
[77] ⍝ Branch if deletion field unneeded:
[78]  →(NDEL←ARPS[SET]=RPS[SET])ρL9
[79] ⍝ Branch if deletion field needed:
[80]  →(NDEL←ARPS[SET]>0)↓L8
[81] ⍝ Active records are leading records:
[82]  DATA←(ARPS[SET],(WID>1)ρWID)ρDATA
[83]  →L9
[84] ⍝ Read and apply deletion field for set SET:
[85] L8:BIT←⎕FREAD TIE,DEL+SDISP
[86]  DATA←BIT⌿DATA
[87] ⍝ Branch if a matrix field:
[88] L9:→(WID>1)ρL10
[89] ⍝ Search algorithm for vector field:
[90]  IND←DATAιVALS
[91]  →L12
```

```
    ∇ IOTA (continued)
[92]  ⍝ Branch if a numeric matrix field:
[93]  L10:→NUMρL11
[94]  ⍝ Search algorithm for character matrix field:
[95]   IND←DATA CMIOTA VALS
[96]   →L12
[97]  ⍝ Search algorithm for numeric matrix field:
[98]  L11:IND←1++/∧\VALS∨.≠⍉DATA
[99]  ⍝ Determine those values found in this set:
[100] L12:FOUND←IND≤1ρρDATA
[101] ⍝ Continue to next set if none found:
[102]  IND←FOUND/IND
[103]  →(×ρIND)↓L7
[104] ⍝ Consider deletion field if applicable:
[105]  →NDELρL13
[106]  IND←(BIT/ιρBIT)[IND]
[107] ⍝ Insert indices in result:
[108] L13:INDS[;FOUND/VIND]←SET,[0.5]IND
[109] ⍝ Compress indices of remaining values:
[110]  BIT←~FOUND
[111]  VIND←BIT/VIND
[112] ⍝ Exit if all found:
[113]  →(×ρVIND)↓L14
[114] ⍝ Else, compress remaining values:
[115]  VALS←BIT≠VALS
[116]  →L7
[117] L14:INDS←(2,SHAPE)ρINDS
    ∇


                                       [WSID: MULTIFLO]
    ∇ INDS←IOTARHO FP;ARPS;BIT;CMPS;DEL;FT;I;LV;NSET;RPS;
      SETS;START;TIE;⎕IO
[1]   ⍝ Returns 2 row matrix of indices of all active records
[2]   ⍝ on file.  First row is set number (origin 1); second
[3]   ⍝ row is index (origin 1) within set.
[4]    ⎕ERROR(1≠ρρFP)/'RANK ERROR'
[5]    ⎕ERROR(11≠ρFP)/'LENGTH ERROR'
[6]    ⎕IO←1
[7]   ⍝ Branch if some records of file:
[8]    →(×FP[10])ρL1
[9]    INDS← 2 0 ρ0
[10]   →L8
[11]  L1:TIE←FP[2]
[12]   ARPS←RPS←⎕FREAD TIE,8
[13]  ⍝ Branch unless ARPS should be read:
[14]   →(=/FP[7 10])ρL2
[15]   ARPS←⎕FREAD TIE,10
[16]  ⍝ Consider only nonempty sets:
[17]  L2:SETS←0≠ARPS
[18]  ⍝ ...and sets specified in <layers>:
[19]   →(×FP[1])↓L6
[20]   →(×⎕NC 'layers')↓L6
```

```
    ∇ IOTARHO (continued)
  ⍝ Read layer values and fld types:
   LV←⎕FREAD TIE,9
   FT←⎕FREAD TIE,4
  ⍝ Branch if matrix layers field:
   →(2=⍴⍴LV)⍴L3
   SETS←SETS∧LV∊layers
   →L6
  ⍝ Convert <layers> to matrix if not already:
  L3:⎕ERROR(FT[1;FP[1]]≠¯1↑⍴layers)/'LENGTH ERROR'
   →(2=⍴⍴layers)⍴L4
   layers←((×/¯1↓⍴layers),¯1↑⍴layers)⍴layers
  ⍝ Branch if numeric matrix field:
  L4:→(0=1↑0⍴LV)⍴L5
   SETS←SETS∧(layers CMIOTA LV)≤1⍴⍴layers
   →L6
  L5:SETS←SETS∧∨/LV∧.=⍉layers
  ⍝ Convert to indices; erase <layers>:
  L6:SETS←SETS/⍳⍴SETS
   LV←⎕EX 'layers'
  ⍝ Construct 2 rows: set inds, rec inds:
   INDS←|ARPS[SETS]
  ⍝ I←MONIOTA INDS:
   I←I+⍳⍴I←INDS/¯1↓0,+\-INDS
   INDS←(INDS/SETS),[0.5]I
  ⍝ Exit if no deleted records:
   →(ARPS[SETS]∧.=RPS[SETS])⍴L8
  ⍝ Deletion field number:
   DEL←|FP[11]
  ⍝ Set no.s for which deletion bits are to be read:
   SETS←(ARPS[SETS]<0)/SETS
   NSET←⍴SETS
  ⍝ Exit if none:
   →(×NSET)↓L8
  ⍝ Component numbers of deletion bit fields:
   CMPS←(DEL+FP[4])+FP[3]×SETS+¯1
  ⍝ Starts of each selected set's indices in result:
   ARPS←|ARPS
   START←(0,+\ARPS)[SETS]
  ⍝ Compress ARPS to selected sets:
   ARPS←ARPS[SETS]
  ⍝ Next set index:
   I←1
  ⍝ Loop by set; read deletion bits:
  L7:BIT←⎕FREAD TIE,CMPS[I]
  ⍝ Insert correct indices in result:
   INDS[2;START[I]+⍳ARPS[I]]←BIT/⍳⍴BIT
  ⍝ Increment and repeat if more:
   I←I+1
   →(I≤NSET)⍴L7
  L8:I←⎕EX 'layers'
    ∇
```

```
                                        [WSID: MULTIFLO]
      ∇ ΔΔINDS←ΔΔEXPR SLASHIOTARHO ΔΔFP;ΔΔACTIVE;ΔΔALL;ΔΔARPS;
        ΔΔBIT;ΔΔDATA;ΔΔDEL;ΔΔDISP;ΔΔF;ΔΔFLD;ΔΔFLDS;ΔΔFNAM;ΔΔFT
        ;ΔΔINCR;ΔΔIND;ΔΔLV;ΔΔNDEL;ΔΔNFLD;ΔΔNSET;ΔΔRPS;ΔΔS;
        ΔΔSDISP;ΔΔSEL;ΔΔSET;ΔΔSETS;ΔΔTIE;ΔΔWID;⎕IO
[1]   ⍝ Loops through active sets doing the following:
[2]   ⍝ reads the fields specified in 11↓FP (calling them
[3]   ⍝ F5, F9, etc. for fields 5, 9, etc.), executes the
[4]   ⍝ character vector EXPR, converts the resulting bit
[5]   ⍝ vector to indices and returns the indices of all
[6]   ⍝ records found for which EXPR returns a 1.  Result
[7]   ⍝ is a 2 row matrix: first row is set number (origin
[8]   ⍝ 1); second row is index (origin 1) within set. If
[9]   ⍝ EXPR and 11↓FP are empty, all records are selected.
[10]  ⍝ Note that EXPR is executed in origin 1; e.g.
[11]  ⍝ 'F3[;2]' always refers to 2nd column.
[12]   ⎕ERROR((1≠⍴⍴ΔΔFP)∨1<⍴⍴ΔΔEXPR)/'RANK ERROR'
[13]   ⎕ERROR(0=1↑0⍴ΔΔEXPR)/'DOMAIN ERROR'
[14]   ⎕IO←1
[15]   ΔΔALL←0=ΔΔNFLD←⍴ΔΔFLDS←11↓ΔΔFP
[16]   ⎕ERROR((ΔΔALL=ΔΔEXPR∨.≠' ')∨11>⍴ΔΔFP)/'LENGTH ERROR'
[17]   ΔΔINCR←ΔΔFP[3]
[18]   ΔΔDEL←|ΔΔFP[11]
[19]   ⎕ERROR((∧/ΔΔFLDS∊⍳ΔΔINCR)≤ΔΔDEL∊ΔΔFLDS)/'INVALID FIELD
        NUMBER'
[20]   ΔΔTIE←ΔΔFP[2]
[21]   ΔΔFT←⎕FREAD ΔΔTIE,4
[22]  ⍝ Width (no. columns) of specified fields:
[23]   ΔΔWID←ΔΔFT[1;ΔΔFLDS]
[24]   ⎕ERROR(0∊ΔΔWID)/'INACTIVE FIELD'
[25]  ⍝ Initialize result as empty:
[26]   ΔΔINDS← 2 0 ⍴0
[27]  ⍝ Exit if no records on file:
[28]   →(×ΔΔFP[10])↓0
[29]  ⍝ Field names (e.g. 'F5 F9') to be erased below:
[30]   ΔΔFNAM←⍕(ΔΔNFLD,1)⍴ΔΔFLDS
[31]   ΔΔFNAM←'F',(+/' '=ΔΔFNAM)⌽ΔΔFNAM
[32]   ΔΔDISP←ΔΔFP[4]
[33]   ΔΔARPS←ΔΔRPS←⎕FREAD ΔΔTIE,8
[34]  ⍝ Branch unless ARPS should be read:
[35]   →(=/ΔΔFP[7 10])⍴ΔΔL1
[36]   ΔΔARPS←⎕FREAD ΔΔTIE,10
[37]  ⍝ Consider only nonempty sets:
[38]  ΔΔL1:ΔΔSETS←0≠ΔΔARPS
[39]  ⍝ ...and sets specified in <layers>:
[40]   →(×ΔΔFP[1])↓ΔΔL5
[41]   →(×⎕NC 'layers')↓ΔΔL5
[42]  ⍝ Read layer values:
[43]   ΔΔLV←⎕FREAD ΔΔTIE,9
[44]  ⍝ Branch if matrix layers field:
[45]   →(2=⍴⍴ΔΔLV)⍴ΔΔL2
[46]   ΔΔSETS←ΔΔSETS∧ΔΔLV∊layers
[47]   →ΔΔL5
```

```
    ∇ SLASHIOTARHO (continued)
[48] ⍝ Convert <layers> to matrix if not already:
[49] ΔΔL2:⎕ERROR(ΔΔFT[1;ΔΔFP[1]]≠¯1↑⍴layers)/'LENGTH ERROR'
[50]  →(2=⍴⍴layers)⍴ΔΔL3
[51]  layers←((×/¯1↓⍴layers),¯1↑⍴layers)⍴layers
[52] ⍝ Branch if numeric matrix field:
[53] ΔΔL3:→(0=1↑0⍴ΔΔLV)⍴ΔΔL4
[54]  ΔΔSETS←ΔΔSETS∧(layers CMIOTA ΔΔLV)≤1⍴⍴layers
[55]  →ΔΔL5
[56] ΔΔL4:ΔΔSETS←ΔΔSETS∧∨/ΔΔLV∧.=⍉layers
[57] ⍝ Convert to indices; erase <layers>:
[58] ΔΔL5:ΔΔSETS←ΔΔSETS/⍳⍴ΔΔSETS
[59]  ΔΔNSET←⍴ΔΔSETS
[60]  ΔΔLV←⎕EX 'layers'
[61] ⍝ Last set index:
[62]  ΔΔS←0
[63] ⍝ Loop by nonempty set:
[64] ΔΔL6:→(ΔΔS≥ΔΔNSET)⍴0
[65] ⍝ Current set index and number:
[66]  ΔΔS←ΔΔS+1
[67]  ΔΔSET←ΔΔSETS[ΔΔS]
[68] ⍝ Displacement (no. components) before this set:
[69]  ΔΔSDISP←ΔΔDISP+ΔΔINCR×ΔΔSET+¯1
[70] ⍝ Branch if deletion field unneeded:
[71]  →(ΔΔNDEL←ΔΔARPS[ΔΔSET]>0)⍴ΔΔL7
[72] ⍝ Read deletion field for set SET:
[73]  ΔΔBIT←⎕FREAD ΔΔTIE,ΔΔDEL+ΔΔSDISP
[74] ⍝ Branch if no selection expression:
[75] ΔΔL7:ΔΔSEL←1
[76]  →ΔΔALL⍴ΔΔL12
[77] ⍝ Are all records in this set active?
[78]  ΔΔACTIVE←ΔΔARPS[ΔΔSET]=ΔΔRPS[ΔΔSET]
[79] ⍝ Last field index:
[80]  ΔΔF←0
[81] ⍝ Loop by field specified:
[82] ΔΔL8:→(ΔΔF≥ΔΔNFLD)⍴ΔΔL11
[83] ⍝ Current field index and number:
[84]  ΔΔF←ΔΔF+1
[85]  ΔΔFLD←ΔΔFLDS[ΔΔF]
[86] ⍝ Read field FLD for set SET:
[87]  ΔΔDATA←⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP
[88] ⍝ Branch if all records in this set active:
[89]  →ΔΔACTIVE⍴ΔΔL10
[90] ⍝ Branch if deletion field needed:
[91]  →ΔΔNDEL↓ΔΔL9
[92] ⍝ Active records are leading records:
[93]  ΔΔDATA←(ΔΔARPS[ΔΔSET],(ΔΔWID[ΔΔF]>1)⍴ΔΔWID[ΔΔF])⍴
      ΔΔDATA
[94]  →ΔΔL10
[95] ⍝ Apply deletion field:
[96] ΔΔL9:ΔΔDATA←ΔΔBIT⌿ΔΔDATA
[97] ⍝ Assign data to global variable Fn:
[98] ΔΔL10:⍎'F',(⍕ΔΔFLD),'←ΔΔDATA'
[99]  →ΔΔL8
```

```
    ∇ SLASHIOTARHO (continued)
[100] ⍝ Once all fields have been read, execute EXPR:
[101] ΔΔL11:ΔΔSEL←⍎ΔΔEXPR
[102] ⍝ Erase field variables (e.g. F5, F9,...):
[103]  ΔΔIND←⎕EX ΔΔFNAM
[104] ⍝ Continue to next set if none found:
[105] ΔΔL12:ΔΔIND←ΔΔSEL/⍳|ΔΔARPS[ΔΔSET]
[106]  →(×⍴ΔΔIND)↓ΔΔL6
[107] ⍝ Consider deletion field if applicable:
[108]  →ΔΔNDEL⍴ΔΔL13
[109]  ΔΔIND←(ΔΔBIT/⍳⍴ΔΔBIT)[ΔΔIND]
[110] ⍝ Catenate indices to result:
[111] ΔΔL13:ΔΔINDS←ΔΔINDS,ΔΔSET,[0.5]ΔΔIND
[112]  →ΔΔL6
    ∇
```

[WSID: MULTIFLO]

```
    ∇ NFP←FP DELREC INDS;BIT;CMP;DEL;DISP;F;FLD;FLDS;INCR;
      IND;NEW;NFLD;NSET;OLD;RPS;S;SDISP;SET;SETS;TIE;UNQ;⎕IO
[1]  ⍝ Deletes records identified by file indices matrix
[2]  ⍝ INDS.  First row is set number (origin 1); second
[3]  ⍝ row is index (origin 1) within set.  INDS may be
[4]  ⍝ of any dimension as long as its first coordinate
[5]  ⍝ is 2.
[6]   ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
[7]   ⎕ERROR((11≠⍴FP)∨2≠1↑⍴INDS)/'LENGTH ERROR'
[8]  ⍝ Exit if nothing to delete:
[9]   NFP←FP
[10]  →(0∊⍴INDS)⍴0
[11]  SETS←, 1 0 ⌿INDS
[12]  INDS←, 0 1 ⌿INDS
[13]  ⎕IO←1
[14]  TIE←FP[2]
[15]  INCR←FP[3]
[16]  DISP←FP[4]
[17]  DEL←|FP[11]
[18] ⍝ RPS will be changed if DEL=0; ARPS will be changed
[19] ⍝ if DEL>0.  Read and replace only one or the other:
[20]  RPS←⎕FREAD TIE,8+2××DEL
[21] ⍝ Determine active fields if no deletion field:
[22]  →(×DEL)⍴L1
[23]  NFLD←⍴FLDS←(×(⎕FREAD TIE,4)[1;])/⍳INCR
[24] ⍝ Determine distinct set numbers (deleting ¯1s):
[25] L1:UNQ←SETS[⍋SETS]
[26]  NSET←⍴UNQ←(UNQ≠¯1↓¯1,UNQ)/UNQ
[27] ⍝ Last set index:
[28]  S←0
[29] ⍝ Loop by distinct set:
[30] L2:→(S≥NSET)⍴L5
[31] ⍝ Current set index and number:
[32]  S←S+1
[33]  SET←UNQ[S]
```

```
    ∇ DELREC (continued)
[34] ⍝ Displacement (no. components) before this set:
[35]  SDISP←DISP+INCR×SET+¯1
[36] ⍝ Indices to delete in this set:
[37]  IND←(SET=SETS)/INDS
[38] ⍝ Branch unless deletion field exists:
[39]  →(×DEL)↓L3
[40] ⍝ Read deletion field for set SET:
[41]  BIT←⎕FREAD CMP←TIE,DEL+SDISP
[42] ⍝ Turn off specified indices and replace:
[43]  BIT[IND]←0
[44]  BIT ⎕FREPLACE CMP
[45] ⍝ Compute new no. records:
[46]  NEW←+/BIT
[47]  OLD←|RPS[SET]
[48] ⍝ Reset parameters:
[49]  NFP[9]←NFP[9]-NEW=0
[50]  NFP[10]←NFP[10]+NEW-OLD
[51]  RPS[SET]←NEW×(¯1 1)[1+NEW=+/∧\BIT]
[52]  →L2
[53] ⍝
[54] ⍝ Turn off specified indices:
[55] L3:OLD←RPS[SET]
[56]  BIT←OLDρ1
[57]  BIT[IND]←0
[58] ⍝ Compute new no. records:
[59]  NEW←+/BIT
[60] ⍝ Reset parameters:
[61]  NFP[9]←NFP[9]-NEW=0
[62]  NFP[7]←NFP[10]←NFP[7]+NEW-OLD
[63]  RPS[SET]←NEW
[64] ⍝ Last field index:
[65]  F←0
[66] ⍝ Loop by active field:
[67] L4:→(F≥NFLD)ρL2
[68] ⍝ Current field index and number:
[69]  F←F+1
[70]  FLD←FLDS[F]
[71] ⍝ Read, compress, replace field FLD for set SET:
[72]  CMP←TIE,FLD+SDISP
[73]  (BIT/⎕FREAD CMP)⎕FREPLACE CMP
[74]  →L4
[75] ⍝ File either RPS or ARPS:
[76] L5:RPS ⎕FREPLACE TIE,8+2××DEL
[77]  NFP ⎕FREPLACE TIE,7
    ∇
```

[WSID: MULTIFLO]

```
     ∇ ΔΔNFP←ΔΔEXPR COMPRESS ΔΔFP;ΔΔAFLDS;ΔΔALL;ΔΔARPS;ΔΔBIT;
       ΔΔCMP;ΔΔDATA;ΔΔDEL;ΔΔDISP;ΔΔF;ΔΔFLD;ΔΔFLDS;ΔΔFNAM;ΔΔFT
       ;ΔΔINCR;ΔΔLV;ΔΔNAFLD;ΔΔNDEL;ΔΔNEW;ΔΔNFLD;ΔΔNSET;ΔΔOLD;
       ΔΔRPS;ΔΔS;ΔΔSDISP;ΔΔSEL;ΔΔSET;ΔΔSETS;ΔΔTIE;ΔΔWID;⎕IO
[1]    ⍝ Loops through active sets doing the following:
[2]    ⍝ reads the fields specified in 11↓FP (calling
[3]    ⍝ them F5, F9, etc. for fields 5, 9, etc.),
[4]    ⍝ executes the character vector EXPR, and deletes
[5]    ⍝ the records of that set which correspond to 0s
[6]    ⍝ in the resulting bit vector.  If EXPR and 11↓FP
[7]    ⍝ are empty, all records are selected (none are
[8]    ⍝ deleted).  Note that EXPR is executed in origin
[9]    ⍝ 1; e.g. 'F3[;2]' always refers to 2nd column.
[10]    ⎕ERROR((1≠⍴⍴ΔΔFP)∨1<⍴⍴ΔΔEXPR)/'RANK ERROR'
[11]    ⎕ERROR(0=1↑0⍴ΔΔEXPR)/'DOMAIN ERROR'
[12]    ⎕IO←1
[13]    ΔΔALL←0=ΔΔNFLD←⍴ΔΔFLDS←11↓ΔΔFP
[14]    ⎕ERROR((ΔΔALL=ΔΔEXPR∨.≠' ')∨11>⍴ΔΔFP)/'LENGTH ERROR'
[15]   ⍝ Exit if all records selected:
[16]    →ΔΔALL⍴0
[17]    ΔΔINCR←ΔΔFP[3]
[18]    ΔΔDEL←|ΔΔFP[11]
[19]    ⎕ERROR((∧/ΔΔFLDS∊⍳ΔΔINCR)≤ΔΔDEL∊ΔΔFLDS)/'INVALID FIELD
         NUMBER'
[20]    ΔΔTIE←ΔΔFP[2]
[21]    ΔΔFT←⎕FREAD ΔΔTIE,4
[22]   ⍝ Width (no. columns) of fields:
[23]    ΔΔWID←ΔΔFT[1;]
[24]    ⎕ERROR(0∊ΔΔWID[ΔΔFLDS])/'INACTIVE FIELD'
[25]   ⍝ Exit if no records on file:
[26]    ΔΔNFP←11⍴ΔΔFP
[27]    →(×ΔΔFP[10])↓0
[28]   ⍝ Field names (e.g. 'F5 F9') to be erased below:
[29]    ΔΔFNAM←⍕(ΔΔNFLD,1)⍴ΔΔFLDS
[30]    ΔΔFNAM←'F',(+/' '=ΔΔFNAM)⌽ΔΔFNAM
[31]    ΔΔDISP←ΔΔFP[4]
[32]    ΔΔARPS←ΔΔRPS←⎕FREAD ΔΔTIE,8
[33]   ⍝ Branch unless ARPS should be read:
[34]    →(=/ΔΔFP[7 10])⍴ΔΔL1
[35]    ΔΔARPS←⎕FREAD ΔΔTIE,10
[36]   ⍝ Determine active fields if no deletion field:
[37]   ΔΔL1:→(×ΔΔDEL)⍴ΔΔL2
[38]    ΔΔNAFLD←⍴ΔΔAFLDS←(×ΔΔWID)/⍳ΔΔINCR
[39]   ⍝ Consider only nonempty sets:
[40]   ΔΔL2:ΔΔSETS←0≠ΔΔARPS
[41]   ⍝ ...and sets specified in <layers>:
[42]    →(×ΔΔFP[1])↓ΔΔL6
[43]    →(×⎕NC 'layers')↓ΔΔL6
[44]   ⍝ Read layer values:
[45]    ΔΔLV←⎕FREAD ΔΔTIE,9
[46]   ⍝ Branch if matrix layers field:
[47]    →(2=⍴⍴ΔΔLV)⍴ΔΔL3
[48]    ΔΔSETS←ΔΔSETS∧ΔΔLV∊layers
```

```
    ∇ COMPRESS (continued)
[49]  →ΔΔL6
[50] ⍝ Convert <layers> to matrix if not already:
[51] ΔΔL3:⎕ERROR(ΔΔFT[1;ΔΔFP[1]]≠¯1↑ρlayers)/'LENGTH ERROR'
[52]  →(2=ρρlayers)ρΔΔL4
[53]  layers←((×/¯1↓ρlayers),¯1↑ρlayers)ρlayers
[54] ⍝ Branch if numeric matrix field:
[55] ΔΔL4:→(0=1↑0ρΔΔLV)ρΔΔL5
[56]  ΔΔSETS←ΔΔSETS∧(layers CMIOTA ΔΔLV)≤1ρρlayers
[57]  →ΔΔL6
[58] ΔΔL5:ΔΔSETS←ΔΔSETS∧∨/ΔΔLV∧.=⍉layers
[59] ⍝ Convert to indices; erase <layers>:
[60] ΔΔL6:ΔΔSETS←ΔΔSETS/⍳ρΔΔSETS
[61]  ΔΔNSET←ρΔΔSETS
[62]  ΔΔLV←⎕EX 'layers'
[63] ⍝ Last set index:
[64]  ΔΔS←0
[65] ⍝ Loop by nonempty set:
[66] ΔΔL7:→(ΔΔS≥ΔΔNSET)ρΔΔL17
[67] ⍝ Current set index and number:
[68]  ΔΔS←ΔΔS+1
[69]  ΔΔSET←ΔΔSETS[ΔΔS]
[70] ⍝ Displacement (no. components) before this set:
[71]  ΔΔSDISP←ΔΔDISP+ΔΔINCR×ΔΔSET+¯1
[72] ⍝ Branch if deletion field unneeded:
[73]  →(ΔΔNDEL←ΔΔARPS[ΔΔSET]>0)ρΔΔL8
[74] ⍝ Read deletion field for set SET:
[75]  ΔΔBIT←⎕FREAD ΔΔTIE,ΔΔDEL+ΔΔSDISP
[76] ⍝ Are all records in this set active:
[77] ΔΔL8:ΔΔALL←ΔΔARPS[ΔΔSET]=ΔΔRPS[ΔΔSET]
[78] ⍝ Last field index:
[79]  ΔΔF←0
[80] ⍝ Loop by field specified:
[81] ΔΔL9:→(ΔΔF≥ΔΔNFLD)ρΔΔL12
[82] ⍝ Current field index and number:
[83]  ΔΔF←ΔΔF+1
[84]  ΔΔFLD←ΔΔFLDS[ΔΔF]
[85] ⍝ Read field FLD for set SET:
[86]  ΔΔDATA←⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP
[87] ⍝ Branch if all records in this set active:
[88]  →ΔΔALLρΔΔL11
[89] ⍝ Branch if deletion field needed:
[90]  →ΔΔNDEL↓ΔΔL10
[91] ⍝ Active records are leading records:
[92]  ΔΔDATA←(ΔΔARPS[ΔΔSET],(ΔΔWID[ΔΔFLD]>1)ρΔΔWID[ΔΔFLD])ρ
      ΔΔDATA
[93]  →ΔΔL11
[94] ⍝ Apply deletion field:
[95] ΔΔL10:ΔΔDATA←ΔΔBIT⌿ΔΔDATA
[96] ⍝ Assign data to global variable Fn:
[97] ΔΔL11:⍎'F',(⍕ΔΔFLD),'←ΔΔDATA'
[98]  →ΔΔL9
[99] ⍝ Once all fields have been read, execute EXPR:
[100] ΔΔL12:ΔΔSEL←⍎ΔΔEXPR
```

```
    ∇ COMPRESS (continued)
[101] ⍝ Erase field variables (e.g. F5, F9,...):
[102]  ΔΔOLD←⎕EX ΔΔFNAM
[103] ⍝ Continue to next set if all records selected:
[104]  →(∧/ΔΔSEL)⍴ΔΔL7
[105] ⍝ Branch unless deletion field exists:
[106]  →(×ΔΔDEL)↓ΔΔL15
[107] ⍝ Branch if deletion field not read in:
[108]  →ΔΔNDEL⍴ΔΔL13
[109] ⍝ Reset BIT:
[110]  ΔΔBIT←ΔΔBIT\ΔΔSEL
[111]  →ΔΔL14
[112] ΔΔL13:ΔΔBIT←ΔΔRPS[ΔΔSET]↑ΔΔSEL
[113] ⍝ Replace deletion field:
[114] ΔΔL14:ΔΔBIT ⎕FREPLACE ΔΔTIE,ΔΔDEL+ΔΔSDISP
[115] ⍝ Compute new no. records:
[116]  ΔΔNEW←+/ΔΔBIT
[117]  ΔΔOLD←|ΔΔARPS[ΔΔSET]
[118] ⍝ Reset parameters:
[119]  ΔΔNFP[9]←ΔΔNFP[9]-ΔΔNEW=0
[120]  ΔΔNFP[10]←ΔΔNFP[10]+ΔΔNEW-ΔΔOLD
[121]  ΔΔARPS[ΔΔSET]←ΔΔNEW×(¯1 1)[1+ΔΔNEW=+/∧\ΔΔBIT]
[122]  →ΔΔL7
[123] ⍝ Compute new no. records:
[124] ΔΔL15:ΔΔNEW←+/ΔΔSEL
[125]  ΔΔOLD←ΔΔRPS[ΔΔSET]
[126] ⍝ Reset parameters:
[127]  ΔΔNFP[9]←ΔΔNFP[9]-ΔΔNEW=0
[128]  ΔΔNFP[7]←ΔΔNFP[10]←ΔΔNFP[7]+ΔΔNEW-ΔΔOLD
[129]  ΔΔRPS[ΔΔSET]←ΔΔNEW
[130] ⍝ Last field index:
[131]  ΔΔF←0
[132] ⍝ Loop by active field:
[133] ΔΔL16:→(ΔΔF≥ΔΔNAFLD)⍴ΔΔL7
[134] ⍝ Current field index and number:
[135]  ΔΔF←ΔΔF+1
[136]  ΔΔFLD←ΔΔAFLDS[ΔΔF]
[137] ⍝ Read, compress, replace field FLD for set SET:
[138]  ΔΔCMP←ΔΔTIE,ΔΔFLD+ΔΔSDISP
[139]  (ΔΔSEL⌿⎕FREAD ΔΔCMP)⎕FREPLACE ΔΔCMP
[140]  →ΔΔL16
[141] ΔΔL17:ΔΔNFP ⎕FREPLACE ΔΔTIE,7
[142] ⍝ RPS has been changed if DEL=0; ARPS has been changed
[143] ⍝ if DEL>0.  Replace only one or the other.
[144]  →(×ΔΔDEL)⍴ΔΔL18
[145]  ΔΔRPS ⎕FREPLACE ΔΔTIE,8
[146]  →0
[147] ΔΔL18:ΔΔARPS ⎕FREPLACE ΔΔTIE,10
    ∇
```

```
                                          [WSID: MULTIFLO]
      ∇ MAT←INDS INDEX FP;CHR;COL;COLS;DATA;DEL;DISP;F;FLD;
        FLDS;FT;INCR;IND;NFLD;NREC;NSET;S;SDISP;SET;SETS;SHAPE
        ;TIE;UNQ;W;WID;⎕IO
[1]   ⍝ Returns data from fields 11↓FP for records
[2]   ⍝ identified by file indices matrix INDS.
[3]   ⍝ First row of INDS is set number (origin 1);
[4]   ⍝ second row is index (origin 1) within set.
[5]   ⍝ INDS may be of any dimension as long as its
[6]   ⍝ first coordinate is 2.  Result has same number
[7]   ⍝ of columns as fields 11↓FP have columns.
[8]   ⍝ Leading shape of result is 1↓⍴INDS.
[9]     ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
[10]    ⎕ERROR((11>⍴FP)∨2≠1↑⍴INDS)/'LENGTH ERROR'
[11]    ⎕IO←1
[12]    NFLD←⍴FLDS←11↓FP
[13]    INCR←FP[3]
[14]    DEL←|FP[11]
[15]    ⎕ERROR((∧/FLDS∊⍳INCR)≤DEL∊FLDS)/'INVALID FIELD NUMBER'
[16]    TIE←FP[2]
[17]    FT←(⎕FREAD TIE,4)[;FLDS]
[18]  ⍝ Width (no. columns) of specified fields:
[19]    WID←FT[1;]
[20]    ⎕ERROR(0∊WID)/'INACTIVE FIELD'
[21]  ⍝ Fields must be all character or all numeric:
[22]    CHR←2=1↑FT[2;]
[23]    ⎕ERROR(CHR∨.≠2=FT[2;])/'DOMAIN ERROR'
[24]  ⍝ Shape of result (excluding columns):
[25]    SHAPE←1↓⍴INDS
[26]    COLS←+/WID
[27]  ⍝ Break apart indices:
[28]    NREC←⍴SETS←, 1 0 ⌿INDS
[29]    INDS←, 0 1 ⌿INDS
[30]  ⍝ Construct all-zero or all-blank result:
[31]    →CHR⍴L1
[32]    MAT←(NREC,COLS)⍴0
[33]    →L2
[34]  L1:MAT←(NREC,COLS)⍴' '
[35]  ⍝ Exit if no indices or no fields:
[36]  L2:→(×NREC×COLS)↓L6
[37]    DISP←FP[4]
[38]  ⍝ Determine distinct set numbers (deleting ¯1s):
[39]    UNQ←SETS[⍋SETS]
[40]    NSET←⍴UNQ←(UNQ≠¯1↓¯1,UNQ)/UNQ
[41]  ⍝ Last set index:
[42]    S←0
[43]  ⍝ Loop by distinct set:
[44]  L3:→(S≥NSET)⍴L6
[45]  ⍝ Current set index and number:
[46]    S←S+1
[47]    SET←UNQ[S]
[48]  ⍝ Displacement (no. components) before this set:
[49]    SDISP←DISP+INCR×SET+¯1
```

```
    ∇ INDEX (continued)
   ⍝ Indices of INDS to retrieve in this set:
    IND←(SET=SETS)/⍳NREC
   ⍝ Last field index, and columns inserted so far:
    F←COL←0
   ⍝ Loop by specified field:
   L4:→(F≥NFLD)⍴L3
   ⍝ Current field index, number and width:
    F←F+1
    FLD←FLDS[F]
    W←WID[F]
   ⍝ Read data:
    DATA←⎕FREAD TIE,FLD+SDISP
   ⍝ Branch if matrix field:
    →(W>1)⍴L5
   ⍝ Insert vector of data:
    COL←COL+1
    MAT[IND;COL]←DATA[INDS[IND]]
    →L4
   ⍝ Insert matrix of data:
   L5:MAT[IND;COL+⍳W]←DATA[INDS[IND];]
    COL←COL+W
    →L4
   ⍝ Exit if result has correct shape already:
   L6:→(1=⍴SHAPE)⍴0
    MAT←(SHAPE,COLS)⍴MAT
    ∇
```

```
                                   [WSID: MULTIFLO]
    ∇ INDS INDEXWS FP;DATA;DEL;DISP;F;FLD;FLDS;FT;INCR;IND;
      LAB;NFLD;NREC;NSET;S;SDISP;SET;SETS;SHAPE;T;TIE;UNQ;W;
      WID;⎕IO
   ⍝ Retrieves data from fields 11↓FP for records
   ⍝ identified by file indices matrix INDS.
   ⍝ First row of INDS is set number (origin 1);
   ⍝ second row is index (origin 1) within set.
   ⍝ INDS may be of any dimension as long as its
   ⍝ first coordinate is 2.  The retrieved data are
   ⍝ assigned to global variables named Fn where n
   ⍝ is the number of the field retrieved (e.g. F3
   ⍝ and F7 for 11↓FP of 3 7).  Global variables
   ⍝ have same number of columns as corresponding
   ⍝ fields.  Leading shape is 1↓⍴INDS.
    ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
    ⎕ERROR((11>⍴FP)∨2≠1↑⍴INDS)/'LENGTH ERROR'
    ⎕IO←1
    NFLD←⍴FLDS←11↓FP
    INCR←FP[3]
    DEL←|FP[11]
    ⎕ERROR((∧/FLDS∊⍳INCR)≤DEL∊FLDS)/'INVALID FIELD NUMBER'
    TIE←FP[2]
    FT←(⎕FREAD TIE,4)[;FLDS]
```

```
    ∇ INDEXWS (continued)
[21] ⍝ Width (no. columns) of specified fields:
[22]  WID←FT[1;]
[23]  ⎕ERROR(0∊WID)/'INACTIVE FIELD'
[24] ⍝ Shape of result (excluding columns):
[25]  SHAPE←1↓ρINDS
[26] ⍝ Break apart indices:
[27]  NREC←ρSETS←, 1 0 ⌿INDS
[28]  INDS←, 0 1 ⌿INDS
[29] ⍝ Construct all-zero or all-blank globals.
[30] ⍝ Label vector needed below based upon field
[31] ⍝ rank and type (nvec, nmat, cvec, cmat):
[32]  LAB←(L2,L3,L4,L5)[(2⌊WID)+2×2=FT[2;]]
[33] ⍝ Last field index:
[34]  F←0
[35] ⍝ Loop by specified field:
[36] L1:→(F≥NFLD)ρL7
[37] ⍝ Current field index and number:
[38]  F←F+1
[39]  FLD←FLDS[F]
[40] ⍝ Branch based upon type and width:
[41]  →LAB[F]
[42] L2:DATA←NRECρ0
[43]  →L6
[44] L3:DATA←(NREC,WID[F])ρ0
[45]  →L6
[46] L4:DATA←NRECρ' '
[47]  →L6
[48] L5:DATA←(NREC,WID[F])ρ' '
[49] L6:⍎'F',(⍕FLD),'←DATA'
[50]  →L1
[51] ⍝ Exit if no indices or no fields:
[52] L7:→(×NREC×NFLD)↓L11
[53]  DISP←FP[4]
[54] ⍝ Determine distinct set numbers (deleting ¯1s):
[55]  UNQ←SETS[⍋SETS]
[56]  NSET←ρUNQ←(UNQ≠¯1↓¯1,UNQ)/UNQ
[57] ⍝ Last set index:
[58]  S←0
[59] ⍝ Loop by distinct set:
[60] L8:→(S≥NSET)ρL11
[61] ⍝ Current set index and number:
[62]  S←S+1
[63]  SET←UNQ[S]
[64] ⍝ Displacement (no. components) before this set:
[65]  SDISP←DISP+INCR×SET+¯1
[66] ⍝ Indices of INDS to retrieve in this set:
[67]  IND←(SET=SETS)/⍳NREC
[68] ⍝ Last field index:
[69]  F←0
[70] ⍝ Loop by specified field:
[71] L9:→(F≥NFLD)ρL8
[72] ⍝ Current field index, number and width:
[73]  F←F+1
```

```
    ∇ INDEXWS (continued)
[74]   FLD←FLDS[F]
[75]   W←WID[F]
[76]  ⍝ Read data:
[77]   DATA←⎕FREAD TIE,FLD+SDISP
[78]  ⍝ Branch if matrix field:
[79]   →(W>1)⍴L10
[80]  ⍝ Insert vector of data:
[81]   ⍎'F',(⍕FLD),'[IND]←DATA[INDS[IND]]'
[82]   →L9
[83]  ⍝ Insert matrix of data:
[84]  L10:⍎'F',(⍕FLD),'[IND;]←DATA[INDS[IND];]'
[85]   →L9
[86]  ⍝ Exit if globals have correct shape already:
[87]  L11:→(1=⍴SHAPE)⍴0
[88]  ⍝ Last field index:
[89]   F←0
[90]  ⍝ Loop by specified field:
[91]  L12:→(F≥NFLD)⍴0
[92]  ⍝ Current field index and number:
[93]   F←F+1
[94]   FLD←FLDS[F]
[95]  ⍝ Reshape globals Fn to conform with shape of indices:
[96]   DATA←⍎T←'F',⍕FLD
[97]   DATA←(SHAPE,1↓⍴DATA)⍴DATA
[98]   ⍎T,'←DATA'
[99]   →L12
    ∇
```

[WSID: MULTIFLO]

```
    ∇ ΔΔMAT←ΔΔEXPR SELECT ΔΔFP;ΔΔACTIVE;ΔΔALL;ΔΔARPS;ΔΔBIT;
      ΔΔCHR;ΔΔCOL;ΔΔCOLS;ΔΔDATA;ΔΔDEL;ΔΔDISP;ΔΔF;ΔΔFILL;
      ΔΔFLD;ΔΔFLDS;ΔΔFNAM;ΔΔFT;ΔΔINCR;ΔΔIND;ΔΔINDS;ΔΔLAB;
      ΔΔLV;ΔΔNDEL;ΔΔNFLD;ΔΔNSET;ΔΔNSFLD;ΔΔNUM;ΔΔROWS;ΔΔRPS;
      ΔΔS;ΔΔSDISP;ΔΔSEL;ΔΔSET;ΔΔSETS;ΔΔSFLDS;ΔΔT;ΔΔTIE;ΔΔW;
      ΔΔWID;⎕IO
[1]   ⍝ Loops through active sets doing the following:
[2]   ⍝ reads the fields specified in (11+(11↓FP)⍳0)↓FP
[3]   ⍝ (calling them F5, F9, etc. for fields 5, 9, etc.),
[4]   ⍝ executes the character vector EXPR, and returns
[5]   ⍝ data from fields (¯1+(11↓FP)⍳0)↑11↓FP for the
[6]   ⍝ records of that set which correspond to 1s in the
[7]   ⍝ resulting bit vector.  Result has one row per
[8]   ⍝ record found and same number of columns as the
[9]   ⍝ latter set of fields has columns.  If EXPR and
[10]  ⍝ selection fields are empty, all records are
[11]  ⍝ selected.  Note that EXPR is executed in origin 1;
[12]  ⍝ e.g. 'F3[;2]' always refers to 2nd column.
[13]   ⎕ERROR((1≠⍴⍴ΔΔFP)∨1<⍴⍴ΔΔEXPR)/'RANK ERROR'
[14]   ⎕ERROR(0=1↑0⍴ΔΔEXPR)/'DOMAIN ERROR'
[15]   ⎕IO←1
[16]  ⍝ Extract 2 sets of fields from FP:
[17]   ΔΔT←11+(11↓ΔΔFP)⍳0
```

```
     ∇ SELECT (continued)
[18]   ∆∆NFLD←ρ∆∆FLDS←11↓(∆∆T+¯1)ρ∆∆FP
[19]   ∆∆ALL←0=∆∆NSFLD←ρ∆∆SFLDS←∆∆T↓∆∆FP
[20]   ⎕ERROR((∆∆ALL=∆∆EXPR∨.≠' ')∨11>ρ∆∆FP)/'LENGTH ERROR'
[21]   ∆∆INCR←∆∆FP[3]
[22]   ∆∆DEL←|∆∆FP[11]
[23]   ∆∆T←∆∆SFLDS,∆∆FLDS
[24]   ⎕ERROR((∧/∆∆T∊⍳∆∆INCR)≤∆∆DEL∊∆∆T)/'INVALID FIELD
       NUMBER'
[25]   ∆∆TIE←∆∆FP[2]
[26]   ∆∆FT←⎕FREAD ∆∆TIE,4
[27]  ⍝ Width (no. columns) of fields:
[28]   ∆∆WID←∆∆FT[1;]
[29]   ⎕ERROR(0∊∆∆WID[∆∆T])/'INACTIVE FIELD'
[30]  ⍝ Fields must be all character or all numeric:
[31]   ∆∆T←2=∆∆FT[2;∆∆FLDS]
[32]   ∆∆CHR←1↑∆∆T
[33]   ⎕ERROR(∆∆CHR∨.≠∆∆T)/'DOMAIN ERROR'
[34]  ⍝ Columns in result:
[35]   ∆∆COLS←+/∆∆WID[∆∆FLDS]
[36]  ⍝ Construct empty result:
[37]   →∆∆CHRρ∆∆L1
[38]   ∆∆MAT←(0,∆∆COLS)ρ∆∆FILL←0
[39]   →∆∆L2
[40]  ∆∆L1:∆∆MAT←(0,∆∆COLS)ρ∆∆FILL←' '
[41]  ⍝ Exit if no records:
[42]  ∆∆L2:→(×∆∆FP[10])↓0
[43]  ⍝ Field names (e.g. 'F5 F9') to be erased below:
[44]   ∆∆FNAM←⍕(∆∆NSFLD,1)ρ∆∆SFLDS
[45]   ∆∆FNAM←'F',(+/' '=∆∆FNAM)⌽∆∆FNAM
[46]   ∆∆DISP←∆∆FP[4]
[47]  ⍝ Label vector needed below based upon field rank
[48]  ⍝ and whether in WS or onfile when needed
[49]  ⍝ (vfile,mfile,vws,mws):
[50]   ∆∆LAB←(∆∆L17,∆∆L18,∆∆L19,∆∆L21)[(2⌊∆∆WID[∆∆FLDS])+2×
       ∆∆FLDS∊∆∆SFLDS]
[51]   ∆∆ARPS←∆∆RPS←⎕FREAD ∆∆TIE,8
[52]  ⍝ Branch unless ARPS should be read:
[53]   →(=/∆∆FP[7 10])ρ∆∆L3
[54]   ∆∆ARPS←⎕FREAD ∆∆TIE,10
[55]  ⍝ Consider only nonempty sets:
[56]  ∆∆L3:∆∆SETS←0≠∆∆ARPS
[57]  ⍝ ...and sets specified in <layers>:
[58]   →(×∆∆FP[1])↓∆∆L7
[59]   →(×⎕NC 'layers')↓∆∆L7
[60]  ⍝ Read layer values:
[61]   ∆∆LV←⎕FREAD ∆∆TIE,9
[62]  ⍝ Branch if matrix layers field:
[63]   →(2=ρρ∆∆LV)ρ∆∆L4
[64]   ∆∆SETS←∆∆SETS∧∆∆LV∊layers
[65]   →∆∆L7
[66]  ⍝ Convert <layers> to matrix if not already:
[67]  ∆∆L4:⎕ERROR(∆∆FT[1;∆∆FP[1]]≠¯1↑ρlayers)/'LENGTH ERROR'
[68]   →(2=ρρlayers)ρ∆∆L5
```

```
    ∇ SELECT (continued)
[69]  layers←((×/¯1↓⍴layers),¯1↑⍴layers)⍴layers
[70] ⍝ Branch if numeric matrix field:
[71] ΔΔL5:→(0=1↑0⍴ΔΔLV)⍴ΔΔL6
[72]  ΔΔSETS←ΔΔSETS∧(layers CMIOTA ΔΔLV)≤1⍴⍴layers
[73]  →ΔΔL7
[74] ΔΔL6:ΔΔSETS←ΔΔSETS∧∨/ΔΔLV∧.=⍉layers
[75] ⍝ Convert to indices; erase <layers>:
[76] ΔΔL7:ΔΔSETS←ΔΔSETS/⍳⍴ΔΔSETS
[77]  ΔΔNSET←⍴ΔΔSETS
[78]  ΔΔLV←⎕EX 'layers'
[79] ⍝ Last set index:
[80]  ΔΔS←0
[81] ⍝ Loop by nonempty set:
[82] ΔΔL8:→(ΔΔS≥ΔΔNSET)⍴0
[83] ⍝ Current set index and number:
[84]  ΔΔS←ΔΔS+1
[85]  ΔΔSET←ΔΔSETS[ΔΔS]
[86] ⍝ Displacement (no. components) before this set:
[87]  ΔΔSDISP←ΔΔDISP+ΔΔINCR×ΔΔSET+¯1
[88] ⍝ Branch if deletion field unneeded:
[89]  →(ΔΔNDEL←ΔΔARPS[ΔΔSET]>0)⍴ΔΔL9
[90] ⍝ Read deletion field for set SET:
[91]  ΔΔBIT←⎕FREAD ΔΔTIE,ΔΔDEL+ΔΔSDISP
[92] ⍝ Branch if no selection expression:
[93] ΔΔL9:ΔΔSEL←1
[94]  →ΔΔALL⍴ΔΔL14
[95] ⍝ Are all records in this set active:
[96]  ΔΔACTIVE←ΔΔARPS[ΔΔSET]=ΔΔRPS[ΔΔSET]
[97] ⍝ Last field index:
[98]  ΔΔF←0
[99] ⍝ Loop by field specified:
[100] ΔΔL10:→(ΔΔF≥ΔΔNSFLD)⍴ΔΔL13
[101] ⍝ Current field index and number:
[102]  ΔΔF←ΔΔF+1
[103]  ΔΔFLD←ΔΔSFLDS[ΔΔF]
[104] ⍝ Read field FLD for set SET:
[105]  ΔΔDATA←⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP
[106] ⍝ Branch if all records in this set active:
[107]  →ΔΔACTIVE⍴ΔΔL12
[108] ⍝ Branch if deletion field needed:
[109]  →ΔΔNDEL↓ΔΔL11
[110] ⍝ Active records are leading records:
[111]  ΔΔDATA←(ΔΔARPS[ΔΔSET],(ΔΔWID[ΔΔFLD]>1)⍴ΔΔWID[ΔΔFLD])⍴
      ΔΔDATA
[112]  →ΔΔL12
[113] ⍝ Apply deletion field:
[114] ΔΔL11:ΔΔDATA←ΔΔBIT⌿ΔΔDATA
[115] ⍝ Assign data to global variable Fn:
[116] ΔΔL12:⍎'F',(⍕ΔΔFLD),'←ΔΔDATA'
[117]  →ΔΔL10
[118] ⍝ Once all fields have been read, execute EXPR:
[119] ΔΔL13:ΔΔSEL←⍎ΔΔEXPR
```

```
    ∇ SELECT (continued)
[120] ⍝ Indices to retrieve (after squeezing deletions):
[121] ∆∆L14:∆∆IND←∆∆SEL/⍳|∆∆ARPS[∆∆SET]
[122] ⍝ Continue to next set if no records selected:
[123]  →(×∆∆NUM←⍴∆∆IND)↓∆∆L23
[124] ⍝ No. records found before this set:
[125]  ∆∆ROWS←1⍴⍴∆∆MAT
[126] ⍝ Expand result for new records:
[127]  ∆∆MAT←∆∆MAT,[1](∆∆NUM,∆∆COLS)⍴∆∆FILL
[128] ⍝ Indices to retrieve (before squeezing deletions):
[129]  ∆∆INDS←∆∆IND
[130] ⍝ Branch if deletion field unneeded:
[131]  →∆∆NDEL⍴∆∆L15
[132] ⍝ Reset INDS, considering deletion field:
[133]  ∆∆INDS←(∆∆BIT/⍳⍴∆∆BIT)[∆∆IND]
[134] ⍝ Last field index and columns inserted so far:
[135] ∆∆L15:∆∆F←∆∆COL←0
[136] ⍝ Loop by field to retrieve:
[137] ∆∆L16:→(∆∆F≥∆∆NFLD)⍴∆∆L23
[138] ⍝ Current field index, number and width:
[139]  ∆∆F←∆∆F+1
[140]  ∆∆FLD←∆∆FLDS[∆∆F]
[141]  ∆∆W←∆∆WID[∆∆FLD]
[142] ⍝ Branch depending on field width and whether
[143] ⍝ already in workspace:
[144]  →∆∆LAB[∆∆F]
[145] ∆∆L17:∆∆DATA←(⎕FREAD ∆∆TIE,∆∆FLD+∆∆SDISP)[∆∆INDS]
[146]  →∆∆L20
[147] ∆∆L18:∆∆DATA←(⎕FREAD ∆∆TIE,∆∆FLD+∆∆SDISP)[∆∆INDS;]
[148]  →∆∆L22
[149] ∆∆L19:∆∆DATA←(⍎'F',⍕∆∆FLD)[∆∆IND]
[150] ∆∆L20:∆∆COL←∆∆COL+1
[151]  ∆∆MAT[∆∆ROWS+⍳∆∆NUM;∆∆COL]←∆∆DATA
[152]  →∆∆L16
[153] ∆∆L21:∆∆DATA←(⍎'F',⍕∆∆FLD)[∆∆IND;]
[154] ∆∆L22:∆∆MAT[∆∆ROWS+⍳∆∆NUM;∆∆COL+⍳∆∆W]←∆∆DATA
[155]  ∆∆COL←∆∆COL+∆∆W
[156]  →∆∆L16
[157] ⍝ Erase field variables (e.g. F5, F9,...):
[158] ∆∆L23:∆∆T←⎕EX ∆∆FNAM
[159]  →∆∆L8
    ∇
```

```
                                          [WSID: MULTIFLO]
     ∇ ΔΔEXPR SELECTWS ΔΔFP;ΔΔACTIVE;ΔΔALL;ΔΔARPS;ΔΔBIT;ΔΔCOL
       ;ΔΔCOLS;ΔΔDATA;ΔΔDEL;ΔΔDISP;ΔΔF;ΔΔFLD;ΔΔFLDS;ΔΔFNAM;
       ΔΔFT;ΔΔINCR;ΔΔIND;ΔΔINDS;ΔΔLAB;ΔΔLAB2;ΔΔLV;ΔΔM1;ΔΔM2;
       ΔΔM3;ΔΔM4;ΔΔNDEL;ΔΔNFLD;ΔΔNSET;ΔΔNSFLD;ΔΔNUM;ΔΔROWS;
       ΔΔRPS;ΔΔS;ΔΔSDISP;ΔΔSEL;ΔΔSET;ΔΔSETS;ΔΔSFLDS;ΔΔSTART;
       ΔΔT;ΔΔTIE;ΔΔTYPES;ΔΔW;ΔΔWID;⎕IO
      ⍝ Loops through active sets doing the following:
      ⍝ reads the fields specified in (11+(11↓FP)⍳0)↓FP
      ⍝ (calling them F5, F9, etc. for fields 5, 9, etc.),
      ⍝ executes the character vector EXPR, and retrieves
      ⍝ data from fields (¯1+(11↓FP)⍳0)↑11↓FP for the
      ⍝ records of that set which correspond to 1s in the
      ⍝ resulting bit vector.  The retrieved data are
      ⍝ assigned to global variables named Fn where n is
      ⍝ the number of the field retrieved (e.g. F3 and F7
      ⍝ if 3 7 are the numbers of the latter set of
      ⍝ fields).  Global variables have same number of
      ⍝ columns as corresponding fields.  If EXPR and
      ⍝ selection fields are empty, all records are
      ⍝ selected.  Note that EXPR is executed in origin 1;
      ⍝ e.g. 'F3[;2]' always refers to 2nd column.
       ⎕ERROR((1≠⍴⍴ΔΔFP)∨1<⍴⍴ΔΔEXPR)/'RANK ERROR'
       ⎕ERROR(0=1↑0⍴ΔΔEXPR)/'DOMAIN ERROR'
       ⎕IO←1
      ⍝ Extract 2 sets of fields from FP:
       ΔΔT←11+(11↓ΔΔFP)⍳0
       ΔΔNFLD←⍴ΔΔFLDS←11↓(ΔΔT+¯1)⍴ΔΔFP
       ΔΔALL←0=ΔΔNSFLD←⍴ΔΔSFLDS←ΔΔT↓ΔΔFP
       ⎕ERROR((ΔΔALL=ΔΔEXPR∨.≠' ')∨11>⍴ΔΔFP)/'LENGTH ERROR'
       ΔΔINCR←ΔΔFP[3]
       ΔΔDEL←|ΔΔFP[11]
       ΔΔT←ΔΔSFLDS,ΔΔFLDS
       ⎕ERROR((∧/ΔΔT∊⍳ΔΔINCR)≤ΔΔDEL∊ΔΔT)/'INVALID FIELD
       NUMBER'
       ΔΔTIE←ΔΔFP[2]
       ΔΔFT←⎕FREAD ΔΔTIE,4
      ⍝ Width (no. columns) of fields:
       ΔΔWID←ΔΔFT[1;]
       ⎕ERROR(0∊ΔΔWID[ΔΔT])/'INACTIVE FIELD'
      ⍝ Exit if no fields:
       →(×ΔΔNFLD)↓ΔΔL36
      ⍝ Datatypes (1,2,3,4) of fields to be retrieved:
       ΔΔTYPES←ΔΔFT[2;ΔΔFLDS]
      ⍝ No. preceding cols. for each fld within its datatype:
       ΔΔT←(⍳4)∘.=ΔΔTYPES
       ΔΔSTART←+⌿ΔΔT×(⍴ΔΔT)↑0,ΔΔS←+\ΔΔT×(⍴ΔΔT)⍴ΔΔWID[ΔΔFLDS]
      ⍝ No. columns of each datatype:
       ΔΔCOLS←, 4 ¯1 ↑ΔΔS
      ⍝ Construct empty result matrices (one per datatype):
       ΔΔM1←(0,ΔΔCOLS[1])⍴0
       ΔΔM2←(0,ΔΔCOLS[2])⍴''
       ΔΔM3←(0,ΔΔCOLS[3])⍴0
       ΔΔM4←(0,ΔΔCOLS[4])⍴0
```

```
    ∇ SELECTWS (continued)
[47] ⍝ Exit if no records:
[48]  →(×ΔΔFP[10])↓19
[49] ⍝ Field names (e.g. 'F5 F9') to be erased below:
[50]  ΔΔFNAM←⍕(ΔΔNSFLD,1)⍴ΔΔSFLDS
[51]  ΔΔFNAM←'F',(+/' '=ΔΔFNAM)⌽ΔΔFNAM
[52]  ΔΔDISP←ΔΔFP[4]
[53] ⍝ Label vector needed below based upon field rank
[54] ⍝ and whether in WS or onfile when needed
[55] ⍝ (vfile,mfile,vws,mws):
[56]  ΔΔLAB←(ΔΔL15,ΔΔL16,ΔΔL17,ΔΔL19)[(2⌊ΔΔWID[ΔΔFLDS])+2×
      ΔΔFLDS∊ΔΔSFLDS]
[57] ⍝ Label vector based upon datatype:
[58]  ΔΔLAB2←(ΔΔL22,ΔΔL23,ΔΔL24,ΔΔL25)[ΔΔTYPES]
[59]  ΔΔARPS←ΔΔRPS←⎕FREAD ΔΔTIE,8
[60] ⍝ Branch unless ARPS should be read:
[61]  →(=/ΔΔFP[7 10])⍴ΔΔL1
[62]  ΔΔARPS←⎕FREAD ΔΔTIE,10
[63] ⍝ Consider only nonempty sets:
[64] ΔΔL1:ΔΔSETS←0≠ΔΔARPS
[65] ⍝ ...and sets specified in <layers>:
[66]  →(×ΔΔFP[1])↓ΔΔL5
[67]  →(×⎕NC 'layers')↓ΔΔL5
[68] ⍝ Read layer values:
[69]  ΔΔLV←⎕FREAD ΔΔTIE,9
[70] ⍝ Branch if matrix layers field:
[71]  →(2=⍴⍴ΔΔLV)⍴ΔΔL2
[72]  ΔΔSETS←ΔΔSETS∧ΔΔLV∊layers
[73]  →ΔΔL5
[74] ⍝ Convert <layers> to matrix if not already:
[75] ΔΔL2:⎕ERROR(ΔΔFT[1;ΔΔFP[1]]≠¯1↑⍴layers)/'LENGTH ERROR'
[76]  →(2=⍴⍴layers)⍴ΔΔL3
[77]  layers←((×/¯1↓⍴layers),¯1↑⍴layers)⍴layers
[78] ⍝ Branch if numeric matrix field:
[79] ΔΔL3:→(0=1↑0⍴ΔΔLV)⍴ΔΔL4
[80]  ΔΔSETS←ΔΔSETS∧(layers CMIOTA ΔΔLV)≤1⍴⍴layers
[81]  →ΔΔL5
[82] ΔΔL4:ΔΔSETS←ΔΔSETS∧∨/ΔΔLV∧.=⍉layers
[83] ⍝ Convert to indices; erase <layers>:
[84] ΔΔL5:ΔΔSETS←ΔΔSETS/⍳⍴ΔΔSETS
[85]  ΔΔNSET←⍴ΔΔSETS
[86]  ΔΔLV←⎕EX 'layers'
[87] ⍝ Last set index:
[88]  ΔΔS←0
[89] ⍝ Loop by nonempty set:
[90] ΔΔL6:→(ΔΔS≥ΔΔNSET)⍴ΔΔL27
[91] ⍝ Current set index and number:
[92]  ΔΔS←ΔΔS+1
[93]  ΔΔSET←ΔΔSETS[ΔΔS]
[94] ⍝ Displacement (no. components) before this set:
[95]  ΔΔSDISP←ΔΔDISP+ΔΔINCR×ΔΔSET+¯1
[96] ⍝ Branch if deletion field unneeded:
[97]  →(ΔΔNDEL←ΔΔARPS[ΔΔSET]>0)⍴ΔΔL7
```

```
    ∇ SELECTWS (continued)
[98] ⍝ Read deletion field for set SET:
[99]  ∆∆BIT←⎕FREAD ∆∆TIE,∆∆DEL+∆∆SDISP
[100] ⍝ Branch if no selection expression:
[101] ∆∆L7:∆∆SEL←1
[102]  →∆∆ALL⍴∆∆L12
[103] ⍝ Are all records in this set active:
[104]  ∆∆ACTIVE←∆∆ARPS[∆∆SET]=∆∆RPS[∆∆SET]
[105] ⍝ Last field index:
[106]  ∆∆F←0
[107] ⍝ Loop by field specified:
[108] ∆∆L8:→(∆∆F≥∆∆NSFLD)⍴∆∆L11
[109] ⍝ Current field index and number:
[110]  ∆∆F←∆∆F+1
[111]  ∆∆FLD←∆∆SFLDS[∆∆F]
[112] ⍝ Read field FLD for set SET:
[113]  ∆∆DATA←⎕FREAD ∆∆TIE,∆∆FLD+∆∆SDISP
[114] ⍝ Branch if all records in this set active:
[115]  →∆∆ACTIVE⍴∆∆L10
[116] ⍝ Branch if deletion field needed:
[117]  →∆∆NDEL↓∆∆L9
[118] ⍝ Active records are leading records:
[119]  ∆∆DATA←(∆∆ARPS[∆∆SET],(∆∆WID[∆∆FLD]>1)⍴∆∆WID[∆∆FLD])⍴
      ∆∆DATA
[120]  →∆∆L10
[121] ⍝ Apply deletion field:
[122] ∆∆L9:∆∆DATA←∆∆BIT⌿∆∆DATA
[123] ⍝ Assign data to global variable Fn:
[124] ∆∆L10:⍎'F',(⍕∆∆FLD),'←∆∆DATA'
[125]  →∆∆L8
[126] ⍝ Once all fields have been read, execute EXPR:
[127] ∆∆L11:∆∆SEL←⍎∆∆EXPR
[128] ⍝ Indices to retrieve (after squeezing deletions):
[129] ∆∆L12:∆∆IND←∆∆SEL/⍳|∆∆ARPS[∆∆SET]
[130] ⍝ Continue to next set if no records selected:
[131]  →(×∆∆NUM←⍴∆∆IND)↓∆∆L26
[132] ⍝ No. records found before this set:
[133]  ∆∆ROWS←1⍴⍴∆∆M1
[134] ⍝ Expand result matrices for new records:
[135]  ∆∆M1←∆∆M1,[1](∆∆NUM,∆∆COLS[1])⍴0
[136]  ∆∆M2←∆∆M2,[1](∆∆NUM,∆∆COLS[2])⍴' '
[137]  ∆∆M3←∆∆M3,[1](∆∆NUM,∆∆COLS[3])⍴0
[138]  ∆∆M4←∆∆M4,[1](∆∆NUM,∆∆COLS[4])⍴0
[139] ⍝ Indices to retrieve (before squeezing deletions):
[140]  ∆∆INDS←∆∆IND
[141] ⍝ Branch if deletion field unneeded:
[142]  →∆∆NDEL⍴∆∆L13
[143] ⍝ Reset INDS, considering deletion field:
[144]  ∆∆INDS←(∆∆BIT/⍳⍴∆∆BIT)[∆∆IND]
[145] ⍝ Last field index:
[146] ∆∆L13:∆∆F←0
[147] ⍝ Loop by field to retrieve:
[148] ∆∆L14:→(∆∆F≥∆∆NFLD)⍴∆∆L26
```

```
    ∇ SELECTWS (continued)
[149] ⍝ Current field index, number and width:
[150]  ΔΔF←ΔΔF+1
[151]  ΔΔFLD←ΔΔFLDS[ΔΔF]
[152]  ΔΔW←ΔΔWID[ΔΔFLD]
[153] ⍝ Branch depending on field width and whether
[154] ⍝ already in workspace:
[155]  →ΔΔLAB[ΔΔF]
[156] ΔΔL15:ΔΔDATA←(⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP)[ΔΔINDS]
[157]  →ΔΔL18
[158] ΔΔL16:ΔΔDATA←(⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP)[ΔΔINDS;]
[159]  →ΔΔL20
[160] ΔΔL17:ΔΔDATA←(⍎'F',⍕ΔΔFLD)[ΔΔIND]
[161] ΔΔL18:ΔΔCOL←ΔΔSTART[ΔΔF]+1
[162]  →ΔΔL21
[163] ΔΔL19:ΔΔDATA←(⍎'F',⍕ΔΔFLD)[ΔΔIND;]
[164] ΔΔL20:ΔΔCOL←ΔΔSTART[ΔΔF]+⍳ΔΔW
[165] ⍝ Branch based upon datatype:
[166] ΔΔL21:→ΔΔLAB2[ΔΔF]
[167] ΔΔL22:ΔΔM1[ΔΔROWS+⍳ΔΔNUM;ΔΔCOL]←ΔΔDATA
[168]  →ΔΔL14
[169] ΔΔL23:ΔΔM2[ΔΔROWS+⍳ΔΔNUM;ΔΔCOL]←ΔΔDATA
[170]  →ΔΔL14
[171] ΔΔL24:ΔΔM3[ΔΔROWS+⍳ΔΔNUM;ΔΔCOL]←ΔΔDATA
[172]  →ΔΔL14
[173] ΔΔL25:ΔΔM4[ΔΔROWS+⍳ΔΔNUM;ΔΔCOL]←ΔΔDATA
[174]  →ΔΔL14
[175] ⍝ Erase field variables (e.g. F5, F9,...):
[176] ΔΔL26:ΔΔT←⎕EX ΔΔFNAM
[177]  →ΔΔL6
[178] ⍝ Label vector by datatype needed below:
[179] ΔΔL27:ΔΔLAB←(ΔΔL31,ΔΔL32,ΔΔL33,ΔΔL34)[ΔΔTYPES]
[180] ⍝ Last field index:
[181]  ΔΔF←0
[182] ⍝ Loop by field to retrieve:
[183] ΔΔL28:→(ΔΔF≥ΔΔNFLD)⍴0
[184] ⍝ Current field index, number and width:
[185]  ΔΔF←ΔΔF+1
[186]  ΔΔFLD←ΔΔFLDS[ΔΔF]
[187]  ΔΔW←ΔΔWID[ΔΔFLD]
[188] ⍝ Branch if matrix field:
[189]  →(ΔΔW>1)⍴ΔΔL29
[190]  ΔΔCOL←ΔΔSTART[ΔΔF]+1
[191]  →ΔΔL30
[192] ΔΔL29:ΔΔCOL←ΔΔSTART[ΔΔF]+⍳ΔΔW
[193] ⍝ Branch based upon datatype:
[194] ΔΔL30:→ΔΔLAB[ΔΔF]
[195] ΔΔL31:ΔΔDATA←ΔΔM1[;ΔΔCOL]
[196]  →ΔΔL35
[197] ΔΔL32:ΔΔDATA←ΔΔM2[;ΔΔCOL]
[198]  →ΔΔL35
[199] ΔΔL33:ΔΔDATA←ΔΔM3[;ΔΔCOL]
[200]  →ΔΔL35
[201] ΔΔL34:ΔΔDATA←ΔΔM4[;ΔΔCOL]
```

```
    ∇ SELECTWS (continued)
[202] ⍝ Assign to Fn vars.:
[203] ΔΔL35:⍎'F',(⍕ΔΔFLD),'←ΔΔDATA'
[204]  →ΔΔL28
[205] ΔΔL36:ΔΔS←⎕EX 'layers'
    ∇
```

[WSID: MULTIFLO]

```
    ∇ R←A ASSIGN B
[1]  ⍝ Used as:  rinds INDEXA (FP,flds) ASSIGN mat
[2]   R←A
[3]   assign←B
    ∇
```

[WSID: MULTIFLO]

```
    ∇ NFP←INDS INDEXA FP;A;AFLD;ARPS;BAD;BIT;BLK;CMP;COL;
      COLS;DATA;DCMP;DEL;DISP;F;FILED;FILL;FLD;FLDS;FREC;FT;
      GOOD;I;IINDS;INCR;IND;LAY;LAYER;LEAD;LSETS;LV;LVAR;M;
      MIN;N;NF;NFLD;NR;NREC;NSET;RPS;S;SD;SDISP;SET;SETS;
      SHAPE;SIND;SINGLE;T;TIE;UNQ;VAR;VEC;W;WID;⎕IO
[1]  ⍝ Used as:  FP←rinds INDEXA (FP,flds) ASSIGN mat
[2]  ⍝ Inserts data from global matrix <assign> into
[3]  ⍝ fields 11↓FP for records identified by file
[4]  ⍝ indices matrix INDS. First row of INDS is set
[5]  ⍝ number (origin 1); second row is index (origin 1)
[6]  ⍝ within set. INDS may be of any dimension as long as
[7]  ⍝ its first coordinate is 2.  <assign> has same number
[8]  ⍝ of columns as fields 11↓FP have columns.
[9]  ⍝ Leading shape of <assign> is 1↓⍴INDS.  <assign> is
[10] ⍝ erased upon completion.
[11]  ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
[12]  ⎕ERROR((11>⍴FP)∨2≠1↑⍴INDS)/'LENGTH ERROR'
[13]  ⎕IO←1
[14]  NFLD←⍴FLDS←11↓FP
[15]  NFP←11⍴FP
[16]  INCR←FP[3]
[17]  DEL←|FP[11]
[18]  FILL←FP[11]<0
[19]  ⎕ERROR((∧/FLDS∊⍳INCR)≤DEL∊FLDS)/'INVALID FIELD NUMBER'
[20]  TIE←FP[2]
[21]  FT←⎕FREAD TIE,4
[22] ⍝ Width (no. columns) of specified fields:
[23]  WID←FT[1;]
[24]  COLS←+/W←WID[FLDS]
[25]  ⎕ERROR(0∊W)/'INACTIVE FIELD'
[26] ⍝ Fields must be all character or all numeric:
[27]  S←2=1↑T←FT[2;FLDS]
[28]  ⎕ERROR(S∨.≠2=T)/'DOMAIN ERROR'
[29] ⍝ Break apart indices:
[30]  SHAPE←1↓⍴INDS
[31]  NREC←⍴SETS←, 1 0 ⌿INDS
[32]  INDS←, 0 1 ⌿INDS
```

```
    ∇ INDEXA (continued)
[33] ⍝ Singleton data?
[34]  SINGLE←1∧.=⍴assign
[35]  →SINGLE⍴L3
[36] ⍝ Branch unless singleton (vector) record:
[37]  →((COLS≠1)∧1∧.=¯1↓⍴assign)↓L1
[38]  ⎕ERROR(COLS≠¯1↑⍴assign)/'LENGTH ERROR'
[39]  →L2
[40] L1:⎕ERROR((⍴SHAPE)∧.≠(⍴⍴assign)+¯1,(COLS=1)⍴0)/'RANK
      ERROR'
[41]  S←(1+⍴SHAPE)⍴(⍴assign),1
[42]  ⎕ERROR(S∨.≠SHAPE,COLS)/'LENGTH ERROR'
[43]  →(2=⍴⍴assign)⍴L3
[44] L2:assign←(NREC,COLS)⍴assign
[45] ⍝ Exit if no records or no fields:
[46] L3:→(×NREC×NFLD)↓L43
[47]  LAY←FP[1]
[48]  DISP←FP[4]
[49]  BLK←FP[5]
[50] ⍝ Read LV, RPS, ARPS if layer fld being assigned:
[51]  →(LAY∊FLDS)↓L4
[52]  LV←⎕FREAD TIE,9
[53]  RPS←ARPS←⎕FREAD TIE,8
[54]  →(=/FP[7 10])⍴L4
[55]  ARPS←⎕FREAD TIE,10
[56] ⍝ Determine distinct set numbers (deleting ¯1s):
[57] L4:UNQ←SETS[⍋SETS]
[58]  NSET←⍴UNQ←(UNQ≠¯1↓¯1,UNQ)/UNQ
[59] ⍝ Last set index:
[60]  SIND←0
[61] ⍝ Loop by distinct set:
[62] L5:→(SIND≥NSET)⍴L43
[63] ⍝ Current set index and number:
[64]  SIND←SIND+1
[65]  SET←UNQ[SIND]
[66] ⍝ Displacement (no. components) before this set:
[67]  SDISP←DISP+INCR×SET+¯1
[68] ⍝ Indices and elts of INDS to retrieve in this set:
[69]  IND←(SET=SETS)/⍳NREC
[70]  I←INDS[IND]
[71] ⍝ Elts of INDS of recs whose layer value changes:
[72]  BAD←⍳0
[73] ⍝ Last field index, and columns filed so far:
[74]  F←COL←0
[75] ⍝ Loop by specified field:
[76] L6:→(F≥NFLD)⍴L13
[77] ⍝ Current field index, number and width:
[78]  F←F+1
[79]  FLD←FLDS[F]
[80] ⍝ Is this the layer field being changed?
[81]  LAYER←LAY=FLD
[82]  W←WID[FLD]
[83] ⍝ Read data:
[84]  DATA←⎕FREAD CMP←TIE,FLD+SDISP
```

```
    ∇ INDEXA (continued)
[85] ⍝ Branch if matrix field:
[86]  →(W>1)⍴L9
[87] ⍝ Insert vector of data.
[88] ⍝ Branch if singleton:
[89]  →SINGLE⍴L7
[90]  COL←COL+1
[91]  DATA[I]←assign[IND;COL]
[92]  →L8
[93] L7:DATA[I]←assign
[94] ⍝ Branch unless layer field:
[95] L8:→LAYER↓L12
[96] ⍝ Flag recs whose layer val has changed:
[97]  BAD←(LV[SET]≠DATA[I])/I
[98]  →L12
[99] ⍝ Insert matrix of data.
[100] ⍝ Branch if singleton:
[101] L9:→SINGLE⍴L10
[102]  DATA[I;]←assign[IND;COL+⍳W]
[103]  COL←COL+W
[104]  →L11
[105] L10:DATA[I;]←assign
[106] ⍝ Branch unless layer field:
[107] L11:→LAYER↓L12
[108] ⍝ Flag recs whose layer val has changed:
[109]  BAD←(DATA[I;]∨.≠LV[SET;])/I
[110] ⍝ Replace data on file:
[111] L12:DATA ⎕FREPLACE CMP
[112]  →L6
[113] ⍝ Next set if layer values unchanged:
[114] L13:→(×⍴BAD)↓L5
[115] ⍝ Indices of active fields:
[116]  AFLD←(×WID)/⍳INCR
[117] ⍝ Exclude deletion field:
[118]  NF←⍴AFLD←(AFLD≠DEL)/AFLD
[119] ⍝ Flag records filed so far:
[120]  FILED←(⍴BAD)⍴0
[121] ⍝ Look at layer field:
[122]  LVAR←⎕FREAD TIE,LAY+SDISP
[123] ⍝ Branch if vector field:
[124]  VEC←WID[LAY]=1
[125] L14:→VEC⍴L15
[126] ⍝ Flag records with layer of 1st unfiled rec:
[127]  LAYER←LVAR[BAD[FILED⍳0];]
[128]  GOOD←LVAR[BAD;]∧.=LAYER
[129] ⍝ ...and sets with this record:
[130]  LSETS←LV∧.=LAYER
[131]  →L16
[132] L15:LAYER←LVAR[BAD[FILED⍳0]]
[133]  GOOD←LVAR[BAD]=LAYER
[134]  LSETS←LV=LAYER
[135] ⍝ Update FILED; convert to indices:
[136] L16:FILED←FILED∨GOOD
[137]  GOOD←GOOD/BAD
```

```
    ∇ INDEXA (continued)
[138]  NR←⍴GOOD
[139] ⍝ Consider only non-full sets in this layer or
[140] ⍝ empty sets in any layer:
[141]  LSETS←(ARPS=0)∨LSETS∧ARPS≠BLK
[142] ⍝ Convert to indices:
[143]  LSETS←LSETS/⍳⍴LSETS
[144] ⍝ No. records filed so far:
[145]  FREC←0
[146] ⍝ Branch if no slots available in existing sets:
[147] L17:→(BLK≤MIN←⌊/A←|ARPS[LSETS])⍴L32
[148] ⍝ Branch if more than 1 set needed:
[149]  T←NR-FREC
[150]  →(T>BLK-MIN)⍴L18
[151] ⍝ Choose fullest set which will hold all recs:
[152]  S←BLK-A
[153]  S←LSETS[S⍳⌊/(S≥T)/S]
[154]  →L19
[155] ⍝ Choose set with most empty slots:
[156] L18:S←LSETS[A⍳MIN]
[157] ⍝ No. records to be inserted within the set:
[158] L19:M←T⌊RPS[S]-|ARPS[S]
[159] ⍝ No. records to be catenated within the set:
[160]  N←(T-M)⌊BLK-RPS[S]
[161] ⍝ Displacement (no. components) before this set:
[162]  SD←DISP+INCR×S+¯1
[163] ⍝ Branch if no deletion field:
[164]  →(×DEL)↓L20
[165] ⍝ Read deletion field for set S:
[166]  BIT←⎕FREAD DCMP←TIE,DEL+SD
[167] ⍝ Branch if no records to be inserted:
[168]  →(×M)↓L20
[169] ⍝ Indices of available insertion slots:
[170]  IINDS←M⍴(~BIT)/⍳⍴BIT
[171] ⍝ Next field index:
[172] L20:F←1
[173] ⍝ Loop by active field:
[174] L21:→(F>NF)⍴L26
[175] ⍝ Field number:
[176]  FLD←AFLD[F]
[177] ⍝ Field width (no. columns):
[178]  W←WID[FLD]
[179] ⍝ Look at original set:
[180]  VAR←⎕FREAD TIE,FLD+SDISP
[181] ⍝ Read field F for set S:
[182]  DATA←⎕FREAD CMP←TIE,FLD+SD
[183] ⍝ Branch if a matrix field:
[184]  →(W>1)⍴L23
[185] ⍝ Branch if no records to insert:
[186]  →(×M)↓L22
[187]  DATA[IINDS]←VAR[GOOD[FREC+⍳M]]
[188] ⍝ Branch if no records to catenate:
[189]  →(×N)↓L25
[190] L22:DATA←DATA,VAR[GOOD[(FREC+M)+⍳N]]
```

```
    ∇ INDEXA (continued)
[191]  →L25
[192] ⍝ Branch if no records to insert:
[193] L23:→(×M)↓L24
[194]  DATA[IINDS;]←VAR[GOOD[FREC+⍳M];]
[195] ⍝ Branch if no records to catenate:
[196]  →(×N)↓L25
[197] L24:DATA←DATA,[1]VAR[GOOD[(FREC+M)+⍳N];]
[198] L25:DATA ⎕FREPLACE CMP
[199]  F←F+1
[200]  →L21
[201] ⍝ Branch if no deletion field:
[202] L26:LEAD←1
[203]  →(×DEL)↓L29
[204] ⍝ Branch if no records to insert:
[205]  →(×M)↓L27
[206] ⍝ Turn active record bits on:
[207]  BIT[IINDS]←1
[208]  LEAD←∧/BIT=∧\BIT
[209] ⍝ Branch if no records to catenate:
[210]  →(×N)↓L28
[211] L27:BIT←BIT,N⍴1
[212] L28:BIT ⎕FREPLACE DCMP
[213] ⍝ Increment FREC by no. records added to this set:
[214] L29:FREC←FREC+M+N
[215]  RPS[S]←RPS[S]+N
[216]  NFP[9]←NFP[9]+T←0=ARPS[S]
[217] ⍝ Replace layer value if set initially empty:
[218]  →T↓L31
[219] ⍝ Branch if a vector layer field:
[220]  →VEC⍴L30
[221]  LV[S;]←LAYER
[222]  →L31
[223] L30:LV[S]←LAYER
[224] L31:ARPS[S]←(¯1 1)[1+LEAD]×M+N+|ARPS[S]
[225]  NFP[7]←NFP[7]+N
[226] ⍝ Exit if all of data in field vars. filed:
[227]  →(NR=FREC)⍴L40
[228]  →L17
[229] ⍝ No. records to be appended in next set:
[230] L32:N←BLK⌊NR-FREC
[231] ⍝ Next field number:
[232]  F←1
[233] ⍝ Loop by field:
[234] L33:→(F>INCR)⍴L39
[235]  W←WID[F]
[236] ⍝ Branch unless a latent field:
[237]  →(×W)⍴L34
[238]  DATA←⍳0
[239]  →L38
[240] ⍝ Branch unless it's the deletion field:
[241] L34:→(DEL≠F)⍴L35
[242]  DATA←N⍴1
[243]  →L37
```

```
    ∇ INDEXA (continued)
[244] ⍝ Look at original set:
[245] L35:VAR←⎕FREAD TIE,F+SDISP
[246] ⍝ Branch if a matrix field:
[247]  →(W>1)⍴L36
[248]  DATA←VAR[GOOD[FREC+⍳N]]
[249]  →L37
[250] L36:DATA←VAR[GOOD[FREC+⍳N];]
[251] ⍝ Branch unless set must be padded to BLK records:
[252] L37:→FILL↓L38
[253]  DATA←(BLK,1↓⍴DATA)↑DATA
[254] L38:T←DATA ⎕FAPPEND TIE
[255]  F←F+1
[256]  →L33
[257] ⍝ Increment FREC by no. records added to this set:
[258] L39:FREC←FREC+N
[259]  ARPS←ARPS,N
[260]  RPS←RPS,T←N⌈BLK×FILL
[261]  NFP[7]←NFP[7]+T
[262]  NFP[6]←NFP[6]+1
[263]  NFP[9]←NFP[9]+1
[264] ⍝ Catenate layer value:
[265]  LV←LV,[1]LAYER
[266] ⍝ Continue unless all of data in field vars. filed:
[267]  →(NR≠FREC)⍴L32
[268] ⍝ Branch if no data left to file:
[269] L40:→(∧/FILED)↓L14
[270] ⍝
[271] ⍝ Branch unless deletion field exists:
[272]  →(×DEL)↓L41
[273] ⍝ Read deletion field for set SET:
[274]  BIT←⎕FREAD CMP←TIE,DEL+SDISP
[275] ⍝ Turn off specified indices and replace:
[276]  BIT[BAD]←0
[277]  BIT ⎕FREPLACE CMP
[278] ⍝ Compute new no. records:
[279]  N←+/BIT
[280]  M←|RPS[SET]
[281] ⍝ Reset parameters:
[282]  NFP[9]←NFP[9]-N=0
[283]  ARPS[SET]←N×(¯1 1)[1+N=+/∧\BIT]
[284]  →L5
[285] ⍝
[286] ⍝ Turn off specified indices:
[287] L41:M←RPS[SET]
[288]  BIT←M⍴1
[289]  BIT[BAD]←0
[290] ⍝ Compute new no. records:
[291]  N←+/BIT
[292] ⍝ Reset parameters:
[293]  NFP[9]←NFP[9]-N=0
[294]  NFP[7]←NFP[7]+N-M
[295]  RPS[SET]←N
```

```
    ∇ INDEXA (continued)
[296] ⍝ Last field index:
[297]  F←0
[298] ⍝ Loop by active field:
[299] L42:→(F≥NF)⍴L5
[300] ⍝ Current field index and number:
[301]  F←F+1
[302]  FLD←AFLD[F]
[303] ⍝ Read, compress, replace field FLD for set SET:
[304]  CMP←TIE,FLD+SDISP
[305]  (BIT/⎕FREAD CMP)⎕FREPLACE CMP
[306]  →L42
[307] ⍝
[308] ⍝ Erase <assign> and exit:
[309] L43:F←⎕EX 'assign'
[310] ⍝ Replace LV, RPS, ARPS if layer field assigned:
[311]  →(LAY∊FLDS)↓0
[312]  LV ⎕FREPLACE TIE,9
[313]  NFP ⎕FREPLACE TIE,7
[314]  RPS ⎕FREPLACE TIE,8
[315]  →(×DEL)↓0
[316]  ARPS ⎕FREPLACE TIE,10
    ∇
```

[WSID: MULTIFLO]

```
    ∇ NFP←INDS INDEXWSA FP;A;AFLD;ARPS;BAD;BIT;BLK;CMP;DATA;
      DCMP;DEL;DISP;F;FILED;FILL;FLD;FLDS;FREC;FT;GOOD;I;
      IINDS;INCR;IND;LAY;LAYER;LEAD;LSETS;LV;LVAR;M;MIN;N;NF
      ;NFLD;NR;NREC;NSET;RPS;S;SD;SDISP;SET;SETS;SHAPE;SIND;
      T;TIE;UNQ;VAR;VEC;W;WID;⎕IO
[1]  ⍝ Inserts data from global field variables (e.g. F3,
[2]  ⍝ F5,... for fields 3, 5,...) into fields 11↓FP for
[3]  ⍝ records identified by file indices matrix INDS.
[4]  ⍝ First row of INDS is set number (origin 1);
[5]  ⍝ second row is index (origin 1) within set.
[6]  ⍝ INDS may be of any dimension as long as its
[7]  ⍝ first coordinate is 2. Global field variables
[8]  ⍝ have same number of columns as corresponding
[9]  ⍝ fields.  Leading shape is 1↓⍴INDS.  Global field
[10] ⍝ variables are erased upon completion.
[11]  ⎕ERROR(1≠⍴⍴FP)/'RANK ERROR'
[12]  ⎕ERROR((11>⍴FP)∨2≠1↑⍴INDS)/'LENGTH ERROR'
[13] ⍝ Exit if no fields:
[14]  NFLD←⍴FLDS←11↓FP
[15]  NFP←11⍴FP
[16]  →(×NFLD)↓0
[17]  ⎕IO←1
[18]  INCR←FP[3]
[19]  DEL←|FP[11]
[20]  FILL←FP[11]<0
[21]  ⎕ERROR((∧/FLDS∊⍳INCR)≤DEL∊FLDS)/'INVALID FIELD NUMBER'
[22]  TIE←FP[2]
[23]  FT←⎕FREAD TIE,4
```

```
    ∇ INDEXWSA (continued)
[24] ⍝ Width (no. columns) of specified fields:
[25]  WID←FT[1;]
[26]  ⎕ERROR(0∊WID[FLDS])/'INACTIVE FIELD'
[27] ⍝ Shape of field variables (excluding columns):
[28]  SHAPE←1↓ρINDS
[29] ⍝ Last field index:
[30]  F←0
[31] ⍝ Loop by specified field to verify field vars
[32] ⍝ (bypass this loop to make the function faster
[33] ⍝ and to live dangerously):
[34] L1:→(F≥NFLD)ρL3
[35] ⍝ Current field index, number and width:
[36]  F←F+1
[37]  FLD←FLDS[F]
[38]  W←WID[FLD]
[39] ⍝ Look at field variable:
[40]  S←ρDATA←⍎'F',⍕FLD
[41]  ⎕ERROR((2≠FT[2;FLD])≠0=1↑0ρDATA)/'DOMAIN ERROR'
[42] ⍝ Done if singleton data:
[43]  →(1∧.=S)ρL1
[44] ⍝ Branch unless 'singleton' record:
[45]  →((W≠1)∧1∧.=¯1↓S)↓L2
[46]  ⎕ERROR(W≠¯1↑S)/'LENGTH ERROR'
[47]  →L1
[48] L2:⎕ERROR((ρSHAPE)∧.≠(ρS)+¯1,(W=1)ρ0)/'RANK ERROR'
[49]  N←(1+ρSHAPE)ρS,1
[50]  ⎕ERROR(N∨.≠SHAPE,W)/'LENGTH ERROR'
[51]  →L1
[52] ⍝ Break apart indices:
[53] L3:NREC←ρSETS←, 1 0 ⌿INDS
[54]  INDS←, 0 1 ⌿INDS
[55] ⍝ Exit if no indices:
[56]  →(×NREC)↓L43
[57]  LAY←FP[1]
[58]  DISP←FP[4]
[59]  BLK←FP[5]
[60] ⍝ Read LV, RPS, ARPS if layer fld being assigned:
[61]  →(LAY∊FLDS)↓L4
[62]  LV←⎕FREAD TIE,9
[63]  RPS←ARPS←⎕FREAD TIE,8
[64]  →(=/FP[7 10])ρL4
[65]  ARPS←⎕FREAD TIE,10
[66] ⍝ Determine distinct set numbers (deleting ¯1s):
[67] L4:UNQ←SETS[⍋SETS]
[68]  NSET←ρUNQ←(UNQ≠¯1↓¯1,UNQ)/UNQ
[69] ⍝ Last set index:
[70]  SIND←0
[71] ⍝ Loop by distinct set:
[72] L5:→(SIND≥NSET)ρL43
[73] ⍝ Current set index and number:
[74]  SIND←SIND+1
[75]  SET←UNQ[SIND]
```

```
    ∇ INDEXWSA (continued)
[76] ⍝ Displacement (no. components) before this set:
[77]  SDISP←DISP+INCR×SET+¯1
[78] ⍝ Indices of INDS to retrieve in this set:
[79]  IND←(SET=SETS)/⍳NREC
[80]  I←INDS[IND]
[81] ⍝ Elts of INDS of recs whose layer value changes:
[82]  BAD←⍳0
[83] ⍝ Last field index:
[84]  F←0
[85] ⍝ Loop by specified field:
[86] L6:→(F≥NFLD)⍴L13
[87] ⍝ Current field index, number and width:
[88]  F←F+1
[89]  FLD←FLDS[F]
[90] ⍝ Is this the layer field being changed?
[91]  LAYER←LAY=FLD
[92]  W←WID[FLD]
[93] ⍝ Read data:
[94]  DATA←⎕FREAD CMP←TIE,FLD+SDISP
[95] ⍝ Current field var:
[96]  VAR←⍎'F',⍕FLD
[97] ⍝ Continue if singleton:
[98]  →(1∧.=⍴VAR)⍴L10
[99] ⍝ Branch if 'singleton' record:
[100]  →((W≠1)∧1∧.=¯1↓⍴VAR)⍴L7
[101] ⍝ Branch if rank OK already:
[102]  →((⍴⍴VAR)=2⌊W)⍴L8
[103] L7:VAR←(NREC,(W>1)⍴W)⍴VAR
[104] ⍝ Branch if matrix:
[105] L8:→(W>1)⍴L9
[106]  VAR←VAR[IND]
[107]  →L10
[108] L9:VAR←VAR[IND;]
[109] ⍝ Branch if matrix:
[110] L10:→(W>1)⍴L11
[111] ⍝ Insert data:
[112]  DATA[I]←VAR
[113] ⍝ Branch unless layer field:
[114]  →LAYER↓L12
[115] ⍝ Flag recs whose layer val has changed:
[116]  BAD←(LV[SET]≠VAR)/I
[117]  →L12
[118] L11:DATA[I;]←VAR
[119] ⍝ Branch unless layer field:
[120]  →LAYER↓L12
[121] ⍝ Flag recs whose layer val has changed:
[122]  BAD←(VAR∨.≠LV[SET;])/I
[123] ⍝ Replace data on file:
[124] L12:DATA ⎕FREPLACE CMP
[125]  →L6
[126] ⍝ Next set if layer values unchanged:
[127] L13:→(×⍴BAD)↓L5
```

```
    ∇ INDEXWSA (continued)
⍝ Indices of active fields:
 AFLD←(×WID)/⍳INCR
⍝ Exclude deletion field:
 NF←⍴AFLD←(AFLD≠DEL)/AFLD
⍝ Flag records filed so far:
 FILED←(⍴BAD)⍴0
⍝ Look at layer field:
 LVAR←⎕FREAD TIE,LAY+SDISP
⍝ Branch if vector field:
 VEC←WID[LAY]=1
L14:→VEC⍴L15
⍝ Flag records with layer of 1st unfiled rec:
 LAYER←LVAR[BAD[FILED⍳0];]
 GOOD←LVAR[BAD;]∧.=LAYER
⍝ ...and sets with this record:
 LSETS←LV∧.=LAYER
 →L16
L15:LAYER←LVAR[BAD[FILED⍳0]]
 GOOD←LVAR[BAD]=LAYER
 LSETS←LV=LAYER
⍝ Update FILED; convert to indices:
L16:FILED←FILED∨GOOD
 GOOD←GOOD/BAD
 NR←⍴GOOD
⍝ Consider only non-full sets in this layer or
⍝ empty sets in any layer:
 LSETS←(ARPS=0)∨LSETS∧ARPS≠BLK
⍝ Convert to indices:
 LSETS←LSETS/⍳⍴LSETS
⍝ No. records filed so far:
 FREC←0
⍝ Branch if no slots available in existing sets:
L17:→(BLK≤MIN←⌊/A←|ARPS[LSETS])⍴L32
⍝ Branch if more than 1 set needed:
 T←NR-FREC
 →(T>BLK-MIN)⍴L18
⍝ Choose fullest set which will hold all recs:
 S←BLK-A
 S←LSETS[S⍳⌊/(S≥T)/S]
 →L19
⍝ Choose set with most empty slots:
L18:S←LSETS[A⍳MIN]
⍝ No. records to be inserted within the set:
L19:M←T⌊RPS[S]-|ARPS[S]
⍝ No. records to be catenated within the set:
 N←(T-M)⌊BLK-RPS[S]
⍝ Displacement (no. components) before this set:
 SD←DISP+INCR×S+¯1
⍝ Branch if no deletion field:
 →(×DEL)↓L20
⍝ Read deletion field for set S:
 BIT←⎕FREAD DCMP←TIE,DEL+SD
```

```
    ∇ INDEXWSA (continued)
⍝ Branch if no records to be inserted:
 →(×M)↓L20
⍝ Indices of available insertion slots:
 IINDS←Mρ(~BIT)/⍳ρBIT
⍝ Next field index:
L20:F←1
⍝ Loop by active field:
L21:→(F>NF)ρL26
⍝ Field number:
 FLD←AFLD[F]
⍝ Field width (no. columns):
 W←WID[FLD]
⍝ Look at original set:
 VAR←⎕FREAD TIE,FLD+SDISP
⍝ Read field F for set S:
 DATA←⎕FREAD CMP←TIE,FLD+SD
⍝ Branch if a matrix field:
 →(W>1)ρL23
⍝ Branch if no records to insert:
 →(×M)↓L22
 DATA[IINDS]←VAR[GOOD[FREC+⍳M]]
⍝ Branch if no records to catenate:
 →(×N)↓L25
L22:DATA←DATA,VAR[GOOD[(FREC+M)+⍳N]]
 →L25
⍝ Branch if no records to insert:
L23:→(×M)↓L24
 DATA[IINDS;]←VAR[GOOD[FREC+⍳M];]
⍝ Branch if no records to catenate:
 →(×N)↓L25
L24:DATA←DATA,[1]VAR[GOOD[(FREC+M)+⍳N];]
L25:DATA ⎕FREPLACE CMP
 F←F+1
 →L21
⍝ Branch if no deletion field:
L26:LEAD←1
 →(×DEL)↓L29
⍝ Branch if no records to insert:
 →(×M)↓L27
⍝ Turn active record bits on:
 BIT[IINDS]←1
 LEAD←∧/BIT=∧\BIT
⍝ Branch if no records to catenate:
 →(×N)↓L28
L27:BIT←BIT,Nρ1
L28:BIT ⎕FREPLACE DCMP
⍝ Increment FREC by no. records added to this set:
L29:FREC←FREC+M+N
 RPS[S]←RPS[S]+N
 NFP[9]←NFP[9]+T←0=ARPS[S]
⍝ Replace layer value if set initially empty:
 →T↓L31
```

```
    ∇ INDEXWSA (continued)
⍝ Branch if a vector layer field:
 →VEC⍴L30
 LV[S;]←LAYER
 →L31
L30:LV[S]←LAYER
L31:ARPS[S]←(¯1 1)[1+LEAD]×M+N+|ARPS[S]
 NFP[7]←NFP[7]+N
⍝ Exit if all of data in field vars. filed:
 →(NR=FREC)⍴L40
 →L17
⍝ No. records to be appended in next set:
L32:N←BLK⌊NR-FREC
⍝ Next field number:
 F←1
⍝ Loop by field:
L33:→(F>INCR)⍴L39
 W←WID[F]
⍝ Branch unless a latent field:
 →(×W)⍴L34
 DATA←⍳0
 →L38
⍝ Branch unless it's the deletion field:
L34:→(DEL≠F)⍴L35
 DATA←N⍴1
 →L37
⍝ Look at original set:
L35:VAR←⎕FREAD TIE,F+SDISP
⍝ Branch if a matrix field:
 →(W>1)⍴L36
 DATA←VAR[GOOD[FREC+⍳N]]
 →L37
L36:DATA←VAR[GOOD[FREC+⍳N];]
⍝ Branch unless set must be padded to BLK records:
L37:→FILL↓L38
 DATA←(BLK,1↓⍴DATA)↑DATA
L38:T←DATA ⎕FAPPEND TIE
 F←F+1
 →L33
⍝ Increment FREC by no. records added to this set:
L39:FREC←FREC+N
 ARPS←ARPS,N
 RPS←RPS,T←N⌈BLK×FILL
 NFP[7]←NFP[7]+T
 NFP[6]←NFP[6]+1
 NFP[9]←NFP[9]+1
⍝ Catenate layer value:
 LV←LV,[1]LAYER
⍝ Continue unless all of data in field vars. filed:
 →(NR≠FREC)⍴L32
⍝ Branch if no data left to file:
L40:→(∧/FILED)↓L14
⍝
```

```
     ∇ INDEXWSA (continued)
[284] ⍝ Branch unless deletion field exists:
[285]  →(×DEL)↓L41
[286] ⍝ Read deletion field for set SET:
[287]  BIT←⎕FREAD CMP←TIE,DEL+SDISP
[288] ⍝ Turn off specified indices and replace:
[289]  BIT[BAD]←0
[290]  BIT ⎕FREPLACE CMP
[291] ⍝ Compute new no. records:
[292]  N←+/BIT
[293]  M←|RPS[SET]
[294] ⍝ Reset parameters:
[295]  NFP[9]←NFP[9]-N=0
[296]  ARPS[SET]←N×(¯1 1)[1+N=+/∧\BIT]
[297]  →L5
[298] ⍝
[299] ⍝ Turn off specified indices:
[300] L41:M←RPS[SET]
[301]  BIT←M⍴1
[302]  BIT[BAD]←0
[303] ⍝ Compute new no. records:
[304]  N←+/BIT
[305] ⍝ Reset parameters:
[306]  NFP[9]←NFP[9]-N=0
[307]  NFP[7]←NFP[7]+N-M
[308]  RPS[SET]←N
[309] ⍝ Last field index:
[310]  F←0
[311] ⍝ Loop by active field:
[312] L42:→(F≥NF)⍴L5
[313] ⍝ Current field index and number:
[314]  F←F+1
[315]  FLD←AFLD[F]
[316] ⍝ Read, compress, replace field FLD for set SET:
[317]  CMP←TIE,FLD+SDISP
[318]  (BIT/⎕FREAD CMP)⎕FREPLACE CMP
[319]  →L42
[320] ⍝
[321] ⍝ Erase global field variables:
[322] L43:F←⍕(NFLD,1)⍴FLDS
[323]  F←⎕EX 'F',(+/' '=F)⌽F
[324] ⍝ Replace LV, RPS, ARPS if layer field assigned:
[325]  →(LAY∊FLDS)↓0
[326]  LV ⎕FREPLACE TIE,9
[327]  NFP ⎕FREPLACE TIE,7
[328]  RPS ⎕FREPLACE TIE,8
[329]  →(×DEL)↓0
[330]  ARPS ⎕FREPLACE TIE,10
     ∇
```

```
                                        [WSID: MULTIFLO]
    ∇ R←A FOR B
[1]   ⍝ Catenates arguments together, separating by a
[2]   ⍝ newline character.  Used as:
[3]   ⍝  (FP,¯7 7 2 4 0 3) EXECUTE 'F7←F7⌈F2÷F4' FOR 'F3>1'
[4]     R←A,⎕TCNL,B
    ∇
```

```
                                        [WSID: MULTIFLO]
    ∇ ∆∆NFP←∆∆FP EXECUTE ∆∆EXPR;∆∆A;∆∆AFLDS;∆∆ALL;∆∆ARPS;
      ∆∆BAD;∆∆BIT;∆∆BLK;∆∆CMP;∆∆DATA;∆∆DCMP;∆∆DEL;∆∆DISP;∆∆F
      ;∆∆FILED;∆∆FILL;∆∆FLD;∆∆FLDS;∆∆FN1;∆∆FN2;∆∆FN3;∆∆FREC;
      ∆∆FT;∆∆GOOD;∆∆IINDS;∆∆INCR;∆∆IND;∆∆INDS;∆∆LAB;∆∆LAY;
      ∆∆LAYER;∆∆LEAD;∆∆LSETS;∆∆LV;∆∆LVAR;∆∆M;∆∆MIN;∆∆N;
      ∆∆NAFLD;∆∆NDEL;∆∆NF;∆∆NFLD;∆∆NR;∆∆NSET;∆∆NSFLD;∆∆RPS;
      ∆∆S;∆∆SD;∆∆SDISP;∆∆SEL;∆∆SET;∆∆SETS;∆∆SFLDS;∆∆SIND;∆∆T
      ;∆∆TIE;∆∆VAR;∆∆VEC;∆∆W;∆∆WID;∆∆XEQ;⎕IO
[1]   ⍝ Used as: (FP,¯7 7 2 4) EXECUTE 'F7←F7⌈F2÷F4' or:
[2]   ⍝ (FP,¯7 7 2 4 0 3) EXECUTE 'F7←F7⌈F2÷F4' FOR 'F3>1'
[3]   ⍝ Loops through active sets doing the following:
[4]   ⍝ reads the fields specified beyond the 0 in the
[5]   ⍝ left argument (calling them F3, F5, etc. for
[6]   ⍝ fields 3, 5, etc.); executes the character vector
[7]   ⍝ expression beyond the newline character (inserted
[8]   ⍝ by FOR) in the right argument; retrieves data
[9]   ⍝ from fields specified before the 0 (and positive)
[10]  ⍝ in the left argument (calling them F7, F2, etc.
[11]  ⍝ for fields 7, 2, etc.) for the records of that
[12]  ⍝ set which correspond to 1s in the resulting bit
[13]  ⍝ vector (or all active records if no selection
[14]  ⍝ expression provided); executes the character
[15]  ⍝ vector expression before the newline character in
[16]  ⍝ the right argument; replaces on file data for
[17]  ⍝ fields specified before the 0 (and negative) in
[18]  ⍝ the left argument (assuming their existence in
[19]  ⍝ variables F7, F8, etc. for fields ¯7, ¯8, etc.)
[20]  ⍝ for the records selected.  All "Fn" variables are
[21]  ⍝ erased upon completion.  Note that EXPR is
[22]  ⍝ executed in origin 1; e.g. 'F3[;2]' always refers
[23]  ⍝ to 2nd column.
[24]    ⎕ERROR((1≠⍴⍴∆∆FP)∨1<⍴⍴∆∆EXPR)/'RANK ERROR'
[25]    ⎕ERROR(0=1↑0⍴∆∆EXPR)/'DOMAIN ERROR'
[26]    ⎕IO←1
[27]  ⍝ Extract 3 sets of fields from FP:
[28]    ∆∆T←11+(11↓∆∆FP)⍳0
[29]    ∆∆FLDS←11↓(∆∆T-1)⍴∆∆FP
[30]    ∆∆SFLDS←∆∆T↓∆∆FP
[31]    ∆∆AFLDS←|(∆∆T←∆∆FLDS<0)/∆∆FLDS
[32]    ∆∆FLDS←(~∆∆T)/∆∆FLDS
[33]  ⍝ Extract 2 expressions from EXPR:
[34]    ∆∆EXPR←,∆∆EXPR
[35]    ∆∆T←∆∆EXPR⍳⎕TCNL
[36]    ∆∆XEQ←(∆∆T-1)⍴∆∆EXPR
```

```
     ∇ EXECUTE (continued)
[37]   ΔΔEXPR←ΔΔT↓ΔΔEXPR
[38]   ⎕ERROR((0=ρΔΔFLDS)∨(11≥ρΔΔFP)∨(' '∧.=ΔΔEXPR)≠0=ρ
       ΔΔSFLDS)/'LENGTH ERROR'
[39]   ΔΔINCR←ΔΔFP[3]
[40]   ΔΔDEL←|ΔΔFP[11]
[41]   ΔΔFILL←ΔΔFP[11]<0
[42]   ΔΔT←ΔΔFLDS,ΔΔAFLDS,ΔΔSFLDS
[43]   ⎕ERROR((∧/ΔΔT∊⍳ΔΔINCR)≤ΔΔDEL∊ΔΔT)/'INVALID FIELD
       NUMBER'
[44]   ΔΔTIE←ΔΔFP[2]
[45]   ΔΔFT←⎕FREAD ΔΔTIE,4
[46]  ⍝ Width (no. columns) of fields:
[47]   ΔΔWID←ΔΔFT[1;]
[48]   ⎕ERROR(0∊ΔΔWID[ΔΔT])/'INACTIVE FIELD'
[49]  ⍝ Numbers of sets with records in wrong layer:
[50]   ΔΔBAD←⍳0
[51]  ⍝ Exit if no records:
[52]   ΔΔNFP←11ρΔΔFP
[53]   →(×ΔΔFP[10])↓ΔΔL29
[54]  ⍝ Make field vectors distinct:
[55]   ΔΔT←ΔΔINCRρ0
[56]   ΔΔT[ΔΔSFLDS]←1
[57]   ΔΔNSFLD←ρΔΔSFLDS←ΔΔT/⍳ρΔΔT
[58]   ΔΔT[]←0
[59]   ΔΔT[ΔΔAFLDS]←1
[60]   ΔΔNAFLD←ρΔΔAFLDS←ΔΔT/⍳ρΔΔT
[61]   ΔΔT[]←0
[62]   ΔΔT[ΔΔFLDS]←1
[63]   ΔΔNFLD←ρΔΔFLDS←ΔΔT/⍳ρΔΔT
[64]  ⍝ Names of fields to erase:
[65]   ΔΔFN1←(~ΔΔSFLDS∊ΔΔFLDS)/ΔΔSFLDS
[66]   ΔΔFN1←⍕((ρΔΔFN1),1)ρΔΔFN1
[67]   ΔΔFN1←'F',(+/' '=ΔΔFN1)⌽ΔΔFN1
[68]   ΔΔFN2←(~ΔΔFLDS∊ΔΔAFLDS)/ΔΔFLDS
[69]   ΔΔFN2←⍕((ρΔΔFN2),1)ρΔΔFN2
[70]   ΔΔFN2←'F',(+/' '=ΔΔFN2)⌽ΔΔFN2
[71]   ΔΔFN3←⍕(ΔΔNAFLD,1)ρΔΔAFLDS
[72]   ΔΔFN3←'F',(+/' '=ΔΔFN3)⌽ΔΔFN3
[73]  ⍝ Label vector needed below based upon whether
[74]  ⍝ field in ws or on file when needed (file,ws):
[75]   ΔΔLAB←(ΔΔL17,ΔΔL19)[1+ΔΔFLDS∊ΔΔSFLDS]
[76]   ΔΔLAY←ΔΔFP[1]
[77]   ΔΔDISP←ΔΔFP[4]
[78]   ΔΔBLK←ΔΔFP[5]
[79]   ΔΔARPS←ΔΔRPS←⎕FREAD ΔΔTIE,8
[80]  ⍝ Branch unless ARPS should be read:
[81]   →(=/ΔΔFP[7 10])ρΔΔL1
[82]   ΔΔARPS←⎕FREAD ΔΔTIE,10
[83]  ⍝ Consider only nonempty sets:
[84]  ΔΔL1:ΔΔSETS←0≠ΔΔARPS
[85]  ⍝ ...and sets specified in <layers>:
[86]   →(×ΔΔLAY)↓ΔΔL5
[87]   ΔΔT←×⎕NC 'layers'
```

```
    ∇ EXECUTE (continued)
  →(ΔΔT⍱ΔΔLAY∊ΔΔAFLDS)ρΔΔL5
 ⍝ Read layer values:
  ΔΔLV←⎕FREAD ΔΔTIE,9
  →ΔΔT↓ΔΔL5
 ⍝ Branch if matrix layers field:
  →(ΔΔWID[ΔΔLAY]>1)ρΔΔL2
  ΔΔSETS←ΔΔSETS∧ΔΔLV∊layers
  →ΔΔL5
 ⍝ Convert <layers> to matrix if not already:
 ΔΔL2:⎕ERROR(ΔΔFT[1;ΔΔLAY]≠¯1↑ρlayers)/'LENGTH ERROR'
  →(2=ρρlayers)ρΔΔL3
  layers←((×/¯1↓ρlayers),¯1↑ρlayers)ρlayers
 ⍝ Branch if numeric matrix field:
 ΔΔL3:→(0=1↑0ρΔΔLV)ρΔΔL4
  ΔΔSETS←ΔΔSETS∧(layers CMIOTA ΔΔLV)≤1ρρlayers
  →ΔΔL5
 ΔΔL4:ΔΔSETS←ΔΔSETS∧∨/ΔΔLV∧.=⍉layers
 ⍝ Convert to indices; erase <layers>:
 ΔΔL5:ΔΔSETS←ΔΔSETS/ιρΔΔSETS
  ΔΔNSET←ρΔΔSETS
  ΔΔT←⎕EX 'layers'
 ⍝ Last set index:
  ΔΔS←0
 ⍝ Loop by nonempty set:
 ΔΔL6:→(ΔΔS≥ΔΔNSET)ρΔΔL29
 ⍝ Current set index and number:
  ΔΔS←ΔΔS+1
  ΔΔSET←ΔΔSETS[ΔΔS]
 ⍝ Displacement (no. components) before this set:
  ΔΔSDISP←ΔΔDISP+ΔΔINCR×ΔΔSET-1
 ⍝ Branch if deletion field unneeded:
  →(ΔΔNDEL←ΔΔARPS[ΔΔSET]>0)ρΔΔL7
 ⍝ Read deletion field for set SET:
  ΔΔBIT←⎕FREAD ΔΔTIE,ΔΔDEL+ΔΔSDISP
 ⍝ Are all records in this set active:
 ΔΔL7:ΔΔALL←ΔΔARPS[ΔΔSET]=ΔΔRPS[ΔΔSET]
 ⍝ Branch if a selection expression:
  →(×ΔΔNSFLD)ρΔΔL9
 ⍝ Branch if all records active:
  →ΔΔALLρΔΔL15
 ⍝ Branch if deletion field unneeded:
  →ΔΔNDELρΔΔL8
 ⍝ Indices of active records:
  ΔΔINDS←ΔΔBIT/ιρΔΔBIT
  →ΔΔL15
 ΔΔL8:ΔΔINDS←ιΔΔARPS[ΔΔSET]
  →ΔΔL15
 ⍝ Last field index:
 ΔΔL9:ΔΔF←0
 ⍝ Loop by selection field:
 ΔΔL10:→(ΔΔF≥ΔΔNSFLD)ρΔΔL13
 ⍝ Current field index and number:
  ΔΔF←ΔΔF+1
```

```
    ∇ EXECUTE (continued)
[141]  ΔΔFLD←ΔΔSFLDS[ΔΔF]
[142] ⍝ Read field FLD for set SET:
[143]  ΔΔDATA←⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP
[144] ⍝ Branch if all records in set active:
[145]  →ΔΔALLρΔΔL12
[146] ⍝ Branch if deletion field unneeded:
[147]  →ΔΔNDELρΔΔL11
[148] ⍝ Apply deletion field:
[149]  ΔΔDATA←ΔΔBIT⌿ΔΔDATA
[150]  →ΔΔL12
[151] ⍝ Active records are leading records:
[152] ΔΔL11:ΔΔDATA←(ΔΔARPS[ΔΔSET],1↓ρΔΔDATA)ρΔΔDATA
[153] ⍝ Assign data to global variable Fn:
[154] ΔΔL12:⍎'F',(⍕ΔΔFLD),'←ΔΔDATA'
[155]  →ΔΔL10
[156] ⍝ Once all selection fields read, execute EXPR:
[157] ΔΔL13:ΔΔSEL←⍎ΔΔEXPR
[158] ⍝ Erase fields in SFLDS and not in FLDS:
[159]  ΔΔT←⎕EX ΔΔFN1
[160] ⍝ Indices to retrieve (after squeezing deletions):
[161]  ΔΔIND←ΔΔSEL/ιρΔΔSEL
[162] ⍝ Branch if some records selected:
[163]  →(×ρΔΔIND)ρΔΔL14
[164] ⍝ Erase field variables; get next set:
[165]  ΔΔT←⎕EX ΔΔFN2
[166]  ΔΔT←⎕EX ΔΔFN3
[167]  →ΔΔL6
[168] ⍝ Branch if all records on file selected:
[169] ΔΔL14:→(ΔΔALL←ΔΔALL∧(ρΔΔIND)=ρΔΔSEL)ρΔΔL15
[170] ⍝ Indices to retrieve (before squeezing deletions):
[171]  ΔΔINDS←ΔΔIND
[172] ⍝ Branch if deletion field unneeded:
[173]  →ΔΔNDELρΔΔL15
[174] ⍝ Reset INDS, considering deletion field:
[175]  ΔΔINDS←(ΔΔBIT/ιρΔΔBIT)[ΔΔIND]
[176] ⍝ Last field index:
[177] ΔΔL15:ΔΔF←0
[178] ⍝ Loop by execution field:
[179] ΔΔL16:→(ΔΔF≥ΔΔNFLD)ρΔΔL22
[180] ⍝ Current field index and number:
[181]  ΔΔF←ΔΔF+1
[182]  ΔΔFLD←ΔΔFLDS[ΔΔF]
[183] ⍝ Branch depending on whether already in ws:
[184]  →ΔΔLAB[ΔΔF]
[185] ΔΔL17:ΔΔDATA←⎕FREAD ΔΔTIE,ΔΔFLD+ΔΔSDISP
[186]  →ΔΔALLρΔΔL21
[187] ⍝ Branch if matrix field:
[188]  →(ΔΔWID[ΔΔFLD]>1)ρΔΔL18
[189]  ΔΔDATA←ΔΔDATA[ΔΔINDS]
[190]  →ΔΔL21
[191] ΔΔL18:ΔΔDATA←ΔΔDATA[ΔΔINDS;]
[192]  →ΔΔL21
[193] ΔΔL19:→ΔΔALLρΔΔL16
```

```
    ∇ EXECUTE (continued)
[194]  ∆∆DATA←⍎'F',⍕∆∆FLD
[195] ⍝ Branch if matrix field:
[196]  →(∆∆WID[∆∆FLD]>1)⍴∆∆L20
[197]  ∆∆DATA←∆∆DATA[∆∆IND]
[198]  →∆∆L21
[199] ∆∆L20:∆∆DATA←∆∆DATA[∆∆IND;]
[200] ⍝ Assign data to global variable Fn:
[201] ∆∆L21:⍎'F',(⍕∆∆FLD),'←∆∆DATA'
[202]  →∆∆L16
[203] ⍝ Once all execution fields read, execute XEQ:
[204] ∆∆L22:⍎∆∆XEQ
[205] ⍝ Erase fields in FLDS and not in AFLDS:
[206]  ∆∆T←⎕EX ∆∆FN2
[207] ⍝ Last field index:
[208]  ∆∆F←0
[209] ⍝ Loop by assignment field:
[210] ∆∆L23:→(∆∆F≥∆∆NAFLD)⍴∆∆L28
[211] ⍝ Current field index, number and width:
[212]  ∆∆F←∆∆F+1
[213]  ∆∆FLD←∆∆AFLDS[∆∆F]
[214]  ∆∆W←∆∆WID[∆∆FLD]
[215]  ∆∆CMP←∆∆TIE,∆∆FLD+∆∆SDISP
[216]  ∆∆DATA←⍎'F',⍕∆∆FLD
[217] ⍝ Branch if all records selected:
[218]  →∆∆ALL⍴∆∆L25
[219]  ∆∆VAR←∆∆DATA
[220] ⍝ Read field from file:
[221]  ∆∆DATA←⎕FREAD ∆∆CMP
[222] ⍝ Branch if matrix field:
[223]  →(∆∆W>1)⍴∆∆L24
[224]  ∆∆DATA[∆∆INDS]←∆∆VAR
[225]  →∆∆L25
[226] ∆∆L24:∆∆DATA[∆∆INDS;]←∆∆VAR
[227] ⍝ Check that data is OK before filing:
[228] ∆∆L25:⎕ERROR((2⌊∆∆W)≠⍴⍴∆∆DATA)/'RANK ERROR'
[229]  ⎕ERROR((∆∆W≠¯1↑1,1↓⍴∆∆DATA)∨∆∆RPS[∆∆SET]≠1⍴⍴∆∆DATA)/'
      LENGTH ERROR'
[230]  ⎕ERROR((2≠∆∆FT[2;∆∆FLD])≠0=1↑0⍴∆∆DATA)/'DOMAIN ERROR'
[231] ⍝ Replace data on file:
[232]  ∆∆DATA ⎕FREPLACE ∆∆CMP
[233] ⍝ Continue loop unless layer field:
[234]  →(∆∆LAY≠∆∆FLD)⍴∆∆L23
[235]  →∆∆ALL↓∆∆L26
[236]  ∆∆VAR←∆∆DATA
[237] ⍝ Branch if matrix field:
[238] ∆∆L26:→(∆∆W>1)⍴∆∆L27
[239] ⍝ Continue if layer values ok:
[240]  →(∆∆VAR∧.=∆∆LV[∆∆SET])⍴∆∆L23
[241] ⍝ Else keep track of bad set:
[242]  ∆∆BAD←∆∆BAD,∆∆SET
[243]  →∆∆L23
[244] ∆∆L27:→(∧/∆∆VAR∧.=∆∆LV[∆∆SET;])⍴∆∆L23
[245]  ∆∆BAD←∆∆BAD,∆∆SET
```

```
    ∇ EXECUTE (continued)
[246]  →∆∆L23
[247] ⍝ Erase fields in AFLDS:
[248] ∆∆L28:∆∆T←⎕EX ∆∆FN3
[249]  →∆∆L6
[250] ∆∆L29:∆∆T←⎕EX 'layers'
[251] ⍝ Exit if no sets with wrong layer recs:
[252]  →(×∆∆NSET←⍴∆∆BAD)↓0
[253]  ∆∆SETS←∆∆BAD
[254] ⍝ Indices of active fields:
[255]  ∆∆AFLDS←(×∆∆WID)/⍳∆∆INCR
[256] ⍝ Exclude deletion field:
[257]  ∆∆NF←⍴∆∆AFLDS←(∆∆AFLDS≠∆∆DEL)/∆∆AFLDS
[258] ⍝ Is layer field a vector field?
[259]  ∆∆VEC←∆∆WID[∆∆LAY]=1
[260] ⍝ Last set index:
[261]  ∆∆SIND←0
[262] ⍝ Loop by bad set:
[263] ∆∆L30:→(∆∆SIND≥∆∆NSET)⍴∆∆L64
[264] ⍝ Current set index and number:
[265]  ∆∆SIND←∆∆SIND+1
[266]  ∆∆SET←∆∆SETS[∆∆SIND]
[267] ⍝ Displacement (no. components) before this set:
[268]  ∆∆SDISP←∆∆DISP+∆∆INCR×∆∆SET-1
[269] ⍝ Branch if deletion field unneeded:
[270]  →(∆∆NDEL←∆∆ARPS[∆∆SET]>0)⍴∆∆L31
[271] ⍝ Read deletion field for set SET:
[272]  ∆∆BIT←⎕FREAD ∆∆TIE,∆∆DEL+∆∆SDISP
[273] ⍝ Indices of active records:
[274]  ∆∆INDS←∆∆BIT/⍳⍴∆∆BIT
[275]  →∆∆L32
[276] ∆∆L31:∆∆INDS←⍳∆∆ARPS[∆∆SET]
[277] ⍝ Look at layer field:
[278] ∆∆L32:∆∆LVAR←⎕FREAD ∆∆TIE,∆∆LAY+∆∆SDISP
[279] ⍝ Branch if vector field:
[280]  →∆∆VEC⍴∆∆L33
[281] ⍝ Indices of recs with changed layer value:
[282]  ∆∆BAD←(∆∆LVAR[∆∆INDS;]∨.≠∆∆LV[∆∆SET;])/∆∆INDS
[283]  →∆∆L34
[284] ∆∆L33:∆∆BAD←(∆∆LVAR[∆∆INDS]≠∆∆LV[∆∆SET])/∆∆INDS
[285] ⍝ Flag records filed so far:
[286] ∆∆L34:∆∆FILED←(⍴∆∆BAD)⍴0
[287] ⍝ Branch if vector field:
[288] ∆∆L35:→∆∆VEC⍴∆∆L36
[289] ⍝ Flag records with layer of 1st unfiled rec:
[290]  ∆∆LAYER←∆∆LVAR[∆∆BAD[∆∆FILED⍳0];]
[291]  ∆∆GOOD←∆∆LVAR[∆∆BAD;]∧.=∆∆LAYER
[292] ⍝ ...and sets with this record:
[293]  ∆∆LSETS←∆∆LV∧.=∆∆LAYER
[294]  →∆∆L37
[295] ∆∆L36:∆∆LAYER←∆∆LVAR[∆∆BAD[∆∆FILED⍳0]]
[296]  ∆∆GOOD←∆∆LVAR[∆∆BAD]=∆∆LAYER
[297]  ∆∆LSETS←∆∆LV=∆∆LAYER
```

```
    ∇ EXECUTE (continued)
[298] ⍝ Update FILED; convert to indices:
[299] ∆∆L37:∆∆FILED←∆∆FILED∨∆∆GOOD
[300]  ∆∆GOOD←∆∆GOOD/∆∆BAD
[301]  ∆∆NR←⍴∆∆GOOD
[302] ⍝ Consider only non-full sets in this layer or
[303] ⍝ empty sets in any layer:
[304]  ∆∆LSETS←(∆∆ARPS=0)∨∆∆LSETS∧∆∆ARPS≠∆∆BLK
[305] ⍝ Convert to indices:
[306]  ∆∆LSETS←∆∆LSETS/⍳⍴∆∆LSETS
[307] ⍝ No. records filed so far:
[308]  ∆∆FREC←0
[309] ⍝ Branch if no slots available in existing sets:
[310] ∆∆L38:→(∆∆BLK≤∆∆MIN←⌊/∆∆A←|∆∆ARPS[∆∆LSETS])⍴∆∆L53
[311] ⍝ Branch if more than 1 set needed:
[312]  ∆∆T←∆∆NR-∆∆FREC
[313]  →(∆∆T>∆∆BLK-∆∆MIN)⍴∆∆L39
[314] ⍝ Choose fullest set which will hold all recs:
[315]  ∆∆S←∆∆BLK-∆∆A
[316]  ∆∆S←∆∆LSETS[∆∆S⍳⌊/(∆∆S≥∆∆T)/∆∆S]
[317]  →∆∆L40
[318] ⍝ Choose set with most empty slots:
[319] ∆∆L39:∆∆S←∆∆LSETS[∆∆A⍳∆∆MIN]
[320] ⍝ No. records to be inserted within the set:
[321] ∆∆L40:∆∆M←∆∆T⌊∆∆RPS[∆∆S]-|∆∆ARPS[∆∆S]
[322] ⍝ No. records to be catenated within the set:
[323]  ∆∆N←(∆∆T-∆∆M)⌊∆∆BLK-∆∆RPS[∆∆S]
[324] ⍝ Displacement (no. components) before this set:
[325]  ∆∆SD←∆∆DISP+∆∆INCR×∆∆S+¯1
[326] ⍝ Branch if no deletion field:
[327]  →(×∆∆DEL)↓∆∆L41
[328] ⍝ Read deletion field for set S:
[329]  ∆∆BIT←⎕FREAD ∆∆DCMP←∆∆TIE,∆∆DEL+∆∆SD
[330] ⍝ Branch if no records to be inserted:
[331]  →(×∆∆M)↓∆∆L41
[332] ⍝ Indices of available insertion slots:
[333]  ∆∆IINDS←∆∆M⍴(~∆∆BIT)/⍳⍴∆∆BIT
[334] ⍝ Next field index:
[335] ∆∆L41:∆∆F←1
[336] ⍝ Loop by active field:
[337] ∆∆L42:→(∆∆F>∆∆NF)⍴∆∆L47
[338] ⍝ Field number:
[339]  ∆∆FLD←∆∆AFLDS[∆∆F]
[340] ⍝ Field width (no. columns):
[341]  ∆∆W←∆∆WID[∆∆FLD]
[342] ⍝ Look at original set:
[343]  ∆∆VAR←⎕FREAD ∆∆TIE,∆∆FLD+∆∆SDISP
[344] ⍝ Read field F for set S:
[345]  ∆∆DATA←⎕FREAD ∆∆CMP←∆∆TIE,∆∆FLD+∆∆SD
[346] ⍝ Branch if a matrix field:
[347]  →(∆∆W>1)⍴∆∆L44
[348] ⍝ Branch if no records to insert:
[349]  →(×∆∆M)↓∆∆L43
[350]  ∆∆DATA[∆∆IINDS]←∆∆VAR[∆∆GOOD[∆∆FREC+⍳∆∆M]]
```

```
    ∇ EXECUTE (continued)
[351] ⍝ Branch if no records to catenate:
[352]  →(×∆∆N)↓∆∆L46
[353] ∆∆L43:∆∆DATA←∆∆DATA,∆∆VAR[∆∆GOOD[(∆∆FREC+∆∆M)+⍳∆∆N]]
[354]  →∆∆L46
[355] ⍝ Branch if no records to insert:
[356] ∆∆L44:→(×∆∆M)↓∆∆L45
[357]  ∆∆DATA[∆∆IINDS;]←∆∆VAR[∆∆GOOD[∆∆FREC+⍳∆∆M];]
[358] ⍝ Branch if no records to catenate:
[359]  →(×∆∆N)↓∆∆L46
[360] ∆∆L45:∆∆DATA←∆∆DATA,[1]∆∆VAR[∆∆GOOD[(∆∆FREC+∆∆M)+⍳∆∆N]
      ;]
[361] ∆∆L46:∆∆DATA ⎕FREPLACE ∆∆CMP
[362]  ∆∆F←∆∆F+1
[363]  →∆∆L42
[364] ⍝ Branch if no deletion field:
[365] ∆∆L47:∆∆LEAD←1
[366]  →(×∆∆DEL)↓∆∆L50
[367] ⍝ Branch if no records to insert:
[368]  →(×∆∆M)↓∆∆L48
[369] ⍝ Turn active record bits on:
[370]  ∆∆BIT[∆∆IINDS]←1
[371]  ∆∆LEAD←∧/∆∆BIT=∧\∆∆BIT
[372] ⍝ Branch if no records to catenate:
[373]  →(×∆∆N)↓∆∆L49
[374] ∆∆L48:∆∆BIT←∆∆BIT,∆∆N⍴1
[375] ∆∆L49:∆∆BIT ⎕FREPLACE ∆∆DCMP
[376] ⍝ Increment FREC by no. records added to this set:
[377] ∆∆L50:∆∆FREC←∆∆FREC+∆∆M+∆∆N
[378]  ∆∆RPS[∆∆S]←∆∆RPS[∆∆S]+∆∆N
[379]  ∆∆NFP[9]←∆∆NFP[9]+∆∆T←0=∆∆ARPS[∆∆S]
[380] ⍝ Replace layer value if set initially empty:
[381]  →∆∆T↓∆∆L52
[382] ⍝ Branch if a vector layer field:
[383]  →∆∆VEC⍴∆∆L51
[384]  ∆∆LV[∆∆S;]←∆∆LAYER
[385]  →∆∆L52
[386] ∆∆L51:∆∆LV[∆∆S]←∆∆LAYER
[387] ∆∆L52:∆∆ARPS[∆∆S]←(¯1 1)[1+∆∆LEAD]×∆∆M+∆∆N+|∆∆ARPS[∆∆S
      ]
[388]  ∆∆NFP[7]←∆∆NFP[7]+∆∆N
[389] ⍝ Exit if all of data in field vars. filed:
[390]  →(∆∆NR=∆∆FREC)⍴∆∆L61
[391]  →∆∆L38
[392] ⍝ No. records to be appended in next set:
[393] ∆∆L53:∆∆N←∆∆BLK⌊∆∆NR-∆∆FREC
[394] ⍝ Next field number:
[395]  ∆∆F←1
[396] ⍝ Loop by field:
[397] ∆∆L54:→(∆∆F>∆∆INCR)⍴∆∆L60
[398]  ∆∆W←∆∆WID[∆∆F]
[399] ⍝ Branch unless a latent field:
[400]  →(×∆∆W)⍴∆∆L55
[401]  ∆∆DATA←⍳0
```

```
    ∇ EXECUTE (continued)
[402]  →∆∆L59
[403] ⍝ Branch unless it's the deletion field:
[404] ∆∆L55:→(∆∆DEL≠∆∆F)⍴∆∆L56
[405]  ∆∆DATA←∆∆N⍴1
[406]  →∆∆L58
[407] ⍝ Look at original set:
[408] ∆∆L56:∆∆VAR←⎕FREAD ∆∆TIE,∆∆F+∆∆SDISP
[409] ⍝ Branch if a matrix field:
[410]  →(∆∆W>1)⍴∆∆L57
[411]  ∆∆DATA←∆∆VAR[∆∆GOOD[∆∆FREC+⍳∆∆N]]
[412]  →∆∆L58
[413] ∆∆L57:∆∆DATA←∆∆VAR[∆∆GOOD[∆∆FREC+⍳∆∆N];]
[414] ⍝ Branch unless set must be padded to BLK records:
[415] ∆∆L58:→∆∆FILL↓∆∆L59
[416]  ∆∆DATA←(∆∆BLK,1↓⍴∆∆DATA)↑∆∆DATA
[417] ∆∆L59:∆∆T←∆∆DATA ⎕FAPPEND ∆∆TIE
[418]  ∆∆F←∆∆F+1
[419]  →∆∆L54
[420] ⍝ Increment FREC by no. records added to this set:
[421] ∆∆L60:∆∆FREC←∆∆FREC+∆∆N
[422]  ∆∆ARPS←∆∆ARPS,∆∆N
[423]  ∆∆RPS←∆∆RPS,∆∆T←∆∆N⌈∆∆BLK×∆∆FILL
[424]  ∆∆NFP[7]←∆∆NFP[7]+∆∆T
[425]  ∆∆NFP[6]←∆∆NFP[6]+1
[426]  ∆∆NFP[9]←∆∆NFP[9]+1
[427] ⍝ Catenate layer value:
[428]  ∆∆LV←∆∆LV,[1]∆∆LAYER
[429] ⍝ Continue unless all of data in field vars. filed:
[430]  →(∆∆NR≠∆∆FREC)⍴∆∆L53
[431] ⍝ Branch if no data left to file:
[432] ∆∆L61:→(∧/∆∆FILED)↓∆∆L35
[433] ⍝
[434] ⍝ Branch unless deletion field exists:
[435]  →(×∆∆DEL)↓∆∆L62
[436]  ∆∆BIT←⎕FREAD ∆∆CMP←∆∆TIE,∆∆DEL+∆∆SDISP
[437] ⍝ Turn off specified indices and replace:
[438]  ∆∆BIT[∆∆BAD]←0
[439]  ∆∆BIT ⎕FREPLACE ∆∆CMP
[440] ⍝ Compute new no. records:
[441]  ∆∆N←+/∆∆BIT
[442]  ∆∆M←|∆∆RPS[∆∆SET]
[443] ⍝ Reset parameters:
[444]  ∆∆NFP[9]←∆∆NFP[9]-∆∆N=0
[445]  ∆∆ARPS[∆∆SET]←∆∆N×(¯1 1)[1+∆∆N=+/∧\∆∆BIT]
[446]  →∆∆L30
[447] ⍝
[448] ⍝ Turn off specified indices:
[449] ∆∆L62:∆∆M←∆∆RPS[∆∆SET]
[450]  ∆∆BIT←∆∆M⍴1
[451]  ∆∆BIT[∆∆BAD]←0
[452] ⍝ Compute new no. records:
[453]  ∆∆N←+/∆∆BIT
```

```
    ∇ EXECUTE (continued)
[454] ⍝ Reset parameters:
[455]  ∆∆NFP[9]←∆∆NFP[9]-∆∆N=0
[456]  ∆∆NFP[7]←∆∆NFP[7]+∆∆N-∆∆M
[457]  ∆∆RPS[∆∆SET]←∆∆N
[458] ⍝ Last field index:
[459]  ∆∆F←0
[460] ⍝ Loop by active field:
[461] ∆∆L63:→(∆∆F≥∆∆NF)⍴∆∆L30
[462] ⍝ Current field index and number:
[463]  ∆∆F←∆∆F+1
[464]  ∆∆FLD←∆∆AFLDS[∆∆F]
[465] ⍝ Read, compress, replace field FLD for set SET:
[466]  ∆∆CMP←∆∆TIE,∆∆FLD+∆∆SDISP
[467]  (∆∆BIT⌿⎕FREAD ∆∆CMP)⎕FREPLACE ∆∆CMP
[468]  →∆∆L63
[469] ⍝
[470] ⍝ Replace LV, FP, RPS, ARPS:
[471] ∆∆L64:∆∆LV ⎕FREPLACE ∆∆TIE,9
[472]  ∆∆NFP ⎕FREPLACE ∆∆TIE,7
[473]  ∆∆RPS ⎕FREPLACE ∆∆TIE,8
[474]  →(×∆∆DEL)↓0
[475]  ∆∆ARPS ⎕FREPLACE ∆∆TIE,10
    ∇
```

[WSID: MULTIFLO]

```
    ∇ R←A LAYERS B
[1]  ⍝ Used as:
[2]  ⍝
[3]  ⍝  RINDS←(FP,KFLD)IOTA VALUES LAYERS Z
[4]  ⍝  RINDS←IOTARHO FP LAYERS Z
[5]  ⍝  RINDS←SVEC SLASHIOTARHO FP,SFLDS LAYERS Z
[6]  ⍝  FP←SVEC COMPRESS FP,SFLDS LAYERS Z
[7]  ⍝  MAT←SVEC SELECT FP,FLDS,0,SFLDS LAYERS Z
[8]  ⍝  SVEC SELECTWS FP,FLDS,0,SFLDS LAYERS Z
[9]  ⍝  (FP,XFLDS)EXECUTE XVEC LAYERS Z
[10] ⍝  (FP,XFLDS,0,SFLDS)EXECUTE XVEC FOR SVEC LAYERS Z
[11] ⍝
[12] ⍝ where Z is a vector (if FP[1] is a vector field)
[13] ⍝ or a matrix (if FP[1] is a matrix field) of the
[14] ⍝ larger values for those sets of records to be
[15] ⍝ considered in the operation being performed.  The
[16] ⍝ global variable <layers> is erased after performing
[17] ⍝ the operation (e.g. by IOTA or EXECUTE).
[18] ⍝
[19]  R←A
[20]  layers←B
    ∇
```

5. The APL⋆PLUS file utility functions listed above should be changed as follows to work on a SHARP APL system:

   A. Make the following direct replacements wherever they occur:

APL⋆PLUS	→	SHARP APL
⎕FREAD		⎕READ
⎕FREPLACE		⎕REPLACE
⎕FAPPEND		⎕APPENDR
⎕FNUMS		⎕NUMS
⎕TCNL		⎕AV[156+⎕IO]

   B. Wherever ⎕ERROR is used, replace the expression with an equivalent ⎕SIGNAL expression. For example:

```
⎕ERROR (1≠⍴⍴FP)/'RANK ERROR'

'RANK ERROR' ⎕SIGNAL (1≠⍴⍴FP)/599
```

   While not listed in this book, the SHARP APL versions of the file utility functions are available on disk. See the Postscript.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Unfortunately, the modifications required to make the file utility functions operational in APL2 are much more dramatic. Since files are limited to fixed length records rather than variable size components, we must take a different approach.

One approach is to use a set of file access functions which emulate the behavior of the file access functions on APL⋆PLUS and SHARP APL systems. For example, the functions in the IBM public workspace 2 VAPLFILE (CREATE, USE, SET, GET,...) perform such an emulation. When the file is created, you specify the record length, the number of records and the number of components. The utility functions break apart arrays which are too large for a single record and maintain directories to locate and reassemble the pieces of such arrays. If you take this approach, your most difficult tasks are choosing a record length and estimating the number of records the file will ultimately require.

Another approach is to assume that a record length is chosen which is sufficiently large that any of the objects of the file may fit within it. Since the data components of a multi-set transposed file are limited to a specified block size (FP[5]), the maximum object size can be determined in advance. The maximum size of each data component is a function of its field type, its field width and the block size. Therefore, given the parameters of a file (FT and FP), you can determine a record length which is just long enough to accomodate any single file component. Further, if the maximum number
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

of records and sets are specified, you can compute the total number of records required.

Given the record length and number of records, the file may be created with all of its records. These records may then be accessed directly (using BDAM) to read or replace objects. In fact, it is a simple matter to write functions FREAD and FREPLACE which behave like the APL★PLUS system functions ⎕FREAD and ⎕FREPLACE.

Appending new records, however, is not so easy. Therefore, the file should be initialized with its maximum number of records, all of which are flagged deleted. The subsequent "catenation" of new records will actually be performed by replacing these deleted records.

Let's pursue the latter approach. To implement it, you can do the following:

A. Write a dyadic function NREC∆RECL whose left argument is FP, whose right argument is FT and whose result is the 2 element vector indicating the number of records and the record length required for the file. FP[6] must be the maximum number of sets needed, FP[7] the maximum number of records, and FP[11] the number of the deletion bit field as a negative number. For example:

```
              FT←2 10⍴12 1 1 1 1 1 1 0 0 0 2 3 3 2 2 4 1 1 1 1
              FP←5 987 10 50 2000 25 50000 7 0 0 ¯7
              NL←FP NREC∆RECL FT
              NL
        300 24020
```

B. Write a dyadic function FCREATE whose left argument is the name of the file to be created, whose right argument is a 3 element integer vector: "tie number", number of records and record length. The file is created, and variables (e.g. CTL987, REC987) are shared with BDAM using the tie number to distinctly identify this file from other files. The records are appended via QSAM or via the FMT option of BDAM. For example:

```
              'POLDATA.DATA' FCREATE FP[2],NL
```

C. Write functions FREAD and FREPLACE to emulate the APL★PLUS system functions ⎕FREAD and ⎕FREPLACE. Assume variables shared with BDAM using the tie number to distinctly identify the file from other files.

D. Modify the INITFILE function (listed above for APL★PLUS systems) to require FP[11] to be negative and to fill the entire file (FP[6] sets) with "deleted" records. Use FREPLACE in place of ⎕FAPPEND since the file already contains all of its "components". For example:

```
FP INITFILE FT
```

E. Modify the APL★PLUS file utility functions (CATREC, CATRECWS, IOTA, IOTARHO,...) with the following direct replacements:

APL★PLUS	→	APL2
⎕FREAD		FREAD
⎕FREPLACE		FREPLACE
⎕TCNL		⎕TC[1+⎕IO]
⎕ERROR		⎕ES

F. Modify CATREC, CATRECWS, INDEXA, INDEXWSA and EXECUTE to eliminate the "append new sets" logic since the file is initialized with the maximum needed number of sets. If all sets are full and more records are to be catenated, signal a FILE FULL error.

G. Write functions FTIE, FUNTIE and FERASE to emulate the APL★PLUS system functions ⎕FTIE, ⎕FUNTIE and ⎕FERASE. Share variables with BDAM using the tie number to distinctly identify one file from another. For example:

```
FUNTIE FP[2]
```

While not listed in this book, the APL2 versions of the file utility functions (including NREC∆RECL, FCREATE, FREAD,...) are available on disk. See the Postscript.

Chapter 15 Solutions

BOOLEAN TECHNIQUES

1.

```
∧/NVEC=⌈NVEC
```

or

```
∧/0=1|NVEC
```

(Note: The first expression checks whether the elements are integers within comparison tolerance, i.e. ⎕CT. The second expression checks whether the elements are exactly integers to full internal precision. Comparison tolerance is not meaningful for "0=".)

2.

```
(⌽∨\' '≠⌽CVEC)/CVEC
```

or

```
(-+/∧\' '=⌽CVEC)↓CVEC
```

3.

```
(+/∧\CMAT=' ')⌽CMAT
```

4.

```
MAP←INPUT∊'0123456789'
+/MAP>¯1↓0,MAP
```

5.

```
                                        [WSID: TIMING]
     ∇ TIME∆DEFINE ∆∆NAME;⎕IO;∆∆ALP;∆∆AN;∆∆BL;∆∆I;∆∆N;∆∆NL;
       ∆∆NUM;∆∆PAN;∆∆PBL;∆∆PNUM;∆∆T;∆∆TCNL;∆∆VR
[1]    ⎕IO←1
[2]   ⍝ Remove blanks from fn name:
[3]    ∆∆NAME←(∆∆NAME≠' ')/∆∆NAME
[4]   ⍝ APL*PLUS:
[5]    ⎕ERROR(3≠⎕NC ∆∆NAME)/'NOT A DEFINED FUNCTION'
[6]    ⎕ERROR(∨/(,'*',⎕SI)⎕SS '*',∆∆NAME,'[')/'FUNCTION
       SUSPENDED'
[7]    ⎕ERROR(0=⍴∆∆VR←⎕VR ∆∆NAME)/'FUNCTION LOCKED'
[8]    ∆∆TCNL←⎕TCNL
[9]   ⍝ SHARP APL:
[10]  ⍝ 'NOT A DEFINED FUNCTION' ⎕ER (3≠⎕NC NAME)⍴599
[11]  ⍝ T←2 ⎕WS 2 ⍝ STATE INDICATOR
[12]  ⍝ 'FUNCTION SUSPENDED' ⎕ER (∨/(((1⍴⍴T),1+⍴NAME)↑T)∧.=
       NAME,'[')⍴599
[13]  ⍝ 'FUNCTION LOCKED' ⎕ER (0=⍴VR←1 ⎕FD NAME)⍴599
[14]  ⍝ TCNL←⎕AV[157]
[15]  ⍝ Flag newline chars (Boolean partition vector):
[16]   ∆∆NL←¯1⌽∆∆VR=∆∆TCNL
[17]  ⍝ Flag digits:
[18]   ∆∆NUM←∆∆VR∊'0123456789'
[19]  ⍝ Pole vector of contiguous digits:
[20]   ∆∆PNUM←∆∆NUM≠¯1⌽∆∆NUM
[21]  ⍝ Flag char following ] after line no.:
[22]   ∆∆NL←¯1⌽∆∆PNUM\¯1⌽∆∆PNUM/¯1⌽∆∆NL
[23]   ∆∆PNUM←0
[24]  ⍝ Flag blanks:
[25]   ∆∆BL←∆∆VR=' '
[26]  ⍝ Pole vector of contiguous blanks:
[27]   ∆∆PBL←∆∆BL≠¯1⌽∆∆BL
[28]  ⍝ Flag 1st nonblank char in each line:
[29]   ∆∆NL←(∆∆NL>∆∆BL)∨∆∆PBL\¯1⌽∆∆PBL/∆∆NL
[30]   ∆∆BL←∆∆PBL←0
[31]  ⍝ Flag letters:
[32]   ∆∆ALP←∆∆VR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklmno
       pqrstuvwxyz⍙'
[33]  ⍝ Flag digits or letters:
[34]   ∆∆AN←∆∆ALP∨∆∆NUM
[35]   ∆∆NUM←0
[36]  ⍝ Pole vector of contiguous digits/letters:
[37]   ∆∆PAN←∆∆AN≠¯1⌽∆∆AN
[38]   ∆∆AN←0
[39]  ⍝ Pole vector of identifiers:
[40]   ∆∆PAN←∆∆PAN\∆∆T∨¯1⌽∆∆T←∆∆PAN/∆∆ALP
[41]   ∆∆ALP←0
[42]  ⍝ Pole vec of identifiers at start of line:
[43]   ∆∆PAN←∆∆PAN\∆∆T∨¯1⌽∆∆T←∆∆PAN/∆∆NL
[44]  ⍝ Pole vector of labels:
[45]   ∆∆PAN←∆∆PAN\∆∆T∨1⌽∆∆T←':'=∆∆PAN/∆∆VR
[46]  ⍝ Flag 1st nonblank after label or at start:
[47]   ∆∆NL←(∆∆NL>∆∆PAN)∨¯1⌽∆∆PAN\¯1⌽∆∆PAN/∆∆NL
```

```
    ∇ TIME∆DEFINE (continued)
[48]  ∆∆PAN←0
[49] ⍝ Expand VR for 2 more chars on each line (⍙◇):
[50]  ∆∆VR←(1+∆∆NL+∆∆NL)/∆∆VR
[51]  ⎕IO←0
[52]  ∆∆I←⍳∆∆N←+/∆∆NL
[53] ⍝ Insert the 2 chars:
[54]  ∆∆VR[((∆∆NL/⍳⍴∆∆NL)+∆∆I+∆∆I)∘.+ 0 1]←(∆∆N,2)⍴'⍙◇'
[55]  ∆∆I←⎕DEF ∆∆VR ⍝ APL⋆PLUS
[56] ⍝ Add a last fn line too:
[57]  ∆∆I←⎕DEFL ∆∆NAME,'[',(⍕∆∆N+1),']⍙' ⍝ APL⋆PLUS
[58] ⍝ I←3 ⎕FD (¯6↓VR),'[',(⍕1++/NL),']⍙∇' ⍝ SHARP APL
[59] ⍝ Initialize stopwatch var:
[60]  ⍙M←(∆∆N,4)⍴ 0 0 ,(⌊/⍳0),0
[61] ⍝ Last line run:
[62]  ⍙L←0
    ∇
```

6.

```
                                  [WSID: FNIDS]
    ∇ IDENTIFY VR;ALP;AN;ARGS;AVAR;BL;COLS;HDR;IDLEN;IDS;
      IDSTART;IND;LAB;LEN;LOC;N;NC;NCCON;NCMT;NID;NL;NQ;NUM;
      PAN;PARSE;PBL;PID;PIDA;PLAB;PNUM;PSID;R;RESULT;RVAR;S;
      START;T;TCNL;⎕IO
[1]  ⍝ Evaluates the vector representation (VR) of a
[2]  ⍝ function and displays all possible inconsistencies
[3]  ⍝ involving the use of identifiers.
[4]  ⍝ Requires subfunction: CMIOTA.
[5]   ⎕IO←0
[6]  ⍝ Construct a scalar newline character:
[7]   TCNL←⎕TCNL
[8]  ⍝ TCNL←⎕TC[1] ⍝ APL2
[9]  ⍝ TCNL←⎕AV[156] ⍝ SHARP APL
[10] ⍝ Where does header end?
[11]  T←VR⍳TCNL
[12] ⍝ Grab header of fn (less newline):
[13]  HDR←T⍴VR
[14] ⍝ Drop header from vis rep:
[15]  VR←T↓VR
[16] ⍝ Where does fn syntax end?
[17]  T←HDR⍳';'
[18] ⍝ Localized vars:
[19]  LOC←T↓HDR
[20] ⍝ Drop local vars:
[21]  HDR←T⍴HDR
[22] ⍝ Drop leading junk:
[23]  HDR←(∨\~HDR∊' ∇')/HDR
[24] ⍝ Is there an explicit result?
[25]  T←(⍴HDR)>IND←HDR⍳'←'
[26] ⍝ Explicit result (if any)
[27]  RESULT←(T×IND)⍴HDR
```

```
    ∇ IDENTIFY (continued)
[28] ⍝ Remove result from header:
[29]  HDR←' ',(T×IND+1)↓HDR
[30] ⍝ Starting indices of fn name, args:
[31]  START←1+(HDR=' ')/⍳⍴HDR
[32] ⍝ Lengths of names:
[33]  LEN←(1↓START,1+⍴HDR)-1+START
[34] ⍝ No. of args:
[35]  T←¯1+⍴LEN
[36] ⍝ Indices of args (if any):
[37]  N←⍴IND←((T=2)⍴0),(×T)⍴T
[38] ⍝ Don't consider fn name:
[39]  START←START[IND]
[40]  LEN←LEN[IND]
[41] ⍝ Length of longest arg:
[42]  COLS←⌈/0,LEN
[43] ⍝ Raveled, blank mat of args:
[44]  ARGS←(N×COLS)⍴' '
[45] ⍝ Fill them in (T←MONIOTA LEN):
[46]  T←T+⍳⍴T←LEN/-¯1↓0,+\LEN
[47]  ARGS[T+LEN/COLS×⍳N]←HDR[T+LEN/START]
[48] ⍝ Reshape to mat of args:
[49]  ARGS←(N,COLS)⍴ARGS
[50] ⍝ Flag newline chars (Boolean partition vector):
[51]  NL←¯1⌽VR=TCNL
[52] ⍝ Flag nonquotes:
[53]  NQ←VR≠''''
[54] ⍝ Map of chars not in quote pairs (i.e. char constants)
[55] ⍝ within each fn line (NCCON←NL pEQSCAN NQ):
[56]  NCCON←=\NQ≠NL\T≠¯1↓0,T←~NL/=\¯1↓1,NQ
[57]  NQ←0
[58] ⍝ Flag non-⍝ chars (includes ⍝s in quotes):
[59]  NC←NCCON⍲VR='⍝'
[60] ⍝ Map of chars which do not follow a ⍝ (ignoring ⍝s
[61] ⍝ within quotes) within each fn line. ⍝s are flagged 0.
[62] ⍝ (NCMT←NL pANDSCAN NC):
[63]  S←NL≥NC
[64]  NCMT←~≠\S\T≠¯1↓0,T←~S/NC
[65]  S←T←NC←0
[66] ⍝ Map of chars which are not included within ⍝s or '':
[67]  PARSE←NCMT∧NCCON
[68]  VR←PARSE/VR
[69]  NL←PARSE/NL
[70]  PARSE←NCCON←NCMT←0
[71] ⍝ Flag digits, letters, blanks:
[72]  NUM←VR∊'0123456789'
[73]  ALP←VR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklmnopqrs
      tuvwxyz⍙'
[74]  BL←VR=' '
[75] ⍝ Flag alphanumeric chars:
[76]  AN←NUM∨ALP
[77] ⍝ Pole vec of contiguous digits:
[78]  PNUM←NUM≠¯1↓0,NUM
[79]  NUM←0
```

```
    ∇ IDENTIFY (continued)
[80] ⍝ Pole vec of contiguous digits/letters:
[81]  PAN←AN≠¯1↓0,AN
[82]  AN←0
[83] ⍝ Pole vec of identifiers:
[84]  PID←PAN\T∨¯1⌽T←PAN/ALP
[85]  ALP←PAN←0
[86] ⍝ Flag '⎕' before identifiers (⎕names):
[87]  T←1⌽PID
[88]  T←T\'⎕'=T/VR
[89] ⍝ Shift leading poles of ⎕names to include ⎕:
[90]  PID←T∨PID>¯1⌽T
[91]  T←0
[92] ⍝ Flag char following ] after line no.:
[93]  START←¯1⌽PNUM\¯1⌽PNUM/¯1⌽NL
[94]  NL←PNUM←0
[95] ⍝ Pole vec of contiguous blanks:
[96]  PBL←BL≠¯1↓0,BL
[97] ⍝ Flag 1st nonblank char in each line:
[98]  START←(START>BL)∨PBL\¯1⌽PBL/START
[99]  BL←PBL←0
[100] ⍝ Pole vec of identifiers at start of line:
[101]  PSID←PID\T∨¯1⌽T←PID/START
[102]  START←0
[103] ⍝ Pole vec of labels:
[104]  PLAB←PSID\T∨1⌽T←':'=PSID/VR
[105]  PSID←0
[106] ⍝ Pole vec of direct assignment identifiers:
[107]  PIDA←PID\T∨1⌽T←'←'=PID/VR
[108] ⍝ Start and end (+1) indices of identifiers:
[109]  IND←PID/⍳⍴PID
[110] ⍝ No. of identifiers:
[111]  NID←(⍴IND)÷2
[112]  IND←(NID,2)⍴IND
[113] ⍝ Start indices of identifiers:
[114]  IDSTART←IND[;0]
[115] ⍝ Lengths of identifiers:
[116]  IDLEN←IND[;1]-IDSTART
[117] ⍝ Starting indices of local vars:
[118]  START←1+(LOC=';')/⍳⍴LOC
[119] ⍝ Lengths of local vars:
[120]  LEN←(1↓START,1+⍴LOC)-1+START
[121] ⍝ Length of longest ident.:
[122]  COLS←(⌈/LEN)⌈(⌈/IDLEN)⌈(⍴RESULT)⌈1↓⍴ARGS
[123] ⍝ Pad arg names to conform:
[124]  ARGS←((1⍴⍴ARGS),COLS)↑ARGS
[125] ⍝ 0 row mat if no result:
[126]  RESULT←((×⍴RESULT),COLS)⍴COLS↑RESULT
[127] ⍝ Raveled blank mat of local vars:
[128]  T←(COLS×N←⍴START)⍴' '
[129] ⍝ Fill them in (S←MONIOTA LEN):
[130]  S←S+⍳⍴S←LEN/-¯1↓0,+\LEN
[131]  T[S+LEN/COLS×⍳N]←LOC[S+LEN/START]
```

```
    ∇ IDENTIFY (continued)
[132] ⍝ Reshape to mat of local vars:
[133]  LOC←(N,COLS)ρT
[134] ⍝ Raveled blank mat of identifiers:
[135]  T←(COLS×N←ρIDSTART)ρ' '
[136] ⍝ Fill them in (S←MONIOTA IDLEN):
[137]  S←S+ιρS←IDLEN/-¯1↓0,+\IDLEN
[138]  T[S+IDLEN/COLS×ιN]←VR[S+IDLEN/IDSTART]
[139] ⍝ Reshape to mat of identifiers:
[140]  IDS←(N,COLS)ρT
[141] ⍝ Mat of label names:
[142]  LAB←(S←((N,2)ρPID/PLAB)[;0])⌿IDS
[143]  PLAB←0
[144] ⍝ Mat of assigned vars:
[145]  AVAR←(T←((N,2)ρPID/PIDA)[;0])⌿IDS
[146]  PID←PIDA←0
[147] ⍝ Mat of referenced vars:
[148]  RVAR←(S⍱T)⌿IDS
[149] ⍝ Make distinct:
[150]  AVAR←((AVAR CMIOTA AVAR)=ι1ρρAVAR)⌿AVAR
[151]  RVAR←((RVAR CMIOTA RVAR)=ι1ρρRVAR)⌿RVAR
[152] ⍝
[153] ⍝ Flag distinct labels:
[154]  →(∧/T←(LAB CMIOTA LAB)=ι1ρρLAB)ρL1
[155]  N←' '≠S←,' ',(~T)⌿LAB
[156]  LAB←T⌿LAB
[157]  ⎕←'REDUNDANT LABEL: ',1↓(N∨1⌽N)/S
[158] ⍝ Flag distinct locals:
[159] L1:R←''
[160]  →(∧/T←(LOC CMIOTA LOC)=ι1ρρLOC)ρL2
[161]  R←,' ',(~T)⌿LOC
[162]  LOC←T⌿LOC
[163] L2:→(∨/T←(1ρρRVAR)≤RVAR CMIOTA LOC)↓L3
[164]  N←' '≠S←,' ',T⌿LOC
[165]  ⎕←'IDENTIFIER LOCALIZED BUT NOT USED: ',1↓(N∨1⌽N)/S
[166] ⍝ Any unassigned local vars?
[167] L3:→(∨/T←(1ρρAVAR)≤AVAR CMIOTA LOC)↓L4
[168]  N←' '≠S←,' ',T⌿LOC
[169]  ⎕←'IDENTIFIER LOCALIZED BUT NOT ASSIGNED: ',1↓(N∨1⌽N)
      /S
[170] ⍝ Include result and args in local vars:
[171] L4:LOC←(RESULT,[0]ARGS),[0]LOC
[172] ⍝ Flag distinct locals:
[173]  →(∧/T←(LOC CMIOTA LOC)=ι1ρρLOC)ρL5
[174]  R←R,,' ',(~T)⌿LOC
[175]  LOC←T⌿LOC
[176] L5:→(×ρR)↓L6
[177]  N←' '≠R
[178]  ⎕←'REDUNDANT LOCAL VARIABLE: ',1↓(N∨1⌽N)/R
[179] ⍝ Any localized labels?
[180] L6:→(∨/T←(1ρρLOC)>LOC CMIOTA LAB)↓L7
[181]  N←' '≠S←,' ',T⌿LAB
[182]  ⎕←'LOCALIZED LABEL: ',1↓(N∨1⌽N)/S
```

```
    ∇ IDENTIFY (continued)
[183] ⍝ Any unreferenced labels?
[184] L7:→(∨/T←(1⍴⍴RVAR)≤RVAR CMIOTA LAB)↓L8
[185]  N←' '≠S←,' ',T⌿LAB
[186]  ⎕←'UNUSED LABEL: ',1↓(N∨1⌽N)/S
[187] ⍝ Any assigned labels?
[188] L8:→(∨/T←(1⍴⍴AVAR)>AVAR CMIOTA LAB)↓L9
[189]  N←' '≠S←,' ',T⌿LAB
[190]  ⎕←'ASSIGNED LABEL: ',1↓(N∨1⌽N)/S
[191] ⍝ Any unlocalized assigned vars?
[192] L9:→(∨/T←(1⍴⍴LOC)≤LOC CMIOTA AVAR)↓L10
[193]  N←' '≠S←,' ',T⌿AVAR
[194]  ⎕←'IDENTIFIER ASSIGNED BUT NOT LOCALIZED: ',1↓(N∨1⌽N)
      /S
[195] ⍝ Any unlocalized referenced vars?
[196] L10:→(∨/T←((1⍴⍴LOC)+1⍴⍴LAB)≤(LOC,[0]LAB)CMIOTA RVAR)↓
      L11
[197]  N←' '≠S←,' ',T⌿RVAR
[198]  ⎕←'IDENTIFIER USED AND NOT LOCALIZED: ',1↓(N∨1⌽N)/S
[199] L11:→(∨/(1⍴⍴AVAR)≤AVAR CMIOTA RESULT)↓L12
[200]  ⎕←'RESULT NOT ASSIGNED: ',,RESULT
[201] L12:→(∨/T←(1⍴⍴RVAR)≤RVAR CMIOTA ARGS)↓0
[202]  N←' '≠S←,' ',T⌿ARGS
[203]  ⎕←'ARGUMENT NOT USED: ',1↓(N∨1⌽N)/S
    ∇
```

[WSID: FNIDS]

```
    ∇ R←KEEP LOCALIZE VR;ALP;AN;ARGS;COLS;HDR;IDLEN;IDS;
      IDSTART;IND;KIDS;KLEN;KSTART;LLEN;LOCAL;N;NB;NC;NCCON;
      NCMT;NID;NKEEP;NL;NQ;NUM;PAN;PARSE;PID;PIDA;RESULT;S;T
      ;TCNL;⎕IO
[1]   ⍝ Modifies the vector representation (VR) of a
[2]   ⍝ function such that its header contains as localized
[3]   ⍝ variables all and only those variables which are
[4]   ⍝ assigned within the function.  KEEP is a character
[5]   ⍝ matrix or vector (blank delimited) of names of
[6]   ⍝ variables to not localize though assigned, or to
[7]   ⍝ localize though not assigned.
[8]   ⍝ Requires subfunction: CMIOTA.
[9]    ⎕IO←0
[10]  ⍝ Construct scalar newline character:
[11]   TCNL←⎕TCNL
[12]  ⍝ TCNL←⎕TC[1] ⍝ APL2
[13]  ⍝ TCNL←⎕AV[156] ⍝ SHARP APL
[14]  ⍝ Flag newline chars (Boolean partition vector):
[15]   NL←¯1⌽VR=TCNL
[16]  ⍝ Flag nonquotes:
[17]   NQ←VR≠''''
[18]  ⍝ Map of chars not in quote pairs (i.e. char constants)
[19]  ⍝ within each fn line (NCCON←NL pEQSCAN NQ):
[20]   NCCON←=\NQ≠NL\T≠¯1↓0,T←~NL/=\¯1↓1,NQ
[21]   NQ←0
```

```
    ∇ LOCALIZE (continued)
[22] ⍝ Flag non-⍝ chars (includes ⍝s in quotes):
[23]  NC←NCCON⍲VR='⍝'
[24] ⍝ Map of chars which do not follow a ⍝ (ignoring ⍝s
[25] ⍝ within quotes) within each fn line. ⍝s are flagged 0.
[26] ⍝ (NCMT←NL pANDSCAN NC):
[27]  S←NL≥NC
[28]  NCMT←~≠\S\T≠¯1↓0,T←~S/NC
[29]  S←T←NC←0
[30] ⍝ Map of chars which are not included within ⍝s or '':
[31]  PARSE←NCMT∧NCCON
[32]  NCCON←NCMT←0
[33] ⍝ Flag digits, letters:
[34]  NUM←PARSE∧VR∊'0123456789'
[35]  ALP←PARSE∧VR∊'ABCDEFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklm
      nopqrstuvwxyz⍙'
[36]  PARSE←0
[37] ⍝ Flag alphanumeric chars:
[38]  AN←NUM∨ALP
[39]  NUM←0
[40] ⍝ Pole vec of contiguous digits/letters:
[41]  PAN←AN≠¯1↓0,AN
[42]  AN←0
[43] ⍝ Pole vec of identifiers:
[44]  PID←PAN\T∨¯1⌽T←PAN/ALP
[45]  ALP←PAN←0
[46] ⍝ Flag '⎕' before identifiers (⎕names):
[47]  T←1⌽PID
[48]  T←T\'⎕'=T/VR
[49] ⍝ Shift leading poles of ⎕names to include ⎕:
[50]  PID←T∨PID>¯1⌽T
[51]  T←0
[52] ⍝ Pole vec of direct assignment identifiers:
[53]  PIDA←PID\T∨1⌽T←'←'=PID/VR
[54] ⍝ Start and end (+1) indices of identifiers:
[55]  IND←PID/⍳⍴PID
[56] ⍝ No. of identifiers:
[57]  NID←(⍴IND)÷2
[58]  IND←(NID,2)⍴IND
[59] ⍝ Start indices of identifiers:
[60]  IDSTART←IND[;0]
[61] ⍝ Lengths of identifiers:
[62]  IDLEN←IND[;1]-IDSTART
[63] ⍝ Map vec of nonblanks in exception identifiers:
[64]  NB←' '≠KEEP←,' ',KEEP
[65] ⍝ Start indices in KEEP of identifiers:
[66]  NKEEP←⍴KSTART←(NB>¯1⌽NB)/⍳⍴NB
[67] ⍝ Lengths of KEEP identifiers:
[68]  KLEN←(1+(NB>1⌽NB)/⍳⍴NB)-KSTART
[69] ⍝ Length of longest identifier:
[70]  COLS←(⌈/KLEN)⌈⌈/IDLEN
[71] ⍝ Raveled blank matrix of identifier names:
[72]  IDS←(NID×COLS)⍴' '
```

```
    ∇ LOCALIZE (continued)
[73] ⍝ Fill them in (T←MONIOTA IDLEN):
[74]  T←T+⍳⍴T←IDLEN/-¯1↓0,+\IDLEN
[75]  IDS[T+IDLEN/COLS×⍳NID]←VR[T+IDLEN/IDSTART]
[76] ⍝ Reshape to mat of identifiers:
[77]  IDS←(NID,COLS)⍴IDS
[78] ⍝ Prepare to look at header syntax:
[79]  T←VR⍳TCNL,';'
[80] ⍝ Take up to local vars:
[81]  HDR←(⌊/T)⍴VR
[82] ⍝ Drop leading junk:
[83]  HDR←(∨\~HDR∊' ∇')/HDR
[84]  T←(⍴HDR)>IND←HDR⍳'←'
[85] ⍝ Explicit result (if any):
[86]  RESULT←(T,COLS)⍴COLS↑IND⍴HDR
[87] ⍝ Remove result from header:
[88]  HDR←(T×IND+1)↓HDR
[89] ⍝ No. of arguments:
[90]  S←+/' '=HDR
[91] ⍝ Arguments (if any):
[92]  ARGS←IDS[T+((S=2)⍴0),(×S)⍴S;]
[93] ⍝ Mat of assigned identifiers:
[94]  LOCAL←(((NID,2)⍴PID/PIDA)[;0])⌿IDS
[95]  PID←PIDA←0
[96] ⍝ Remove redundancies:
[97]  LOCAL←((LOCAL CMIOTA LOCAL)=⍳1⍴⍴LOCAL)⌿LOCAL
[98] ⍝ Raveled blank matrix of exception identifiers:
[99]  KIDS←(NKEEP×COLS)⍴' '
[100] ⍝ Fill in (T←MONIOTA KLEN):
[101]  T←T+⍳⍴T←KLEN/-¯1↓0,+\KLEN
[102]  KIDS[T+KLEN/COLS×⍳NKEEP]←KEEP[T+KLEN/KSTART]
[103] ⍝ Reshape to mat of exception identifiers:
[104]  KIDS←(NKEEP,COLS)⍴KIDS
[105] ⍝ No. of syntax local vars:
[106]  N←1⍴⍴T←RESULT,[0]ARGS
[107] ⍝ Squeeze out identifiers in KEEP (globally assigned)
[108] ⍝ or syntax local vars:
[109]  LOCAL←(S←(N+NKEEP)=IND←(KIDS,[0]T)CMIOTA LOCAL)⌿LOCAL
[110] ⍝ Include identifiers in KEEP which are not assigned
[111] ⍝ (assigned within subfns):
[112]  T←(1+N+NKEEP)⍴1
[113]  T[IND]←0
[114]  LOCAL←LOCAL,[0](NKEEP⍴T)⌿KIDS
[115] ⍝ Place local vars in sorted order:
[116]  LOCAL←LOCAL[⎕AV⍋LOCAL;]
[117] ⍝ Lengths of local vars:
[118]  LLEN←+/LOCAL≠' '
[119] ⍝ Indices after each header local semicolon:
[120]  T←+\1+0,LLEN
[121] ⍝ Initialize local var segment of header:
[122]  HDR←(¯1+¯1↑T)⍴';'
[123] ⍝ Insert (S←MONIOTA LLEN):
[124]  S←S+⍳⍴S←LLEN/-¯1↓0,+\LLEN
[125]  HDR[S+LLEN/¯1↓T]←(,LOCAL)[S+LLEN/COLS×⍳⍴LLEN]
```

```
    ∇ LOCALIZE (continued)
[126]  T←VR⍳⎕TCNL,';'
[127]  R←((⌊/T)⍴VR),HDR,T[0]↓VR
    ∇
```

[WSID: COMMENTS]

```
    ∇ R←UNCOMMENT VR;BL;NC;NCCON;NCMT;NL;NQ;NUM;PARSE;PBL;
      PNUM;S;START;T
[1]  ⍝ Modifies the vector representation (VR) of a
[2]  ⍝ function such that its comments are removed (but
[3]  ⍝ the lamp symbols remain for full-line comments).
[4]  ⍝ Flag newline chars (Boolean partition vector):
[5]   NL←¯1⌽VR=⎕TCNL
[6]  ⍝ NL←¯1⌽VR=⎕TC[1+⎕IO] ⍝ APL2
[7]  ⍝ NL←¯1⌽VR=⎕AV[156+⎕IO] ⍝ SHARP APL
[8]  ⍝ Flag nonquotes:
[9]   NQ←VR≠''''
[10] ⍝ Map of chars not in quote pairs (i.e. char constants)
[11] ⍝ within each fn line (NCCON←NL pEQSCAN NQ):
[12]  NCCON←=\NQ≠NL\T≠¯1↓0,T←~NL/=\¯1↓1,NQ
[13]  NQ←0
[14] ⍝ Flag non-⍝ chars (includes ⍝s in quotes):
[15]  NC←NCCON⍲VR='⍝'
[16] ⍝ Map of chars which do not follow a ⍝ (ignoring ⍝s
[17] ⍝ within quotes) within each fn line. ⍝s are flagged 0.
[18] ⍝ (NCMT←NL pANDSCAN NC):
[19]  S←NL≥NC
[20]  NCMT←~≠\S\T≠¯1↓0,T←~S/NC
[21]  S←T←NC←0
[22] ⍝ Always include newline chars (extend maps left by 1):
[23]  NCMT←NCMT∨1⌽NCMT
[24] ⍝ Map of chars which are not included within ⍝s or '':
[25]  PARSE←NCMT∧NCCON
[26]  NCCON←0
[27] ⍝ Flag digits, blanks:
[28]  NUM←PARSE∧VR∊'0123456789'
[29]  BL←PARSE∧VR=' '
[30]  PARSE←0
[31] ⍝ Pole vec of contiguous digits:
[32]  PNUM←NUM≠¯1↓0,NUM
[33]  NUM←0
[34] ⍝ Flag char following ] after line no.:
[35]  START←¯1⌽PNUM\¯1⌽PNUM/¯1⌽NL
[36]  NL←PNUM←0
[37] ⍝ Pole vec of contiguous blanks:
[38]  PBL←BL≠¯1↓0,BL
[39] ⍝ Flag 1st nonblank char in each line:
[40]  START←(START>BL)∨PBL\¯1⌽PBL/START
[41]  BL←PBL←0
[42] ⍝ Squeeze out cmnts except cmnt symb for full line:
[43]  R←(NCMT∨START)/VR
    ∇
```

Chapter 16 Solutions

IRREGULAR ARRAYS

1.

```
AVG←∊(+/¨SALES)÷ρ¨SALES                  (APL2)

AVG←,↑(+/¨SALES)÷ρ¨SALES                 (APL*PLUS)

T←,ρ⍥>SALES                              (SHARP APL)
AVG←(+/T↑⍥>SALES)÷T

AVG←(SALES NNSUMCOLS 1 1)÷NNLEN SALES    (Conventional APL)
```

2.

```
UNAMES←((NAMESιNAMES)=ιρNAMES)/NAMES     (APL2, APL*PLUS,
                                          SHARP APL)

UNAMES←NAMES CNIDX ((NAMES CNIOTA NAMES)=
       ι1↑NAMES)/ι1↑NAMES                (Conventional APL)
```

or

```
SORTED←NAMES CNIDX ⎕AV CNGRADEUP NAMES   (Conventional APL)
UNAMES←SORTED CNIDX (¯1↓1,~SORTED CNEQ
       SORTED CNIDX 1⌽ι1↑SORTED)/ι1↑SORTED
```

3.

```
                                    [WSID: CNFNS]
    ∇ R←CNEST M;A;B;L;N;S;E;I;⎕IO
[1]   ⍝ CNEST is used to convert a character vector or
[2]   ⍝ matrix into a character nest.  If a character
[3]   ⍝ vector, the first character is taken as the
[4]   ⍝ delimiter; each set of characters between
[5]   ⍝ delimiters defines one segment.  If a character
[6]   ⍝ matrix, each row (less the trailing blanks)
[7]   ⍝ defines one segment. The rows of a character
[8]   ⍝ matrix are assumed to be left justified, i.e.
[9]   ⍝ to have no blanks to the left of the leftmost
[10]  ⍝ nonblank character.
[11]  ⍝ Branch if scalar or vector:
[12]   →(1≥ρρM)ρL2
[13]   ⎕IO←0
```

```
    ∇ CNEST (continued)
 ⍝ Insure even no. cols:
  →(0=2|1↓ρM)ρL1
  M←M,' '
 ⍝ Lengths:
 L1:L←+/∨\' '≠⌽M
 ⍝ Cvt to int mat, incl lengths:
  M←L,163 ⎕DR M
 ⍝ No. elts to take from each row, incl length:
  N←ρL←1+⌈L÷2
 ⍝ Start elts of raveled mat:
  S←(1↓ρM)×⍳N
 ⍝ R←MONIOTA L:
  R←R+⍳ρR←L/-¯1↓0,+\L
  R←(N,(+\¯1↓0,L)+1+N),(,M)[R+L/S]
  →0
 ⍝ Word starts,lengths:
 L2:⎕IO←1
  B←M=1↑M
  N←ρS←B/⍳ρB
  L←¯1+(1↓S,1+ρB)-S
  I←+\¯1↓(1+N),E←1+B←⌈L÷2
 ⍝ 8224=163 ⎕DR '  ':
  ⎕IO←0
  R←(N,I),E/8224
  R[I]←L
  R←82 ⎕DR R
  ⎕IO←1
 ⍝ A←MONIOTA L:
  A←A+⍳ρA←L/-¯1↓0,+\L
  R[A+L/2+I+I]←M[A+L/S]
  R←163 ⎕DR R
    ∇
```

[WSID: CNFNS]

```
    ∇ R←W CN∆M M;J;L;N;S;X;⎕IO
 ⍝ CN∆M is used to convert a character nest (M) to a
 ⍝ character matrix (R) with W columns and one row
 ⍝ per segment.  If W is ⍳0, the matrix will have as
 ⍝ many columns as the longest segment in M.  If W
 ⍝ has two elements, the first is the number of columns
 ⍝ of the result and the second is the justification
 ⍝ indicator: 1 (left justify each segment within its
 ⍝ row); 2 (right justify each segment within its row);
 ⍝ 3 (center each segment within its row).  If the
 ⍝ second element is omitted, a value of 1 (left
 ⍝ justify) is assumed.  If the first element is
 ⍝ negative (e.g. ¯1), the matrix will have as many
 ⍝ columns as the longest segment in M.
 ⍝ J∊1 2 3 (L,R,C):
  J←1↑(1↓W),1
 ⍝ Handle char vec or scalar arg:
  →(82≠⎕DR M)ρL1
```

```
    ∇ CN∆M (continued)
[18]  L←ρM←,M
[19]  W←1↑(((0=ρ,W)∨0>1↑W)/L),W
[20]  R←W↑M
[21]  R←(1,W)ρ((J≠1)×0⌈⌊(W-L)÷1⌈J-1)⌽R
[22]  →0
[23] ⍝ No., starts, lengths:
[24] L1:⎕IO←1
[25]  X←⍳N←M[⎕IO]
[26]  ⎕IO←0
[27]  L←M[S←M[X]]
[28]  →(0>1↑W,¯1)↓L2
[29]  W←0⌈⌈/L
[30] ⍝ Truncate too-long segments:
[31] L2:L←L⌊W←1↑W
[32] ⍝ Blank, raveled result:
[33]  R←(N×W)ρ' '
[34]  S←S+1
[35] ⍝ X←MONIOTA L:
[36]  X←X+⍳ρX←L/-¯1↓0,+\L
[37]  M←(82 ⎕DR M)[X+L/S+S]
[38]  →(L3,L4,L5)[J+¯1]
[39] ⍝ Left:
[40] L3:R[X+L/W×⍳N]←M
[41]  →L6
[42] ⍝ Right:
[43] L4:R[X+L/(W×⍳N)+W-L]←M
[44]  →L6
[45] ⍝ Center:
[46] L5:R[X+L/(W×⍳N)+⌊0.5×W-L]←M
[47] L6:R←(N,W)ρR
    ∇
```

[WSID: CNFNS]

```
    ∇ R←C CN∆V CNEST;N;L;S;T;E;⎕IO
[1]  ⍝ CN∆V is used to convert a character nest
[2]  ⍝ CNEST to a character vector with the
[3]  ⍝ character string C between each segment.
[4]   →(82≠⎕DR CNEST)ρL1
[5]   R←CNEST
[6]   →0
[7]  ⍝ No., starts, lengths:
[8]  L1:⎕IO←1
[9]   T←⍳N←CNEST[⎕IO]
[10]  ⎕IO←0
[11]  L←CNEST[S←CNEST[T]]
[12]  S←S+1
[13]  R←ρC←,C
[14]  E←(0⌈¯1+N+N)ρ 1 0
[15] ⍝ Lengths of segments (incl separators):
[16]  L←R+E\L-R
[17] ⍝ Starts of segments (in C, CNEST):
[18]  S←E\R+S+S
```

```
    ∇ CN∆V (continued)
     ⍝ R←MONIOTA L:
      R←R+⍳⍴R←L/-¯1↓0,+\L
      R←(C,82 ⎕DR CNEST)[R+L/S]
    ∇
```

[WSID: CNFNS]

```
    ∇ R←A CNCAT B;D;E;F;G;L;⎕IO
     ⍝ CNCAT is used to catenate two character nests and
     ⍝ return the catenated character nest result.  Either
     ⍝ argument may be a character nest or a character
     ⍝ vector (i.e. a character nest "scalar").
      ⎕IO←0
     ⍝ V∘V,V∘S,S∘V,S∘S:
      →(L1,L2,L3,L4)[2⊥82=(⎕DR A),⎕DR B]
     L1:⎕IO←1
      D←⍳E←A[⎕IO]
      F←⍳G←B[⎕IO]
      ⎕IO←0
      D←A[D]
      F←B[F]
      R←((G+E,D),F+¯1+⍴A),((1+E)↓A),(1+G)↓B
      →0
     L2:L←⍴B←,B
      F←⌈L÷2
      ⎕IO←1
      D←⍳E←A[⎕IO]
      ⎕IO←0
      D←A[D]
      R←(1+E,D,⍴A),((1+E)↓A),L,163 ⎕DR(F+F)↑B
      →0
     L3:L←⍴A←,A
      F←⌈L÷2
      ⎕IO←1
      D←⍳E←B[⎕IO]
      ⎕IO←0
      D←B[D]
      R←((E+ 1 2),(D+2+F),L,163 ⎕DR(F+F)↑A),(1+E)↓B
      →0
     L4:D←⍴A←,A
      E←⌈D÷2
      F←⍴B←,B
      G←⌈F÷2
      R←(2 3 ,E+4),(D,163 ⎕DR(E+E)↑A),F,163 ⎕DR(G+G)↑B
    ∇
```

[WSID: CNFNS]

```
    ∇ R←CNEST CNIDX INDS;L;S;N
     ⍝ CNIDX is used to extract one or more segments
     ⍝ from a character nest (CNEST).  INDS is an
     ⍝ integer vector or scalar of the indices of the
     ⍝ segments to be extracted.  If a vector, the
     ⍝ array extracted (R) will be a character nest.
     ⍝ If a scalar, the array will be a character vector.
      L←CNEST[⎕IO+S←CNEST[1+INDS]]
     ⍝ Branch if nest result:
      →(0≠ρN←ρINDS)ρL1
      R←Lρ82 ⎕DR CNEST[(S+1)+⍳⌈L÷2]
      →0
     L1:L←1+⌈L÷2
     ⍝ R←MONIOTA L:
      R←R+⍳ρR←L/-¯1↓0,+\L
      R←(N,+\¯1↓(1+N),L),CNEST[R+L/S]
    ∇
```

[WSID: CNFNS]

```
    ∇ R←A ASSIGN B
      R←A ⋄ assign←B
    ∇
```

[WSID: CNFNS]

```
    ∇ R←CNEST CNIDXA INDS;L;S;NL;N;A;B;C;NS;I;U
     ⍝ CNIDXA is used to replace one or more segments
     ⍝ into a character nest (CNEST).  INDS is an
     ⍝ integer vector or scalar of the indices of the
     ⍝ segments to be replaced.  If a vector, the
     ⍝ array replaced (assign) will be a character nest.
     ⍝ If a scalar, the array will be a character
     ⍝ vector.  The function ASSIGN simply assigns its
     ⍝ right argument to the global variable <assign>
     ⍝ and returns its left argument.  After being
     ⍝ replaced, <assign> is erased.
      N←CNEST[⎕IO]
     ⍝ Branch unless a scalar index:
      →(×ρρINDS)ρL2
      L←CNEST[⎕IO+S←CNEST[1+INDS]]
      NL←ρassign←,assign
     ⍝ Index assign if same length:
      A←⌈L÷2
      B←⌈NL÷2
      →(A≠B)ρL1
      R←CNEST
      R[S+⍳1+B]←NL,163 ⎕DR(B+B)↑assign
      →L9
     L1:R←(S+1)ρCNEST
      I←(1+INDS+∼⎕IO)+⍳N-INDS+∼⎕IO
      R[I]←R[I]+B-A
      R[S+⎕IO]←NL
```

```
    ∇ CNIDXA (continued)
[27]  R←R,(163 ⎕DR(B+B)↑assign),(S+A+1)↓CNEST
[28]  →L9
[29] ⍝ Branch if nothing to assign:
[30] L2:→(0=U←⍴INDS)⍴L7
[31]  I←1+ιN
[32]  L←CNEST[⎕IO+S←CNEST[I]]
[33]  A←⌈L÷2
[34] ⍝ Branch if char vec to assign:
[35]  →(C←82=⎕DR assign)⍴L3
[36]  R←ιU
[37]  NL←assign[⎕IO+NS←assign[1+R]]
[38]  B←⌈NL÷2
[39]  →L4
[40] L3:R←ι1
[41]  NL←⍴assign←,assign
[42]  NS←0
[43]  B←⌈NL÷2
[44]  assign←NL,163 ⎕DR(B+B)↑assign
[45] ⍝ Index assign if same length.
[46] ⍝ Branch if any lengths change:
[47] L4:→(A[INDS]∨.≠B)⍴L8
[48] ⍝ Drop header data off nest if multi-segments:
[49]  →C⍴L5
[50]  assign←(U+1)↓assign
[51]  →L6
[52] L5:B←U⍴B
[53] ⍝ Repeat data if scalar assignment:
[54]  assign←(U×⍴assign)⍴assign
[55] L6:R←CNEST
[56]  I←S[INDS]
[57]  L←1+B
[58] ⍝ S←MONIOTA L:
[59]  S←S+ι⍴S←L/-¯1↓0,+\L
[60]  R[S+L/I]←assign
[61]  →L9
[62] L7:R←CNEST
[63]  →L9
[64] L8:I[INDS]←N+R
[65]  L←1+(A,B)[I]
[66]  S←(S,NS+⍴CNEST)[I]
[67] ⍝ R←MONIOTA L:
[68]  R←R+ι⍴R←L/-¯1↓0,+\L
[69]  R←(N,+\¯1↓(1+N),L),(CNEST,assign)[R+L/S]
[70] L9:⎕ERASE 'assign'
    ∇
```

[WSID: CNFNS]

```
    ∇ L←CNLEN CNEST;⎕IO
[1]  ⍝ CNLEN is used to return the lengths of the
[2]  ⍝ segments of the character nest right argument.
[3]   ⎕IO←1
[4]   L←⍳CNEST[1]
[5]   ⎕IO←0
[6]   L←CNEST[CNEST[L]]
    ∇
```

[WSID: CNFNS]

```
    ∇ R←CSEQ CNGRADEUP CNEST;A;B;D;G;I;L;M;N;P;R;S;T;Z
[1]  ⍝ CNGRADEUP is used to determine the grade vector
[2]  ⍝ which can be used (with CNIDX) to sort the
[3]  ⍝ segments of CNEST into ascending order.  The
[4]  ⍝ collating sequence used is in CSEQ.
[5]   N←CNEST[⎕IO]
[6]  ⍝ Start, lengths:
[7]   S←⎕IO+CNEST[1+⍳N]
[8]   L←⌈CNEST[S]÷2
[9]  ⍝ Empty segments precede others in coll. seq.:
[10]  R←⍋B←L>0
[11] ⍝ Z:indices into CNEST of remaining elts (i.e. R[A]):
[12]  Z←B/⍳N
[13] ⍝ Done if 0 or 1 segments or all empty:
[14]  →((N>1)∧×⍴Z)↓0
[15] ⍝ A:indices into R of remaining elts (always
[16] ⍝ ascending), B[R]/⍳N:
[17]  A←(N-⍴Z)+⍳⍴Z
[18] ⍝ 1st col (2 chars as an integer) of data:
[19]  D←CNEST[S[Z]+I←1]
[20] ⍝ Reorder grade vec:
[21]  R[A]←Z[G←CSEQ⍋((⍴D),2)⍴82 ⎕DR D]
[22]  D←D[G]
[23] ⍝ Flag 1st elts of grps of like values (partition vec):
[24]  P←D≠¯1⌽D
[25] ⍝ Flag elts of >1 elt grps (map vec):
[26] LP:M←P⍲1⌽P
[27]  →(⍴A←M/A)↓0
[28]  P←M/P
[29] ⍝ Skip following logic if no segments end here:
[30]  →(∧/B←L[Z←R[A]]>I)⍴L1
[31] ⍝ Shift enders to front or end of grps:
[32]  G←⍋B
[33]  G←G[⍋(T←+\P)[G]]
[34]  R[A]←Z[G]
[35]  B←B[G]
[36]  D←B/T
[37]  P←D≠¯1⌽D
[38]  M←P⍲1⌽P
[39]  →(⍴A←M/B/A)↓0
[40]  Z←R[A]
[41]  P←M/P
```

```
    ∇ CNGRADEUP (continued)
   L1:I←I+1
    D←CNEST[S[Z]+I]
    G←CSEQ⍋((⍴D),2)⍴82 ⎕DR D
    G←G[⍋(+\P)[G]]
    R[A]←Z[G]
    D←D[G]
    P←P∨D≠¯1⌽D
    →LP
    ∇
```

[WSID: CNFNS]

```
    ∇ R←A CNEQ B;C;D;E;S;T;L;K;Z;⎕IO
   ⍝ CNEQ is used to compare two character nests of the
   ⍝ same length (i.e. number of segments) for matching
   ⍝ segments.  The result is a Boolean vector with as
   ⍝ many elements as segments and with a 1 for each
   ⍝ segment which is the same in both arguments.  A
   ⍝ character vector argument (or arguments) is
   ⍝ treated as a character nest "scalar" and is
   ⍝ compared against all the segments of the other
   ⍝ argument.  Thus, if both arguments are character
   ⍝ vectors, the result will be a scalar 1 if the
   ⍝ vectors are equivalent and will be 0 otherwise.
   ⍝ V∘V,V∘S,S∘V,S∘S:
    ⎕IO←0
    →(L1,L2,L3,L4)[2⊥82=(⎕DR A),⎕DR B]
   L1:⎕IO←1
    T←⍳A[1]
    ⎕IO←0
   ⍝ Extract starts, lengths:
    S←A[T]
    T←B[T]
    Z←0≠L←B[T]
   ⍝ Compare values when same nonzero lengths:
    E←Z∧R←A[S]=L
    L←⌈(E/L)÷2
    S←E/S
    T←E/T
    ⎕IO←1
   ⍝ Z←MONIOTA L:
    Z←Z+⍳⍴Z←L/-¯1↓0,+\L
    S←Z+L/S
    T←Z+L/T
    ⎕IO←0
   ⍝ Compare values:
    K←+\A[S]=B[T]
    ⎕IO←1
    K←K[+\L]
   ⍝ Are they all equal?
    R[E/⍳⍴E]←L=K-¯1↓0,K
    →0
   L2:⎕IO←1
```

```
    ∇ CNEQ (continued)
[41]  S←⍳A[1]
[42]  ⎕IO←0
[43]  S←A[S]
[44]  R←A[S]=L←⍴B←,B
[45]  →(×L)↓0
[46]  ⎕IO←1
[47]  S←(R/S)∘.+⍳L←⌈L÷2
[48]  ⎕IO←0
[49]  R←R\A[S]∧.=163 ⎕DR(L+L)↑B
[50]  →0
[51] ⍝ Reverse S∘V args and use V∘S logic:
[52] L3:T←A
[53]  A←B
[54]  B←T
[55]  →L2
[56] L4:A←,A
[57]  B←,B
[58]  →(R←(⍴A)∧.=⍴B)↓0
[59]  R←A∧.=B
    ∇
```

```
                                   [WSID: CNFNS]
    ∇ INDS←CNEST CNIOTA VALS;A;B;BASE;C;F;G;I;L;LAB;LL;M;S;
      SHAPE;SS;cseq
[1]  ⍝ CNIOTA searches through the character nest left
[2]  ⍝ argument CNEST for the character nest, vector
[3]  ⍝ or scalar right argument VALS.  The result is
[4]  ⍝ an integer vector (for nest right arg.) or
[5]  ⍝ scalar (for character right arg.) of the
[6]  ⍝ segment index in which the character vector was
[7]  ⍝ first located (as an exact, not partial, match)
[8]  ⍝ or 1 greater than the number of segments if not
[9]  ⍝ found (ala dyadic ⍳).
[10] ⍝ Requires subfns: CNGRADEUP,CNCAT,CNIDX,CNEQ.
[11] ⍝ Branch if right arg a nest:
[12]  C←CNEST[⎕IO]
[13]  →(82≠⎕DR VALS)⍴L1
[14] ⍝ Handle char. vec or scalar right arg:
[15]  L←⍴VALS←,VALS
[16]  SHAPE←⍳0
[17] ⍝ Length of arg. in integers:
[18]  M←⌈L÷2
[19] ⍝ Convert arg. to integers:
[20]  VALS←163 ⎕DR(M+M)↑VALS
[21]  →L5
[22] L1:A←VALS[⎕IO]
[23] ⍝ Branch unless no segments in either arg:
[24]  →(×F←A⌊C)⍴L2
[25] ⍝ Handle empty arg:
[26]  INDS←A⍴⎕IO
[27]  →0
```

```
    ∇ CNIOTA (continued)
[28] ⍝ Branch if both args have more than 1 segment:
[29] L2:→(F≠1)⍴L7
[30] ⍝ Branch unless left arg has 1 segment:
[31]  →(C≠1)⍴L4
[32] ⍝ Handle 1 segment left arg:
[33]  L←CNEST[2+⎕IO]
[34]  M←⌈L÷2
[35] ⍝ Extract segment as integers:
[36]  CNEST←CNEST[3+⍳M]
[37]  S←⎕IO+VALS[1+⍳A]
[38]  B←L=VALS[S]
[39] ⍝ Branch if segment empty:
[40]  →(×L)↓L3
[41] ⍝ Indices of same-length segments:
[42]  S←((~⎕IO)+B/S)∘.+⍳M
[43]  B←B\VALS[S]∧.=CNEST
[44] L3:INDS←⎕IO+~B
[45]  →0
[46] ⍝ Handle 1 segment right arg:
[47] L4:SHAPE←1
[48]  L←VALS[2+⎕IO]
[49]  M←⌈L÷2
[50]  VALS←VALS[3+⍳M]
[51] L5:S←⎕IO+CNEST[1+⍳C]
[52] ⍝ Flag same-length segments in left arg:
[53]  B←L=CNEST[S]
[54] ⍝ Branch if segment empty:
[55]  →(×L)↓L6
[56] ⍝ Indices of same length segments:
[57]  S←((~⎕IO)+B/S)∘.+⍳M
[58]  B←B\CNEST[S]∧.=VALS
[59] L6:INDS←SHAPE⍴B⍳1
[60]  →0
[61] ⍝ Branch if sort alg. costs more than looping alg.:
[62] ⍝    (remove ⍝ after replacing C1,C2,C3,C4 by
[63] ⍝    computed constants):
[64] L7: ⍝→((C4+C5×L+A)>C1+A×C2+C3×L)⍴L5
[65] ⍝ Combine args. and sort (like values together):
[66]  cseq←⎕AV
[67]  F←G←⎕AV CNGRADEUP A←CNEST CNCAT VALS
[68] ⍝ F←⍋G:
[69]  F[G]←⍳⍴G
[70] ⍝ Flag 1st of distinct rows by shifting and comparing:
[71]  F←(~A CNEQ A CNIDX(¯1⌽G)[F])[G]
[72] ⍝ Insure 1st elt is 1 (in case all rows the same):
[73]  F[⎕IO]←1
[74] ⍝ Indices of 1st distinct rows:
[75]  I←F/G
[76] ⍝ Replicate for each like row:
[77]  F[⎕IO]←⎕IO
[78]  I←I[+\F]
[79] ⍝ Unsort indices (to catenated order):
[80]  INDS←I
```

```
    ∇ CNIOTA (continued)
[81]   INDS[G]←I
[82]  ⍝ Keep those corresponding to right arg:
[83]   INDS←C↓INDS
[84]  ⍝ Set 'not found' inds to 'one greater':
[85]   INDS←INDS⌊C+⎕IO
[86]   →0
[87]  ⍝ Use looping algorithm if more efficient:
[88]  L8:INDS←A⍴0
[89]   LAB←(A⍴L9),0
[90]   LL←CNEST[SS←⎕IO+CNEST[1+⍳CNEST[⎕IO]]]
[91]  ⍝ Starting indices from which to add ⍳M:
[92]   SS←SS+~⎕IO
[93]   I←⎕IO
[94]  L9:B←LL=L←VALS[S←⎕IO+VALS[1+I]]
[95]   M←⌈L÷2
[96]   B←B\CNEST[(B/SS)∘.+⍳M]∧.=VALS[(S+~⎕IO)+⍳M]
[97]   INDS[I]←B⍳1
[98]   →LAB[I←I+1]
    ∇
```

4.

[WSID: NNFNS]

```
    ∇ R←P NNEST V;⎕IO;E;I;N
[1]   ⍝ NNEST is used to convert a numeric vector V into
[2]   ⍝ a numeric nest.  P is the replication vector used
[3]   ⍝ to partition the vector.  That is,
[4]   ⍝ (⍴P/0)=(⍴,V).  If P is a singleton, it is
[5]   ⍝ replicated as much as necessary to encompass V.
[6]   ⍝ Replicate left arg.:
[7]    ⎕IO←0
[8]    V←,V
[9]    →(1≠N←⍴P←,P)⍴L1
[10]  ⍝ Repeat P if singleton:
[11]   N←⍴P←((⍴V)÷P)⍴P
[12]  L1:I←+\¯1↓(1+N),E←1+P
[13]  ⍝ Fill with 2s for now:
[14]   R←(N,I),E/2
[15]  ⍝ Insert lengths:
[16]   R[I]←P
[17]   ⎕IO←1
[18]  ⍝ E←MONIOTA P:
[19]   E←E+⍳⍴E←P/-¯1↓0,+\P
[20]   ⎕IO←0
[21]  ⍝ Insert data:
[22]   R[E+P/I]←V
    ∇
```

```
                                        [WSID: NNFNS]
     ∇ R←A NNCATSS B;D;F
[1]   ⍝ NNCATSS is used to catenate two numeric vectors (i.e.
[2]   ⍝ numeric nest "scalars") to form a 2-segment numeric
[3]   ⍝ nest.
[4]    D←ρA←,A
[5]    F←ρB←,B
[6]    R←(2 3 ,D+4),D,A,F,B
     ∇
```

```
                                        [WSID: NNFNS]
     ∇ R←A NNCATVS B;D;E;L;⎕IO
[1]   ⍝ NNCATVS is used to catenate a numeric nest (A) to a
[2]   ⍝ numeric vector (B, i.e. a numeric nest "scalar").
[3]    L←ρB←,B
[4]    ⎕IO←1
[5]    D←⍳E←A[1]
[6]    ⎕IO←0
[7]    D←A[D]
[8]    R←(1+E,D,ρA),((1+E)↓A),L,B
     ∇
```

```
                                        [WSID: NNFNS]
     ∇ R←A NNCATSV B;D;E;L;⎕IO
[1]   ⍝ NNCATSV is used to catenate a numeric vector
[2]   ⍝ (A, i.e. a numeric nest "scalar") to a
[3]   ⍝ numeric nest (B).
[4]    L←ρA←,A
[5]    ⎕IO←1
[6]    D←⍳E←B[1]
[7]    ⎕IO←0
[8]    D←B[D]
[9]    R←((E+ 1 2),(D+2+L),L,A),(1+E)↓B
     ∇
```

```
                                        [WSID: NNFNS]
     ∇ R←A NNCATVV B;D;E;F;G;⎕IO
[1]   ⍝ NNCATVV is used to catenate two numeric nests to form
[2]   ⍝ a longer numeric nest.
[3]    ⎕IO←1
[4]    D←⍳E←A[1]
[5]    F←⍳G←B[1]
[6]    ⎕IO←0
[7]    D←A[D]
[8]    F←B[F]
[9]    R←((G+E,D),F+¯1+ρA),((1+E)↓A),(1+G)↓B
     ∇
```

[WSID: NNFNS]

```
    ∇ R←A NNCAT B;D;E;F;G;L;⎕IO
      ⍝ NNCAT is used to catenate two numeric nests
      ⍝ and/or vectors.  The non-nest arguments are
      ⍝ provided as matrices.
       ⎕IO←0
      ⍝ S∘S;V∘S;S∘V;V∘V:
       →(L1,L2,L3,L4)[(1=ρρA)+2×1=ρρB]
      L1:D←ρA←,A
       F←ρB←,B
       R←(2 3 ,D+4),D,A,F,B
       →0
      L2:L←ρB←,B
       ⎕IO←1
       D←⍳E←⍬ρA
       ⎕IO←0
       D←A[D]
       R←(1+E,D,ρA),((1+E)↓A),L,B
       →0
      L3:L←ρA←,A
       ⎕IO←1
       D←⍳E←⍬ρB
       ⎕IO←0
       D←B[D]
       R←((E+ 1 2),(D+2+L),L,A),(1+E)↓B
       →0
      L4:⎕IO←1
       D←⍳E←⍬ρA
       F←⍳G←⍬ρB
       ⎕IO←0
       D←A[D]
       F←B[F]
       R←((G+E,D),F+¯1+ρA),((1+E)↓A),(1+G)↓B
    ∇
```

[WSID: NNFNS]

```
    ∇ R←NEST NNIDX INDS;I;L;S;N
      ⍝ NNIDXA is used to replace one or more segments
      ⍝ into a numeric nest (NEST).  INDS is a
      ⍝ numeric vector or scalar of the indices
      ⍝ of the segments to be replaced.  If a vector,
      ⍝ the array replaced (assign) will be a
      ⍝ numeric nest.  If a scalar, the array will be
      ⍝ a numeric vector.  The function ASSIGN simply
      ⍝ assigns its right argument to the global
      ⍝ variable <assign> and returns its left argument.
      ⍝ After being replaced, <assign> is erased.
       L←NEST[⎕IO+S←NEST[1+INDS]]
      ⍝ Branch if nest result:
       →(0≠ρN←ρINDS)ρL1
       R←NEST[(S+1)+⍳L]
       →0
      L1:L←1+L
```

```
    ∇ NNIDX (continued)
[17] ⍝ I←MONIOTA L:
[18]  I←I+⍳⍴I←L/-¯1↓0,+\L
[19]  R←(N,+\¯1↓(1+N),L),NEST[I+L/S]
    ∇
```

[WSID: NNFNS]

```
    ∇ R←A ASSIGN B
[1]   R←A ◇ assign←B
    ∇
```

[WSID: NNFNS]

```
    ∇ R←NEST NNIDXA INDS;L;S;D;NL;N;NS;I;U
[1]  ⍝ NNIDXA is used to replace one or more segments
[2]  ⍝ into a numeric nest (NEST).  INDS is a
[3]  ⍝ numeric vector or scalar of the indices of the
[4]  ⍝ segments to be replaced.  If a vector, the
[5]  ⍝ array replaced (assign) will be a numeric
[6]  ⍝ nest.  If a scalar, the array will be a
[7]  ⍝ numeric vector.  The function ASSIGN simply
[8]  ⍝ assigns its right argument to the global
[9]  ⍝ variable <assign> and returns its left argument.
[10] ⍝ After being replaced, <assign> is erased.
[11]  N←NEST[⎕IO]
[12] ⍝ Branch unless a scalar index:
[13]  →(×⍴⍴INDS)⍴L2
[14]  L←NEST[⎕IO+S←NEST[1+INDS]]
[15]  NL←⍴D←,assign
[16] ⍝ Index assign if same length:
[17]  →(L≠NL)⍴L1
[18]  R←NEST
[19]  R[(S+1)+⍳L]←D
[20]  →L5
[21] L1:R←(S+1)⍴NEST
[22]  I←(1+INDS+~⎕IO)+⍳N-INDS+~⎕IO
[23]  R[I]←R[I]+NL-L
[24]  R[S+⎕IO]←NL
[25]  R←R,D,(S+L+1)↓NEST
[26]  →L5
[27] ⍝ Branch if nothing to assign:
[28] L2:→(0=⍴INDS)⍴L3
[29]  I←1+⍳N
[30]  R←⍳U←(D←assign)[⎕IO]
[31]  L←NEST[⎕IO+S←NEST[I]]
[32]  NL←D[⎕IO+NS←D[1+R]]:
[33] ⍝ Index assign if same length:
[34]  →(L[INDS]∨.≠NL)⍴L4
[35]  R←NEST
[36]  I←S[INDS]
[37]  L←1+NL
[38] ⍝ N←MONIOTA L:
[39]  N←N+⍳⍴N←L/-¯1↓0,+\L
```

```
    ∇ NNIDXA (continued)
  R[N+L/I]←(U+1)↓D
  →L5
 L3:R←NEST
  →L5
 L4:I[INDS]←N+R
  L←1+(L,NL)[I]
  S←(S,NS+⍴NEST)[I]
 ⍝ I←MONIOTA L:
  I←I+⍳⍴I←L/-¯1↓0,+\L
  R←(N,+\¯1↓(1+N),L),(NEST,D)[I+L/S]
 L5:⎕ERASE 'assign'
    ∇
```

[WSID: NNFNS]

```
    ∇ L←NNLEN NEST;⎕IO
 ⍝ NNLEN is used to return the lengths of the
 ⍝ segments of the numeric nest right argument.
  ⎕IO←1
  L←⍳NEST[1]
  ⎕IO←0
  L←NEST[NEST[L]]
    ∇
```

[WSID: NNFNS]

```
    ∇ R←NEST NNSUMCOL COLS;I;L;N;S
 ⍝ NNSUMCOL is used to sum the Nth column of each
 ⍝ M column matrix item (raveled) in the numeric
 ⍝ nest NEST.  COLS is M,N.
  N←NEST[⎕IO]
  S←NEST[1+⍳N]+⎕IO
  L←NEST[S]÷COLS[⎕IO]
  I←I+⍳⍴I←L/-¯1↓0,+\L
  R←+\NEST[(L/S+(~⎕IO)+COLS[1+⎕IO])+COLS[⎕IO]×I-⎕IO]
  R←R[(-~⎕IO)++\L]
  R←R-¯1↓0,R
    ∇
```

Chapter 17 Solutions

CURVE FITTING

1.
```
            AMT←60.62 59.57 56.70 60.42
            MAT←4 4⍴32 61 15 82 35 104 10 82 37 83 5 85 25 62 14 85
            AMT⌹MAT
      0.15 0.05 0.73 0.51
```

2. The formula is: SUPPLY = A+(B×TIME)

   The supply is exhausted when SUPPLY=0, i.e. when:

```
       0 = A+(B×TIME)
      -A = B×TIME
    TIME = -(A÷B)
```

   Use ⌹ to determine A and B and then plug into the last equation:

```
          SUPPLY←1850 1772 1705 1508 1490 1250
          TIME←1 2 3 5 6 9
          C←SUPPLY⌹1,[1.5]TIME          (C is A,B)
          -÷/C
    25.6
```

   The supply will be exhausted between weeks 25 and 26.

3. The formula is:
```
The formula is:   (R⋆2) = ((X-CX)⋆2)+((Y-CY)⋆2)
           or:    (R⋆2) = (X⋆2)+(Y⋆2)+(CX⋆2)+(CY⋆2)+(¯2×X×CX)+(¯2×Y×CY)
        so:   (X⋆2)+(Y⋆2) = ((R⋆2)+(-CX⋆2)+(-CY⋆2))+(2×X×CX)+(2×Y×CY)
```

Use ⌹ to determine the coefficients:

```
             X←4 3 2 4 7 6
             Y←1 2 3 5 2 4
             LEFT←(X*2)+(Y*2)
             MAT←1,2×X,[1.5]Y
             C←LEFT⌹MAT
             ⎕←CX←C[2]
       4.6
             ⎕←CY←C[3]
       2.9
```

```
Since:    C[1] = (R*2)+(-CX*2)+(-CY*2)
 then:    (R*2) = C[1]+(CX*2)+(CY*2)
   so:
             ⎕←R←(C[1]+(CX*2)+(CY*2))*.5
       2.2
```

Therefore, the center is (4.6,2.9) and the radius is 2.2.

4. For: T1 = C4+(C5×(R+L))

```
             C←T1⌹1,[1.5]R+L
             C4←C[1]
             C5←C[2]
```

For: T2 = C1+(R×(C2+(C3×L)))
T2 = C1+(C2×R)+(C3×R×L)

```
             C←T2⌹1,R,[1.5]R×L
             C1←C[1]
             C2←C[2]
             C3←C[3]
```

Chapter 18 Solutions

FINANCIAL UTILITIES

1.
```
        1000×(1+365 EFFECTIVE .11)*1.5
  1179.36
or
        1000×(1+.11÷365)*365×1.5
  1179.36
```

2.
```
        tERM←40÷52
        pAY←10×52
        pER←52
        pDEF←0
        dEF←-tERM
        cONV←12
        iNT←.08
        VALUE
  412.84
or
        V←÷(1+.08÷12)*12
        10×(V*-40÷52)×(1-V*40÷52)÷1-V*÷52
  412.84
```

3.
```
        PI←SCHEDULE 4 12 12 0 12 .14                (48 rows)
        TOTPRIN←+/PI[;1]       (Loan amt. at $1 per month)
        PI←PI×10000÷TOTPRIN    (Cvt. loan amt. to $10,000)
        +/PI[1;]               (Monthly pmt.; sum any row)
  273.26
        +/¯12↑PI[;1]          (Repayment amt. is principal
  3043.47                      outstanding, i.e. remaining
                               principal payments)
        +/36↑PI[;2]           (Sum 3 years of interest pmts.)
  2881.01
```

4.

```
          DATES←19870101 19870701,101+10000×1991+⍳15
          AMTS←1000×¯10 ¯5,15⍴3
          DATES IROR AMTS
      0.1058
```

5.

```
                                             [WSID: INTERESTI]
        ∇ BV←DATE FCBOOK PARAMS;COUP;CRATE;D;DAYS;F;MDATE;MVAL;N
          ;NCOUPS;P;PAR;S;YLD
    [1]  ⍝ Returns book values as of DATE (YYYMMDD), a scalar,
    [2]  ⍝ for fixed-coupon securities defined in the matrix
    [3]  ⍝ PARAMS, one row per security.  Value is computed
    [4]  ⍝ after coupon is received if DATE is a coupon date.
    [5]  ⍝ Result has shape: (1↑⍴PARAMS).
    [6]  ⍝
    [7]  ⍝   PARAMS[;1]  par value
    [8]  ⍝         [;2]  maturity date (YYYYMMDD)
    [9]  ⍝         [;3]  annual coupon rate
    [10] ⍝         [;4]  number of coupons per year
    [11] ⍝         [;5]  couponly yield rate
    [12] ⍝         [;6]  (optional) maturity value (par value
    [13] ⍝                if omitted)
    [14] ⍝
    [15]  PAR←MVAL←PARAMS[;1]
    [16]  MDATE←PARAMS[;2]
    [17]  CRATE←PARAMS[;3]÷NCOUPS←PARAMS[;4]
    [18]  YLD←PARAMS[;5]
    [19]  →(5=1↓⍴PARAMS)⍴START
    [20]  MVAL←PARAMS[;6]
    [21] ⍝
    [22] ⍝ Formula:
    [23] ⍝
    [24] ⍝  BV(t) = (MVAL×(1+Y)*-W(t))+PAR×CRATE×(1-(1+Y)*
    [25] ⍝          -W(t))÷Y
    [26] ⍝
    [27] ⍝ where:  BV(t) = book value at time t (a coupon date)
    [28] ⍝             Y = couponly yield rate
    [29] ⍝          W(t) = the number of (whole) coupon periods
    [30] ⍝                 remaining from time t to MDATE
    [31] ⍝
    [32] ⍝ Compute approx days (360 days/yr) from specified
    [33] ⍝ DATE to maturity (change 31 days to 30):
    [34] START:DAYS← 360 30 1 +.× 0 100 100 ⊤MDATE-31=100|MDATE
    [35]  DAYS←DAYS- 360 30 1 +.× 0 100 100 ⊤DATE-31=100|DATE
    [36] ⍝ No. coupon periods from specified date to maturity:
    [37]  N←(DAYS×NCOUPS)÷360
    [38] ⍝ Coupon payment:
    [39]  COUP←PAR×CRATE
    [40] ⍝ Book value at prior coupon:
    [41]  D←(1+YLD)*-⌈N
    [42]  P←(MVAL×D)+COUP×(1-D)÷YLD
```

```
    ∇ FCBOOK (continued)
[43] ⍝ Book value at subsequent coupon:
[44]  D←(1+YLD)⋆-⌊N
[45]  S←(MVAL×D)+COUP×(1-D)÷YLD
[46] ⍝ Fraction of per. from specified date to next coupon:
[47]  F←N-⌊N
[48] ⍝ Interpolate between prior and subsequent book vals:
[49]  BV←(P×F)+S×1-F
    ∇
```

Chapter 19 Solutions

EXCEPTION HANDLING

1. APL⋆PLUS:

```
     ∇ SHOWFILE NAME;LIM;N;⎕ALX;⎕ELX
[1]    ⎕ALX←⎕ELX←'⎕DM'
[2]    LIM←TIEFILE NAME
[3]    ⎕ALX←'→ASK'
[4]    ⎕ELX←'→(''ATTN''∧.=4↑⎕DM)/ASK ⋄ ⎕DM'
[5]   ASK:⍞←'BEGIN WITH WHICH RECORD?'
etc.
```

SHARP APL:

```
     ∇ SHOWFILE NAME;LIM;N;⎕TRAP
[1]    ⎕TRAP←''
[2]    LIM←TIEFILE NAME
[3]    ⎕TRAP←'∇1000 C →ASK ∇0 C →(''ATTN''∧.=4↑5↓⎕ER[⎕IO;])
       /ASK'
[4]   ASK:⍞←'BEGIN WITH WHICH RECORD?'
etc.
```

APL2: Cannot be solved since attentions are not detected as exceptions

2.

```
                                            [WSID: INPUT]
     ∇ R←NUM NXPROMPTE PROMPT;⎕ELX
[1]   ⍝ Displays character vector PROMPT, allows keyboard
[2]   ⍝ input on same line and returns numeric vector
[3]   ⍝ response of length NUM (or of any length if NUM=0).
[4]   ⍝ Returns numeric scalar escape code if escape word
[5]   ⍝ entered.  Allows and executes primitive APL
[6]   ⍝ expressions.  Requires: CPROMPTE.
[7]   ⍝ APL⋆PLUS version.
[8]   ⍝ SHARP APL: localize ⎕TRAP instead of ⎕ELX.
[9]   ⍝ APL2: localize neither ⎕ELX or ⎕TRAP.
[10]   ⎕ELX←'⎕DM' ⍝ APL⋆PLUS
[11] ⍝ ⎕TRAP←'' ⍝ SHARP APL
[12] L1:R←CPROMPTE PROMPT
```

```
    ∇ NXPROMPTE (continued)
[13] ⍝ Exit if scalar escape code:
[14]  →(⍴⍴R)↓0
[15] ⍝ Branch if any letters (but E) in response:
[16]  →(∨/R∊'ABCDFGHIJKLMNOPQRSTUVWXYZ∆abcdefghijklmnopqrstu
      vwxyz⍙')/L2
[17] ⍝ Branch if any E not used in exponential notation:
[18]  →(∨/(R='E')∧¯1↓1,~R∊'01234567890')/L2
[19] ⍝ Execute expression and ravel (trapping error):
[20]  ⎕ELX←'→L4' ⍝ APL*PLUS
[21] ⍝ ⎕TRAP←'∇0 E →L4' ⍝ SHARP APL
[22]  R←,⍎R ⍝ APL*PLUS or SHARP APL
[23] ⍝ '→L4' ⎕EA 'R←,',R ⍝ APL2
[24] ⍝ Check that result is numeric:
[25]  →(0=1↑0⍴R)/L3
[26] L2:⍞←'** ENTER NUMBERS OR APL EXPRESSIONS ONLY **'
[27]  →L1
[28] ⍝ Exit if NUM is 0 or is length of input:
[29] L3:→NUM↓0
[30]  →(NUM=⍴R)/0
[31]  ⍞←'** ENTER ',(⍕NUM),' NUMBER',(NUM=1)↓'S **'
[32]  →L1
[33] L4:⍞←'** INVALID EXPRESSION CAUSING: ',⎕DM ⍝ APL*PLUS
[34] ⍝ L4:⍞←'** INVALID EXPRESSION CAUSING: ',5↓⎕ER[⎕IO;] ⍝
      SHARP APL
[35] ⍝ ⎕←1 0↓⎕ER ⍝ SHARP APL
[36] ⍝ L4:⍞←'** INVALID EXPRESSION CAUSING: ',⎕EM[⎕IO;] ⍝
      APL2
[37] ⍝ ⎕←1 0↓⎕EM ⍝ APL2
[38]  ⍞←''
[39]  →L1
    ∇
```

3. Below are the SHARP APL (ERRATTNS) and APL*PLUS (ERRATTNP) solutions to the problem. The problem cannot be solved with APL2 because attentions are not considered exceptions (to be handled) and because the ⎕EA and ⎕EC system functions are not designed to allow the type of "environment conditioning" required in the problem.

[WSID: ERROR]

```
    ∇ ELX ERRATTNS ALX
[1]  ⍝ SHARP APL version.
[2]  ⍝ Sets ⎕TRAP so that error will cause →ELX at
[3]  ⍝ the SI level at which ⎕TRAP is local; and
[4]  ⍝ break will cause →ALX at that level.  If ELX
[5]  ⍝ or ALX is a character vector, it is executed
[6]  ⍝ at the level at which the error ar attention
[7]  ⍝ occurs.  If ELX or ALX is empty, no special
[8]  ⍝ error or attention handling is included in
[9]  ⍝ the local ⎕TRAP.  Be sure to localize ⎕TRAP
```

```
    ∇ ERRATTNS (continued)
[10] ⍝ in the calling function.
[11]  ⎕TRAP←''
[12]  →(0∊⍴ELX)⍴L2
[13] ⍝ Branch if numeric ELX:
[14]  →(0=1↑0⍴ELX)⍴L1
[15]  ⎕TRAP←'∇ 0 E ',ELX
[16]  →L2
[17] L1:⎕TRAP←'∇ 0 C →',⍕ELX
[18] L2:→(0∊⍴ALX)⍴0
[19] ⍝ Branch if numeric ALX:
[20]  →(0=1↑0⍴ALX)⍴L3
[21]  ⎕TRAP←⎕TRAP,'∇ 1000 E ',ALX
[22]  →0
[23] L3:⎕TRAP←⎕TRAP,'∇ 1000 C →',⍕ALX
    ∇
```

[WSID: ERROR]

```
    ∇ ELX ERRATTNP ALX
[1]  ⍝ APL*PLUS version.
[2]  ⍝ Sets ⎕ELX and possibly ⎕ALX so that error
[3]  ⍝ will cause →ELX at the SI level at which
[4]  ⍝ ⎕ELX is local; and break will cause →ALX
[5]  ⍝ at the SI level at which ⎕ALX is local (uses
[6]  ⍝ ⎕ERROR to percolate to proper level).  If
[7]  ⍝ ELX or ALX is a character vector, it is
[8]  ⍝ executed at the level at which the error or
[9]  ⍝ attention occurs.  If ELX is empty, ⎕ELX is
[10] ⍝ set to '⎕DM'.  If ALX is empty, ⎕ALX is set
[11] ⍝ to '⎕DM' if ⎕ALX is localized in the calling
[12] ⍝ function, and otherwise is not modified.  Be
[13] ⍝ sure to localize ⎕ELX in the calling function.
[14] ⍝ Localize ⎕ALX only if ALX is not empty.
[15]  →(0∊⍴ALX)⍴L2
[16] ⍝ Branch if numeric ALX:
[17]  →(0=1↑0⍴ALX)⍴L1
[18] ⍝ Character ALX:
[19]  ⎕ALX←ALX
[20]  →L3
[21] ⍝ Numeric ALX:
[22] L1:⎕ALX←'⍎(¯1=1⍴⎕IDLOC''⎕ALX'')/'''⎕ERROR''''ATTN''''''◇
      →',⍕ALX
[23]  →L3
[24] ⍝ Empty ALX; branch if ⎕ALX not local to calling fn:
[25] L2:→(¯1=1↑1↓,⎕IDLOC '⎕ALX')⍴L3
[26]  ⎕ALX←'⎕DM'
[27] L3:→(0∊⍴ELX)↓L4
[28]  ELX←'⎕DM'
[29] ⍝ Branch if numeric ELX:
[30] L4:→(0=1↑0⍴ELX)⍴L6
[31] ⍝ Branch if non-empty numeric ALX:
[32]  →((0≠1↑0⍴ALX)∨0∊⍴ALX)↓L5
```

```
     ∇ ERRATTNP (continued)
[33] ⍝ Character ELX; character or empty ALX:
[34]  ⎕ELX←ELX
[35]  →0
[36] ⍝ Character ELX; numeric ALX:
[37] L5:⎕ELX←'⍎((¯1≠1⍴⎕IDLOC''⎕ALX'')∧''ATTN''∧.=4↑⎕DM)/''→'
      ,(⍕ALX),'''◇⎕ERROR(''ATTN''∧.=4↑⎕DM)/4↑⎕DM◇',ELX
[38]  →0
[39] L6:→((0≠1↑0⍴ALX)∨0∊⍴ALX)↓L7
[40] ⍝ Numeric ELX; character or empty ALX:
[41]  ⎕ELX←'⍎(¯1=1⍴⎕IDLOC''⎕ELX'')/''⎕ERROR(∧\⎕DM≠⎕TCNL)/⎕DM
      ''◇→',⍕ELX
[42]  →0
[43] ⍝ Numeric ELX; numeric ALX:
[44] L7:⎕ELX←'⍎(¯1=1⍴⎕IDLOC''⎕ELX'')/''⎕ERROR(∧\⎕DM≠⎕TCNL)/
      ⎕DM''◇→(''ATTN''∧.=4↑⎕DM)⌽',⍕(1↑,ELX),1↑,ALX
     ∇
```

INDEX

APL

ADVANCED TECHNIQUES AND UTILITIES

Bergquist

ZARK