York

APL

by J. Morgan Smyth

SMYTH • York *APL*

York APL

by J. Morgan Smyth

Published by

Ryerson Polytechnical Institute

Toronto

CONTENTS

INTRODUCTION

York APL is a terminal oriented computer language. It derives its name from two sources. The word "York" refers to the fact that this particular version of the APL language was designed and developed at York University, primarily by Gord Ramer. The letters APL are an acronym from a book entitled, "A Programming Language", written by Ken Iverson and published by John Wiley and Sons in 1962. The APL language is based on the mathematical notation expressed by Mr. Iverson in this book.

Unlike most computer lanuages, York APL is ideal for the person who knows very little about the computer and its inner workings. There are no punched cards or complex coding which are usual requirements associated with other languages. The person enters statements to the computer via a terminal, which is often at a remote location, and the computer uses the same terminal to type its responses to these statements. Here is an example of two such statements and the computer's respective replies:

```
      67+43
110

      6×3
18
```

The statements typed in by the person at the terminal, called the APL user, are slightly indented from the left margin to make it easier to distinguish who typed what.

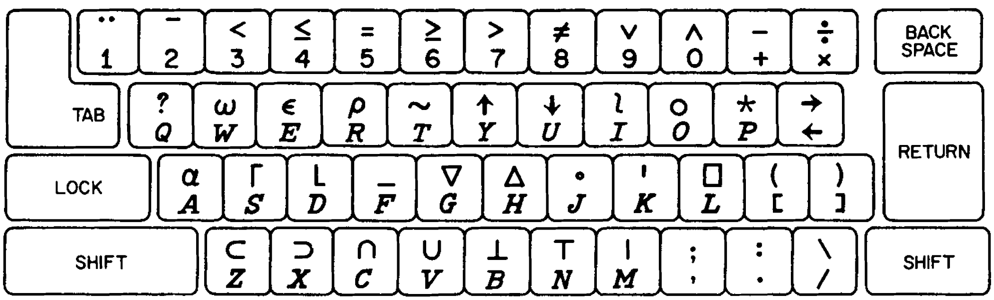
As soon as each of the two statements above was entered by the user, the computer performed an evaluation and returned its result, just as any desk calculator would do. But besides being able to use the computer like a desk calculator, it can also be used to write and store several statements that may be executed at any time. These statements make up a program or function as it's called in APL. Each function has an associated name which is typed in by the user in order to execute the statements. Below, is a function called *STAT* that, when executed, finds the mean, highest and lowest values, and the range between the highest and lowest values contained in a set of numbers. Here is how it works:

```
      STAT
ENTER DATA.
⎕     2 6 5 1 9 7

THE MEAN IS 5
THE MAXIMUM VALUE IS 9
THE MINIMUM VALUE IS 1
THE RANGE IS 8
```

The user typed in the word *STAT* and the requested set of numbers, and the computer typed the rest.

The characters found on the APL terminal keyboard are quite different from the usual characters on most typewriters. Here is the format of a typical APL terminal keyboard:



TYPICAL APL TERMINAL KEYBOARD

(Although there are several types of terminals that can be used to access the York APL system, all references to terminals in this book are to the IBM 2741 Communications Terminal because it is presently the most common.)

Notice that all the letters found on the lowershift part of most of the keys are letters of the alphabet in capitalized italic form, while the uppershift characters consist of some familiar and many unfamiliar symbols. It is these symbols that are used to make up the York APL system, as will be illustrated in the remainder of this text.

Chapter One

SIGNING ON AND OFF

To use the York APL system, the first step is to "Sign On" at any of the available terminals. This is done by typing in an APL user account number.

Every user of APL has his own 4 digit APL number, usually obtained from the Computing Centre secretary or the APL Coordinator. Once this number has been added to the system, generally within a day following the request, the user is able to "Sign On" to the APL system.

Signing On

The following steps must be taken to Sign On:

1. The switch on the left hand side of the APL terminal must be in the "COM" position.

2. The ON/OFF switch on the right hand side of the terminal keyboard is then switched to ON.

   If the terminal is not connected to the computer via a telephone, proceed to step 5.

3. Telephone the computer.

4. When an uninterrupted high-pitched tone is heard, if using a data set, press the "talk" button and hang up the receiver. With an acoustic coupler, place the receiver firmly into the coupler. If the phone keeps ringing, the computer and/or APL are not functioning.

(A busy signal signifies all the telephone lines are presently in use.)

5. Press the "RETURN" key once.

6. Enter a right parenthesis immediately followed by your APL user number.

   Here's an example of a user signing on:

```
)1234
```

The terminal should respond with a message similar to the one below, stating when that user number was last used and the user's name.

```
YORK APL  LAST USED 12.44/ 72.220/JONES
```

If this message is not received, but instead a response of *LOCKED* or *NUMBER NOT IN SYSTEM* or *NUMBER IN USE* is typed, see the APL Coordinator.

Signing Off

To terminate an APL session, the following is typed:

```
)OFF    (followed by pressing "RETURN")
```

The "RETURN" key must be pressed after each instruction is entered. It signals to the computer to start evaluating what was typed.

A message, like the one below, will be typed by the computer in reply to the *)OFF* command.

```
025  12.52.04  09/08/72
CONNECTED     0.23.15  TO DATE    25.42.41
CPU TIME      0.00.21  TO DATE     0.09.33
```

The first line printed contains the terminal number, the time of day, and the date in the form MMDDYY. The next line is a report of how long the user was connected to the APL system during this session and how much time he's logged so far this month. The last line indicates how much of the computer's time was used during the session. All four sets of figures in the last two lines are in hours, minutes and seconds. They are reset at the beginning of each month.

Along with the *)OFF* command is an option that will protect a user's number from unauthorized use. He can include a "lock code" in the Sign Off command that will be required each time he signs on from then on. Here is an example of a user signing off with a lock code of *BERT* :

```
)OFF:BERT
```

To add a lock code to a user's number, a colon plus any combination of up to 8 symbols and characters follows the *)OFF* command.

Now, if the user tries to sign on without it, this will happen:

```
      )1234
LOCKED
```

Here it is again with the lock code included:

```
      )1234:BERT
YORK APL  LAST USED 15.32/ 72.221/JONES
```

As mentioned earlier, this lock code is required <u>each time</u> the user wishes to use APL. He is able to change it to something else anytime he wishes, or he may even drop the lock code from his number.

The procedure for changing the lock code is the same as the initial addition. The existing code is ignored while the new one is added as previously described.

initial addition	*)OFF:BERT*
change	*)OFF:JONES*

To erase the lock code from the user number, the Sign Off command is typed in as usual, followed by a colon, then the "RETURN" key is immediately pressed.

)OFF: (press "RETURN")

Chapter Two

ONCE SIGNED ON

As soon as the Sign On procedure is completed, a section of the computer's internal storage (or "memory") is made available to the user. This section of storage is called the "Active Workspace" since this is where the user performs all his APL activities. Programs may be created in this area along with any calculations the user may want to do. All his activities take place in either of two modes: one is called Immediate Execution or Calculator Mode and the other is called Definition Mode.

When a user first signs onto APL, he is issued a clear Active Workspace and is placed into Calculator Mode. This means that every statement typed at the terminal is immediately evaluated by the computer as soon as the "RETURN" key is pressed and the result is then printed out at the terminal. Here are some examples.

```
      3+8      (press "RETURN")
11

      107+66   (press "RETURN")
173
```

Notice that the typing element indents 6 spaces before the user is allowed to enter any input.

Unlike some other computer languages, there are no restrictions on calculations involving both integers and real numbers.

```
      47+13.5  (press "RETURN")
60.5
```

APL makes a distinction between a minus operation and a negative number by employing two different symbols. The minus sign, (uppershift plus sign (+) on the APL keyboard), is situated at the mid-point of the number, whereas the symbol to indicate a negative value, (uppershift 2), is placed level with the top of the number.

```
      11-7   (press "RETURN")
4

      11-15  (press "RETURN")
¯4

      11-¯7  (press "RETURN")
18
```

The last expression above reads "eleven minus negative seven".

The multiplication and division key is to the right of the plus/minus key.

```
      9×6
54

      9×¯6
¯54
```

In mathematics, it is quite common to omit the multiplication sign by substituting parentheses, but this practice does not apply in APL. The user must be specific in his operations.

```
      9(5)
9(5)
 ? SYNTAX ERROR
```

However, an error of this sort causes no harm. The user can either re-enter the same statement, employing the proper syntax, or type in some other expression, ignoring the computer's answer to the mistake.

```
      9×5
45

      12÷4
3
```

One thing that is not allowed in APL is having zero as a divisor.

```
      6÷0
6÷0
 ? DIVISION BY ZERO
```

All calculations are carried out to approximately sixteen positions, then rounded to the first ten significant digits for printout. All leading zeros are suppressed.

```
      8÷3
2.666666667
```

Later, it will be seen how the number of digits printed may be varied by the user from 1 to 16.

Error Correction

```
      7×6
42
```

Suppose, in the above example, the user meant to add six to seven but pressed the wrong key by accident. If the mistake is noticed before the "RETURN" key is pressed, it can be corrected by pressing the "BACKSPACE" key (above the "RETURN" key) as many times as is required until the typing element or typeball is directly over the error. Then the "ATTN" key is pressed. This signals to the system to ignore whatever was typed at that point and everything to its right. The correction would read as follows:

```
            Ball here before error detected
      7×6
       ^    Backspaced to here, then "ATTN" hit
       +6   Caret printed by computer

          Correct character and rest of line typed in
          by user
```

When the "ATTN" key is pressed, the computer causes the terminal to line-feed and type out the symbol ^, called the Caret symbol. The computer then evokes another line-feed and waits for the corrected input. In this case, it's +6.

If the "RETURN" key was pressed before the error was noticed, the operation would be carried out as entered, forcing the user to re-enter the statement with the necessary corrections to get the answer he initially wanted.

If the statement 7×6 were typed and the user noticed his error in time and backspaced to the appropriate position, but forgot to press the "ATTN" key and just typed in the correction, this is what would happen:

```
      7*6
7*6
 ? CHARACTER ERROR
```

* is not a valid APL character and the computer acknowledges this. The question mark preceding the error message is printed under the error.

The "ATTN" key is also convenient when a user wishes to terminate a lengthy printout. He need only press this key to stop the printout and return the system back to Calculator Mode.

Order Of Execution

When solving mathematical equations, certain operations are performed before others. For instance, statements involving exponential operations are done before any multiplication and division, which are always carried out before addition and subtraction. This hierarchy of execution does not exist in APL. Here, the order of execution is simply from right to left.

```
      2×7+3
20

      18÷9-5
4.5
```

APL subtracts 5 from 9 to obtain a result of 4. It then divides 18 by 4 to arrive at the answer 4.5. The only exception to this rule of right to left sequence of execution is in the use of parentheses.

```
      (2×7)+3
17

      (18÷9)-5
¯3
```

The operations enclosed in parentheses are evaluated first to produce a result which is then used as input to the operations outside the parentheses. Execution time is slowed by the use of parentheses because the system must interrupt its regular order of execution to evaluate the contents of the parentheses before it can continue. However, they can be eliminated in most cases if the user is aware of the order of execution and types his statements accordingly.

```
      3+2×7
17
```

One stipulation with using parentheses is that there must always be an equal number of pairs in each expression. Here is an example where there's not:

```
      2+(7-3))÷10
2+(7-3))÷10
  ? UNBALANCED PARENS
```

Comment Statements

The only statement that is not immediately executed by the computer while it is in Calculator Mode is one that is preceded by the Lamp symbol ⍝.

```
      ⍝ THIS STATEMENT ISN'T EXECUTED
```

The Lamp symbol (uppershift *C* overstruck with uppershift *J*) indicates to the computer that the following characters are not to be executed. The computer produces no response to the statement. The Lamp symbol is typed in as either ∩ backspace ∘ or ∘ backspace ∩. APL accepts typed in statements exactly as they appear to the user. This is called "visual fidelity".

Variables

Operations in APL can also be performed with variables.

```
      A←20
```

The left pointing arrow (←) performs the function of assigning or specifying values to variables. In the above operation, *A* was assigned the value 20. Below, *A* is displayed and used in an operation.

```
      A
20

      A+12.5
32.5
```

Here are some more examples of variables:

```
      B←12.5

      A+B
32.5

      SUM←A+B

      SUM
32.5

      SUM-15
17.5

      A
20
```

To list the names of the variables in the Active Workspace, the system command)*VARS* is typed.

```
      )VARS
A         B          SUM
```

There are 3 variables in the Active Workspace.

```
      A
20
```

A contains the value 20. To give *A* a new value, the user re-assigns *A* the new desired value.

```
      A←6.7

      A
6.7

      A+B
19.2
```

To increase the number of characters that represent variables, the underscore symbol, _ (uppershift *F*) can be employed.

```
      A̲←¯4
```

A̲ is typed *A* backspace _. This new variable, called *A* underscored, is completely separate from the variable *A* already in the Active Workspace.

```
      A
6.7

      A̲
¯4
```

Here is a list of the valid characters that can be used in variable names:

```
      ABCDEFGHIJKLMNOPQRSTUVWXYZ
      A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲
      ∆⍙0123456789
```

Variable names may consist of any number of characters, but the first 8 characters of each name must be unique. Variable names may also contain numbers, but the first character of the variable name must be either a letter, a ∆ (uppershift *H*) or ⍙.

```
      1NUM←26.75
1NUM←26.75
? SYNTAX ERROR

      NUM1←26.75

      NUM1-6
20.75
```

Variables need not represent only numerical values. They may contain literal data as well.

```
      LETTER1←'A'

      LETTER1
A
```

Literal data are identified by being enclosed between a pair of single quotation marks (uppershift *K*).

```
      LETTER2←'B'

      LETTER1+LETTER2
LETTER1+LETTER2
       ? DOMAIN ERROR
```

Obviously, literals cannot be arguments to some numerical operation. Although the domain of the + operation is limited to numerical values only, some APL functions, such as comparisons and other logical operations, can be performed on literal data. How these are done will be discussed later.

Arrays

So far, all the examples have shown one value performing some operation on another. This could become quite time consuming and very confusing if there were several values to be computed. If the value 5 were to be added to the numbers 6 and 9, two lines of operations would be required to carry out this task.

```
      5+6
11

      5+9
14
```

Another way of doing this is as follows:

```
      5+6 9
11  14
```

The numbers 6 and 9 are called elements. Together, they are called a vector. A vector in APL is defined as a string or chain of elements. 5 is called a 1-element vector. To create a vector, a space is placed between each element, if there is more than one. Here are some examples of functions employing vectors as arguments:

```
      27 22-2
25  20

      A←6 7.5 ¯3

      2×A
12  15  ¯6
```

The vector 27 22 has a length of 2 and *A* has a length of 3. "Dimension" is another word that could be used to express the length. One could say that 27 22 and *A* have dimension values of 2 and 3 respectively, meaning there are 2 elements in 27 22 and 3 elements in *A*.

Literal vectors are also allowed.

```
      'BOB'
BOB

      NAME←'BILL'

      NAME
BILL
```

BOB and *BILL* are 3 and 4 element vectors respectively. Notice that the quotation marks are always the first and last characters typed when defining literal data. This can be very important, especially if one of them is accidently omitted. Here is an example of what happens if one of the quotes is not entered before the "RETURN" key is pressed.

```
      WORD←'HELLO
STOP
HELP
)OFF
'
```

After the word *HELLO* was typed and the "RETURN" key pressed, the typing element returned to position zero of the carriage and just "twitched". No matter what else was typed, APL responded the same way until a second quote was entered. So now *WORD* contains the following:

```
      WORD
HELLO
STOP
HELP
)OFF
```

The problem of entering an odd number of quotes is usually experienced when trying to enter a quote as part of the data. The way a literal quote is made part of the text is as follows:

```
      X←'HAVEN''T'

      X
HAVEN'T
```

Arrays with Rank Greater than 1

Apart from just having vectors, it is also possible to have such things as matrices and multidimensional arrays. A vector has only one value to represent its dimension. The vectors 27 22 and *A* used in the previous examples had dimension values of 2 and 3 respectively. Because only one value is used to express its dimension, a vector is said to be an array of rank 1. Arrays can be created to any rank as long as they are small enough to fit into the Active Workspace. Here are some examples of arrays with more than one dimension.

```
      D
10    5    7    4
 8    9   12    2
 1    3    6   11
```

D is a 2-dimensional array, usually called a matrix. It has 3 rows and 4 columns.

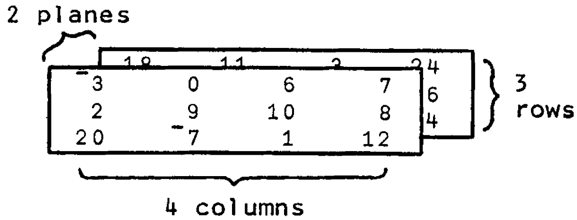
```
      100+D
110   105   107   104
108   109   112   102
101   103   106   111
```

When a single value is added to an array, it is in fact added to each element of the array.

```
      F
¯3    0    6    7
 2    9   10    8
20   ¯7    1   12

18   11    3   24
17   14    5   86
 5   23   22   64
```

F is an array of rank 3 consisting of 2 planes, each plane containing 3 rows and 4 columns. Because the terminal is unable to print *F* in its 3-dimensional form, it distinguishes the two planes by leaving a blank line between them. But, when performing calculations involving *F*, it should be thought of in the format expressed on the next page.



Operations involving arrays are very simple to perform, no matter what their dimension.

```
      F-2
¯5    ¯2     4     5
 0     7     8     6
18    ¯9    ¯1    10

16     9     1    22
15    12     3    84
 3    21    20    62
```

Like vectors, there may also be literal arrays.

```
      NAMES
DICK
BILL
GORD
```

This literal matrix has a rank of 2 consisting of 3 rows and 4 columns. The creation of arrays with varying ranks or dimensions, such as *D*, *F* and *NAMES*, and the determination of their ranks, will be discussed later.

Chapter Three

PRIMITIVE FUNCTIONS

The term "Primitive Functions" refers to the operators of the APL system such as + - × ÷. They are called "primitive" because they are predefined by the APL system and therefore do not have to be created by the user each time he wishes to use them. The word "functions" best describes these operators because, when using them, they must be accompanied by at least one argument. Therefore, the whole expression is thought of as a function that produces a result. Every primitive APL function is either Monadic or Dyadic.

Monadic functions consist of one operator and <u>only one argument</u>.

```
operator
   ↓
   +  6
      ↑
   argument
```

The argument of every monadic function must always be situated to the right of the operator.

Dyadic functions involve an operator and <u>two arguments</u>.

```
    operator
        ↓
    8   +   9
    ↑       ↑
   left   right
argument  argument
```

An argument must be on <u>each side</u> of the operator to produce a Dyadic Function.

Types Of Functions

The primitive APL functions are divided into 2 different groups; scalar and mixed.

Scalar Functions

The term "scalar" refers to an individual number or value, whereas a vector means a string or chain of numbers or values. In York APL, an element of a vector is a scalar and a scalar is a 1-element vector. The term "Scalar Functions" best describes the following functions because they operate on a one-for-one basis.

```
      2+8
10

      6 9-8 2
¯2  7
```

Here, the definition becomes a little clearer. The operation is on an element-for-element basis or parallel processing.

```
      1.5  3×6
9  18
```

In the above example, the right argument consists of only one number so it is repeated as many times as there are elements in the left argument before the operation takes place. The operation is the same as this:

```
      1.5  3×6  6
9  18
```

The number of elements contained in the result is equal to the number of elements in the longer of the two arguments. If the arguments both contain more than one element, they must be of equal length.

```
      16 10÷4 8 5
16 10÷4 8 5
     ? LENGTH ERROR
```

This operation failed because the system didn't know which numbers of the right argument were to be divided into which numbers of the left argument.

```
      8 16 10÷4 8 5
2  2  2
```

Some Monadic Uses

So far, all the examples have shown the addition, subtraction, multiplication and division functions used dyadically. They can also be used monadically.

```
      +2
2

      +6 ¯7 0
6  ¯7  0
```

The above expressions perform the same way as the dyadic Plus function when a zero is the left argument.

The monadic minus operator changes positive numbers to negatives and negative numbers to positives. Because zeros are defined as being neither positive nor negative, there is no sign to change.

```
      -2
¯2

      --2
2

      -6 ¯7 0
¯6  7  0
```

When the multiplication operator is used monadically, the sign(s) of its argument are determined and a 1 is printed to indicate a positive value, 0 for a zero value, and ¯1 for a negative value. This is called the Signum function.

```
      ×6 ¯7 0
1 ¯1 0

      ×¯27 .1 ¯1.1
¯1 1 ¯1
```

The division operator produces the reciprocal of its argument just as the dyadic divide function does when its left argument is a one.

```
      ÷5
0.2

      ÷÷5
5

      ÷1 2 4
1 0.5 0.25
```

Chapter Four

MORE SCALAR FUNCTIONS

Exponential

When used monadically, the exponential symbol, * (uppershift *P*), raises e, the base of the natural logarithm, 2.71828... to the power of the right argument.

```
      *1
2.718281828

      *¯3 .5
0.04978706837  1.648721271
```

Exponentiation

In mathematics, a number raised to a power is written as $\text{number}^{\text{power}}$. For example, the square of three is written as 3^2. In APL it is written as 3*2.

```
      3*2
9

      2 3*4 3
16  27
```

Taking the square root of a number is the same as raising it to the power of 0.5.

```
      64*0.5
8

      25 16*.5
5  4
```

A number raised to a negative power is equivalent to the reciprocal of the number raised to its positive power. For example, 5*¯2 is the same as ÷5*2.

```
      5*¯2
0.04

      ÷5*2
0.04
```

What happens when a value is raised to a fairly high power?

```
      100*6
1E12
```

The *E* is interpreted as meaning "times 10 to the power of". The number above when written in long form, would look like this:

1,000,000,000,000

which is very hard to read and could easily result in errors if it had to be typed in this fashion.

Numbers may also be taken to very small powers.

```
      .01*6
1E¯12
```

Of course there's a limit to the size of number that can be created.

```
      100*100
100*100
   ? NUMBER TOO BIG
```

The largest and the smallest numbers possible are listed in the last chapter.

Values containing *E*'s may also be used in calculations.

```
      2E1+4
24

      4+2E2
204

      4+2E¯2
4.02
```

Natural Logarithm - Logarithm

The base of the natural logarithm is e or 2.718281828... The natural logarithm of the value 10 would be written as $\log_e 10$ and read as "log 10, base e".

```
      ⍟10
2.302585093

      ⍟100 20
4.605170186  2.995732274

      *⍟100 20
100  20
```

The natural log function is the inverse of the exponential function, thus they negate each other.

The symbol ⍟ does not appear on the APL keyboard. It is a combination of ○ (uppershift *O*) overstruck with * (uppershift *P*). This is typed as ○ backspace *.

```
      2⍟8
3
```

The above expression is read as "log 8, base 2" or "log 8".

PI Times - Trigonometric Functions

The uppershift *O* symbol, ○, has interesting characteristics. When used monadically, in the form ○*A*, it means PI times *A*, (PI representing 3.14519...).

```
      ○1
3.141592654
```

If 1 radian = $\frac{180 \text{ degrees}}{\text{PI}}$, how many radians are there in 30 degrees?

```
      (30×○1)÷180
0.5235987756
```

In its dyadic use, the large circular symbol performs various trigonometric functions, depending on the value of its left argument. Here is a table of all the dyadic operations possible with this symbol:

SYNTAX	FUNCTION
7○*A*	hyperbolic tangent of *A* (tanh *A*)
6○*A*	hyperbolic cosine of *A* (cosh *A*)
5○*A*	hyperbolic sine of *A* (sinh *A*)
4○*A*	(1+*A*⋆2)⋆0.5
3○*A*	tangent *A*
2○*A*	cosine *A*
1○*A*	sine *A*
0○*A*	(1-*A*⋆2)⋆0.5
¯1○*A*	arcsin *A*
¯2○*A*	arccos *A*
¯3○*A*	arctan *A*
¯4○*A*	(¯1+*A*⋆2)⋆0.5
¯5○*A*	arcsinh *A*
¯6○*A*	arccosh *A*
¯7○*A*	arctanh *A*

For all the trigonometric functions, *A* is in radians and the left argument is an integer from 7 to ¯7.

What is the sine of 3 radians?

```
      1○3
0.1411200081
```

Show that $\sin^2\theta+\cos^2\theta = 1$. (Give θ the value 2 radians.)

```
      (((1○2)*2)+(2○2)*2)
1
```

Ceiling - Maximum And Floor - Minimum

The two functions ⌈ and ⌊ (uppershift *S* and *D* respectively) are very similar, so they will be discussed here together.

Ceiling - Floor

```
      ⌈2 2.6
2  3

      ⌈.01 ¯6.7
1  ¯6

      ⌊2 2.6
2  2

      ⌊.01 ¯6.7
0  ¯7
```

Monadically, the Ceiling function, ⌈, rounds the value(s) of its argument to its next highest integer and the Floor function, ⌊, rounds its argument down. If the argument is already an integer, no rounding takes place. An application for these functions would be in the rounding of numbers to their nearest whole numbers. In the case of numbers containing .5, it would depend on the user whether to round them up or down. If numbers ending in .5 were rounded up, 0.5 would be added to the numbers before the floor operation.

```
      X←4.2 7.6 5.5 3 6.69

      ⌊0.5+X
4  8  6  3  7
```

To round the .5's down, 0.5 is subtracted from the numbers, before the ceiling operation.

```
      ⌈X-0.5
4  8  5  3  7
```

Maximum - Minimum

```
      4⌈6
6

      6⌈4 7 5
6  7  6

      4⌊6
4

      6⌊4 7 5
4  6  5
```

Dyadically, the Maximum function, ⌈, determines which argument is greater. The opposite of this, the Minimum function, ⌊, determines which argument is of lesser value.

Factorial - Combination

How many different ways can 4 items be arranged? In mathematics, the expression to represent the equation is 4! meaning 4×3×2×1 which is equal to 24. In APL, it is written as !4. The Factorial function, !, is created by overstriking the ' (uppershift *K*) with the period or decimal point.

```
      !4
24

      !5
120

      !6  3
720  6
```

Calculating the number of permutations of "n" different things, taking "r" at a time, without repetitions using the formula

$$\frac{n!}{(n-r)!}$$

would be expressed in APL by the following algorithm:

```
      (!N)÷!N-R
```

How many words can be formed from the letters of the word "computer", taking 6 letters at a time? (Obviously, most of the resulting "words" will not be part of the English language).

```
      N←8
      R←6

      (!N)÷!N-R
20160
```

The difference between a permutation and a combination is that in a permutation, order is taken into account, while in a combination, it is not. The equation for calculating the number of combinations of ways in which objects can be selected from a group without regard to their order is as follows:

$$\frac{n!}{r!(n-r)!}$$

How many ways can 2 marbles be selected from a population of 6? The APL algorithm for solving this problem is as follows:

```
      2!6
15
```

Where n=6 and r=2.

Absolute Value - Residue

To find the absolute value of the variable *X*, the mathematical notation is $|X|$. In APL, it is simply |*X*.

```
      |8 ¯7 0 ¯2.5
8  7  0  2.5
```

```
      |6×¯7
42

      8+|¯4
12

      A
¯6       2
¯5.1    ¯0.1

      |A
6       2
5.1     0.1
```

But when used dyadically, the function performs quite differently.

```
      3|8
2

      5|6 10 12 124
1  0  2  4
```

In the above two operations, the right argument is divided by the left to find the residue or remainder.

```
      3|¯4
2
```

When the right argument is negative, the left argument is added to the right until their sum is a positive value. It is this positive value which is then printed.

Relational Functions

The APL language has six relational functions, < ≤ = ≥ > ≠, which are uppershift 3 through 8 respectively. They represent comparisons such as less than, less than or equal to, equal to, etc. All these functions are dyadic. The result from each of these is always either 1 or 0. The 1 represents "yes" or "true" while the 0 means "no" or "false".

Here is a list of the six different functions and their meaning:

Function	Meaning
<	Les′ than
≤	Less than or equal to
=	Equal to
≥	Greater than or equal to
>	Greater than
≠	Not equal to

Here is how they are used:

```
      47<70
1

      47>70 50 30
0  0  1

      3≤6 2
1  0

      3≤3
1

      2 4=4 2
0  0

      2 4=2 4
1  1

      2 6 5≥4
0  1  1

      4 10 0>2 7 ¯4
1  1  1

      6≠6
0
      6≠6+2
1

      'A'='A'
1

      'A'='B'
0
```

Literal arguments are allowed for all relational functions.

The relationship between characters other than alphabetics may be found by treating them as literals also.

```
      '+'≠'÷'
1
```

Logical Functions

Logical functions are similar to the relational functions in that all, with the exception of one, are dyadic and that they too produce only 1 or 0 results, indicating a "yes" or "no" reply. The part where they differ from the relational functions is that they accept only 1's and 0's as arguments.

Or

```
      1∨0
1

      0 1∨1 1
1 1

      0 0∨0 1
0 1
```

Either corresponding argument of the Or function, ∨, (uppershift 9), must contain a 1 before the result is one.

And

The And function, ∧, (uppershift 0), expects both corresponding arguments to be equal to 1 before a 1 is returned.

```
      1∧0 1 0
0  1  0

      0 1∧1 1
0  1

      2∧0
2∧0
 ? DOMAIN ERROR
```

As stated earlier, the values of the arguments are limited to either 1's or 0's. The left argument is outside the "domain" of the And function in the above example.

Nor

The Nor function produces the opposite result to that of Or. It is created by overstriking the ∨ with the tilde, ~, (uppershift *T*).

```
      1⍱0
0

      0⍱0
1

      0⍱1 0 1
0  1  0
```

Nand

The Nand function is the inverse of And. It is produced by overstriking the ∧ and the ~.

```
      1⍲1
0

      1⍲1 0 1
0  1  0

      0⍲0
1
```

Not

The one scalar function that may only be used monadically is the Not function, ~.

```
      ~1
0

      ~0
1

      ~0 1 0 0 1
1  0  1  1  0
```

The Not function performs a logical negation on its argument. If its argument is a 1, its result is a 0, and vice versa.

Chapter Five

REDUCTIONS

Previously, it was shown how to perform operations on vectors and arrays by parallel processing or on a one-for-one basis. Such things as adding the elements of one vector to the corresponding elements of another vector is a simple operation. But what about summing the elements of a vector, or the columns and rows of an array? This could prove to be quite tedious using methods discussed earlier. To eliminate this laborious task, APL has incorporated the solidus symbol (/) to aid in performing operations on the individual members of vectors and arrays.

```
      +/2 6 7 9
24

      2+6+7+9
24
```

The "Plus-Reduction" function above operates the same way as if plus signs had been inserted between each pair of elements of the vector. Here is how it works:

```
      ← order of execution

      +/2   6   7 + 9

            6  +  16

       2  +  22

            24
```

The order of execution is not important for the "Plus-Reduction" function but it is for the "Minus-Reduction" function.

```
      -/6 3 2 4
1
```

```
       ← order of execution

      -/6   3   2 - 4
            3 - ¯2
        6 - 5
            1
```

Routines involving "Minus-Reductions" and "Divide-Reductions" should be fully tested before they are used.

Here are examples of some of the other Scalar functions used with the Solidus symbol:

```
      ×/2 4 3 5
120

      ⌈/2 ¯6 4 0
4

      ∨/1 0 0
1

      ∧/1 0 0
0

      ≥/8 6
1
```

The Relational functions along with the Reduction symbol can also be used with literals as arguments.

```
      =/'AB'
0

      ≠/'AB'
1
```

When performing Reduction operations on arrays of rank greater than one, the user must specify along which coordinate the operation is to apply. If none is indicated, the system assumes the last coordinate.

```
      M
2     3     1
8     9     7

      +/M
6   24
```

M is a matrix with 2 rows and 3 columns. The above "Plus-Reduction" of *M* summed along the last coordinate, the columns.

```
      +/[2]M
6   24
```

The last operation summed along the second coordinate of *M*, which, in this case, represents the columns because *M* has only two coordinates. Therefore, it performed the same as the previous example.

```
      +/[1]M
10   12   8
```

Summing along the first coordinate of the 2-dimensional matrix *M* adds the corresponding values found in each row together. These distinctions are not required with vectors because they have only one dimension so that +/ or +/[1] means the same thing to vectors.

Another way to sum the rows of *M* is as follows:

```
      +⌿M
10   12   8
```

⌿ is the solidus overstruck with the minus sign. When this is used, the system always carries out the operation along the first coordinate of its argument. If it is a matrix, the operation is along the rows; if it's a 3-dimensional array, each plane is evaluated.

```
      N
0  1  0
1  0  0

1  1  1
0  0  0

1  0  1
1  1  1
```

N is a 3-dimensional array with 3 planes, each consisting of 2 rows and 3 columns.

```
      ∨/N
1  1
1  0
1  1
```

Above, the "Or-Reduction" of *N* shows that all the columns in row 2 of plane 2 are equal to zero. All the rest have at least one 1 in them.

```
      ∨⌿N
1  1  1
1  1  1
```

Each plane of *N* has at least one 1 in it.

```
      ∧/[2]N
0  0  0
0  0  0
1  0  1
```

Just the rows of columns 1 and 3 of plane 3 contain all 1's.

Chapter Six

INNER AND OUTER PRODUCTS

Generalized Inner Product

How many elements are in the vector *X*? This could easily be determined by two means. First, by asking "does *X* equal *X*" the result has to be a vector of 1's as each element of *X* is obviously equal to itself. And by summing up all these 1's, the user could easily find out how many elements there are in *X*.

```
      X
1  2  3  4  5  6

      X=X
1  1  1  1  1  1

      +/X=X
6
```

Or, it could be written in the following format:

```
      X+.=X
6
```

+.= is read as "plus dot equal".

The last two illustrations work in exactly the same manner. As in Reduction operations, all Scalar functions, except the Not function, can be used in the Generalized Inner Product format.

```
      N←4 2 3

      M←3 5 1

      O←N×M
      O
12  10  3

      +/O
25

      N+.×M
25

      N×.+M
196

      N⌈.⌈M
5

      N+.+M
18
```

If both arguments are vectors, the result is a single value. But, if one argument is a vector containing more than one number and the other argument is an array, certain restrictions and procedures are imposed. For example, if the left argument is a 3-element vector and the right argument is a matrix, the matrix must have three rows. The general rule states that the dimension of the last coordinate of the left argument must be equal to the first coordinate of the right argument.

```
      K
10    11
12    13
14    15

      L
1  2  3

      L+.×K
76  82
```

The solution to this last example was accomplished in the following manner:

```
          (+/1 2 3×10 12 14)   (+/1 2 3×11 13 15)
```

If a 2-dimensional array is the left argument, and a vector containing more than one element is the right, the length of the vector must be equal to the dimension of the last coordinate of the right argument.

```
      K+.×1 2 3
K+.×1 2 3
 ? LENGTH ERROR

      K+.×2 3
53  63  73
```

The dimensions of the result are a combination of all but the last coordinate of the left argument and all but the first coordinate of the right argument. In the above example, *K* is a three by two matrix operating on a 2-element vector. Therefore, the result is a 3-element vector. If the left argument was a 4 2 matrix and the right was a 2 3 matrix, the coordinates of the result would by 4 by 3.

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

      B
2     3     2
1     0    ¯1

      A+.×B
 4     3     0
10     9     2
16    15     4
22    21     6
```

This result was obtained by the following calculations:

	column 1	column 2	column 3
row 1	+/1 2×2 1	+/1 2×3 0	+/1 2×2 ¯1
row 2	+/3 4×2 1	+/3 4×3 0	+/3 4×2 ¯1
row 3	+/5 6×2 1	+/5 6×3 0	+/5 6×2 ¯1
row 4	+/7 8×2 1	+/7 8×3 0	+/7 8×2 ¯1

If *A* were a 4 by 3 by 2 array and *B* was 2 by 6 by 7, the result of *A*+.×*B* would be a 4-dimensional array with coordinates 4 by 3 by 6 by 7.

Generalized Outer Product

The purpose of the Generalized Outer Product is to allow every element of the right argument to perform a specified operation on every element of the left argument.

```
      2 4∘.×3 5
 6    10
12    20
```

Here, every element of the left argument was multiplied by every element of the right. The ∘.f, where "f" is any Scalar function other than the Not function, represents the Generalized Outer Product function. The above operation reads as "two, four, null dot times three, five", and is represented in the following table:

×	3	5
2	6	10
4	12	20

Unlike the Inner Product function, only one Scalar operation can take place at one time and there are no dimension restrictions placed on the arguments.

```
      'CAT'∘.='CAT'
1  0  0
0  1  0
0  0  1
```

The coordinates of the result are a combination of the coordinates of both arguments. The result's dimension is the sum of the dimensions of the arguments. If both arguments are 3-element vectors, which have only one dimension each, the result is a 3 by 3 two dimensional array.

```
      1 2 3 4∘.≥1 2 3 4
1  0  0  0
1  1  0  0
1  1  1  0
1  1  1  1
```

```
      C
¯3    0    2    1
 6    4    3    7
      C∘.+5 ¯1
 2    ¯4
 5    ¯1
 7     1
 6     0

11     5
 9     3
 8     2
12     6
```

The Null symbol, ∘, when used in this context, does not perform any real function other than to indicate to the system that an Outer Product operation is being carried out. When used by itself, the Null symbol does perform an operation which will be discussed later.

Chapter Seven

COMPRESSION AND EXPANSION

Compression

To eliminate some, none, or all the elements of an array, the expression stated in the following format is used:

```
V/[I]A
```

V represents a logical vector of 1's and 0's; *A* is an array; and *I* indicates along which coordinate the compression is to be applied. The length of the left argument must be equal to the length of the *I*th coordinate of the right argument, unless either of the two arguments contains only one element. In that case, the argument containing the one element would be extended until it was the same length as the other argument.

```
      1 0 1/6 2 4
6  4
```

In the above example, only the first and the third elements of the right argument were selected. When dealing with arrays of rank greater than 1, entire planes, rows, and columns may be omitted from the result.

```
      M
1     2     3     4
5     6     7     8
9    10    11    12

      1 0 0 1/M
1     4
5     8
9    12
```

```
      1 0 0 1/[2]M
1     4
5     8
9    12
```

The [2] indicates that the compression is to occur along the second dimension of *M*. In this case it's the columns or last dimension. If this pointer is omitted, the system defaults to the last dimension, as in the previous example. To eliminate the rows of a matrix, either of the following two methods could be used:

```
      1 0 1/[1]M
1     2     3     4
9    10    11    12

      1 0 1⌿M
1     2     3     4
9    10    11    12
```

⌿ refers to the first dimension of the array.

To illustrate the use of the Logical Reduction operation, suppose that, at the end of a semester, a teacher wanted to find out who, among his students, attained honours standing. He had given three tests during the term and the marks went as follows:

NAME	TEST 1	TEST 2	TEST3
Bateson	25	36	20
Atkin	17	24	18
Chapin	24	33	21
Kirby	20	25	17

He created two matrices, one called *MARKS*, containing the grades achieved by each student for each test, and one called *NAMES* containing the corresponding student's name.

```
      MARKS
25    36    20
17    24    18
24    33    21
20    25    17


      NAMES
BATESON
ATKIN
CHAPIN
KIRBY
```

The first step would be to sum the marks.

```
      SUM←+/MARKS

      SUM
81  59  78  62
```

The next step would be to find out which values of *SUM* were equal to or greater than 75. In this case, it could easily be accomplished by simply looking at the totals. But in the case of a hundred or so students, there is a chance that an honours student could be overlooked in the scanning process.

```
      HIGH←SUM≥75

      HIGH
1  0  1  0
```

The final step is to find out who got honours.

```
      HIGH/NAMES
BATESON
CHAPIN
```

This whole process could easily be done all on one line:

```
      (75≤+/MARKS)/NAMES
BATESON
CHAPIN
```

Expansion

The Expansion expression is very similar to the Compression function. It has the same properties as the Compression function except that the number of 1's contained in the logical vector left argument must be equal to the length of the right argument, except if either argument contains only one element.

```
      1 0 0 1 1\66 67 68
66  0  0  67  68

      1 1 0 0 0 0 1 1\'ABCD'
AB    CD
```

The size of the result is determined by the length of the left argument. If the right argument is numeric, zeros are used to expand the result. If a literal is situated to the right of the Expansion function, then blanks or spaces are used.

```
      X
1     2     3     4
5     6     7     8
9    10    11    12

      1 0 1 1 1\X
1     0     2     3     4
5     0     6     7     8
9     0    10    11    12

      Y
BIG
BAD
JOE

      1 1 0 1\Y
BI G
BA D
JO E
```

To increase the number of rows in *X*, the expansion is along the first coordinate.

```
      1 1 0 0 1\[1]X
1     2     3     4
5     6     7     8
0     0     0     0
0     0     0     0
9    10    11    12
```

Or, as with the Reduction function, there is an Expansion symbol that applies to only the first coordinate of the right argument.

```
      0 1 1 1⍀X
0     0     0     0
1     2     3     4
5     6     7     8
9    10    11    12

      1 1 0 1⍀Y
BIG
BAD

JOE
```

Primitive Function Drill

There's a drill available on most York APL systems to practice using the primitive scalar functions. The workspace containing this drill is called *APLCOURSE*.

To select the functions desired, a *Y* is typed under the appropriate ones. When the user has had enough practice, he can type in the word *STOP* or *STOPNOW* to terminate the exercise.

Here's a typical session:

```
      )LOAD APLCOURSE,7
SAVED 12.42/ 71.203/ 8768

      EASYDRILL
TYPE Y UNDER EACH FUNCTION FOR WHICH YOU WANT EXERCISE
SCALAR DYADIC FUNCTIONS
+-×÷*⌈⌊<≤=≥>≠!|∧∨⍟⍲⍱
YYYYYYYYYYYYYYYYYYYY
SCALAR MONADIC FUNCTIONS
+-×÷*⌈⌊!|~
YYYYYYYYYY
TYPE Y IF EXERCISE IN VECTORS IS DESIRED, N OTHERWISE
Y
TYPE Y IF EXERCISE IN REDUCTION IS DESIRED, N OTHERWISE
Y

                    ⌈/¯7  ¯2  8  9  9
⎕     9

                    |¯6
⎕     6

                    ⍟/10  100
⎕     3
TRY AGAIN
⎕     2

                    ÷/12  ¯6  ¯4  3  6
⎕      PLEASE
ANSWER IS 16

                    ÷/¯10  ¯10
⎕     1

                    ⌈1.75  2.45
⎕     2 3

                    ~1  0  1  0  0
⎕     STOPNOW
```

Chapter Eight

MIXED FUNCTIONS

The previous chapters dealt with the Scalar functions that are avialable on the APL system. They showed that the main characteristic of all Scalar functions is that the length of the result is in direct relationship with the length of the arguments. But there are many more functions in APL which produce results with lengths that are only remotely similar to the lengths of the arguments. These are called "mixed functions".

Index Generating - Index Of

```
      ⍳5
1  2  3  4  5

      ⍳3
1  2  3
```

The Iota operator, ⍳, (uppershift *I*), generates all the indices from 1 to *N* (*N* being the right argument).

```
      2×⍳4
2  4  6  8
```

To alter the starting point 1, a user may perform some calculation on the generated integers.

```
      ¯5+⍳10
¯4  ¯3  ¯2  ¯1  0  1  2  3  4  5

      .5-⍳3
¯0.5  ¯1.5  ¯2.5
```

Because of the limited amount of space in an Active WS, a limited amount of numbers can be generated.

```
      ⍳5000
⍳5000
? WORKSPACE FULL
```

The maximum number of integers that can be generated is 3,598.

If the right argument of the monadic iota function is zero, the result is an empty vector. Because there are no elements in an empty vector, there is nothing to print, so the typing element simply returns and indents 6 spaces. Empty vector results are denoted by the symbol ⍉.

```
      ⍳0
⍉

      2×⍳0
⍉
```

When used dyadically, the iota operator indicates where, in the vector left argument, the element(s) in the right argument are located.

```
      12 27 ¯13⍳27
2

      12 27 ¯13⍳¯13
3

      12 27 ¯13⍳6
4
```

If an element in the right argument is not present in the left, the computer returns a result equal to the length of the left argument plus 1 as seen in the last example. Because 6 is not a member of the left argument, the resultant value is 4.

This operator may also have literals as arguments, when used dyadically.

```
      'ABCD'⍳'C'
3

      'A'⍳'ABC'
1  2  2

      'ABC'⍳'AC'
1  3
```

Dimension - Restructure

```
      D←2 6 ¯4 0

      ρD
4

      ρ'HELLO'
5

      ι6
1  2  3  4  5  6

      ρι6
6
```

The Rho operator, ρ, (uppershift *R*), when used monadically, indicates how many elements are in the above vector right arguments.

```
      ρ6
1
```

The number 1, in this last example, represents a vector containing only one element.

```
      ρι0
0
```

Because the argument of the above Rho operator is an empty vector, it has a length of zero, to indicate it contains no elements.

```
      MAT1
8    ¯2    0
6     3    4

      ρMAT1
2  3
```

The variable *MAT*1 represents a 2-dimensional array consisting of two rows and three columns. The above example of the Rho function produced the vector 2 3 indicating the coordinates of the array. If the argument is a vector, the response to the Rho function is a single value indicating how many elements are in the vector. By only printing one number, it also states that its argument has only one rank, which means it's a vector. The result of the Rho operation when *MAT*1 was the right argument was a 2-element vector which said that *MAT*1 was a 2-dimensional array, or a matrix.

```
      ⍴MAT2
2  3  4

      MAT2
 8    ¯2     0     6
 3     4     2     7
20    15     9    11

24     5    17    22
18    19    10    13
23    14    18    12
```

*MAT*2 is a 3-dimensional array with 2 planes, each consisting of 3 rows and 4 columns. A way to determine the rank of variables such as *MAT*1 and *MAT*2 is to do a "Rho" of the vector produced from the first Rho operation.

```
      A←⍴MAT1

      A
2  3

      ⍴A
2
```

or just

```
      ⍴⍴MAT1
2

      ⍴⍴MAT2
3

      ⍴⍴D
1
```

D has a rank of 1, *MAT*2 has a rank of 3, and *MAT*1 has a rank of 2. Using three ⍴'s together will always produce a 1.

```
      ⍴⍴⍴D
1

      ⍴⍴⍴MAT1
1

      ⍴⍴⍴MAT2
1
```

When the Rho operator is used dyadically, in the form *AρB*, *A* determines the size and dimensions of the result and *B* contains the values for the result.

```
      4ρ1
1  1  1  1

      5ρ3 4
3  4  3  4  3

      2ρ4 5 6
4  5
```

The first example above produced a 4-element vector of all ones. The second illustration produced the resultant vector of five elements, all of which are contained in the right argument. Because the left argument asked for more numbers than were contained in the right argument, the right argument was repeated until its length equalled the value of the integer left argument. In the third example, only 2 of the 3 numbers were asked for.

```
      ι4
1  2  3  4

      2 2ρι4
1    2
3    4
```

Previously, the left argument of the dyadic Rho function contained only one integer which always produced a vector result. By placing more than one element in the left argument, results of greater dimensions can be achieved. In the last example, two values represented the left argument and the result was a 2-dimensional array. Therefore, the number of elements contained in the left argument determine the rank of the result. Here is how a 2 3 4 array containing the values 1 to 24 is created:

```
      2 3 4ρι24
 1     2     3     4
 5     6     7     8
 9    10    11    12

13    14    15    16
17    18    19    20
21    22    23    24
```

Here are some more examples:

```
       MAT1←2 3ρ8 ¯2 0 6 3 4

       MAT1
8      ¯2       0
6       3       4

       ρMAT1
2   3

       2 2ρMAT1
8      ¯2
0       6

       0ρMAT1
ƀ

       A←0ρMAT1

       ρA
0

       (ι0)=A
1
```

A is a vector with zero dimensions.

The left argument must always be either a zero or a positive integer.

```
       ¯2ρMAT1
¯2ρMAT1
  ? DOMAIN ERROR
```

Literals may also be restructured.

```
       2 5ρ'HELLOTHERE'
HELLO
THERE
```

A literal empty vector is created by typing two adjacent quotes.

```
       L←''

       L
 ƀ

       ρL
0
```

Ravel - Catenate

To ravel an array or turn it into a vector, the monadic syntax of the comma is used.

```
      MAT1
8   ¯2    0
6    3    4

      ,MAT1
8  ¯2  0  6  3  4

      ⍴MAT1
2  3

      ⍴,MAT1
6

      NAMES
BILL
AL
FRED

      ,NAMES
BILLAL  FRED

      ⍴NAMES
3  4
```

Dyadically, the comma appends the right argument to the left.

```
      3,4
3  4

      8 9,10
8  9  10

      4 6,3 7 1
4  6  3  7  1
```

This is nice to know when an application arises requiring several values to be assigned to a variable, so many as to make it impossible to type them all on one line. An easy way around this problem is to assign a small group of numbers at a time, and catenate the remaining in the following manner:

```
      NUM←6 8.5 0 ¯1 6 12 20 18 2
      NUM←NUM,4 10 1 5 11 5.2 3

      ⍴NUM
16
```

Arrays of greater rank may also be catenated.

```
      A←2 3⍴6

      B←2 2⍴3

      A,B
6    6    6    3    3
6    6    6    3    3
```

When two matrices are catenated along their second or last coordinate, (columns), the number of rows contained in each matrix must be equal. And, when the catenation occurs along the first coordinate, the number of columns in both arguments must be equal.

```
      C←3 3⍴9
      A,[1]C
6    6    6
6    6    6
9    9    9
9    9    9
9    9    9
```

The syntax for catenation is *A*,[*I*]*B* or *A*,*B*. *A*,*B* performs the catenation along the last coordinate of each argument. *I* is any integer from 1 to ⌈/(⍴⍴*A*),⍴⍴*B* and all corresponding dimensions of *A* and *B*, except the *I*th must be equal. If *A* and *B* are not of the same rank, then they can differ by only one rank. For instance, if *A* is a matrix, then *B* can be either a vector or a 3-dimensional array; if *A* is a 5-dimensional array, then *B* must be a 4- or 6-dimensional array. The only exception to this rule is if either *A* or *B* is a one element vector.

```
      B←7 7

      A,B
6    6    6    7
6    6    6    7
```

In this last example, the ⍴*B* (i.e., 2) must equal the number of rows contained in *A*. The general rule is (⍴*B*) = (*I*≠⍳⍴⍴*A*)/⍴*A*.

Here is a one element vector catenated along the first coordinate of *A*.

```
      A,[1]4
6    6    6
6    6    6
4    4    4
```

Lamination

Lamination means joining two variables along a new coordinate. The syntax for lamination is *A*,[*I*]*B* or *A*,*B*. It's almost the same as the catenation syntax except that *I* must be a real number from 0 to 1+⌈/(ρρ*A*),ρρ*B* and the expression ∧/(ρ*A*)=ρ*B* must be equal to 1, except where either *A* or *B* contains only one value. Here are some examples:

```
      A←2 3ρι6
      B←2 3ρ100+ι6

      A,[.5]B
  1      2      3
  4      5      6

101    102    103
104    105    106

      A,[1.1]B
  1      2      3
101    102    103

  4      5      6
104    105    106

      A,[2.7]B
1    101
2    102
3    103

4    104
5    105
6    106

      ρA,[2.7]B
2  3  2
```

The results are three dimensional arrays with the size and content of the last two coordinates determined by the value of *I* and the contents of the arguments. The rank of the result is always one greater than the rank of the arguments. Where *A* and *B* are matrices, the result is a 3-dimensional array. When *I* is less than 1, the right argument, *A*, makes up the first plane of the result and *B* makes up the second plane. When *I* is greater than 1 but less than 2, *A* is placed in the first row of each plane of the result and *B* is placed in the second. If *I* is greater than 2 but less than 3, the two arguments are placed in the corresponding columns of the result.

Semicolon

The semicolon performs very similar to the comma, except for two distinct differences. It always ravels its arguments and converts any numeric data into a literal string.

```
      A←;6

      A+10
A+10
 ? DOMAIN ERROR
```

A contains the character 6, not the value 6.

```
      '6'=A
1

      B←;6 10+2

      B
8  12
      ρB
5
```

B contains five characters because the spaces that separated the once numeric 8 12 are now elements of the literal vector *B*.

```
      ' '=B
0  1  1  0  0
```

When the semicolon is used dyadically, it not only turns numerical data into literals, but also performs a catenation operation with the other argument.

```
      'HE IS ';6;' YEARS OLD.'
HE IS 6 YEARS OLD.

      'THE SUM OF SIXTY-NINE AND FOUR IS ';69+4
THE SUM OF SIXTY-NINE AND FOUR IS 73
```

The semicolon is also used in indexing arrays as seen in the next chapter.

Chapter Nine

MORE MIXED FUNCTIONS

Indexing

Selecting specific elements from arrays was illustrated earlier with the use of the Logical Reduction function. The desired elements were indicated by 1's and 0's. Another method of extracting array data is by indexing the array with the actual locations of the elements. Here are some examples of various values being indexed:

```
      X←2 ¯7 0 9 3 8

      X[3]
0
```

Above, the third element of *X* is indexed. Below, the fourth element is asked for.

```
      X[4]
9
```

The indexing brackets are called a dyadic function which encloses its right argument whose value(s) are dependent on the left argument. This means that the numbers contained in the right argument must be integers whose values are within the dimensions and coordinates of the left argument.

```
          X[10]
X[10]
 ? DOMAIN ERROR
```

Because *X* does not contain ten elements, the index request cannot be executed.

Indexed variables can also be used as arguments to other operations.

```
      6+X[2]
¯1

      X[2]+X[5]
¯4
```

They may also be used as arguments to other indexing operations.

```
      X[X[1]]
¯7
```

The order of the indices dictates the order in which the result is printed.

```
      Z←'PORK LAY'

      Z[8 2 3 4 5 7 1 6]
YORK APL
```

Not only can specific elements be extracted from arrays, but they can also be replaced by other values.

```
      X[1]←20

      X
20  ¯7  0  9  3  8
```

And, as the extraction sequence depends on the arrangement of the indices, the same is true for replacement.

```
      X[1 6 3]←10 ¯8 1

      X
10  ¯7  1  9  3  ¯8
```

So far, indexing has been with only vector arguments which have only one dimension. But when indexing arrays of greater rank, how are the rows distinguished from the planes and columns? Easily, with the use of the semicolon. Here is an example of indexing the matrix *MAT*1.

```
      MAT1
8     ¯2     0
6      3     4

      ⍴MAT1
2  3
```

What is the value contained in the first row, second column of *MAT*1?

```
      MAT1[1;2]
¯2
```

The semicolon separates the coordinate values indicating to the system which value is desired.

What element is in row 2, column 1?

```
      MAT1[2;1]
6
```

The same restrictions and freedoms that apply to vector arguments are also valid for arrays of other dimensions.

```
      MAT1[6;2]
MAT1[6;2]
    ? DOMAIN ERROR

      MAT1[2 1;3]
4  0
```

The number of semicolons required to index a variable is always one less than the number of dimensions of the variable.

```
      ⍴MAT2
2  3  4

      MAT2
 8    ¯2     0     6
 3     4     2     7
20    15     9    11

24     5    17    22
19    18    10    13
23    14    18    12

      MAT2[1;2;3]
2

      MAT2[1;2;2]
4

      MAT2[1;2;1]
3
```

If no value is placed before, between, or after the semicolons, the system will produce the entire plane, row or column that was not specified.

```
      MAT2[1;2;]
3  4  2  7
```

The above example asked for all the columns of row 2 of plane 1. The result is a vector.

```
      MAT2[1;;]
 8     ¯2     0     6
 3      4     2     7
20     15     9    11
```

This last illustration called for all the rows and columns of the first plane.

```
      ⍴MAT2[1;;]
3  4
```

Because only one plane and all the rows and columns of that plane were wanted, the result is a 2-dimensional array with coordinates equalling that of the rows and columns of *MAT*2. The result takes on the dimensions of the portion of the argument indexed. If only one column or one row is indexed, the result is a vector. But if more than one row or column is requested, the dimensions and coordinates are established accordingly.

If an array is to be indexed and one of the coordinates has to be calculated, the calculation must be enclosed in parentheses if it precedes a semicolon.

```
      MAT1[(1+1);]
6  3  4

      MAT1[;1+1]
¯2  3
```

Only the last coordinate is exempt from this rule.

Something that can be tried, though not highy recommended without first learning the indexing operation thoroughly, is omitting the semicolons when indexing an array. To determine the value contained in the first row, second column of *MAT*1, the following would be typed:

```
      MAT1[1 2]
¯2
```

The values at coordinates 1 1 and 2 3 of *MAT*1 could be obtained by typing the following:

```
      MAT1[1 1 2 3]
8  4
```

When not employing the semicolon, there must be a value stated for each coordinate of the array being indexed. Here is an example where there's not:

```
      MAT1[2]
MAT1[2]
    ? RANK ERROR
```

Because only one value was contained in the brackets, and no semicolon was included, the above operation failed. It would have worked if the left argument were a vector which has a rank of one but *MAT*1 has a rank of two.

This form of indexing becomes even more complex as the number of dimensions of the argument increases.

Grade Up

```
      ⍋8 ¯2 6 0
2  4  3  1
```

The Grade Up function ⍋ (uppershift *H* overstruck with uppershift *M*), in the above example, determined that its argument could be rearranged in ascending sequence if the second element came first, followed by element number 4, then 3, then 1. The result is a list of indices which, if used to index the argument, would print out the argument in ascending order.

```
      8 ¯2 6 0[2 4 3 1]
¯2  0  6  8
```

Here are some more examples:

```
      ⍋¯16 ¯17
2  1

      ⍋'CDBA'
4  3  1  2

      'CDBA'[4 3 1 2]
ABCD
```

Using this function, the problem of sorting both numbers and characters is greatly simplified.

```
      X←'CDBA'

      X[⍋X]
ABCD
```

Grade Down

```
      ⍒8 ¯2 6 0
1  3  4  2

      A←8 ¯2 6 0

      A[⍒A]
8  6  0  ¯2
```

The Grade Down function ⍒ (uppershift *G* overstruck with uppershift *M*) is the inverse of the Grade Up function.

```
      ⍒'ABACK'
5  4  2  1  3
```

If the left argument contains elements of equal value, the operator ranks them according to their position in the argument.

```
      ⍒3  3  3
1  2  3
```

Take

```
      3↑1 2 3 4 5
1  2  3

      8↑1 2 3 4 5
1  2  3  4  5  0  0  0

      ¯3↑1 2 3 4 5
3  4  5

      ¯8↑1 2 3 4 5
0  0  0  1  2  3  4  5
```

In the expression *X*↑*Y*, if the left argument *X* is a positive integer, the result is the first *X* elements of *Y*. If *Y* does not contain *X* elements, then zeros or blanks, depending on whether *Y* is numeric or literal, are appended to the result to give it a dimension of *X*.

If *X* is a negative integer, the result is the last *X* elements of *Y*. If *Y*'s dimension is less than the absolute value of *X*, the same rule applies as for the positive *X*.

```
      MAT1
8    ¯2     0
6     3     4

      2 2↑MAT1
8    ¯2
6     3
```

The result above is the first two rows and the first two columns of *MAT*1.

```
      4 4↑MAT1
8    ¯2     0     0
6     3     4     0
0     0     0     0
0     0     0     0

      ¯6↑'TOM'
   TOM
```

In this last example, three blanks were placed before the word *TOM*.

Drop

The Drop function ↓, (uppershift *U*), is the inverse of the Take function.

```
      2↓'ABCDE'
CDE

      ¯2↓'ABCDE'
ABC

      5↓'ABCDE'
b̸
```

In the expression *X*↓*Y*, if *X* is a positive integer, the result is the remainder of *Y* after the first *X* elements have been removed. If *X* is a negative integer, the result is *Y* minus its last *X* elements.

```
      MAT1
8    ¯2     0
6     3     4

      1 2↓MAT1
4
```

Transposition

Often during computation, it is desirable to restructure arrays in such a way that the rows are interchanged with the columns. This is accomplished in APL by employing the monadic operator ⍉ , (uppershift *O* overstruck with the reverse solidus).

```
      MAT1
8    ¯2     0
6     3     4

      ⍴MAT1
2  3

      ⍉MAT1
 8    6
¯2    3
 0    4
```

```
      M1←⍉MAT1

      ⍴M1
3  2

      A←3 4⍴'ACRESLAVHUGE'

      A
ACRE
SLAV
HUGE

      ⍉A
ASH
CLU
RAG
EVE

      X←2 3 4⍴⍳24

      X
 1     2     3     4
 5     6     7     8
 9    10    11    12

13    14    15    16
17    18    19    20
21    22    23    24

      T←⍉X

      T
 1     5     9
 2     6    10
 3     7    11
 4     8    12

13    17    21
14    18    22
15    19    23
16    20    24

      ⍴T
2  4  3
```

The result of the monadic Transpose function contains all the elements that are in the argument with the only difference being that the rows of the argument are now the columns of the result and the columns of the argument are now the rows of the result. Only the last two coordinates of the array argument are reversed in the result. If the argument is a vector, the result is identical to the argument as there are not two coordinates to transpose. Here is an example of the Transpose operation being attempted on a vector:

```
      ⍉7 3 2
7  3  2
```

The monadic Transpose only allows for the reversal of the last two coordinates. But, when dealing with multi-dimensional arrays, the interchange of more than just the last two coordinates may be desired. The dyadic use of the Transpose operator is used in this case.

```
      A←3 2 4⍴⍳24

      A
 1     2     3     4
 5     6     7     8

 9    10    11    12
13    14    15    16

17    18    19    20
21    22    23    24

      R←3 1 2⍉A

      R
1     9    17
2    10    18
3    11    19
4    12    20

5    13    21
6    14    22
7    15    23
8    16    24

      ⍴R
2  4  3
```

The left argument of the dyadic Transpose function is a vector of positive integers indicating the dimensions of the result *R*. In the above example, the argument 3 1 2 states that the first coordinate of *A* shall be the third coordinate of *R*, the second coordinate of *A* is to be the first coordinate of *R*, and the last coordinate of *A* will become the second of *R*. The number of integers contained in the left argument is equal to the number of dimensions of the right argument. The contents of the left argument are positive integers from 1 to *N*, where *N* is the number of dimensions of the result. None of these integers can be greater than the number of dimensions of the right argument. For instance, when transposing a 3-dimensional array, the right argument could be 1 2 3 or 3 2 1 or 1 1 2 or 2 2 1 or 1 2 2 but could not be 1 3 3 or 2 2 3.

Along with redimensioning is the relocation of the elements of the argument. For example, the value 7 in array *A* is located at position [1;2;3]; that is, first plane, second row, third column. In the resultant array *R*, it is located at position [2;3;1]; or plane two, row three, column 1. The general algorithm for this transpose is *A*[*X*;*Y*;*Z*] with the coordinates of *A* being relocated in *R*[*Y*;*Z*;*X*] where the function is (*Y*,*Z*,*X*)⍉*A*.

```
      S←3 2 1⍉A

      S
1     9    17
5    13    21

2    10    18
6    14    22

3    11    19
7    15    23

4    12    20
8    16    24

      ⍴S
4  2  3
```

The result of a dyadic Transpose need not always be of the same dimensions as the right argument. For example, to create a 2-dimensional array from *A*, the following would be used:

```
      T←2 2 1⍉A

      T
1    13
2    14
3    15
4    16

      ⍴T
4  2
```

The dimensions of the result are determined by finding the maximum value contained in the left argument as shown by the following calculation:

```
      ⌈/2 2 1
```

Therefore, *T* is a 2-dimensional array. The values that *T* represents are located according to the following:

```
      T[X;Y]←A[Y;Y;X]
```

T has the dimensions *X*, *Y*. Just which *Y* coordinate of *A* to pick is determined by finding the smaller of the two coordinates referenced by *Y* in the following algorithm:

```
      ⌊/2 3
2
```

Another example is:

```
      T2←1 1 2⍉A

      T2
 1      2      3      4
13     14     15     16

      ρT2
2  4
```

T2 is a (⌈/1 1 2) or 2-dimensional array. Its elements are organized in the following format:

```
      T2[X;Y]←A[X;X;Y]
```

X ranges from 1 to 2 because (⌊/3 2) is 2 and ranges from 1 to 4. For a 2-dimensional array, a dyadic Transpose is the same as a monadic Transpose.

```
      M←3 4ρι12

      M
1      2      3      4
5      6      7      8
9     10     11     12

      2 1⍉M
1      5      9
2      6     10
3      7     11
4      8     12

      ⍉M
1      5      9
2      6     10
3      7     11
4      8     12
```

To find the major diagonal of a 2-dimensional array, the following algorithm is used:

```
      1 1⍉M
1  6  11
```

Reversal - Rotation

The Reversal function ⌽, (uppershift *O* overstruck with uppershift *M*) reverses the order of its right argument if the argument is a vector.

```
      ⌽4 3 2 1
1  2  3  4

      ⌽'RAT'
TAR
```

If the argument is of a greater dimension, the columns are reversed.

```
      M←3 4⍴⍳12

      M
1     2     3     4
5     6     7     8
9    10    11    12

      ⌽M
 4     3     2     1
 8     7     6     5
12    11    10     9
```

To reverse the rows of a matrix, there are two methods.

```
      ⌽[1]M
9    10    11    12
5     6     7     8
1     2     3     4
```

This reads as "reverse the order of array *M* along its first coordinate".

Here's another way to do the same thing:

```
      ⊖M
9     10     11     12
5      6      7      8
1      2      3      4
```

The function ⊖ (uppershift *O* overstruck with the minus sign -) also reverses along the first coordinate of its argument.

If the argument were a 3-dimensional array and the user wished to reverse the order of the rows, he would type the following:

```
      ⌽[2]X
```

where *X* is the 3-dimensional array with its rows being the second coordinate.

The dyadic format of the ⌽ operator is *X*⌽*Y*. If *X* is a single integer value and *Y* is a vector, then *X*⌽*Y* is a cyclic rotation of *Y*. For example:

```
      3⌽2 6 0 ¯3 8
¯3  8  2  6  0
```

In the above function, the vector right argument was rotated, an element at a time, placing the first element of the vector at the back and repeating the process over again as many times as prescribed by the left argument.

```
      2⌽1 2 3 4
3  4  1  2
```

The left argument may be a negative integer. If it is, the rotation of the right argument is in a back-to-front direction.

```
      ¯3⌽2 6 0 ¯3 8
0  ¯3  8  2  6

      ¯1⌽1 2 3 4
4  1  2  3

      ¯5⌽'NOONAFTER'
AFTERNOON

      4⌽'NOONAFTER'
AFTERNOON
```

If the right argument is an array with dimensions greater than one, and the left argument is a single value, the entire columns of the array are rotated by the amount specified by the left argument.

```
      A←3 4⍴⍳12

      A
1     2     3     4
5     6     7     8
9    10    11    12

      2⌽A
 3    4     1     2
 7    8     5     6
11   12     9    10
```

Which is the same as:

```
      2⌽[2]A
 3    4     1     2
 7    8     5     6
11   12     9    10
```

The rows of the matrix may be cyclicly rotated by the following method:

```
      2⌽[1]A
9    10    11    12
1     2     3     4
5     6     7     8
```

Or, another symbol to rotate an array along its first coordinate is ⊖.

```
     2⊖A
9    10    11    12
1     2     3     4
5     6     7     8

      M←3 5⍴'EATBLACKQUUNTGR'

      M
EATBL
ACKQU
UNTGR
```

```
      3⌽M
BLEAT
QUACK
GRUNT

      2⊖3⌽M
GRUNT
BLEAT
QUACK
```

It may not be desirable to have all the rows of each column rotated by the same amount, as were the last examples. So, by expanding the left argument to a vector whose length equals the number of rows contained in the right argument, this limitation is overcome.

```
      0 1 2⌽A
 1     2     3     4
 6     7     8     5
11    12     9    10
```

The same is true for columns.

```
      0 1 2 3⊖A
1     6    11     4
5    10     3     8
9     2     7    12
```

Membership

To find out if a certain element is contained in a particular variable, the Membership function, ∊, (uppershift *E*), may be used. Here is an example of its use. The value 2 is being looked for in the vector 6 3 4.

```
      2∊6 3 4
0
```

The response, like that of the Relational functions, is either 1's or 0's representing "yes" or "no" respectively. Because the 2 was not found in the vector the computer returned the value 0.

Here are some more examples:

```
      6 2 4∊2
0 1 0

      2 4 5∊'ABCDE'
0 0 0

      'ABCDEF'∊'BAD DAY'
1 1 0 1 0 0

      A←3 3ρι9

      A∊6 10 4
0 0 0
1 0 1
0 0 0
```

The size and shape of the result is always equal to the size and shape of the left argument.

Roll-Deal

The symbol *?*, (uppershift *Q*), produces some rather interesting results when used both monadically and dyadically. Here are some examples:

```
      ?10
5

      ?10
1

      ?10
7
```

Notice that each result above is different. The reason is that the Roll function selects a number at random from 1 to *N* where *N* is the value of the argument. This argument may be any positive integer.

One can easily see the possibilities for simulations that require numbers to be selected at random. For instance, simulating the rolling of two dice would be done in the following manner:

```
      ?6 6
1  4

      ?6 6
3  4

      ?6 6
1  1
```

The number of elements contained in the result is equal to the number of elements contained in the argument.

The same is not true when the *?* is used dyadically.

```
      3?10
7  3  9

      3?10
1  8  3
```

Here, the *?* function, (called Deal when used dyadically), uses the format

R?N

which means to select *R* integers from 1 to *N* without replacement. Each value contained in the result will be unique. Both *R* and *N* must be single integers and *N* must be greater than or equal to *R*.

Chapter Ten

USER DEFINED FUNCTIONS

York APL has over forty "primitive" functions that do a wide variety of operations on differing arguments. It was designed to give the user almost instant response to his input by immediately executing his typed statements as soon as the "RETURN" key is pressed. In other words, it performs like a desk calculator. But, as the user becomes more familiar with the APL symbols and the way the system handles their execution, he will want to execute more complex and elaborate equations. He could do this quite easily by executing his problem a line at a time and storing the required intermediate results in variables to be used later on in the algorithm. But this process becomes quite cumbersome and time consuming. And, what if he wishes to execute the same equation several times, varying the parameters each time? He would have to retype the entire procedure every time he made a change to one of his initial variables. Here is a simple illustration.

A student wishes to take the attendance readings for his class over the last two weeks and calculate the average, the lowest and highest readings, and the range between the lowest and highest. After obtaining the attendance numbers, he assigns them to a variable called *X*.

```
      X←23 25 24 26 28 23 27 28 27 26
```

To calculate the average, he first of all must find out how many elements are in *X* and then divide this number into the sum of *X*. To do this, he types the following:

```
      ⍴X
10
```

This determines how many elements are contained in *X*.

Now he must sum up *X*

```
      TOT←+/X

      TOT
257
```

and then to find the average, he types the following:

```
      TOT÷⍴X
25.7
```

The highest, lowest and range are found in the following manner:

```
      ⌈/X
28

      ⌊/X
23

      (⌈/X)-⌊/X
5
```

But, suppose he made a typing error and the last value of *X* should have been a 27 instead of a 26. He would have to correct *X* and then type the above steps over again. Or, suppose he wanted to save his procedure so that it could be re-executed when new data was made available. To avoid these problems, the APL user can write a program, or function as it's called in APL, that will execute the required calculations.

So far, the terminal has been like a desk calculator because it has been in "Calculator Mode". Everytime something was entered by the user, it was immediately executed. But now the user wants to define a function, not execute an algorithm. So he must signal his intentions to the computer to prevent it from trying to execute his input. To do this, the "Mode" of operation must be changed. The system must be taken out of Calculator Mode and placed into Function Definition Mode. It sounds complex, but it's really quite simple.

But before this is done, the user should first determine what he wants his function to do. In this case, the student wishes to perform the following calculations on a set of attendance data:

(1) calculate the average attendance
(2) find the highest attendance reading

(3) find the lowest attendance reading
(4) determine the range between the highest and lowest readings

Finding the average of *X* involves two steps. First, the number of elements of *X* must be found. Second, this number is then divided into the sum of *X* to calculate the average attendance reading. Now the function to carry out all the above computations can be defined.

Above the letter *G* on the keyboard, there's a symbol called "Del" (∇). To switch from Calculator Mode to Function Definition Mode, this Del must be typed, followed by the name of the function.

```
      ∇ATTEND
[ 1  ]
```

ATTEND is the name of the function that will hold the algorithms to calculate the mean, high, low, and range for the class attendance. The system responds to this line by typing out [1] signifying that it is in Function Definition Mode and is ready to accept the first line. The user then types in the first line and presses the "RETURN" key.

```
      ∇ATTEND
[ 1  ]N←⍴X
[ 2  ]
```

In this case, it's *N*←⍴*X* which determines the number of elements in *X* and places the result in *N*. The system again responds with [2], asking for the next line. So the user keeps entering his algorithms until he gets to line 6.

```
      ∇ATTEND
[ 1  ]N←⍴X
[ 2  ](+/X)÷N
[ 3  ]⌈/X
[ 4  ]⌊/X
[ 5  ](⌈/X)-⌊/X
[ 6  ]
```

At this point, all the calculations to be performed by *ATTEND* have been entered. Now he would like to end his function definition so that he may get back into Calculator Mode to use *ATTEND*. To do this, he simply types another Del.

```
[ 6  ]∇
```

This places him back into Calculator Mode. To check, he types in a simple problem to see if the computer will execute it.

```
      2+2
4
```

Once the function *ATTEND* has been defined, it may be displayed. To do this, he would type the following:

```
      )FNS ATTEND
```

To which the system would respond:

```
∇ATTEND
 [1] N←⍴X
 [2] (+/X)÷N
 [3] ⌈/X
 [4] ⌊/X
 [5] (⌈/X)-⌊/X
∇
```

To execute *ATTEND*, the user need only type in its name.

```
      ATTEND
25.7
28
23
5
```

The above 4 values should look familiar; being the mean, highest, lowest and range values of *X*.

Once all the procedures to calculate the results are contained in a function, the user is able to alter his input all he wants and then execute *ATTEND* to obtain new statistics.

```
      X←24 25 26 28 28 27 27 26 28 24

      ATTEND
26.3
28
24
4

      X←25

      ATTEND
25
25
25
0
```

Function Editing

Functions usually undergo several changes to their content from the initial definition to the final function. This could be due to many reasons; anything from additional features being inserted once the original function has been tested and found to be inadequate, all the way to just correcting typing errors. But whatever the reason, it is best to know the three basic techniques that are available to change the structure and content of a function. They are Line Insertion, Line Modification, and Line Deletion.

Line Insertion

It would be nice to have *ATTEND* display *X* before any computations took place, just to make sure it contains the proper values. This means inserting a line before line 1 of *ATTEND* that would allow *X* to be printed out. The user would type the following:

```
      ∇ATTEND
[ 6  ][.1]X
[ 1  ]∇
```

The user typed in the Del, followed by *ATTEND*, signalling to the system to open up *ATTEND* for the purposes of making editions. The system responded by typing [6], indicating that it is in Function Definition Mode and that it is ready to add more lines onto *ATTEND*. Line 6 is the first available free line. But the user wants to place his statement before any of the others so that it will be printed first whenever *ATTEND* is executed. Therefore he must redirect the computer away from line 6 and point it to some point before line 1. The user pointed to .1 . He could have typed in any value between 0 and 1 to obtain the same objective.

On the same line that he typed [.1], he entered his new statement. After pressing the "RETURN" key, the system asked if there were any modifications to be made to the line immediately following the inserted line; in this case, it's line 1. There weren't, so the user closed the function by typing in another Del. This does not affect the contents of line 1 in any way. Just to make sure, *ATTEND* is again displayed.

```
      )FNS ATTEND
∇ATTEND
 [.1] X
 [1] N←ρX
 [2] (+/X)÷N
 [3] ⌈/X
 [4] ⌊/X
 [5] (⌈/X)-⌊/X
∇
```

Upon displaying *ATTEND,* the line insertion seems to have been successful. On execution of *ATTEND,* it proves it was.

```
      X←24 26 26 20 25

      ATTEND
24  26  26  20  25
24.2
26
20
6
```

Line Renumbering

The line numbers remain as they were originally entered in case there are more modifications to be made. The function does not have to be displayed constantly to make sure the right line gets altered.

But once all insertions and changes to the function have been completed, the lines of the function can be renumbered to make it easier to read and neater in appearance. To do this, a comma and the letter *R* are placed after the command to display the function.

```
      )FNS ATTEND,R
OK
```

The system responds with *OK* to indicate the line numbers have been renumbered. This is what *ATTEND* now looks like:

```
      )FNS ATTEND
∇ATTEND
 [1] X
 [2] N←ρX
 [3] (+/X)÷N
 [4] ⌈/X
 [5] ⌊/X
 [6] (⌈/X)-⌊/X
∇
```

Line Modification

Line 3 of *ATTEND*, which reads (+/*X*)÷*N* , could be modified to read (+/*X*)÷ρ*X* which would achieve the same result as lines 2 and 3 do now since *N* is really ρ*X* anyway. So, to modify line 3, the same procedure is followed as with Line Insertions.

```
      ∇ATTEND
[ 7  ][3](+/X)÷ρX∇
```

Something to note here. After the system was redirected back to line 3 and the modification typed in, a Del was typed at the end of the line, just before the "RETURN" key was pressed. This is just another way of signalling to the system that all the modifications are complete and the function can be closed. It's just quicker than having the system come back with [4] and then closing the function.

```
      )FNS ATTEND
∇ATTEND
 [1] X
 [2] N←ρX
 [3] (+/X)÷ρX
 [4] ⌈/X
 [5] ⌊/X
 [6] (⌈/X)-⌊/X
∇
```

It can be seen that line 3 has indeed been changed.

Line Deletion

There is no longer any need to have *N* defined in line 2 since it is not used in the function anymore. So, to get rid of it, the following would be typed:

```
      ∇ATTEND
[ 7  ][2]                (press "RETURN" key only)
[ 3  ]∇
```

After pointing the system back to line 2, only the "RETURN" key is pressed. This erases line 2 from the function.

```
      )FNS ATTEND
∇ATTEND
 [1] X
 [3] (+/X)÷ρX
 [4] ⌈/X
 [5] ⌊/X
 [6] (⌈/X)-⌊/X
∇
```

And, just to renumber the lines to eliminate the gap left by the deletion of line 2, the Renumber command is again typed.

```
      )FNS ATTEND,R
OK
```

And to make sure it still works:

```
      ATTEND
24  26  26  20  25
24.2
26
20
6
```

X still has the values 24 26 26 20 25 that were assigned to it earlier.

The only function that can't be displayed or modified in any way is a "locked" function. To lock a function, either the beginning or ending Del is overstruck with a Tilde to form the symbol, ⍫. Locked functions cannot be unlocked, so if the user wishes to lock his functions he should make sure they work perfectly or else have unlocked versions saved privately in his library just in case.

One other feature about displaying a function is that the display may begin at any line. For instance, lines 4 and 5 only of *ATTEND* could be displayed by typing the following:

```
      )FNS ATTEND,4
```

The ,4 tells the computer to list the statements contained in *ATTEND* beginning at line 4. To which the computer responds:

```
 [4] ⌊/X
 [5] (⌈/X)-⌊/X
∇
```

Chapter Eleven

TYPES OF FUNCTIONS

A function is basically made up of two parts, the body and the Header Line. The body runs from line 1 to the last line of the function. The Header Line is that line which contains the name of the function. It is always the first line printed whenever a function is displayed. Another feature of the Header Line is that it contains the syntax of the function. For instance, just as there are Monadic and Dyadic primitive functions such as ÷2 and 6÷2, so too are there Monadic and Dyadic user defined functions. Here is what the Header Line of a Monadic user defined function looks like:

```
∇SORT X
```

SORT is the name of the function and *X* is its argument. The name of a Monadic function always precedes its argument just as the primitive functions do The argument is separated from the function name by a space.

The Header Line of a Dyadic function would look like this:

```
∇A HYP B
```

The name of the function in this case is *HYP* and the two arguments are *A* and *B*. When relating this type of function to the primitive functions, *HYP* could be thought of as the operator, such as ÷, and *A* and *B* could be the 6 and 2 mentioned above.

In addition to having both Monadic and Dyadic user defined functions, there is also one called "Niladic", or a function that has no arguments. *ATTEND* was defined as being a Niladic function.

Here is a table of all the different types of user defined functions that can be written:

	Niladic	Monadic	Dyadic
No Explicit Result	∇*ATTEND*	∇*SORT X*	∇*A HYP B*
Explicit Result	∇*R*←*ROLL*	∇*R*←*SQRT N*	∇*C*←*A RND B*

The difference between "No Explicit Result" and "Explicit Result" functions will be discussed in a moment, but first the three different functions in the top row will be illustrated.

Suppose for the time being that all of the functions listed have already been defined in the Active Workspace. The function *ATTEND* is the same one that was created earlier. Just to make sure, it's displayed.

```
      )FNS ATTEND
∇ATTEND
 [1] X
 [2] (+/X)÷ρX
 [3] ⌈/X
 [4] ⌊/X
 [5] (⌈/X)-⌊/X
∇
```

And to see if it still works, *X* is assigned the values 1 to 10 and the function is executed.

```
      X←ι10

      ATTEND
1  2  3  4  5  6  7  8  9  10
5.5
10
1
9
```

Everything seems to work okay.

The function *SORT* is a Monadic function and, when displayed, looks like this:

```
      )FNS SORT
∇SORT X
 [1] X[⍋X]
∇
```

Notice that only the name of the function has to be typed in order to display its contents. The argument(s) is not included when the system is asked to display a function.

Obviously, the function *SORT* sorts a vector of values in ascending sequence. So, it's tried out:

```
      SORT
SORT
? SYNTAX ERROR
```

What happened here? *ATTEND* worked okay and it used the variable *X*. So why doesn't *SORT*? The reason is that both functions have different types of Header Lines. *ATTEND* is defined as being a Niladic function requiring no accompanying argument while *SORT* is a monadic function expecting a right argument each time it's executed. Here it's tried again; this time with a right argument.

```
      SORT 2 10 6 2.5 0 ¯4
¯4  0  2  2.5  6  10
```

Something rather odd just happened here. *X* had previously been assigned the values 1 through 10, but upon executing *SORT* which uses a variable called *X*, different values were returned than were originally assigned to *X*. When *X* is displayed, it still has the values that were assigned to it before *ATTEND* was executed.

```
      X
1  2  3  4  5  6  7  8  9  10
```

This means that two different *X*'s were used. And this is in fact what happened. Because *X* is used in the Header Line of the function *SORT* to define the syntax of that function, it is classified as being a "local variable". This means that it becomes a valid variable only while *SORT* is executing. Once *SORT* has successfully completed its computations, the local variable *X* is automatically erased from the system. The *X* assigned the values 1 to 10 is called a "global variable"; meaning it was defined while the system was in Calculator Mode and can be used outside any of the functions. The)*VARS* command will list all the global variables.

```
      )VARS
X
```

X is the only global variable presently in the Active Workspace. The function *HYP* should further illustrate the local-global variable aspect.

HYP calculates the hypotenuse of a right angled triangle.

```
      3 HYP 4
5
```

Displaying *HYP* should reveal how the hypotenuse is calculated.

```
      )FNS HYP
∇A HYP B
 [1] ((A*2)+B*2)*0.5
∇
```

Two more local variables, *A* and *B* are used here. And to prove that they too are erased from the Active WS as soon as *HYP* is successfully completed, a listing of the current global variables is again requested.

```
      )VARS
X
```

The following example re-executes *HYP* using predefined variables for its arguments.

```
      SIDE1←2
      SIDE2←5

      SIDE1 HYP SIDE2
5.385164807
```

When *HYP* was executed this last time, the local variables *A* and *B* took on the values contained in *SIDE*1 and *SIDE*2 respectively until the hypotenuse was printed.

```
      )VARS
SIDE1       SIDE2       X
```

Therefore, the variable names that appear in Header Lines of functions become unique variables only while their respective functions are executing, even if they have the same names as previously defined global variables.

Explicit vs No Explicit Result Functions

The difference between an Explicit Result function and a No Explicit Result function is that the Explicit Result function, at the end of its computations, produces a result that may be used immediately as an argument of another user defined or primitive function.

The way a function is denoted as one that produces an Explicit Result is by the presence of a specification arror (←) in the Header Line. The second row of the function Header Lines in the table listed before are all Explicit Result functions.

The function *ROLL* is an Explicit Result, Niladic function that selects, at random, two numbers from two different sixes.

```
      )FNS ROLL
∇R←ROLL
 [1] R←?6 6
∇
```

Here is how it works:

```
      ROLL
3  4

      ROLL
1  1

      +/ROLL
8

      +/ROLL
3
```

By defining *ROLL* as an Explicit Result function, it can be used as the right argument to the "Plus-Reduction" operations in the last two examples. If the same type of thing were tried with a No Explicit Result function, such as *SORT*, the following would occur:

```
      +/SORT 2 6 ¯1
¯1  2  6
+/SORT 2 6 ¯1
  ? VALUE ERROR
```

The function *SORT* rearranged its right argument into ascending sequence and printed it out just as it did before, but, because

It was not defined as being an Explicit Result function, it caused the *VALUE ERROR* to occur.

The next function, *SQRT*, is a Monadic, Explicit Result function used to find the square root(s) of its argument.

```
      SQRT 4 25 64
2  5  8

      SQS←SQRT 16 36

      SQS
4  6

      SQS*2
16  36

      )FNS SQRT
∇Z←SQRT N
 [1] Z←N*0.5
∇
```

And lastly, the function *RND*, used to round off the value(s) contained in the right argument according to the number of digits specified in the left.

```
      2 RND 76.826
76.83

      10+0 RND ¯2.3 6 4.7
8  16  15

      )FNS RND
∇C←A RND B
 [1] C←(10*-A)×⌊0.5+B×10*A
∇
```

There are just a few more points to mention concerning function Header Lines. The names of the functions must adhere to the same rules as variables do. They are mentioned in Chapter 2. The variables contained in the Header Line serve only to indicate the syntax required for that function each time it is executed. They become valid variables only while the function whose Header Line in which they reside is executing. Once it is finished, they are automatically erased from the Active WS and any existing global variables with the same name are then "reactivated". The variable name to the left of the specification arrow found in the Header Line of an Explicit Result function is also a local variable. Within the body of the function, this variable must be assigned a value before normal completion of the function is reached. If this is not done, the function will terminate with an error message.

Additional Local Variables

Most user defined functions require more than one line to complete their prescribed calculations. This means that intermediate results obtained as each line is executed have to be stored in variables so that they can be used in future calculations. But, once the function has completed execution, these intermediate results serve no further purpose. After a while, if several functions are executed, these intermediate values start to clutter up the Active WS and cut down on available space that may be needed for other calculations. They also make it very hard to remember which variables are useful and which are not. Therefore, to aid the user in the general "housekeeping" of his Active WS, APL allows several variables to be defined as being local to certain functions. This means that instead of just having the arguments of a function automatically erased by the system after the function has completed its computations, the user may state any number of variables to be eliminated this way.

Here is the Header Line of a Dyadic, Explicit Result function with five local variables, *R*, *A*, *B*, *PROD*, and *TOT*.

```
      ∇R←A TIMES B;PROD;TOT
```

Each local variable not required to define the general syntax of the function must be preceded by a semicolon.

```
      )FNS TIMES
∇R←A TIMES B;PROD;TOT
 [1] PROD←A×B
 [2] TOT←+/PROD
 [3] R←TOT+TOT×0.05
∇
```

An invoice contains the following:

Quantity	Unit Price
6	5.95
23	.98
16	1.59
9	2.25

The function *TIMES* is used to calculate the total amount of the bill, including a 5% sales tax on line three.

```
      6 23 16 9 TIMES 5.95 .98 1.59 2.25
109.1265

      )VARS
SIDE1      SIDE2      X
```

There are still only three variables listed in the Active WS. To make the variables *PROD* and *TOT* global, they must be taken out of the Header Line and *TIMES* must again be executed.

Changing the Header Line of a function is the same as changing any other line of the function.

```
      ∇TIMES[0]R←A TIMES B∇
```

Upon opening the above function, the system is immediately pointed to line zero, the Header Line, the change is made and the function closed. This is just another way to further reduce the number of lines needed to modify part of a function.

Below, *TIMES* is displayed to see what the new Header Line looks like, then it's executed.

```
      )FNS TIMES
∇R←A TIMES B
 [1] PROD←A×B
 [2] TOT←+/PROD
 [3] R←TOT+TOT×0.05
∇

      6 23 16 9 TIMES 5.95 .98 1.59 2.25
109.1265
```

Now there are two more variables added to the global variable list.

```
      )VARS
PROD      SIDE1      SIDE2      TOT      X
```

```
      PROD
35.7  22.54  25.44  20.25

      TOT
103.93
```

If the name of the function is changed in any way, there will not be an additional function added to the Active WS, but rather the "new" functions will replace the "old".

```
      ∇TIMES[0]R←A INVOICE B∇
```

It must be remembered that whenever a line of a function is changed in any way, the entire new line must be entered. Partial changes are not allowed. This means that even if one character is in error, the entire line must be typed in to make the proper correction.

```
      )FNS TIMES
)FNS TIMES
     ? VALUE ERROR
```

The system command to list all the functions in the Active WS is just)*FNS* .

```
      )FNS
ATTEND
SORT X
A HYP B
R←ROLL
Z←SQRT N
C←A RND B
R←A INVOICE B
```

It not only lists the names of all the functions but also the syntax of each.

Sub-functions

When writing fairly complex functions, it is easy to accidentally use duplicate names for different variables, or make the flow of logic very difficult to follow and maintain. The best approach to writing functions that contain a lot of calculations is to modularize specific routines and use a main function to "call" them when they are needed. Another reason for breaking up one big function into several small ones is that certain routines may be executed many times. This not only adds an unnecessary number of lines to the function, but it also increases the chance of errors while designing and typing the function.

The function *INVOICE* is used below to illustrate how a sub-function is used to perform a specific calculation for the main function. First, *INVOICE* is displayed, just to refresh the user on how it works.

```
      )FNS INVOICE
∇R←A INVOICE B
 [1] PROD←A×B
 [2] TOT←+/PROD
 [3] R←TOT+TOT×0.05
∇
```

Then it is tried again with a new set of data.

```
      UNITS←6 22 10 5
      COST←.95 1.5 2 1.75

      UNITS INVOICE COST
70.8225
```

This bill would normally be rounded off to the nearest penny. This is where the function *RND* comes into play. It can do the rounding of the bill for the function *INVOICE*.

Here is what the function *RND* looks like:

```
      )FNS RND
∇R←A RND B
 [1] R←(10*-A)×⌊0.5+B×10*A
∇
```

Here it is in use:

```
      1 RND 56.46
56.5
```

Line 3 of *INVOICE* must be changed to include *RND*.

```
      ∇INVOICE[3]R←2 RND TOT+TOT×0.05∇
```

The function *INVOICE* now looks like this:

```
      )FNS INVOICE
∇R←A INVOICE B
 [1] PROD←A×B
 [2] TOT←+/PROD
 [3] R←2 RND TOT+TOT×0.05
∇
```

And works like this:

```
      UNITS INVOICE COST
70.82
```

Chapter Twelve

BRANCHING AND INPUT-OUTPUT

Branching

So far, the statements contained in the previous functions have executed in an orderly fashion. The contents of line 1 were computed before those of line 2, which was done before line 3, and so on to the end of the function. But, in many computer applications, there is often the need to branch to a statement that is not immediately below the one presently being executed. Or a user may want to execute a certain set of statements several times depending on the prevailing conditions. This latter situation is called "looping". A typical function that requires looping of a certain routine is one that sorts a group of names.

Assume the user has defined a matrix called *M* to contain the following names:

```
      M
FRED
BILL
BERT

      ⍴M
3  4
```

M is a matrix consisting of 3 rows and 4 columns. The names contained in *M* were entered in a random order with the intention of sorting them in ascending sequence. So now the user must set about to define a function to do this task.

The first step is to decide what type of function to use. The user has chosen to make this function a Monadic, Explicit Result function with the following Header Line:

```
      ∇R←SORT A
```

The next step is to pick a method of sorting the contents of the matrix argument. Most computer and manual sorters start at the right column of the matrix and work to the left, sorting the matrix according to the relative positions of the letters in each column as it's sorted.

Therefore, a counter must be set to equal the number of columns contained in the argument to assure that the right column is sorted first. This counter is then decremented by one before each new column sort is attempted so that the same column isn't sorted twice.

Here is how this counter would be initialized in line one of the function:

```
[ 1  ]I←(⍴A)[2]
```

Line 1 contains a statement that calculates the dimensions of the argument *A* and places the second of the two dimensions into the counter *I*.

Line 2 states that *R* is assigned the values contained in *A*. Doing this function now just avoids the need to do it later. Either method is equally acceptable.

```
[ 2  ]R←A
```

The next step is to perform a sorting operation on the extreme righthand column. This is done by the following algorithm:

```
[ 3  ]R←R[(⍋R[;I]);]
```

R is replaced by the values contained in *R* after they have been rearranged according to the sorted sequence of the indices in the column number specified by *I*. The first time through, the contents of *R* remain unchanged because the letters *D*, *L* and *T* are already in ascending sequence.

Line 4 will decrement the counter by 1 so that the next column can by sorted.

```
[ 4  ]I←I-1
```

Now, the system must be redirected back up to line 3 to perform the next sort. The symbol used to change the normal sequence of execution is the right pointing arrow, →, (uppershift ←).

```
[ 5  ]→3
```

This last statement is an Unconditional Branch, telling the system to always go back to line 3, no matter what happens. It is read as "go to line 3". This "loop" between lines 3 and 5 will be repeated over and over until *I* reaches 0. The reason it stops when *I* equals zero is that *R* cannot be indexed by the value 0. If this does happen, the function will abnormally terminate. So *I* must be checked for this condition and the appropriate action taken to avoid this happening. That means a line must be inserted between lines 4 and 5 to make the function terminate its execution when *I* equals 0. This can be accomplished by the following line:

```
      [ 4.1 ]→(I=0)/0
```

The Logical Reduction function is used here to terminate the function *SORT* when *I* equals 0. The statement reads, "if *I* equals 0, branch to line 0, otherwise continue to the next statement". Line 0 was chosen as the target of the branch because it does not appear as a line number of the function. Actually, any invalid line number would do the same thing; namely terminate *SORT*. But line 0 is used because, no matter how many lines are added to the function, it will never have a line 0.

If *I* is not equal to 0, no branch occurs. The reason for this is that the comparison *I*=0 returns a result of 0, and 0/0 yields an empty vector. A branch to an empty vector means no branch at all.

Here is the way *SORT* should look:

```
∇R←SORT A
 [1] I←(ρA)[2]
 [2] R←A
 [3] R←R[(⍋R[;I]);]
 [4] I←I-1
 [5] →(I=0)/0
 [6] →3
∇
```

Now for the sorting of *M*.

```
      SORT M
BERT
BILL
FRED
```

The branch operation may occur anywhere within a statement. For instance, the following branch is performed in the middle of the statement of line 7 of a function.

```
      .
      .
      .
    [7] R←(⍳0)=X←→(0=A)/11
      .
      .
      .
```

If the above branch is successful, execution of the statement on the above line 7 will be halted at the branch arrow, (→), and resume again at line 11. If not, the branch function will return an empty vector result which will then be assigned to the variable *X*. Therefore, if *A* is not equal to 0 in the above statement, *X* will receive the value ⍳0 and *R* will be assigned the value 1. Knowing this capability, lines 5 and 6 of *SORT* could have been combined on the following manner:

```
    [ 5  ] →3,→(I=0)/0
```

Labels

Here is an example of another kind of branching operation. The first line of the function *RANK* is a branching operation that has, as its targets, the names *VECTOR* and *MATRIX*. These two names are called line labels.

```
      )FNS RANK
∇RANK A
 [1] →(1 2=⍴⍴A)/VECTOR,MATRIX
 [2] 'MULTI-DIMENSIONAL ARRAY'
 [3] →0
 [4] VECTOR: 'RANK 1'
 [5] →0
 [6] MATRIX: 'RANK 2'
∇
```

The line labels are positioned in front of the function statements on lines 4 and 6 and are separated from these statements by colons. They are not part of the executable statements but serve as "local constants" to the function. This means that during the execution of the function in which they are contained, they are assigned the values of the numbers on which they reside. For instance, while *RANK* is executing, the labels *VECTOR* and *MATRIX* become local constants with the values 4 and 6 respectively. Therefore, a branch to *VECTOR* is really a branch to line 4.

They have the same properties as local variables in that their domain is limited to their function. But unlike local variables whose values may change many times during the execution of a function, the values associated to local constants remain the same throughout the function's execution.

Local constants are quite useful, especially in the initial stages of writing a function. Usually, the final copy of a function looks only remotely like the original version. Lines are often inserted while others are deleted. And during all these modifications the line numbers change many times, making it a nightmare for branching operations because, no matter what the new numbering sequence of the function, branching operations are always to the same line number; even though the intended statement to receive the branch now resides on a new line. All this chaos can be avoided with the use of line labels because they are assigned values only while their respective function is executing. Therefore, it does not matter how many times a function is renumbered, a branch to a specific label will always do just that.

Here are some examples of how the labels in *RANK* work:

```
      RANK 6 7 8
RANK 1

      RANK 2 3⍴⍳6
RANK 2

      RANK 3 3 3 3 3⍴⍳100
MULTI-DIMENSIONAL ARRAY
```

Examples of Branch Instructions

There are many more ways to evoke a branch to another line. Here are just a few of the different branching techniques that are possible:

```
→2
→2+X×Y≥0
→LABEL
→0
```

Branch To Either Of Two Lines

```
→((X<0),X≥0)/6 2

→(LABEL1,LABEL2)[1+X>0]
```

Conditional Branch

```
→( 1 OR 0 )/LABEL

→( 1 OR 0 )ρLABEL

→( 1 OR 0 )↑LABEL

→LABEL×ι( 1 OR 0 )
```

If a function becomes suspended while trying to execute, the cause of the suspension can be corrected and the execution resumed. For example, assume a function called *EVAL* became suspended on line 12. If, after correcting the error, the user wished to continue execution of *EVAL* at line 12, he would type the following:

```
→12
```

Or he could restart the execution at any other line by typing in the branch to it, just as the branch to line 12 was accomplished.

When a function becomes suspended, its name and the number of the line on which the suspension occured are added to the State Indicator. To find out what is listed in the State Indicator, the)*SI* command is issued.

```
      )SI
*EVAL       [12]
```

The State Indicator lists the function *EVAL* as being suspended on line 12. *EVAL* is preceded by an asterisk because it is a "suspended" function. The State Indicator may also list "pendant" functions - functions that have called other functions that have become suspended, like *EVAL*. The resumption of pendant functions is dependent on the resumption of suspended functions. Pendant functions are also listed in the State Indicator, but are not preceded by an asterisk. Here's an example of a pendant function and a suspended function:

```
      )SI
*SS         [2]
ANALYSIS    [17]
```

The line number listed with the pendant function is the line in which the suspended function was called.

To find out which variables are "local" at the time of suspension, the following command is used:

```
      )SIV
*EVAL       [12] A          SUM       X
```

It's good practice to keep the State Indicator clear of all listings because they take up valuable space within the Active Workspace which could be used for other activities. The State Indicator should be cleared as soon as its contents are no longer needed. This can be done in either of two ways, other than clearing the entire workspace:

```
      →0
```

Or just

```
      →

      )SI
⍉
```

The →0 operation erases the latest suspended function from the State Indicator listing and reactivates any related pendant functions at the line numbers listed by the *)SI* command.

The → operation erases one suspended function and all related dependent functions from the State Indicator list. A branch arrow is required for every suspended function that appears in the State Indicator listing. If there are two such items listed, the following would be required to clear the State Indicator:

```
      →
      →
```

Input - Output

To be completely interactive with the user, a defined function must have the ability to print out items on the terminal as well as accept terminal input at various stages of its execution.

Numeric Input

To accept numeric input from the terminal, the symbol used is called the Quad Symbol, ⎕, (uppershift *L*).

```
∇SORT;X
 [1] 'ENTER DATA'
 [2] X←⎕
 [3] 'THE DATA SORTED IN ASCENDING SEQUENCE IS AS FOLLOWS:'
 [4] X[⍋X]
∇

       SORT
ENTER DATA
⎕      20 16 21 15 17 22 18
THE DATA SORTED IN ASCENDING SEQUENCE IS AS FOLLOWS:
15  16  17  18  20  21  22
```

After the first line of *SORT* is displayed, the Quad Symbol is typed and the typing element spaces into position 6 where the keyboard then unlocks. The user is expected to type in the required data at this point to carry out the rest of the execution of *SORT*.

Here are some examples of the use of the Quad Symbol while the system is not executing a function:

```
       6+⎕
⎕      7
13

       ⎕-3
⎕      10
7
```

A good use for the Quad Symbol outside of functions occurs when several values have to be assigned to a variable; so many in fact that they cannot all be assigned on the same line. Here is a small sample showing how this situation could be handled using the Quad Symbol:

```
      A←1 2 3 4 5,⎕
⎕     6 7 8 9 10

      A
1  2  3  4  5  6  7  8  9  10
```

If a user is executing a function requiring him to type in numeric input, but he would rather exit from the function, he could do so by typing the following:

```
      →
```

He is immediately exited from the function.

Literal Input

To accept literal input, the Quote-Quad Symbol, ⍞, (uppershift *L* overstruck with uppershift *K*) is used. Here it is employed on line 2 of the function *QUES*1 to receive the answers typed by the user:

```
∇QUES1
 [1] 'WHAT IS THE CAPITAL OF CANADA?'
 [2] →Q2×⍳∧/'OTTAWA'=6⍴⍞
 [3] 'WRONG.  TRY AGAIN.'
 [4] →2
 [5] Q2: 'RIGHT'
∇

      QUES1
WHAT IS THE CAPITAL OF CANADA?
TORONTO
WRONG.  TRY AGAIN.
OTTAWA
RIGHT
```

After line 1 of *QUES*1 is typed, the typing element returns to its starting position and "twitches" to indicate it expects literal input.

```
      A←⍞
I CAN'T FIND IT.

      A
I CAN'T FIND IT.

      B←'IT''S OVER THERE.'

      B
IT'S OVER THERE.
```

When a quotation mark is desired within a literal, two quotes must be typed together to represent one. When input is required for the Quote-Quad Symbol, the literal string is stored exactly as it is typed.

If the user is expected to reply to a Quote-Quad operation of a function, but would rather not answer and leave the function entirely, a special symbol is available.

```
∇LOOP;A
 [1] A←⍞
 [2] →1
∇

      LOOP
STOP
HELP
⍝
```

The last input released the user from *LOOP* and took him out of the function. This symbol is typed as *O*, backspace *U*, backspace *T*.

Output

The Quad Symbol, ⎕, is used to print out both numeric and literal data. To do this, the specification arrow is placed to the right of the Quad Symbol instead of its left as it was in the cases where the Quad was used as an input operation.

```
      ⎕←A←6+7
13

      A
13
```

It may be used in calculations also.

```
      A←7+⎕←6+7
13

      A
20

∇MEAN;X
 [1] 'THE MEAN IS ';(+/X)÷ρX←¯1↓⎕,ρ⎕←'ENTER DATA'
∇

      MEAN
ENTER DATA
⎕     ⍳10
THE MEAN IS 5.5
```

The expression ¯1↓⎕,ρ⎕← above allows the literal *ENTER DATA* to be displayed from line 1 while not interfering with the rest of the line. This is accomplished in the following manner:

1. The literal is displayed

2. ρ then determines how many characters were displayed. (In this case it is 10)

3. This is then catenated to the requested input to produce the vector 1 2 3 4 5 6 7 8 9 10 10.

4. The last element is then dropped from this vector (i.e., the 10 that represents the length of the literal output) and the result is assigned to the variable *X*.

Here is another example of the same thing:

```
∇QUIZ
 [1] →1×⍳(×/X)=1↑⎕,⍴⍞←'WHAT IS ';(X[1]);'×';1↓X←?10 10
 [2] →1,⍴⍞←'NO.  THE ANSWER IS ';×/X
∇
```

The expression 1↑⎕,⍴⍞ in line 1 above, the rho function, ⍴, takes the size of the output literal statement. If this were not done, a *DOMAIN ERROR* message would occur because literal data cannot be catenated to numeric data. The 1↑ operation allows only the first element typed in to be compared to the product of *X*. In line 2, the Rho function is used again to perform the same task. Because the branch function only recognizes the first element of a vector, the system will always return to line 1.

Here is an example of the function *QUIZ* being used:

```
      QUIZ
WHAT IS 7×4
⎕     12
NO.  THE ANSWER IS 28
WHAT IS 6×9
⎕     54
WHAT IS 2×2
⎕     4
WHAT IS 1×8
⎕     →
```

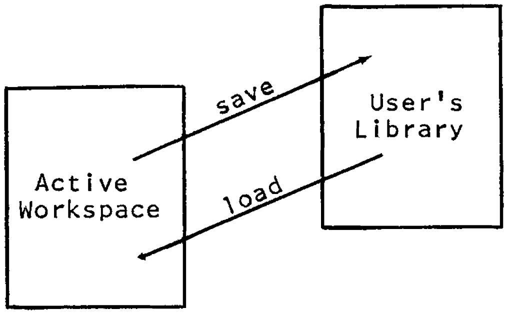
The → terminated the exercise.

Chapter Thirteen

LIBRARIES

So far, all computer activities have taken place in an area known as the Active Workspace. Calculations and writing of functions are done in this area. But it is only a temporary area set up for the user while he is signed on. As soon as he terminates his APL session with the sign off command, the entire contents of his Active WS are erased from the system. Any functions or variables he may have created since signing on immediately vanish and cannot be recalled at a later session. This is not a desirable feature, especially if the user has created something which he would like to use again at a later date. But it is necessary because the computer would not be able to store every Active WS after every terminal session that takes place. This would require a computer of enormous size to have this capability. APL does make provision for the user who does want to sign on later and re-use certain routines he has created, by allowing him to store these routines in an area called his Library.

The Library space assigned to each user when he receives his APL account number is a permanent storage area that resides in the system as long as the user's account number is valid. Here, items from the Active WS can be saved indefinitly and loaded back into the Active WS area only when requested.



To place an item into the library, the)*SAVE* command is used. As an example, suppose there is a function called *ASORT*1 currently in the Active WS which the user wishes to save in his library for future use. Here is how he would perform the task:

```
      )SAVE ASORT1
SAVED  9.06/ 72.264/  608
```

After the)*SAVE* command was issued, the system printed out a message stating the time and date, and how much library space was required to save the item. *ASORT*1 was saved at 9:06 AM on the 264th day of 1972 (September 20), and it took up 608 bytes of library space.

When *ASORT*1 was saved in the library area, only a copy of the original function *ASORT*1 in the Active WS was saved. A quick check to make sure that *ASORT*1 still resides in the Active WS should verify this.

```
      )FNS
R←ASORT1
```

Therefore, *ASORT*1 is still in the Active WS and can still be used.

```
      ASORT1
ENTER DATA.
TOM
DICK
HARRY
b̸

DICK
HARRY
TOM
```

(*ASORT*1 sorts literal data, a column at a time)

```
      )FNS ASORT1
∇R←ASORT1;I;J;L;V
 [1] I←2+L←ρV←0ρ⎕←'ENTER DATA.'
 [2] →(0≠¯1↑L←L,(ρV←V,⍞)-+/L)/2
 [3] R←(((ρL)-1),⌈/L←¯1↓L)ρ' '
 [4] J←⍳(+/IρL)-+/(I-1)ρL
 [5] R[(I-1);J]←(+/(I-1)ρL)↓(+/IρL)↑V
 [6] →((ρL)≥I←I+1)/4
 [7] L←1↓ρR
 [8] R←R[(⍋R[;L]);]
 [9] →8×⍳0<L←L-1
∇
```

And to make sure a copy of *ASORT*1 did in fact get stored in this particular user's library area, the following command is used:

```
      )LIB
ASORT1
```

Notice the *)LIB* command lists only the names of the functions while the *)FNS* command includes their syntax.

If the user wished to load the copy of *ASORT*1 back from his library into his Active WS, he would issue the following command:

```
      )LOAD ASORT1
DATA IN WS
```

After the user typed the *)LOAD* command, the computer replied with the message *DATA IN WS* which means, "you already have a thing called *ASORT*1 in your Active WS, therefore this command is being ignored". Obviously, two items with the same name cannot be in the Active WS area at the same time. So instead of replacing the present one with the stored copy, the system leaves the decision up to the user as to whether he really wants the stored copy in the Active WS in place of the present one. The only way to get the stored copy back into the Active WS is by erasing the present copy.

```
      )ERASE ASORT1
OK

      )FNS
ø
```

Since *ASORT*1 no longer resides in the Active WS, it cannot be used.

```
      ASORT1
ASORT1
? VALUE ERROR
```

Now a copy of *ASORT*1 may be loaded from the library.

```
      )LOAD ASORT1
SAVED  9.06/ 72.264/  608
```

To indicate that a copy of *ASORT*1 has been loaded successfully, the system replies with the same message it produced for the)*SAVE* command. Now again there are copies of *ASORT*1 in both the Active WS and the library.

```
      )FNS
R←ASORT1

      )LIB
ASORT1
```

Although a member of the Active WS cannot be replaced by the library member, the same is not true for library members.

Suppose the user tried to resave *ASORT*1, even though it already was in his library.

```
      )SAVE ASORT1
REPLACED  9.30/ 72.264/  608
```

The copy of *ASORT*1 that was in his library was replaced by the Active WS copy. Because both copies were identical, no harm was done. But suppose there was a 200 line function in the library and a variable containing only three values in the Active WS, both with the same name. That 200 line function would be lost and the user would be left with a 3-element vector in his library. Therefore, it pays to check the names of the library's contents before any saving is attempted, just to be sure that valuable data isn't lost.

Saved Workspaces

Functions and variables contained in the Active WS may be saved independently or collectively. If they are saved independently, only one item can be saved at one time (i.e., a *)SAVE* command is required for each member to be saved), and the user must be explicit in which item he wishes to save. As in the examples of saving *ASORT*1, the name of the item was stated after each *)SAVE* command. But quite often a user will wish to save several items together so that a separate *)SAVE* and *)LOAD* command won't be required to store and retrieve each one. To do this, he must save the entire workspace. This is done in the following manner:

```
      )SAVE
SAVED  9.45/ 72.264/ 4820
```

No name is stated after the *)SAVE* command to indicate to the system that the whole workspace is to be saved. Notice that even though an Active WS is approximately 32,000 bytes in size, only 4,820 bytes were saved. This is due to the fact that the items in this particular workspace only occupied 4,820 bytes. Only that amount of space required to store the items is used in order to free up as much available library space as possible so that other items can be saved.

Now take a look at what's in the library.

```
      )LIB
ASORT1     *CONTINUE
```

There are two things of interest here. One is that the saved workspace is called *CONTINUE* and the other is that it is preceded by an asterisk. The asterisk is placed there for a very good reason. Because there can be only one Active WS at one time, anytime a saved workspace is loaded into the area occupied by the Active WS, the present contents of the Active WS are replaced by the contents of the saved workspace. The asterisk is placed before the saved workspace name to warn the user of this event. The loading of saved functions and variables is quite different. They are merely appended to the present contents of the Active WS, causing no loss of data at all.

The name of the saved workspace is *CONTINUE* because the system assigns this name to any Active WS when the user signs on. The user is the only one who can change this name to something else. This is illustrated in Chapter 15. Saving workspaces under the name *CONTINUE* is risky because, if there is a break in the connection between the terminal and the computer, the Active WS is automatically saved in the user's library under the name *CONTINUE*. This means that any item called *CONTINUE* in the library is replaced by the present contents of the Active WS when the break occurs. This rule applies even if the)*WSID* is some other name.

Loading a saved workspace is done the same way as the loading of functions and variables.

```
      )LOAD CONTINUE
SAVED  9.45/ 72.264/ 4820

      )FNS
R←ASORT1
PROG

      )VARS
A         B̲         SUM       P1
```

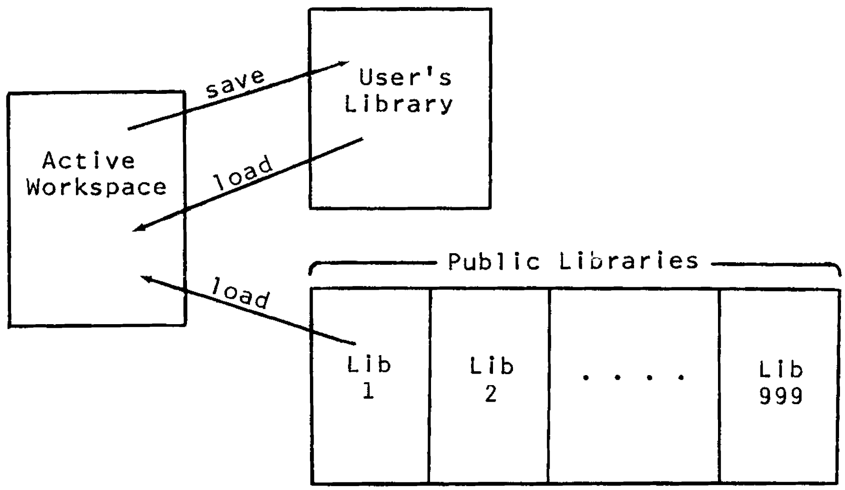
This particular workspace contains two functions and four variables.

Public Libraries

Apart from a user being able to load items from his own library, he is also allowed to load things from other libraries. Every APL system has a set of "Public Libraries" that contain a great variety of functions already written and documented so the user doesn't have to create his own. He need only load them into his Active WS and use them. Here's an example of a user loading a function called *DSTAT* from public library number 4:

```
      )LOAD DSTAT,4
SAVED 13.58/ 71.155/  616
```

To indicate to the system that it is to look in a library other than the user's, a comma and the number of the library which contains the item must follow the item's name.



A user can only load items from public libraries; he cannot save items from his Active WS into these libraries. He is restricted to using his own library for storing things.

To find out what is contained in each library, there is usually a member called *DESCRIBE* in each public library. It contains a brief description of the library. There is also a member called *INDEX* which lists the items contained in its respective library along with a brief description of their uses. Another way of listing the contents of another library is by issuing the)*LIB* command in the following manner:

```
      )LIB 70
*CASHVAL   *DEPRECN   DESCRIBE   INDEX    *LOANSCH   *RATERTN
```

Public library number 70 contains the above items.

Of course these library numbers and their contents will vary depending on the installation.

Private Libraries

Besides being able to list and load the contents of public libraries, it is also possible to do the same thing with other user's libraries. To issue a *)LIB* command on another person's library, his account number must follow the *)LIB* command.

```
      )LIB 3031
A          LINE       *GAMES
```

To load a member from someone else's library, the command is the same as that used to load items from public libraries; only the library number is different.

```
      )LOAD A,3031
SAVED 11.36/ 72.220/   40

      A
1  2  3  4  5
```

This could be very annoying to someone who wants to save something and not let others have access to it. So there is a feature that will accommodate this particular situation. The user has only to type a *,P* after his *)SAVE* command to attain this result.

```
      )SAVE PROG,P
SAVED 12.01/ 72.265/ 2987
```

Or, in the case of a privately saved workspace:

```
      )SAVE,P
SAVED 12.02/ 72.265/ 4089
```

Items saved in this manner will not appear in the computer's response to *)LIB* commands issued by some other user and they cannot be loaded by any other user.

Library Limits

Because the amount of library space allocated to each user is determined when his APL account number is added to the system, only a certain amount of data can be stored before all available space is gone. When this happens, the user is unable to save any more items.

```
      )SAVE REPORT
)SAVE REPORT
? LIBRARY FULL
```

A copy of *REPORT* could not be saved because there just wasn't enough room in the library to accommodate it. The user will have to either abandon his attempts at saving the item or get rid of some other members in his library to free up enough space. First, he must find out what is in there.

```
      )LIB
A          *FORCAST  *GAMES     TEXT
```

Then he must decide which item to drop to make room for *REPORT*. This user is going to see if *A* will give him the needed space.

```
      )DROP A
SAVED 15.33/ 72.215/ 1056
```

The computer responded with the same message that was printed when *A* was saved. This freed up 1,056 bytes, but is it enough?

```
      )SAVE REPORT
)SAVE REPORT
? LIBRARY FULL
```

Apparently *REPORT* requires more space than that. So he continues:

```
      )DROP GAMES
SAVED 16.23/ 72.195/  940

      )SAVE REPORT
SAVED 10.09/ 72.264/ 1357
```

This time it was successful and, as seen below, *REPORT* was saved and both *A* and *GAMES* were dropped from the library.

```
      )LIB
*FORCAST  REPORT    TEXT
```

There is one more system command that may be used to save the contents of an Active WS. Whenever this command is executed, the Active WS members are saved in the library under the name *CONTINUE* and the APL session is automatically terminated.

Here is an example of this command:

```
      )CONTINUE
SAVED 11.26/ 72.264/ 2056
010   11.26.37   09/20/72
CONNECTED    00.16.23   TO DATE   01.06.23
CPU TIME     00.00.15   TO DATE   00.12.04
```

Chapter Fourteen

DIAGNOSTIC AIDS

When a function containing a faulty expression is executed, it will either suspend execution at the point of error, produce incorrect results, or it may just continue executing forever in an "endless loop", unless the "ATTN" key is pressed. Sometimes it is difficult to isolate the "bug" that's caused the problem because the interruption of the function's execution may not be at the line that's at fault, but rather at a line that tries to use the erroneous calculation in some routine. Tracking down problems like this can be very frustrating and time consuming. So, two features have been incorporated into York APL to help the user with such problems.

Trace Feature

To follow the flow of logic through a function, the ability to trace this flow is necessary. The function *IN*, below, determines if *X* is contained in the matrix *Y*, and if so, in what row.

```
      )FNS IN
∇X IN Y
 [1] ROWS←(⍴Y)[1]
 [2] I←0
 [3] L3: I←I+1
 [4] →L8×⍳∧/X=Y[I;]
 [5] →(ROWS≥I)/L3
 [6] 'NO SUCH WORD.'
 [7] →0
 [8] L8: 'THE WORD ',X,' IS IN ROW ';I
∇
```

Here is how it works:

```
      X←'MICE'
      Y←4 4ρ'BIRDKNATMICEFISH'

      X IN Y
THE WORD MICE IS IN ROW 3
```

To trace the order in which the lines were executed, and the values produced by each line, the following would be typed:

```
      TΔIN←ι8
```

The expression *TΔ* (the Δ is uppershift *H*) indicates to the system that the function *IN* will have some of its lines traced. All the lines in this case will be traced. ι8 is the same as writing out all the numbers from 1 to 8.

Here is what the execution of *IN* looks like with the Trace feature:

```
      X IN Y
IN[1] 4
IN[2] 0
IN[3] 1
IN[4]
IN[5] 3
IN[3] 2
IN[4]
IN[5] 3
IN[3] 3
IN[4] 8
IN[8] THE WORD MICE IS IN ROW 3
THE WORD MICE IS IN ROW 3
```

The results obtained for each line are printed after the function name and the line number. For example, the value of *ROWS* in line 1 is set to 4, the number of rows contained in *Y*. Nothing is printed the first two times line 4 is executed because its result is an empty vector both times.

Notice the looping that occurs between lines 3 and 5. Line 5 branches to line 3 twice and then line 4 branches to line 8. Each time line 3 is encountered, the variable *I* is incremented by 1.

Tracing every line of *IN* is not really necessary because the only really relevant lines are 3, 4 and 5 as this is where all the searching for *X* in *Y* takes place. The other lines either set up variables or format the output, so they can be ignored by the Trace feature.

Here is how the Trace is changed to only display lines 3, 4 and 5:

```
      TΔIN←3 4 5
```

Now the function *IN* is executed again, but this time it is being asked to find a word that doesn't exist in *Y*.

```
      'FROG' IN Y
IN[3] 1
IN[4]
IN[5] 3
IN[3] 2
IN[4]
IN[5] 3
IN[3] 3
IN[4]
IN[5] 3
IN[3] 4
IN[4]
IN[5] 3
IN[3] 5
→L8×ι∧/X=Y[I;] :[4]IN
         ? DOMAIN ERROR
```

Why? The last line 3 of the Trace indicates the *I* has been incremented by 1 until it now equals 5. But there are only 4 rows in *Y*. Therefore, there is a flaw in the logic of this function. Line 5 states "if *ROWS* is greater or equal to *I*, branch back to line 3". But what should have happened when *I* equalled 4 is that no branch occur and the system fall through to line 6. The way line 5 is supposed to read is "as long as *ROWS* is greater than *I*, branch back to line 3".

Here is the fix required:

```
      ∇IN[5]→(ROWS>I)/L3∇
```

Now for a second try:

```
      'FROG' IN Y
IN[3] 1
IN[4]
IN[5] 3
IN[3] 2
IN[4]
IN[5] 3
IN[3] 3
IN[4]
IN[5] 3
IN[3] 4
IN[4]
IN[5]

NO SUCH WORD
```

To find out which lines of a function are being traced, the following is typed:

```
      T∆IN
3  4  5
```

In this case, it's lines 3, 4 and 5.

To take a trace off a function, there are two methods:

```
      T∆IN←0
```

or

```
      T∆IN←⍳0
```

Stop Control

Similar to the Trace feature, the Stop Control halts execution of a function at predetermined lines.

To stop the execution of the function *IN* just before it gets to line 3, the following would be typed:

```
      SΔIN←3
```

Now *IN* is executed:

```
      X IN Y
IN[3]
```

The function terminated just before executing line 3 and the system displayed the function name and the next line number to be executed. The user is free to do any calculations or function displays he wants because the system just acts as if the function had become suspended, which in fact it has.

To reactivate *IN* at line 3, the same instruction is used as with suspended functions.

```
      →3
```

The user could also have branched to any other line the same way as the "branch to 3" was accomplished.

To find out at which lines the Stop Control has been employed, the following is used:

```
      SΔIN
3  5  7
```

This function will stop its execution everytime it goes to execute lines 3, 5 and 7.

Removing the Stop Control from a function is the same as with the Trace feature.

```
      SΔIN←0
```

or

```
      SΔIN←⍳0
```

Error Trap

The Error Trap feature of York APL is not really one of the Diagnostic Aids that are available, but rather is used as a "preventive" tool. It is evoked and suppressed in the same manner as the Trace and Stop features, therefore it's included in this chapter.

While executing a function, it is often advantageous to have certain checks and comparisons included to make sure that improper data don't become arguments of certain functions. For instance, a literal should never be allowed to become an argument for a "Plus-Reduction" operation; zeros should never be permitted to act as divisors. These things are hard to control, especially in programs requiring input from the user at certain stages of their execution.

To make sure that he has entered the right data, elaborate routines could be written to do nothing but edit his input. Or, York APL's feature for "trapping" such problems before they cause any damage could be employed. This feature is called the Error Trap.

Here's a sample program being executed without the Error Trap being applied:

```
      INDEX
THIS VECTOR CONTAINS 5 ELEMENTS,
WHICH ONE WOULD YOU LIKE?
⎕     10
A[⎕] :[4]INDEX
 ? DOMAIN ERROR
```

Here it is again; this time with the Error Trap:

```
      INDEX
THIS VECTOR CONTAINS 8 ELEMENTS,
WHICH ONE WOULD YOU LIKE?
⎕     10
THAT NUMBER IS OUTSIDE THE DIMENSIONS
OF THIS VECTOR.  TRY AGAIN.
⎕     7
45
```

By using the Error Trap, the problems experienced in the first example were avoided in the second.

Here is how the Error Trap was evoked:

```
EΔINDEX←5
```

This means that any errors encountered in line 5 of *INDEX* will be handled by the Error Trap feature if they are of a certain type (see list below). This avoids the function from becoming suspended.

Not all errors can be trapped. But here is a list of the ones that do fall within its domain:

1 *DOMAIN ERROR*
2 *SYNTAX ERROR*
3 *VALUE ERROR*
4 *LENGTH ERROR*
5 *RANK ERROR*
6 *NUMBER TOO BIG*
7 *DIVISION BY ZERO*

Along with *E*Δ and the line number(s) to be checked, the first line of the function is reserved for a special I-beam function. I-beam functions are described in chapter 15. If an error is encountered while the function is executing, this I-beam function (I28) is set to the value corresponding to the error committed. If a *VALUE ERROR* should occur, I28 would contain the value 3. For a *RANK ERROR*, I28 would be equalled to 5. Each value that I28 may be set to corresponds to the numbering sequence of the error listing above. If no error occurs, I28 remains as an empty vector (⍳0).

Here is what the function *INDEX* looks like:

```
∇INDEX;A
 [1] →(1=I28)/ERR
 [2] A←(?10)?100
 [3] 'THIS VECTOR CONTAINS ';(⍴A);' ELEMENTS,'
 [4] 'WHICH ONE WOULD YOU LIKE?'
 [5] A[⎕]
 [6] →0
 [7] ERR: 'THAT NUMBER IS OUTSIDE THE DIMENSIONS'
 [8] 'OF THIS VECTOR.  TRY AGAIN.'
 [9] →5
∇
```

The statement in line 1 compares I28 to the value 1 (checking for a *DOMAIN ERROR*). If one is encountered on line 5, as it was in both the previous examples, the system is to branch to line 7, if the Error Trap feature has been evoked. It was only in the second example.

Here is another example.

As most primitive functions can be either Monadic or Dyadic, so too can the user defined functions. In other words, functions that are defined as being Dyadic can be used monadically! The following function will illustrate this.

The function *RND* rounds the values contained in the right argument to the prescribed number of digits specified in the left argument.

```
      2 RND 56.785
56.79
```

But, if the left argument is omitted, it will round the right argument's values to their nearest integers.

```
      RND 56.785
57
```

By displaying the function, how this was done should be evident.

```
      )FNS RND
∇R←N RND X
 [1] →(3=⍸28)/4
 [2] R←(10*-N)×⌊0.5+X×10*N
 [3] →0
 [4] N←0
 [5] →2
∇
```

The only line that can't be trapped is line 1, the line containing ⍸28.

To take the Error Trap off the function, the following is typed:

```
      E∆RND←0
```

or

```
      E∆RND←⍳0
```

To find out which lines are being trapped, the following is typed:

```
      E∆RND
2
```

VALUE ERROR's on line 2 of *RND* were being trapped.

Chapter Fifteen

MORE SYSTEM COMMANDS AND THE I-BEAM FUNCTIONS

When a user signs on to the APL system he is always issued a clear Active Workspace. But this Active WS has many more attributes besides being void of user defined functions and variables.

One such attribute is the origin of the Active WS. If a string of numbers is generated by using the index operator, ⍳, the result is always a vector of integers beginning at 1. This means that the origin of the Index Generator has to be set at 1. A quick way to check it is with the *)ORIGIN* command.

```
      )ORIGIN
IS 1
```

The origin of the Active WS can either be 1 or 0. The way to change it from 1 to 0 is as follows:

```
      )ORIGIN 0
WAS 1
```

In origin 0, the index generator begins at 0 instead of 1.

```
      ⍳4
0  1  2  3
```

Another feature of an Active WS is that the maximum amount of output possible on one line is 130 characters. The *)WIDTH* command is used to check this.

```
      )WIDTH
IS 130
```

To alter the number of characters allowed per line, the system command)*WIDTH* is again used; this time followed by the prescribed number of characters.

```
      )WIDTH 30
WAS 130
```

The width may vary anywhere from 30 to 130 characters. This comes in quite useful when a document of unknown width has to be written on 8.5 by 11 inch paper.

If a value, such as "pi" is displayed, the system prints out only the first 10 digits, rounding the last digit to the appropriate value.

```
      ○1
3.141592654
```

The system rounds at ten because of a default built in. To make sure it is set to ten, the user could find out by typing the following command:

```
      )DIGITS
IS 10
```

The number of digits printed can vary from 1 to 16. The number specified is also done by use of the)*DIGITS* command.

```
      )DIGITS 16
WAS 10

      ○1
3.141592653589791
```

Changing the digits displayed has no effect on the variables or calculations taking place inside the computer. Here, every computation is carried out to sixteen decimal places, independent of the)*DIGITS* setting.

Every Active WS has its own "name". A clear Active WS is called *CONTINUE*. To display the workspace name or ID, the following command is used:

```
      )WSID
CONTINUE
```

To change the name of the current workspace to something else, the user would type the following:

```
      )WSID WS1
WAS CONTINUE
```

where *WS*1 is the new name of the Active WS.

The way to erase all the contents of the present Active WS and reinstate the name *CONTINUE* as its ID is with the following command:

```
      )CLEAR
OK
```

There are features of three other system commands mentioned earlier that may prove useful. One is that the *)ERASE* command is able to erase several members at once from the Active WS. Here's an example:

```
      )ERASE A SIMULATE SUM
OK
```

The names of the members must be separated from each other by at least one space. If one of the names is misspelled, it is not erased, but the others are.

```
      )VARS
CASHFLO   DEBIT     NETVAL

      )ERASE CASHFLOW DEBIT NETVAL
CASHFLOW/ VALUE ERROR

      )VARS
CASHFLO
```

An option to both the *)OFF* and *)CONTINUE* commands is the *HOLD* feature. By typing a space and the word *HOLD* after either of these two commands, the connection between the computer and the terminal is not broken. This is most useful for those terminals that use a telephone to obtain connection to the computer. The computer doesn't have to be redialed each time someone wants to use APL.

An Active WS may have the same name as any of its contents.

Communications With Other Users

Typing messages on one terminal and having them printed out on another is possible. The *)MSG* command is used to do this.

Here is an example of a message being sent to a user with the APL account number 5006:

```
      )MSG 5006,'HEY BILL, WHERE''S THE NEXT LAB? ...TED'
OK
```

The statement enclosed in quotes is the message. The *OK* indicates that the message was sent. If the person who was to receive the message was not signed on at that time, a message of *? USER NOT ON* would be typed to the sender. To see who is signed on, the *)PORTS* command is used.

```
      )PORTS
OPR  5006  5173  5261
```

The *OPR* represents the operator terminal. Messages sent to it are done so in the following manner:

```
      )OPR 'MESSAGE'
OK
```

Messages are typed at the receiving terminal only when its keyboard unlocks after the "RETURN" key is pressed and any printout that is to be done has terminated. This means that if a person is just sitting at his terminal and not pressing the "RETURN" key periodically, the message will be unable to print. Only when that "RETURN" key is pressed does a message have a chance of being displayed. If a person is expecting a message from another user, there is a command that will free him of pressing the "RETURN" key every few seconds to see if the message has been sent.

By simply typing the command *)WAIT*, the keyboard will lock up and only print incoming messages. When the user wishes to unlock the keyboard, he must press the "ATTN" key.

I-beam Functions

In every Active WS there are certain pieces of information concerning several aspects of the APL system. The user has access to this information by means of the I-beam functions which are formed by overstriking the ⊤ and the ⊥.

The first I-beam tells the user how much space is still available in his library.

```
      ⌶17
2364
```

This user has 2,364 bytes of unused library space left.

Here is a break-down of how much space is required for the various elements of the APL system:

> 1 literal character (i.e., '*A*') takes up 1 byte.
> Numeric values 0 and 1 take up only one eighth of a byte each.
> All other integers up to 2*24 can be stored in 4 bytes each.
> Everything else is stored in 8 bytes.

These figures apply to both the user's library and his Active WS.

```
      ⌶20
3259810
```

I-beam 20 contains the time of day in sixtieths of a second. It is more meaningful when converted to hours, minutes, seconds and sixtieths of seconds.

```
      24 60 60 60⊤⌶20
15  5  30  10
```

When this I-beam was executed, it was fifteen hours, five minutes thirty seconds and ten sixtieths of a second into the day, or approximately 3:05 PM. The function ⊤ used above is discussed in the next chapter.

```
      ⌶21
4
```

The ⌶21 function returns the total amount of time, also in sixtieths of a second, that the computer has been utilized since Sign-on time. So far, during this session, 4 sixtieths of a second of the computer's time have been used. This I-beam can be quite useful in timing a function's performance by storing the current ⌶21 , executing the function, then subtracting the old value from the new one.

An Active WS contains approximately 32,000 bytes. As various functions and variables are loaded or created, the number of available bytes decreases. To determine how many remain unused, the following I-beam is used:

```
      ⌶22
17065
```

There are over seventeen thousand bytes of space still left in this Active WS.

```
      )CLEAR
OK

      ⌶22
31672
```

A clear workspace has exactly 31,672 bytes of available space.

```
      ⌶23
6
```

This last I-beam tells how many other users are currently signed on. There are six.

```
      1⌶23
3065
```

The account number of the person using this terminal is 3065.

```
      3065=1⌶23
1

      2⌶23
3001  3031  3065  3243  3313  3400
```

2⌶23 returns the APL numbers of the users presently using the system. It is the same as the *)PORTS* command, except these numbers can be used in calculations.

```
      ⌶24
105535
```

To determine how long a user has been signed on during the current session, the I-beam 24 function is used. It too is calculated in sixtieths of a second.

```
      24 60 60 60⊤⌶24
0  29  18  55
```

This user has been on for almost half an hour.

⌶25 returns a value representing the date in the form month, day, year.

```
      ⌶26
2
```

⌶26 contains a value if there is a suspended function within the Active WS. It indicates on what line the most recent suspension took place. A quick check of the State Indicator should confirm it.

```
      )SI
*GET        [2]
PART2       [5]
MAIN        [9]
```

The astersik preceding *GET* indicates that it is a suspended function. The other function names listed are called pendant functions. Apparently, the function *MAIN* called the function *PART*2, which in turn called *GET* which abnormally terminated. The resumption of these pendant functions is dependent on the resumption and successful completion of *GET*.

```
      ⌶27
2  5  9
```

This last I-beam contains the values of the line numbers listed in the State Indicator above. Notice that the ⌶26 value is the first element of the ⌶27. If the State Indicator is empty, I-beams 26 and 27 return empty numeric vector results.

⌶28 is used in the Error Trap feature discussed earlier. It is always located on the first line of a function and contains the values 1 to 7 or ⍳0 only. For a more detailed description of its use, the Error Trap feature should be consulted.

⌶29 takes, as its left argument, the name of a function enclosed in quotes, and returns a literal vector of that function.

```
      )FNS SQRT
∇R←SQRT X
 [1] R←X*0.5
∇

      A←'SQRT'⌶29

      A
[0]R←SQRT X
[1]R←X*0.5

      ⍴A
23
```

Converting functions to vectors is used in user defined functions to alter portions of lines of other functions. There are usually several of these functions on the system.

Another use of this I-beam is in storing function on OS files. They have to be converted to literals before they can be written on this type of file. OS files refer to the typical kinds of files used by other computer languages that operate in an Operating System environment. The Computer Services Department of York University should be consulted for more information in this area.

I-beam 30 packs or converts its literal vector left argument into a hexadecimal value. The left argument consists of an even number of characters all of which must be of the numbers 0-9 and A-F to be in agreement with the base 16 hexadecimal numbering system. This I-beam is most useful in creating variables that, when displayed, influence the behavior of the terminal. There are six such variables currently available.

```
      CR←'C0'⌶30

      10⍴'*',CR
*
*
*
*
*
```

After typing out each asterisk, the variable *CR* causes the typing element to return to typing position zero and evokes a line feed before the next asterisk is displayed.

```
      LF←'D0'I30

      10ρ'*',LF
*
 *
  *
   *
    *
```

LF causes a line feed only; no carriage return occurs.

```
      BS←'E0'I30

      'TITLE',(5ρBS),5ρ'_'
TITLE
```

The variable *BS* causes the typing element to backspace one position.

```
      TAB←'90'I30

      10ρ'*',TAB
*     *     *     *     *
```

The *TAB* variable allows the typing element to skip across the carriage, according to the tab settings on the terminal, before printing the next character. To insure that the system waits until the typing element has skipped to its new position before it prints the next character, the tab settings should be less than an inch apart. Or, if the spacing between each tab setting is greater than one inch, the following variable could be used.

```
      WAIT←'80'I30

      9ρ'*',TAB,WAIT
*          *          *
```

One *WAIT* is equivalent to the time it takes the terminal to print out one character. On the IBM 2741, this is approximately one tenth of a second.

The last packed hexadecimal literal, '*A0*', prevents the typing element from both line feeding and carriage returning. The following function illustrates its uses.

```
      )FNS ADD
∇ADD
 [1] A←?20 20
 [2] (A[1]);'+';(A[2]);'=',NCR
 [3] →1×⍳⎕=+/A
 [4] 'WRONG.  THE ANSWER IS ';+/A
 [5] →1
∇

      NCR←'A0'⌶30

      ADD
16+12=30
10+19=39
WRONG.  THE ANSWER IS 29
```

Notice the input is requested on the same line as the question is printed and the Quad that usually accompanies the numerical input request is not printed.

I-beam 31 is the inverse of I-beam 30. It unpacks its left argument to produce a literal of hexadecimal notation.

```
      NCR⌶31
A0
```

⌶32 tells how many terminals are currently in use. This figure should be identical to ⌶23.

```
      ⌶32
6
```

If the left argument of this last I-beam is a 1, the computer will return the decimal port number at which the user is signed on.

```
      1⌶32
29
```

This user is signed on at terminal number 29. It's the same number that appears on the first line of the Sign Off message.

A left argument of 2 will list all the terminal numbers that are currently signed on.

```
      2⌶32
29  31  34  35  37  38
```

The terminal numbers listed here are in the same sequence as the port number produced for either 2⌶23 or the)PORTS command.

When comparing two numbers, APL is accurate to the first fifteen significant digits.

```
      2=2.0000000000001
0

      2=2.00000000000001
1
```

To determine the current fuzz setting, the following is typed:

```
      ⌶38
3.330669074E¯15
```

To reduce or increase this accuracy, the "fuzz factor" must be altered. Here it is changed to a significance of 2 decimal places.

```
      3E¯2⌶38
3.330669074E¯15
```

When this last function is executed, the previous fuzz setting is printed.

Now two simple comparisons produce rather interesting results.

```
      2=2.1
0

      2=2.01
1
```

To find out the names of the global variables presently in the Active WS, the)*VARS* commands could be used. Or, I-beam 39 may be executed to return a literal result containing all the global variable names.

```
      ⌶39
A        B          SUM        TOTAL
```

This I-beam, when used in conjunction with the Unquote function, which is discussed later, can usually save some typing when erasing all variables.

```
      ⍎')ERASE ',⌶39

      )VARS
⍉
```

The last I-beam ⌶40, lists the names of all the functions currently residing in the Active WS.

```
      ⌶40
ASORT      REPORT
```

It too can be used with the Unquote function.

```
      ⍎')ERASE ',⌶40

      )FNS
⍉
```

Chapter Sixteen

ADDITIONAL PRIMITIVE FUNCTIONS AND THE IDENTITY ELEMENTS

The following primitive functions are more advanced than those previously described, mainly because their algorithms to solve the problems are more complex, and because they expect the user to be quite familiar with the basic APL language in order to use them properly.

Base Value (Decode)

The Base Value or Decode function ⊥ (uppershift *B*), is used to convert a vector of values from one numbering system to another. This could mean changing the numbering base of a set of values from base 2 to base 10 or vice versa. Or, where there are mixed measurements for related values, the different "weighted" measurements would be stated. An example of this "weighted" problem is in finding the answer to the question, "How many inches are there in 14 yards, 2 feet and 7 inches?". Here is the solution using the Base Value function:

```
      1 3 12⊥14 2 7
535
```

The left argument, called a radix vector, states the relationship between inches, feet and yards (there are 12 inches in a foot and 3 feet to a yard).

How many seconds are there in 8 hours, 45 minutes and 16 seconds?

```
      1 60 60⊥8 45 16
31516
```

Elements two and three of the left argument indicate there are sixty seconds in one minute and sixty minutes in one hour. The number one is the first element because the question only goes as high as hours. If days were involved, the 1 would be changed to 24 to represent 24 hours in a day.

What is the binary value 0 1 0 0 1 in base 10?

```
      2 2 2 2 2⊥0 1 0 0 1
9
```

Or, because the left argument is all 2's:

```
      2⊥0 1 0 0 1
9
```

How many pints are there in 2 gallons, 3 quarts, and 1 pint?

```
      1 4 2⊥2 3 1
23
```

The algorithm for the Base Value function used to solve this last example is as follows:

```
To find out how many pints in one gallon: ×/4 2   or 8
To find out how many pints in one quart:  ×/2     or 2
To find out how many pints in one pint:   ×/      or 1
```

```
      +/8 2 1×2 3 1
23
```

Representation (Encode)

The Representation or Encode function, ⊤ (uppershift *N*), is the inverse of the Decode function. It "breaks up" the right argument according to the values contained in the left argument.

How many days, hours, minutes and seconds are there in 320756 seconds?

```
      1 24 60 60⊤320756
0  17  5  56
```

What is the binary notation for the decimal value 7?

```
      (5⍴2)⊤7
0  0  1  1  1
```

How many yards, feet and inches are there in 436 inches?

```
      1760 3 12⊤436
12  0  4
```

The answer to the above problem was arrived at by the following process:

1. The last element of the left argument (12) was divided into 436 to produce a quotient of 36 and a remainder of 4. This 4 then became the last element in the answer.

2. The quotient 36 was then divided by the next element in the left argument (3) to produce a quotient of 12 and a remainder of 0, the second element in the answer.

3. 1760 was then divided into the quotient of 12 to produce a quotient of 0 and 12 remainder, the first element of the answer.

Dollar Sign

The Dollar Sign function $\mathit{\$}$ (S overstruck with uppershift M), is used to format numerical data. The left argument is the "mask" which determines how the result will look. The right argument is an array of any dimension. Here's an example:

```
      A←⍳4

      '9999.9'$A
   1.0   2.0   3.0   4.0
```

Here's another:

```
      '999.99'$2.21
  2.20
```

What happened here? The result should have been 2.21. The reason for this descrepancy is a combination of both computer limitations and characteristics associated with the Dollar Sign function.

First of all, the computer has only 8 bytes in which to store real numbers. Therefore, a repeating decimal number such as .33333.... must be truncated to fit into the storage space available, thus making the stored value not exactly equal to the repeating decimal value. But how does all this relate to the non-repeating decimal number 2.21? Well, every number used in APL, with few exceptions, is converted from decimal to hexadecimal (base 10 to base 16), for storage reasons. This conversion process causes most real decimal numbers to become repeating hexadecimals. Therefore, the number 2.21, when converted to hexadecimal in the computer, is really only equal to about 2.2099.. Because the left argument of the Dollar Sign function asked for only two digits to the right of the decimal point of the right argument, and because the Dollar Sign truncates its right argument's values instead of rounding them to the desired degree of accuracy, the value 2.20 was printed.

The way to prevent these incorrect values from appearing is to add a "fuzz factor" to each value. So, instead of applying the Dollar Sign to a number like 2.21, it will really be applied to the number 2.215 (the number 2.21 with a "fuzz factor" of 0.005 added to it) to return the result of 2.21.

Here it is with the "fuzz factor" added:

```
      '999.99'⍕2.21+0.005
  2.21
```

When the right argument contains negative values, the "fuzz factor" must be subtracted, as shown in the following examples:

```
      '9999.99'⍕¯2.21
  ¯2.20

      '9999.99'⍕¯2.21-0.005
  ¯2.21
```

But, in most cases, both positive and negative values may appear in the right argument. So, the signs of these values must be determined in order to know whether the fuzz factor is to be added or subtracted. This is a good use of the Signum function.

```
      A←¯6.67 2.21 1

      ×A
¯1  1  1

      '999.99'⍕A+0.005××A
¯6.67  2.21  1.00
```

The vector "mask" can contain a vast assortment of characters, mainly to indicate certain characteristics that the result is to take. Here are some examples of different masks applied to vector *Q*.

```
      Q←1 ¯97 0 6 4726

      '999'⍕Q
   1¯97    6726
```

This last example produced two significant points in its result. Point one is the zero contained in *Q* appeared as a space in the result. Point two, the first digit of the number 4726 is missing in the result. To obtain the correct response, the mask must have a zero as its last character (or as any character) to force all values to be printed, starting at the position of the zero character. To correct the second problem, the length of the mask must be extended by at least the number of digits in the largest value contained in the right argument.

```
      '99990'⍕Q
    1  ¯97    0    6 4726
```

Another useful feature of the mask is that other characters besides decimal points, zeros and nines can be used to give the results more meaning.

```
      '999,990.99'$Q
      1.00      ¯97.00       0.00       6.00  4,726.00
```

In the above example, the comma is printed only for values which are greater than or equal to 1,000. Actually, the comma can be placed anywhere within the mask, but it is usually used to separate numbers into thousands.

Here's an easy way to format the date:

```
      '909/99/99'$I25
 09/20/72
```

Normally, for negative values, the $ operator will "float" the ¯ (the negative characteristic). The user may alter this by specifying any of the following "condition codes" as the <u>first character</u> of the mask, followed by the character to be "floated".

Condition Code	Meaning
+	Float only if data is positive
±	Float under all conditions
∘	Float under no conditions
- or ¯	Float only if data is negative

The desired "float character" must immediately follow the specified condition code.

```
      '±$999,990.99'$Q
     $1.00     $97.00     $0.00      $6.00 $4,726.00
```

But now the negative sign is missing from the value ¯97. So, to indicate negative numbers, the following mask could be used:

```
      '±$999,990.99CR'$Q
     $1.00       $97.00CR      $0.00        $6.00    $4,726.00
```

Another feature used to represent negative numbers is parentheses. Next page, the mask indicates to the computer that parentheses are to be placed around any negative numbers that appear in the right argument.

```
      '-(999,990.99)'$Q
      1.00      (97.00)       0.00         6.00   4,726.00
```

Or, for an application such as filling in dollar amounts on cheques, the following mask could be used:

```
      '-(*999,990.99)'$Q
******1.00*****(97.00)******0.00*******6.00***4,726.00*
```

The right argument is not limited to just vectors. It may be of any dimension.

```
      '9990.99'$2 3⍴⍳6
   1.00    2.00    3.00
   4.00    5.00    6.00
```

The dimensions of the result are arrived at by the following equation:

```
        (⍴RESULT) = (¯1↓⍴DATA),(⍴MASK)×¯1↑⍴DATA
```

where the function format is:

```
                RESULT ← MASK $ DATA
```

⍴*MASK* does not include any leading special control characters, *MASK* is a literal vector, and *RESULT* is character output.

Unquote

One of the most significant features of York APL is the Unquote function, ⍎, (uppershift *V* overstruck with uppershift *K*). This operator effectively adds a new dimension to APL's capabilities.

Primarily useful in user defined functions, the Unquote operates exactly as its name implies. It first removes the quotes from its argument and then executes that argument. A few examples should demonstrate this aptly.

Example 1 -

```
[4] .....
[5] ⍎')LOAD LEMSIM'
[6] .....
```

In the above, when line 5 is reached, the system would load the function *LEMSIM* into the user's Active Workspace and then go on to line 6. The terminal user would not be aware that a new function had been loaded (no *SAVED* .... message occurs in this situation). If an item called *LEMSIM* already resides within the Active WS, the statement is ignored.

It should not take too much imagination to visualize the usefulness of this capability. A main function could cause the loading of subfunctions conditionally, depending solely upon data or program conditions at that moment, or, optionally, different blocks of data could be requested by the program, again, without the terminal user having to issue any commands!

It should be readily seen that this capability helps overcome the Active WS size limitation of 32,000 bytes. A function can be defined so that it contains only the code deemed resident at all times. The rest of the function may be defined as subfunctions which can be loaded and erased when required. For those who are more familiar with programming concepts, this enables one to effectively "overlay".

Example 2 -

```
[6] .....
[7] ⍎⍞
[8] .....
```

In the above, when line 7 is reached, the system will request the terminal user to type in something. That "something", whatever it may be, would then be executed immediately and the system would carry on to line 8. This enables the user to write functions which permit him, or anyone using his functions, to perform calculations during the execution of functions. The user does not have to stop the function's execution in order to do his calculations. This means that the design of functions that will be fully interactive with the user at execution time is possible.

Example 3 -

```
      ∇ALTER
       [1] ⍎'∇PLOT[3]HS←1∇'
      ∇
```

The above demonstrates how one function can be altered via the execution of a second function. This gives the user the ability of altering certain segments of functions depending on the prevailing conditions at that moment.

Example 4 -

One slight problem that new APL users experience lies in the necessity for the user to issue two commands to obtain function execution. The first is the)*LOAD* to bring the function into the Active WS, and the second is the issuance of the function name to instigate function execution. Experience has shown that this does, in fact, cause a lot of difficulty.

With the use of the Unquote, this situation can be overcome to some extent. A function can be made to begin execution immediately after it has been loaded, as follows:

```
      )WSID HYP
WAS CONTINUE

      ∇R←HYP1;A
[ 1   ]⍎')SAVE'
[ 2   ]'THE FUNCTION JUST LOADED CALCULATES THE'
[ 3   ]'HYPOTENUSE OF A RIGHT ANGLED TRIANGLE.'
[ 4   ]'PLEASE ENTER THE LENGTHS OF THE OPPOSITE'
[ 5   ]'SIDES.'
[ 6   ]A←⎕
[ 7   ]R←((A[1]*2)+A[2]*2)*0.5
[ 8   ]∇
```

When *HYP*1 is executed for the first time, a copy of the then Active WS is saved in the user's library. Note that this saved workspace was executing a function at the time it was saved. This means that, whenever it is loaded back into the Active WS by the user, it will immediately resume execution at <u>line 2</u> of *HYP*1. The user does then <u>not</u> have to initiate the execution by typing in the function name.

Note: One prerequisite to using the)*SAVE* command along with the Unquote operator is that a copy of the item to be saved has to have been previously saved in the user's library and its size must be at least slightly larger than the workspace to be saved.

This can be accomplished by the following steps:

1. Once all the desired functions and variables have been created in the Active WS, insert the ⍎')*SAVE*' line in its proper place.

2. Load in or create an extra item that will take up at least 400 bytes of available workspace (i.e., *X*←⍳150 - this takes up approximately 600 bytes)

3. Save the workspace in the usual manner.

)*SAVE*

   Make sure the)*WSID* has the correct name.

4.)*ERASE X* (the variable created in step 2)

5. Execute the function that contains the Unquote.

Not all system commands can be used with the Unquote. The following can:

)*LOAD*)*ERASE*)*SAVE*)*WSID*)*DIGITS*)*ORIGIN*)*OFF*

The Unquote function differs from all other APL functions in that it does not produce an Explicit Result. Therefore, if a user wishes to use the result of an "unquoted" expression as input to another operation, he must do so by the use of a sub-function, like the one below:

```
      )FNS UNQUOTE
∇R←UNQUOTE X
 [1] ⍎'R←',X
∇

      6+UNQUOTE '3×4'
18
```

Because of this feature the ⍎ must be the first item in any APL statement, otherwise everything that appears to its left will be ignored.

There are plans to make the Unquote produce an Explicit Result whenever it does not appear in the first position of a statement. When these plans are implemented, any Unquote that begins a statement will not produce an Explicit Result, only Unquotes within statements will.

Null

The Null symbol ∘ (uppershift *J*) is most commonly used to perform Generalized Outer and Inner Product functions. This is its Dyadic use. Monadically, it acts quite differently.

In the expression

```
B←2∘A←1
```

the variable *A* will be assigned the value 1 and *B* will be assigned the value 2 only. The Null symbol tells the system to treat everything to its right as though it were on a separate line. For instance, the same operation could be carried out without the Null symbol in the following manner:

```
A←1
B←2
```

But by using the Null symbol, both computer time and connect time are reduced.

The ∘ could be employed in user defined functions to cut down on the number of lines required to state the problem solving algorithms which in turn would shorten the amount of time required to run it. Here are a few lines of a typical user defined function that could easily be reduced to one by the use of the Null operator:

```
 .
 .
 .
[4] V←ρJ←N
[5] I←SIGMA[J]
[6] V←V,G[I;J]
 .
 .
 .
```

Here are the same three expressions stated on one line:

```
 .
 .
 .
[4] V←V,G[I;J]∘I←SIGMA[J]∘V←ρJ←N
 .
 .
 .
```

The reason why it's advantageous to try to get as many calculations on one line as possible is that it helps speed up the overall performance of the function. This should not be carried too far because the longer the line is, usually the harder it is to understand what the line does.

After each line of a function is executed, the APL system must carry out certain "housekeeping" routines. Such things as updating ⌶26 and ⌶27 and checking to see whether the Trace and/or Stop features have been employed on the function must all be done before the next line can begin to be evaluated. All these things take time. Therefore, by reducing the number of lines in the function, the time required to execute it is also reduced.

Identity Elements

In Chapter 3, some Monadic uses for a few of the Scalar functions were discussed. The statement +2 produced a result of 2 because the APL system treated it as 0+2. The zero, in this case is called an identity element. Similar results occured for subtraction and division because they too have associated identity elements that are assumed each time these functions are used Monadically. To find the identity element for the Scalar functions that have them, the Reduction operator and an empty vector right argument are used.

Here is how to find the identity element for the Plus function:

```
      +/⍳0
0
```

The division's identity element is found in a similar manner.

```
      ÷/⍳0
1
```

Below is a list of the Scalar functions that have identity elements and what these identity elements are:

Function	Identity Element
+	0
-	0
×	1
÷	1
|	0
!	1
⌈	¯7.237005577E75
⌊	7.237005577E75
∨	0
∧	1
<	0
≤	1
=	1
≥	1
>	0
≠	0

Note: ¯7.237005577E75 and 7.237005577E75 are the smallest and the largest numbers possible in the APL system.

Bibliography

Berry, P.C., APL\360 Primer, White Plains, N.Y., IBM Corp., Form No. GH20-0689, 1969

Falkoff, A.D., and K.E. Iverson, APL\360 User's Manual, Yorktown Heights, N.Y., IBM Corp., Form No. GH20-0683, 1968

Gilman, L., and A.J. Rose, APL\360 An Interactive Approach, New York, John Wiley & Sons, Inc., 1970

Hanson, J.C., and W.F. Manry, APL: an intro, Atlanta, Georgia, Atlanta Public Schools, 1971

Katzan, H., APL User's Guide, New York, Van Nostrand Reinhold Co., 1971

Pakin, S., APL\360 Reference Manual, Chicago, Science Research Assoc., Inc., 1972

INDEX