commodore **SuperPET** computer

Waterloo microAPL

commodore COMPUTER

WATERLOO MICROAPL

Tutorial and Reference Manual

J. C. Wilson

T. A. Wilkinson

First Edition
Second Printing

Disclaimer

Waterloo Computing Systems Limited makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Waterloo Computing Systems Limited, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various users. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

PREFACE

APL is a powerful and concise notation which can be used to communicate algorithms between people or between a person and a computer. The name *APL* is an acronym for "A Programming Language," which was the title of a book published in 1962 by the inventor of *APL,* Kenneth E. Iverson. Since the late 1960's the notation itself has remained relatively unchanged, although features have been added to facilitate its use with the computer.

Waterloo MicroAPL for the SuperPET follows closely the IBM internal standard for *APL* written by A. D. Falkoff and D. L. Orth and published in 1979 by the Association for Computing Machinery. All of the standard language primitives are included. System features are those consistent with a single user environment. Extensions include system functions supporting file access, the reading and modification of memory, and the execution of machine language subroutines.

This manual is presented in two parts. The first part is a tutorial intended to introduce the new user to the language and system features. The second part is a comprehensive reference manual. Much of the reference manual should be read or skimmed by the new user, although Chapter 6, which contains the detailed definitions of all the primitive functions and operators, should be deferred until needed.

Acknowledgment

Many people have made significant contributions to the design of Waterloo MicroAPL and so it is difficult to acknowledge everyone individually. The design is based upon ideas evolved and proven over the past decade in other software projects in which these and other people have been involved. The major portion of the implementation was performed by Geno Coschi, Rick Gallant, Eric Mackie, Steve McDowell and Terry Stepien. Kay Harrison and Paul Dirksen were very helpful in the production of this document.

J. C. Wilson
T. A. Wilkinson,

July, 1981.

Table of Contents

TUTORIAL SECTION

Table of Contents

REFERENCE SECTION

Table of Contents

WATERLOO MICROAPL

Tutorial Manual

J. C. Wilson

T. A. Wilkinson

Introduction

This Tutorial is intended to provide an introduction to the basic concepts and facilities of the *APL* computer language as implemented on the Commodore SuperPET. It is composed of a number of short topics with accompanying notes which illustrate each point.

Getting Started

Before turning the machine on, make sure the switches are set at 6809 and R/W. Then turn on the power switch on the SuperPET and disk (and printer if attached). The following menu should appear.

```
Waterloo microSystems

Select:

  setup
  monitor
  apl
  basic
  edit
  fortran
  pascal
  development
```

Insert the system diskette in drive 1 and a data diskette in drive 0. Then select *APL* by typing apl and pressing return.

There will be a pause of about 1 minute while the *APL* language translator is loaded into the machine. Then a message similar to the following will appear:

```
WATERLOO MICROAPL
COPYRIGHT 1981 BY WATERLOO COMPUTING SYSTEMS
  LIMITED
CLEAR WS
```

Now the *APL* system is ready for use.

Tutorial 1

Simple Arithmetic Functions

The *APL* system manipulates numbers in the usual manner using the functions of addition (+), subtraction (−), multiplication (×), division (÷) and exponentiation (*). Type each of the following simple expressions on the keyboard hitting the RETURN key after each one.

(a)
```
      5+6
11
```

(b)
```
      3−1
 2
```

(c)
```
      18×3
54
```

(d)
```
      3÷4
0.75
```

(e)
```
      2*3
8
```

(f)
```
      3−4
¯1
```

(g)
```
      2*.5
1.41421356
```

NOTES:

1 The symbols + − × ÷ and * are called functions. Since each has 2 arguments (one on the left and the other on the right) they are called dyadic functions.

2 The result of example (f) is a negative number. The symbol for negative (¯) should not be confused with the subtraction symbol (−).

3 Numbers in *APL* are displayed with 9 digits of accuracy as in example (g).

Consider these examples:

```
(h)      (8−4)+1
      5

(i)      8−(4+1)
      3

(j)      8−4+1
      3
```

NOTES:

1 These expressions are more complex, each containing a number of functions and arguments.

2 Example (h) uses a pair of parentheses to force the subtraction (8−4) function to be evaluated before the addition.

3 Example (i) uses parentheses to force the addition to be evaluated first.

4 As shown in (j), expressions are evaluated from right to left if no parentheses exist to indicate otherwise.

5 The numbers used in these examples are referred to in *APL* as scalars.

Tutorial 2

Storing Numbers

Numbers can be remembered by the *APL* system through the use of numeric variables which are defined by the programmer.

(a)
```
      VISA←79.45
```

The assignment function (←) is used to save the number 79.45 in the variable *VISA*. It may represent an amount owed to a credit company.

The names given to variables can be composed of up to about 80 characters. The first must be a letter of the alphabet, while the remainder can be any letter or number, or the underscore symbol (_).

(b)
```
      VISA
79.45
```

The contents of a variable can be displayed by typing its name.

(c)
```
      VISA-50
29.45

      MASTER←301.15
      VISA+MASTER
380.6
```

Variables such as *VISA* and *MASTER* can be used in expressions.

(d)
```
      VISA⌈MASTER
301.15
```

The greater of two numbers can be computed with the dyadic function maximum (⌈).

(e)
```
      VISA⌊MASTER
79.45
```

The lesser of the two numbers can be computed with the dyadic mimimum (⌊).

Tutorial 3

Lists of Numbers

Many applications require manipulation of lists of numbers. *APL* has the ability to store such lists of numbers in a single variable.

(a)
```
      CARDS←79.45 301.45 65 300.2
```

The above list of 4 numbers is stored (using ←) in the variable *CARDS*. Such lists are called vectors. This one could represent the various amounts owed to 4 credit companies.

(b)
```
      CARDS
79.45 301.45 300.2
```

The contents of a vector are displayed by typing its name.

(c)
```
      ρCARDS
4
```

The number of elements in or length of a vector can be computed using the shape function (ρ).

This is an example of a monadic function since it has only one argument.

(d)
```
      CARDS-20
59.45  281.45  45  280.2

      CARDS÷2
39.725  150.725  32.5  150.1
```

The arithmetic functions (+ − × ÷ *) can be used to perform calculations on all elements of a vector.

(e)
```
      +/CARDS
746.1
```

The sum of the elements in a vector can be computed using the plus reduction function (+/)

```
      +/CARDS×.18
134.298
```

Since expressions are evaluated right to left, this multiplies each element in *CARDS* by .18 and then totals the elements of the resulting vector.

```
      (+/CARDS)÷⍴CARDS
186.525
```

This computes the average of the elements in the vector *CARDS*.

(f)
```
      ⌈/CARDS
301.45
```

The function ⌈/ is used to compute the maximum element in a vector.

```
      ⌊/CARDS
65
```

Similarly ⌊/ computes the minimum element in a vector.

(g)
```
      ⍳8
1  2  3  4  5  6  7  8
```

The monadic index generator function (⍳) generates a vector of the integers from 1 to the value of the specified argument.

(h)
```
      ?10
6

      ?10
3
```

The monadic roll function (?) generates a random integer between 1 and the specified argument (in this case 10).

```
      3?10
4 8 2

      3?10
2 7 6
```

The dyadic deal function (?) generates a vector of random numbers without any duplicates. The left argument specifies how many numbers are to be generated and the right argument defines the range.

It is often desirable to perform operations involving more than one list of numbers. In these applications functions are performed on pairs of vectors.

(i)
```
      RATE←.18 .21 .14 .17

      CARDS×RATE
14.301 63.3045 9.1 51.034
```

The first line creates a vector called *RATE*. It might represent the interest rates charged by 4 credit companies. The second line multiplies each element of *RATE* by the corresponding element in *CARDS* and displays the result. Note that both *RATE* and *CARDS* must have the same number of elements (i.e., they must have the same length).

At this point we have created two vectors named *CARDS* and *RATE*. We may wish to leave the system for a time and return later to continue. To preserve the data we have created, it is necessary to save the contents of our workspace.

(j)
```
      )SAVE THISWS
SAVED 81/01/07  00:49:44 THISWS
```

This is a system command that causes the contents of the workspace to be recorded on a diskette in drive 0.

(k) *)LOAD THISWS*
SAVED 81/01/07 00:49:44

This system command is used to restore the workspace to its state at the time when the *)SAVE* was done.

(l) *)LIB*

This system command will display a list of all the workspaces which have been saved.

(m) *)DROP THISWS*

Workspaces can be removed from the library by use of the *)DROP* command.

(n) *)VARS*

This system command will display a list of all the variables defined in this workspace.

Tutorial 4

Manipulating Character Data

As well as performing arithmetic calculations, most computer applications must manipulate character data such as names, addresses, etc. *APL* treats such items of data as arrays of characters.

(a)
```
      NAME←'SMITH'
```

The character data enclosed between the quote symbols is assigned to the character variable *NAME*.

NAME is a vector of 5 characters.

```
      ⍴NAME
5
```

The shape function (⍴) computes the length of a character vector.

(b)
```
      FIRSTNAME←'JACK'
      FULLNAME←FIRSTNAME,NAME
      FULLNAME
JACKSMITH

      FULLNAME←FIRSTNAME,'  ',NAME
      FULLNAME
JACK SMITH
```

The dyadic catenation function (,) is used to create longer vectors from two shorter ones.

(c)
```
      NAME←'SMITH'
      1↑NAME
S
      3↑NAME
SMI
      ¯3↑NAME
ITH
```

The dyadic function take (↑) can be used to select a number of elements from the beginning or end of a vector.

```
      (6↑FIRSTNAME),NAME
JACK  SMITH
```

The take function can be used to pad blank characters onto a character vector.

(d)
```
      NAME[4]
T
```

An index or subscript enclosed in square brackets and following the name of a vector indicates that the element in the specified position is to be selected from the vector. In this case the 4th character of 'SMITH' is selected.

```
      NAME[1 3 5]
SIH
```

If a list or vector of subscripts is specified, the corresponding elements are all selected to form a new vector.

```
      NAME[1+⍳3]
MIT
```

Using vector arithmetic and the index generator function, a substring can be extracted from a vector.

Tutorial 5

Data Comparisons

A very important facility of any computer is the ability to determine the relationships between various elements of data, both numeric and character. *APL* uses the functions equals (=), not equal (≠), greater than (>), greater than or equal (≥), less than (<), less than or equal (≤), and (∧), or (∨) and not (~) to perform these comparisons.

Comparison of items in *APL* is accomplished by the evaluation of a function which is found to be either True (represented by 1) or False (represented by 0).

(a)
```
      VISA←79.45
      VISA=0
0
```

When the expression *VISA*=0 is evaluated it is found to be false since *VISA* is 79.45. Therefore the expression is given the value 0.

```
      VISA>0
1
      (VISA>0)∧(VISA<100)
1
```

These conditions are true and each receives the value 1 indicating truth.

(b)
```
      CARDS←79.45  301.45  65  300.2
      CARDS>100
0 1 0 1
```

This condition requires that each element of *CARDS* be compared to 100. The first and third elements are not greater than 100 and so fail the test. They generate 0 or false. The second and fourth elements are greater than 100 and generate 1 or true. Thus a vector is created which indicates the result of each of the comparisons

(c)
```
      +/(CARDS>100)
2
```

This expression computes how many elements of CARDS are greater than 100 by creating a vector of 1's and 0's and summing.

It is possible to use the vector of 0's and 1's produced in this way to select certain elements from a vector creating a new vector.

(d)
```
      1 0 1 0/CARDS
79.45  65
```

The dyadic function compression (/) is used to select specific elements from the vector CARDS. Compression selects elements from the right-hand vector which have a corresponding 1 in the left-hand vector.

```
      (CARDS>100)/CARDS
301.45  300.2
```

The selection vector on the left of a compression function is often generated by the evaluation of a condition. Here we select all elements of *CARDS* which are greater than 100.

```
      (CARDS>100)/⍳⍴CARDS
2 4
```

Here we determine which elements of *CARDS* are greater than 100 (they are the second and fourth).

Tutorial 6

Tables of Data

Frequently, it is desirable to group items of data together in the form of a table or matrix. The elements of a matrix are arranged in rows and columns.

(a)

```
      TABLE←2 3⍴1 2 3 4 5 6
      TABLE
1 2 3
4 5 6
```

The first line uses the dyadic function shape (⍴) to create a matrix with 2 rows and 3 columns. The elements of the matrix are filled in from the vector on the right side. Note the order in which the elements are inserted into the matrix. If there are not sufficient elements in the right hand argument to fill the matrix, then the elements are re-used as many times as necessary.

```
      ⍴TABLE
2 3
```

The monadic function reshape (⍴) can be used to compute the number of rows and columns in a matrix. This is called the shape of the matrix. Since this is a 2-dimensional matrix, it is said to have a rank of 2.

Consider a consumer who has 3 credit cards which he uses for 2 different categories of purchases (say business and personal).

(b)
```
      CHARGES←2 3⍴14.3 7.9 100.1 17.26 24 40

      CHARGES
14.3    7.9  100.1
17.26   24    40
```

The first line creates a matrix with 2 rows and 3 columns and assigns 6 values from the 6-element vector on the right.

The second line displays the elements of the matrix.

(c)
```
      +/[1]CHARGES
31.56  31.9  140.1
```

The plus reduction function along dimension [1] (i.e., down the columns) causes a total to be produced for each column. This creates a row of totals giving a total for each credit company.

```
      +/[2]CHARGES
122.3  81.26
```

Similarly, plus reduction along dimension [2] (i.e., along the rows) causes a column of totals to be produced, one for each row.

```
      +/+[2]CHARGES
203.56
```

This computes the sum of the rows and then sums the resulting vector to give the total of all the numbers in the table.

(d)
```
      PAYMENTS←2 3⍴20 10 20 10 20 30

      PAYMENTS
20 10 20
10 20 30

      CHARGES—PAYMENTS
¯5.7  ¯2.1 80.1
 7.26   4   10
```

This computes the balance owing in each account.

All the usual arithmetic functions (+ − × ÷ *) apply to matrices.

```
      +/[1]CHARGES—PAYMENTS
1.56 1.9 90.1
```

This computes how much we still owe each credit company.

Character data can be arranged in matrices or tables as well. The elements of a character matrix are single characters.

(e)
```
      COMPANY←3 6⍴'VISA  MASTERAMEX
      COMPANY
VISA
MASTER
AMEX
```

The first line creates a matrix of characters called *COMPANY*. Each of the 3 rows in the matrix is composed of 6 characters.

In general, matrices or arrays can have as many dimensions as the application demands. They can be viewed as tables of tables.

Tutorial 7

Indexing

Previous tutorials have discussed vectors and matrices of numbers and characters. It is sometimes desirable to perform operations on selected elements of these tables. A technique called indexing or subscripting can be used for this purpose.

(a)
```
      DATA←14.2 10 3 41.1 62
      DATA
14.2 10 3 41.1 62
```

Here we have created a vector of 5 numbers.

```
      DATA[2]
10

      DATA[1 3 4]
14.2 3 41.1
```

These lines display selected elements from the vector *DATA*. The numbers in the square brackets are called subscripts. A subscript can be a numeric scalar or vector.

Similar operations are possible with character vectors (see Tutorial 4).

(b)
```
      DATA[3]←1000
      DATA
14.2  10  1000  41.1  62
```

Any element of a vector can be replaced by using this combination of indexing and assignment.

The various elements of a matrix can also be accessed individually with indexing.

(c)
```
      CHARGES←2 3⍴14.3 7.9 100.1 17.26 24 40
      CHARGES
14.3   7.9  100.1
17.26  24    40

      CHARGES[2;3]
40

      CHARGES[2;3]←0
      CHARGES
14.3   7.9  100.1
17.26  24     0
```

Individual elements of a matrix (created as above) must be referenced by two indices or subscripts, specifying the row and column of the element respectively.

In the case of matrices, complete rows or columns can be referenced.

(d)
```
      CHARGES[1;]
14.3  7.9  100.1

      CHARGES[;3]
100.1  0

      CHARGES[;2]←0

      CHARGES
14.3   0  100.1
17.26  0    0
```

Complete rows or columns of a matrix can be extracted or replaced using subscripts.

(e)
```
      COMPANY←3 6⍴'VISA  MASTERAMEX'
      COMPANY
VISA
MASTER
AMEX

      COMPANY[3;]
AMEX

      COMPANY[;1]
VMA

      COMPANY[2;]←'MCHG'
      COMPANY
VISA
MCHG
AMEX
```

Row and column extraction and replacement can also be done on character arrays.

Tutorial 8

Combining Sets of Data

In Tutorial 4, the catenation function was introduced as it pertained to character vectors. However, it has a more general application to vectors of all types.

(a)
```
      DATA←10 13 47
      MORE←6 5 7

      LOTS←DATA,MORE
      LOTS
10 13 47 6 5 7
```

The catenation function creates a single vector from two other vectors.

(b)
```
      CHARGES←2 3⍴14.3 7.9 100.1 17.26 24 40
      CHARGES
14.3   7.9 100.1
17.26  24   40
```

```
      NEW←CHARGES,[1]  16  18  21
      NEW
14.3   7.9  100.1
17.26  24    40
16     18    21
```

These lines show how the catenation function is used to add a new row to the matrix ([1] means in the first dimension).

```
      NEW←CHARGES,[2]  16.1  14.7
      NEW
14.3   7.9  100.1  16.1
17.26  24    40    14.7
```

A new column can be added in a similar manner ([2] means the second dimension).

(c)

```
      CATEGORY←2 8⍴'BUSINESSPERSONAL'
      CATEGORY
BUSINESS
PERSONAL

      CATEGORY,⍕CHARGES
BUSINESS  14.3   7.9   100.1
PERSONAL  17.26  24    40
```

The thorn symbol (⍕) specifies the monadic function format and is formed by overstriking the symbols T and °. It converts the numeric data in the matrix *CHARGES* to character data so catenation can be performed with the matrix *CATEGORY* (row by row). The result is a new character matrix.

```
      CATEGORY,7 2⍕CHARGES
BUSINESS  14.30  7.90  100.10
PERSONAL  17.26  24.00  40.00
```

The dyadic function format converts numeric data to character and formats it. In this case, each number is converted to 7 characters with 2 digits after the decimal point.

Tutorial 9

Storing Instructions as Functions

So far we have created various forms of numeric and character data in the workspace. Each time functions were to be performed on that data, the correct *APL* statements had to be entered. It is often desirable to create a list of such statements or instructions in the workspace. These can then be invoked as a new function, thus avoiding re-entering all the lines again.

(a)
```
      ∇SUM
[1]
```

This line opens the definition of a function (or procedure or program) called *SUM*. We simply enter the statements we want to put in the function in response to the line-number prompt by the system editor.

```
      ∇SUM
[1]  'ENTER A LIST OF NUMBERS'
[2]  X←□                          ⍝ GET NO'S FROM KB
[3]  'SUM = ',⍕+/X                ⍝ DISPLAY SUM OF NO'S
[4]  'AVG = ',⍕(+/X)÷⍴X           ⍝ DISPLAY AVG OF NO'S
[5]  ∇
```

These lines define the function *SUM*.

Line [2] contains the symbol quad (□). Later, when this list of instructions is being executed, it will allow vectors of numbers to be entered from the keyboard.

Some lines contain the comment symbol (⍝) (∩ overstruck with °). Text following this symbol provides documentation only.

The del symbol (∇) is also used to close the function in line [5].

(b)

```
      SUM
ENTER A LIST OF NUMBERS
□:

      1 7 18 4 ¯2
SUM = 28
AVG = 5.6
```

A function can be executed by simply typing its name.

(c)

```
      ∇SUM[□]∇
[0]  SUM
[1]  'ENTER A LIST OF NUMBERS'
[2]  X←□                          ⍝ GET NO'S FROM KB
[3]  'SUM  =  ',⍕+/X              ⍝ DISPLAY SUM OF NO'S
[4]  'AVG  =  ',⍕(+/X)÷           ⍝ DISPLAY AVG OF NO'S
     ⍴X
```

Example of listing a function.

In order to modify the statements of a function definition, it is necessary to open the function. Then changes can be made and the function closed.

(d) ∇*SUM*[□]

This lists the function and leaves it open (i.e., the prompt for a new line [5] is displayed). Lines can now be added, inserted, modified or deleted. The cursor movement and the INST and DEL keys can be used to change existing lines. When all desired changes are made, the ∇ symbol is used to close the function.

The following are some examples of function editing:

```
[4] 'AVERAGE=',⍕(+/X)÷ρX      ⍝ DISPLAY AVG OF NO'S
[5] ∇
```

Example of replacing a line in a function.

```
[2.1] 'THERE ARE',(⍕ρX),' ELEMENTS'
[2.2] ∇
```

Example of inserting a line in a function.

```
[Δ3]
[4] ∇
```

Example of deleting a line from a function.

(e))*FNS*

This system command displays a list of all the functions which are defined in the current workspace.

(f)

```
∇FA
[1] 'ENTER A WORD'
[2] WORD←⍞
[3] 'THE WORD HAS',(⍕ρWORD),' CHARACTERS'
[4] ∇
```

This function illustrates how the input operation is used for character vectors. The symbol used for character input is quote quad (⍞) and is formed by overstriking the symbols ' and □.

(g)
```
     ∇FB WORD
[1]  N←+/'A'=WORD
[2]  'THE LETTER A OCCURS',(⍕N),' TIMES'
[3]  ∇
```

In this example, the word to be examined is passed as a parameter to the function *FB* rather than being entered as input (as in (f)). It would be used as follows:

```
     FB 'ACTUAL'
THE LETTER A OCCURS 2 TIMES
```

(h)
```
     )ERASE FB
```

The system command *)ERASE* is used to erase a function or a variable from the workspace.

Tutorial 10

Controlling the Sequence of Execution

Functions are frequently very complex combinations of *APL* statements. It is usually necessary to control the order of execution in these functions, repeating some statements a number of times (loop structures) and selectively executing others (if structures). This logical complexity is achieved in *APL* through the use of branching statements.

(a)

```
      ∇CALC
[1]   'ENTER SOME NUMBERS'
[2]   DATA←□
[3]   TOTAL←+/DATA
[4]   IF:→(TOTAL≤100)/ENDIF
[5]      'TOTAL GREATER THAN 100'
[6]   ENDIF:
[7]   'TOTAL=',⍕TOTAL
[8]   ∇
```

The function *CALC* computes and displays the total of a list of numbers. If that total is greater than 100, it also prints a message to that effect.

Line [4] causes a branch to the line labelled *ENDIF* ([6]) when the condition $TOTAL \leq 100$ is found to be true. The symbol → indicates a possible branch.

The *IF:* in line [4] and the *ENDIF:* in line [6] are called labels and must be unique within the function. They are followed by the colon (:). Rules for forming label names are the same as those for forming variable names (see Tutorial 2).

It is considered good practice to indent statements (such as line [5]) which are conditionally executed.

(b)

```
      ∇COMP
[1]   'ENTER SOME NUMBERS'
[2]   DATA←□
[3]   TOTAL←+/DATA
[4]   IF:→(TOTAL≤100)/ELSE
[5]     'TOTAL GREATER THAN 100'
[6]     →ENDIF
[7]   ELSE:
[8]     'TOTAL NOT GREATER THAN 100'
[9]   ENDIF:
[10]  'TOTAL=',⍕TOTAL
[11]  ∇
```

In this example, lines [5] and [6] are executed if $TOTAL > 100$ and lines [7] and [8] are executed if $TOTAL \leq 100$.

The →*ENDIF* in line [6] causes an unconditional branch to line [9].

It is important to note that, while the *APL* language does not have the structured language constructs, good program structure can be achieved and revealed using controlled branching, well chosen labels and proper indenting.

(c)

```
      ∇ADDER
[1]  RPT:
[2]     'ENTER SOME NUMBERS'
[3]     DATA←□
[4]     TOTAL←+/DATA
[5]     →(TOTAL<0)/END
[6]     'SUM OF', (⍕DATA), 'IS', (⍕TOTAL)
[7]     →RPT
[8]  END:
[9]  ∇
```

The function *ADDER* repeatedly asks for a list of numbers for which it displays a total. This is done with a loop composed of lines [1] through [7].

Line [5] causes the loop to terminate when the calculated total is less than zero.

(d)

```
      ∇TRANSLATE
[1]  'ENTER A NUMBER'
[2]  I←□
[3]  CASE:→(I=1  2  3)/CS1,CS2,CS3
[4]     'NUMBER NOT IDENTIFIED'
[5]     →END
[6]  CS1:
[7]     'NUMBER IS ONE'
[8]     →END
[9]  CS2:
[10]    'NUMBER IS TWO'
[11]    →END
[12] CS3:
[13]    'NUMBER IS THREE'
[14] END:
[15] ∇
```

Line [3] in this function performs a case test and selectively executes a group of statements based on the value of I. If $I=1$ control transfers to label *CS1*; if $I=2$ control transfers to label *CS2*; etc. If the number entered is not 1 or 2 or 3, control passes to line [4].

Tutorial 11

External Storage of Data as Files

It is often desirable to transfer data between an *APL* workspace and external storage areas known as files. The simplest form of such a file can be viewed as a list of items of data. This is called a sequential file and is stored in the *APL* library and given a name.

(a)
```
'TEST' □CREATE 6
```

A file named *TEST* is created and given the tie-number 6.

```
CHARGES←2 3⍴14.3 7.9 100.1 17.26 24 40
CHARGES □WRITE 6
```

The shape, rank, type and all the data of the variable *CHARGES* are transferred to the file tied by number 6.

```
COMPANY←3 6⍴'VISA  MASTERAMEX
COMPANY □WRITE 6
```

As before, all characteristics of *COMPANY* are written to the file.

```
      □UNTIE 6
```

The file tied by number 6 is released.

(b)
```
      'TEST' □TIE 4
```

The file named *TEST* is tied to the workspace with the number 4.

```
      X←□READ 4
      X
14.3    7.9  100.1
17.26  24    1 40
```

The variable *X* receives the shape, rank, type and all the data from the value stored in the file.

```
      X←□READ 4
      X
VISA
MASTER
AMEX
```

Here the variable *X* receives all the characteristics of the next value stored in the file.

```
      □UNTIE 4
```

The file tied by number 4 is released.

(c)
```
      'TEST' □TIE 4
      'TEST' □ERASE 4
```

Files can be removed from the library with the function □*ERASE*. The file being erased must be currently tied to the workspace.

(d)
```
      ∇CRTFILE A
[1]  ST←A ⎕CREATE 3
[2]  'ENTER LINES (EOF TO STOP)'
[3]  RPT:
[4]     X←⍞
[5]     →(∧/'EOF'=3↑X)/END
[6]     ST←X ⎕WRITE 3
[7]     →RPT
[8]  END:
[9]  ⎕UNTIE 3
[10]∇
```

The above function creates a sequential file of character vectors entered from the keyboard. Line [1] creates the file using ⎕*CREATE* and gives it the name specified by parameter *A*. It also states that this file will be referred to as file number 3.

Items from *X* are written to the file in line [6] using ⎕*WRITE* with *ST* receiving a string indicating the success or failure of the operation.

When the entire file has been written, line [9] releases the file with ⎕*UNTIE* and the file number becomes available for other uses.

(e)
```
      ∇LISTFILE A
[1]  ST←A ⎕TIE 3
[2]  →(0≠⍴ST)/0
[3]  RPT:
[4]     X←⎕READ 3
[5]     →(0≠⍴⎕STATUS 3)/END
[6]     ⎕←X
[7]     →RPT
[8]  END:
[9]  ⎕UNTIE 3
[10]∇
```

This function retrieves the list of items from the file whose name is in *A* and displays them on the screen. Line [1] attaches the specified file using ⎕*TIE* and states that it will be referred to as file number 3.

Line [2] causes a transfer to line [0] (i.e., exit from the function) if an error occurs while attaching the file.

Line [4] uses □*READ* to retrieve one item from the file and assign it to *X*. After this operation, the function □*STATUS* (in line [5]) is used to check the result of the □*READ*. When all the items have been retrieved, □*UNTIE* is used in line [9] to release the file.

WATERLOO MICROAPL

Reference Manual

J. C. Wilson

T. A. Wilkinson

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time to time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various users. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 1

Keyboard and Screen

Keyboard

Although the *APL* keyboard is similar in many ways to the standard keyboard, there are some major differences.

The capital letters A-Z are in their usual positions, but you do not press SHIFT to get them. There are no lower case letters in the *APL* character set.

Many of the remaining characters are peculiar to *APL* and will be explained later.

"Overstruck" Characters

Besides the symbols shown on the keyboard diagram there are 18 other symbols that can be created by typing two symbols on top of each other (e.g., by using the "cursor back" key). The 18 overstruck symbols and the two symbols which produce each of them are shown in the figure below.

Overstruck Symbol	Combine	
⍱	∨	~
⍲	∧	~
⍫	∇	|
⍒	∆	|
⌽	○	|
⊖	○	—
⍉	○	\
⍟	○	*
⍢	∇	~
⍦	⊥	∘
⍪	⊤	∘
⍀	\	—
⌿	/	—
⍵	∘	∩
⍞	□	'
⍰	□	÷
⍬	'	.
⌶	⊤	⊥

Unused Symbols

Some of the symbols in the *APL* character set have no use in MicroAPL except as convenient graphic symbols. These are

diamond	$	⊂
left brace	⊃	∩
right brace	∪	⍺
left tack	⍵	¨
right tack	⌶	⍱

Control Keys

Several of the keys control screen or cursor functions:

<= CRSR	Moves cursor left ("cursor back")
=> CRSR	Moves cursor right
 CRSR	Moves cursor up
 CRSR	Moves cursor down
TAB	Moves cursor to next tab position
CLR	Clears the screen and "homes" the cursor
HOME	Moves cursor to top left hand corner
RUBOUT	"destructive backspace"
EEOL	Erase to end of line
INST	Opens space under the cursor
DEL	Deletes characters and closes text. (Behavior is slightly different depending on whether there are nonblanks to the right of the cursor or not.)

Full Screen Editing and the RETURN Key

Whenever the *APL* system expects you to enter something from the keyboard, you may use the control keys to manipulate the screen in any way you wish, but nothing will be sent to *APL* until you press the RETURN key. This is the only rule you need to remember: when you press RETURN, the entire contents of the line the cursor was on (when you pressed RETURN) gets sent to *APL*. It doesn't matter what may be elsewhere on the screen. The rule applies equally whether you are in immediate mode or function definition mode.

Chapter 2

The Workspace and System Commands

The active workspace is the environment within which you deal with *APL*. The principal contents of a workspace are variables and defined functions. When you begin an *APL* session the active workspace is empty; the message *CLEAR WS* confirms this.

There are a number of system commands which permit you to manipulate workspaces. System commands are distinguished by the fact that they all start with a right parenthesis.

)LIB
)LIB libid

The response is a listing of the directory for the diskette in drive 0 of the disk unit, or else for the device designated by libid (see the System Overview Manual). For example, if you have a disk unit whose address is 9, not the usual 8, then *)LIB DISK9/1* will give you the directory for the diskette in drive 1 of that unit. A lengthy listing can be interrupted by means of the STOP key.

)CLEAR

The active workspace is replaced by one without any defined functions or variables and with the workspace parameters (*□IO, □CT, □PP, □PW, □RL, □LX*) set to their default values. The name of the workspace is set to null and is reported as *CLEAR WS*. The previously active workspace is gone.

)WSID

The response is the name of the active workspace.

)WSID wsid

The name of the active workspace is changed to wsid. The response is its previous name. The parameter wsid is often just a name, like *CHESS,* but it can designate a device as well (e.g., *DISK/1.CHESS*). See the System Overview Manual.

)SAVE

The active workspace is saved under its current name (unless the name is null).

)SAVE wsid

The active workspace is saved under the name wsid unless wsid doesn't match the current name and there already exists a workspace with the name wsid. (This is to prevent you from inadvertently overwriting one workspace with another.) After the SAVE, the active workspace will have the name wsid, whether it did before or not.

The response to a SAVE is a timestamp derived from the current setting of the system clock (see □*TS*) together with the name under which the workspace was saved. The timestamp is saved with the workspace and is reported on subsequent loading.

)LOAD wsid

The named workspace replaces the active workspace. The active workspace is gone. The response is the timestamp saved with the workspace.

)COPY wsid
)COPY wsid names

The named objects (functions and variables) are copied from the named workspace into the active workspace, replacing any objects therein having the same names. The parameter names is a string of names separated by blanks. If it is omitted, all the variables and functions (except objects beginning with □) are copied.

)DROP wsid

The designated workspace is deleted from the library.

)*FNS*

The response is a list of the names of the defined functions in the active workspace.

)*VARS*

The response is a list of the names of the variables currently defined in the active workspace. (Includes local variables if there are suspended functions.)

)*ERASE* names

Names is a string containing the names of variables and functions separated by blanks. The named objects are deleted from the active workspace. Response: normally none, but objects which could not be erased are reported.

)*SI*

The response is the current state indicator (see Defined function execution).

)*SINL*

The response is as for)*SI* but with the local variables shown against each active function.

)*OFF*

APL is discontinued and control is returned to the microlanguage menu.

)*SYMBOLS*

The response is the current maximum number of symbol table entries.

)*SYMBOLS* number

This has the same effect as)*CLEAR* except that the maximum number of symbol table entries is set to number.

)*WSLIMIT*

The response is the first memory address beyond the current end of the workspace.

)*WSLIMIT* number

The end of the workspace is changed to (number-1).

NOTE:

1 Program function key 3 (i.e., shifted "3" on the numeric keypad) is equivalent to typing the three system commands)*FNS*,)*VARS* and)*SI*.

Chapter 3

Expressions

An expression is a string of *APL* variables, functions, operators, numeric and character constants, parentheses and bracketed index expressions.

As a general rule, an *APL* expression is evaluated from right to left, in the absence of parentheses. Although at first this sounds peculiar it is in fact what we are used to when we use the English language. For example, the sentence "The equivalent resistance is the reciprocal of the sum of the reciprocals of the given resistances." makes sense (to an electrical engineer!) when read from left to right, but as a prescription for computation it must be used from right to left, starting with "the given resistances" and ending with the specification of "the equivalent resistance."

The equivalent *APL* expression is $M \leftarrow \div + / \div R$. It, too, can be read from left to right, but is executed from right to left.

Parentheses modify the order of execution in the usual way.

There are no priority rules such as the common convention that "multiplication and division are done before addition and subtraction." For example $3 \times 4 + 5$ is 27, not 17. This makes life much simpler in an environment such as *APL,* in which there are dozens of functions like $\times, \div, +$ and $-$.

Subexpressions containing operators, like $+.\times$ are exceptions to the above.

Chapter 4

Arrays

Data in *APL* is not in general single quantities, but rectangular arrays of quantities. A table, or matrix, like

```
2  3  ¯4
5  0   6.7
```

is a rectangular 2 by 3 array of numbers. We say that its type is numeric, its rank is 2 and its shape is 2 3.

In general, an *APL* array has, besides its elements, a type (numeric or character), a rank (the number of axes, or "coordinates," or "dimensions"), and a shape (a vector giving the length of each axis).

A simpler array than a matrix is a list, or vector, like

```
¯7 2 3 5
```

whose rank is 1 and whose shape is 4.

There is an even simpler array than a vector. This is a single quantity, or scalar, like 3.14 whose rank is 0 and whose shape is empty.

Empty arrays

It is possible in *APL* to create an empty array, that is one having no elements at all. The rank and shape of an empty array are not restricted except that the shape vector contains at least one element which is zero.

In general, the number of elements in an array is the product of the elements in its shape vector.

Internal Representation: Numeric Data

The elements of a numeric array are stored internally in a 5 byte floating point format. Thus the 2 by 3 matrix used in the above example requires $6 \times 5 = 30$ bytes of memory, plus the memory necessary for the shape vector, the rank and the type, plus some further overhead.

MicroAPL does not take advantage of the compression which is possible when numeric arrays are known to be boolean or integer.

Internal Representation: Character Data

The elements of character arrays are stored one per byte, plus the same overhead as for numeric arrays. Thus the 3 by 5 matrix

ABCDE
FGHIJ
KLMNO

requires 15 bytes plus overhead.

All 256 possible bytes are legal as the elements of character arrays.

Numeric data: Input

Numeric data elements are entered through the keyboard using the digits 0-9 and the symbols ".", "¯", and "*E*".

The digits 0-9 and the decimal point are used in the normal way.

The negative sign ¯ (which is located above the "2" on the keyboard, not above the "+") and the "*E*" (which means "times ten to the power . . .") are symbols similar to the decimal point in that they are regarded as being part of the representation of the number and not functions or operators.

Examples:

¯2.35*E*2 is equivalent to ¯235
3.14*E*¯3 is equivalent to 0.00314

There must not be spaces within the representation of a number, and if *E* occurs, the number following it must be an integer.

A numeric vector may be entered by typing a sequence of numeric elements separated by spaces, all on a single line. Larger vectors, and arrays of higher rank, must be formed from smaller ones by applying *APL* functions to them.

Character Data: Input

Character data elements which correspond to *APL* symbols can be entered through the keyboard.

A character vector may be entered as a string of *APL* symbols without unintended spaces (space is an *APL* symbol), all on a single line.

When a character vector is included as a character constant in an expression it must be enclosed in quotes, and any quote symbol which the vector itself contains must be made into two consecutive quote symbols. Thus the contraction of *CANNOT* would be input by means of the string *'CAN' 'T'*. The length of the resulting vector is 5 and it contains one quote symbol.

Fewer than half of all the possible byte values are interpreted internally as *APL* symbols. An application requiring the manipulation of bytes in general will usually create the character arrays by indexing □*AV*, not by getting them through the keyboard.

Again, larger vectors, and arrays of higher rank, must be formed from smaller ones by applying *APL* functions to them.

Variables

Every variable has a name which is a string of letters, digits and the underscore character (_). The first character of a name must be a letter. Names should be limited to 80 characters.

A variable generally acquires a value (which is an array) by having one assigned or specified. Thus

```
TAX_RATE← 27.4
```

assigns the scalar value 27.4 to the variable called *TAX_RATE*.

There may be several assignments within a single *APL* expression. For example $A \leftarrow 1 + B \leftarrow 0$ is sometimes used to initialize A to 1 and B to 0. This tends to reduce readability and should usually be avoided.

Chapter 5

Defined Functions

The defined function in *APL* is similar to a "program" in other languages, and in fact we will often use the term "program" interchangeably with "defined function."

A workspace may contain a large number of defined functions and they need not bear any particular relation to each other.

A defined function has a multiline representation. The first line, or header, establishes the name and syntax of the function, the names used for the parameters and the local names. The subsequent lines, or body, are the statements to be executed when the function itself is executed.

The Header of a Defined Function.

-Function name

The names of defined functions are subject to the same rules as the names of variables.

--Syntax

The syntax of a function describes how the function name may appear in an expression.

A function may have 0, 1 or 2 explicit arguments. The arguments of a function are always data arrays. A function having no arguments is called niladic. A function having one argument is called monadic and the argument appears to the right of the function name. A function having two arguments is called dyadic and the arguments appear on either side of the function name.

In addition to being niladic, monadic or dyadic, a function either produces an explicit result or it doesn't. This gives six syntactical forms to choose from for any given function.

	No Explicit Result	Explicit Result
Niladic	*F*	*Z*←*F*
Monadic	*F R*	*Z*←*F R*
Dyadic	*L F R*	*Z*←*L F R*

-Parameters

The variables represented by *L*, *R* and *Z* in the above figure are called parameters. These are temporary names which exist for the purpose of function definition and execution only. They indicate the syntax of the function and they serve as names by which you can refer to the left argument, the right argument and the result in the body of the function definition. The function name and parameter names must all be distinct, but they may match other names in the workspace; that is, they have local significance only. If a result parameter is included in the syntax of a function then during execution a value must be specified for that variable or a VALUE ERROR will result when execution ends.

-Local names

It is often desirable to have other local names in defining a function. Names can be made local by listing them in the header after the syntax, each name preceded by a semicolon.

Thus the header *A*←P R ;*X*;*B* establishes a monadic function *P* with result *A*, right argument *R* and local names *X* and *B*. (Usually *X* and *B* will be the names of variables, but they may also be the names of functions to be established by □*FX*.)

The name of the function being defined cannot be localized. The names of the parameters and the labels in a function are implicitly local and should not appear in the local variable list.

The Body of a Defined Function

-Statements

An *APL* statement is made up of any combination of the following components, in the order given.

(1) A label; i.e., a name followed by a colon.
(2) A branch arrow (→).
(3) An expression.
(4) A comment; i.e., a lamp symbol (⍝) (formed by overstriking ∩ and ∘) followed by any string of characters.

Figure 5.1 Statement

Execution of a statement consists of evaluating its expression, if it has one, and taking action appropriate to the branch arrow, it there is one.

```
L1:
END: →0
→
M←÷+/÷R                    ⍝ EQUIV RESISTANCE
→5÷H
REPEAT: →(N>I←I+1)/REPEAT  ⍝ A DELAY
⍝HOLD FOR ARRIVAL
```

Figure 5.2 Examples of Statements

-Branches and Labels

The statements in the body of a function are numbered 1, 2, ... If a statement includes a label, the value of the label is the number of the statement. The label is a local name. Care should be taken not to assign the same label name to more than one statement in a given function.

The statements of a function are normally executed in the sequence in which they occur. This normal sequence can be modified by the branch statement, that is, one containing a branch arrow.

The next statement to be executed after a branch statement is determined as follows:

(a) If the branch statement has no expression then execution ceases, the current execution sequence is cleared off the execution stack and *APL* awaits your next request.

This statement is normally used manually to clear out the execution stack and get rid of local variables, but it can also be written into functions and used to abandon automatically the execution of a program when a fatal error is discovered.

(b) If the branch statement has no expression then the value of the expression must be either (1) an integer scalar, or (2) a vector whose first element is an integer, or (3) an empty vector.

In the first two cases the integer referred to is the number of the next statement to be executed. If the integer is not a valid statement number then execution of the function terminates and control returns to whatever caused execution to start. In particular, 0 is not a valid statement number and is often used to cause termination of the function.

In the third case, the empty vector, no branch occurs: the next statement to be executed is the one following the branch statement in normal sequence.

A very common form for a "conditional branch" is exemplified by

$$\rightarrow(I \leq N)/INVERT$$

which can be read as 'if $I \leq N$ then go to *INVERT*'. *INVERT* is assumed to be the label on some statement in the function.

A "case" construction can be obtained by

$$\rightarrow(I=1\ \ 2\ \ 3)/CASE1,CASE2,CASE3$$

or by

$$\rightarrow((\times N)=\overline{\ }1\ \ 0\ \ 1)/NEGATIVE,ZERO,POSITIVE$$

where as before the names to the right of the compression (/) symbol are assumed to be labels.

The branch statement is extremely powerful and must be used with restraint, or you will create programs which are very difficult to understand.

Defining a Function

A function can be defined or established in a workspace in one of three ways.

(1) It can be copied from a stored workspace using the *)COPY* system command.
(2) It can be established by means of the system function □*FX*.
(3) It can be established by the use of the "del" function editor.

The first two features are described elsewhere.

To use the "del" editor to define a function, enter function definition mode by typing the character del (∇) followed by the header of the function. The editor will prompt with a line number in brackets. Enter the statements of the body of the function one by one.

To leave function definition mode, end a line, other than one containing a comment, with a del (∇), or enter del in response to the line number prompt.

Editing a Function

To use the editor to revise the definition of an existing function, enter ∇ followed by the name of the function. You cannot redefine the header this way: the name only is acceptable.

The name of the function may optionally be followed immediately by an editing command. Once you are back in function definition mode, every line you enter must be an editing command.

Following is a list of the possible editing commands.

Editing command	Meaning
[□]	Display the existing definition.
[□n]	Display the function, starting at line n.
[n□]	Display line n only and leave the cursor on the line.
[n] text	Replace the contents of line n by text. (Note that no change will occur if text is blank. This is a safety measure.) The number n need not be an integer: if n falls between two existing lines, then the new line is inserted between those lines.
[Δ n]	Delete line(s) n (n may be a vector).

It can be seen that the bracketed line number prompt that the system displays is merely an editing command. It is not necessary to delete it or move to a new line to enter a different command: the system ignores all but the last recognizable editing command on a line.

During the editing process for a function the lines of the body are not renumbered 1,2,... immediately after deletion or insertion. Renumbering occurs after you leave function definition mode.

For editing purposes only, the header is considered to be line 0, and it is displayed that way when [□] is executed. The entire header may be changed if desired, including the function name.

Editing Hints

Keep functions short. Functions longer than the screen (about 20 lines) are inconvenient to deal with.

When editing an existing function, display it immediately and then use the screen editing controls to adjust the definition of existing lines. Don't forget to strike RETURN when you are satisfied with a line; otherwise your changes will not be recorded.

Conversely if you have accidentally made garbage of a line of the function, don't hit RETURN, but use the cursor to get to a new line and ask for a redisplay. The garbage will not be recorded.

Program function key 1 (i.e., shifted "1" on the numeric keypad) during function editing will close the function definition, then immediately reopen and display it. The effect is to get a clean, renumbered listing of the function.

Errors During Function Editing

The editor detects a variety of errors, all of which are reported as *DEFN ERROR*. Here are some of the possible causes.

—On Opening a Function Definition

—The name of the intended function is already in use as a variable.

—Attempt to respecify the header of an existing function.

—Header is syntactically invalid.

—Invalid function name (e.g., "3D").

—You were already in function definition mode.

—Function is pendent. (Clear the state indicator.)

—Function name in locals list.

—Function name and parameter names are not all distinct.

—During Editing.

—Invalid editing command.

—Attempt to edit the header of a suspended or waiting function.

If you get a *SYNTAX ERROR* during editing you were probably not in the editor at all. You may have executed ∇*F*[□]∇ instead of ∇*F*[□], for example.

Effect of Localization

All names appearing in the header (except the function name itself), together with the names of all labels, are local names. Local names have only the significance assigned to them in their own function regardless of their significance in the calling environment, that is, before execution of the function began. The effect of this is that during the execution of the function (and even if the function becomes suspended) there is no way to "see" the original value associated with a localized name. The original value is restored when execution of the function terminates.

Any name which is not local has the same significance that it had in the calling environment. One of the effects of this is that a function cannot "hide" the value of its variables from any function which it calls.

Executing Defined Functions

In immediate execution mode, if you type a statement and press RETURN the statement is executed immediately. If, during the execution of the statement, the name of a defined function is encountered, then that function is executed. This in turn involves executing statements (those of the function) and any functions that they refer to, and so on. This process normally ceases only when the statement you originally typed is completely executed. If the statement has an explicit result and the last thing executed was not an assignment (←) the system displays the result on the screen. It then awaits your next command.

Suspension of Execution

The execution of a defined function will stop prematurely if an error is encountered, if the STOP key is pressed, or through stop control (see below). The system returns to immediate execution mode.

The function whose execution was interrupted is said to be suspended and all those functions which led to its execution and are not yet completed are referred to as pendent. A dyadic function whose left argument is being evaluated is said to be waiting.

The suspended function can be restarted by entering a branch statement. In the case of stop control, no part of the line has been executed and the function can be safely restarted with a branch to the line number in question. In the other two cases the point of interruption is indicated approximately by a caret (∧). Whether the function can be restarted (even after the error, if any, has been fixed up) normally requires some analysis.

A convenient way to restart a suspended function is to enter →□*LC* since □*LC* is a vector whose first element is the line number at which execution is to be resumed.

In the suspended state, most normal activities are possible, including the evaluation of expressions and the execution of functions, but there are some limitations.

1. All names have their local significance (that is, the significance they had in the suspended function).

2. Space may be limited by the inclusion in the workspace of the local variables of the suspended and pendent functions.

3. Pendent functions cannot be edited.

4. The header of a function which is suspended or waiting cannot be edited.

5. Functions which are suspended, pendent or waiting cannot be erased.

6. The workspace can be saved in this condition but it may not be subsequently loadable by any different release of the MicroAPL system.

NOTE: There may be ways, not prohibited by these limitations, to create an inconsistent workspace by manipulating halted functions.

In general it is best to "punt" (see below) after suspension of function execution unless you have a good reason not to. One good reason not to is if you are not sure what caused the error and wish to investigate further by listing variables or executing subexpressions of the one in error. It is sometimes useful in this case to save a copy of the workspace in its suspended state (under a "temporary" name!) before doing anything that might make the trail hard to follow.

Stop Control

By the use of the system function □*STOP* (see "System Functions") a function can be caused to stop in a suspended state just before executing a given line or lines. The function may be normally restarted safely by branching to the line number of the stop.

Trace Control

A trace of a function line is a display generated on the screen immediately after the execution of the line. The system function □*TRACE* is used to determine which lines are to be traced (see "System Functions"). Execution of the function is not halted. The display generated by a trace consists of a TRACE SET message, the line number and the value, if any, of the expression in the statement.

State Indicator

The system command)*SI* causes the state indicator to be displayed. The state indicator shows all the suspended (marked with an asterisk) and pendent functions. (It does not show the waiting functions.) The order of the display is the same as for □*LC*, that is, most recent first.

For example:

```
      )SI
H[3]  *
G[7]
F[2]  *
      □LC
3 7 2
```

It is good practice to display the state indicator periodically to see that it is clear, and it is especially inportant when something mysterious seems to have happened: a function has disappeared, for instance, or an unusual WS FULL occurs.

"Punt"

The statement → is sometimes called a "punt" (the football term). It may be used as a line of a defined function, as the response to a □ input request, or in immediate execution mode. Its effect in each case is the same: the currently executing function, or the latest suspended function, is terminated, together with all the pendent functions which led to its execution.

The state indicator may always be cleared by executing punt sufficiently many times.

Chapter 6

Primitive Functions and Operations

The term "primitive" refers to things that are available as part of the system without the necessity of defining them.

The primitive functions and operations all have *APL* symbols reserved for them. Almost half of the symbols used for primitive functions actually represent two functions, one monadic and the other dyadic. Which is intended in a given expression must be determined from the context: the dyadic function is denoted if possible, i.e., if there is a left argument.

NOTE: The symbol ←→ used in the following is not *APL* notation. It means "is equivalent to."

-Scalar Functions

Scalar functions are functions defined on scalar arguments, yielding a scalar as a result, and which are extended to array arguments element by element.

--Monadic Scalar Functions

The monadic scalar functions are shown in Table A.1.

Each of these functions has the same syntax as the familiar "negative" function.

Each takes only numeric arguments. Each can be applied element by element to an array argument, yielding an array of the same shape.

---Arithmetic Functions

$R\leftarrow +B$ (Conjugate or indentity) The result is the same as the argument. $+B \leftarrow\rightarrow 0+B \leftarrow\rightarrow B$

$R\leftarrow -B$ (Negative) $-B \leftarrow\rightarrow 0-B$

$R\leftarrow \times B$ (Signum) $\times B$ is $\bar{}1, 0$ or 1 according to whether B is negative, zero or positive. $\times B \leftarrow\rightarrow (B>0)-(B<0)$

$R\leftarrow \div B$ (Reciprocal) $\div B \leftarrow\rightarrow 1\div B$. B must not be zero.

$R\leftarrow \lfloor B$ (Floor) $\lfloor B$ is the greatest integer not greater than B. This result is modified in accordance with the system's comparison tolerance parameter. For example, if the comparison tolerance has its default value of about $1E\bar{}8$ then $\lfloor 7.99999999 \leftarrow\rightarrow 8$. Formally, floor has the following definition.

```
∇R←FL X ; N
[1] N←(×X)×⌊0.5+|X
[2] R←N-(N-X)>⎕CT×1⌈|N
∇
```

$R\leftarrow \lceil B$ (Ceiling) $\lceil B$ is the least integer not less than B. Again the result is modified in accordance with comparison tolerance. $\lceil B \leftarrow\rightarrow -\lfloor -B$

$R\leftarrow *B$ (Exponential) $*B$ is e raised to the B' th power, where e is the base of natural logarithms (approximately 2.71828).

$R\leftarrow \circledast B$ (Natural Logarithm) The inverse of the exponential function. $\circledast *B \leftarrow\rightarrow B \leftarrow\rightarrow *\circledast B$. B must be greater than zero.

$R\leftarrow |B$ (Magnitude) The absolute value of B. $|B \leftarrow\rightarrow B\lceil(-B)$.

$R\leftarrow !B$ (Factorial) $!B \leftarrow\rightarrow B\times(B-1)\times(B-2)\times\ldots\times 2\times 1$ and $0! \leftarrow\rightarrow 1$. B must be a non-negative integer.

$R\leftarrow \circ B$ (Pi times) Pi times B where Pi is approximately 3.14159.

---Random Function

$R \leftarrow ?B$ (Roll) $?B$ is a random choice from the integers ιB. Since ιB is dependent on the current index origin, so is $?B$. B must be a positive integer.

---Logical Function

$R \leftarrow \sim B$ (Not) B must be 0 or 1. $\sim 0\ 1 \leftarrow\rightarrow 1\ 0$

--Dyadic Scalar Functions

The dyadic scalar functions are shown in Table A.2.

Each of these functions has the syntax $R \leftarrow A\ f\ B$ like the familiar "plus" function. Each takes only numeric arguments, except = and $\neq$ which permit both numeric and character arguments. Each is extended to nonscalar arguments according to the following rules.

a) If A and B are the same shape, the function f is applied to corresponding elements of A and B to give a result of the same shape;

b) else, if A and B each have only one element, then the result has one element and is of shape equal to the shape of the argument having the greater rank;

c) else, if one argument has only one element then it is extended to be the same shape as the other argument;

d) else the arguments are not conformable and an error is reported.

---Arithmetic Functions

$R \leftarrow A+B$ (Plus) Addition.

$R \leftarrow A-B$ (Minus) Subtraction.

$R \leftarrow A \times B$ (Times) Multiplication.

$R \leftarrow A \div B$ (Divide) Division. B must not be zero unless A is as well, and then $0 \div 0 \leftarrow\rightarrow 1$.

$R \leftarrow A \lfloor B$ (Minimum) $A \lfloor B$ is the lesser of A and B.

$R \leftarrow A \lceil B$ (Maximim) $A \lceil B$ is the greater of A and B.

$R \leftarrow A \star B$ (Power) A raised to the power B. $A \star B$ is not defined if $A=0$ and $B<0$ or if $A<0$ and B is not an integer. $0 \star 0 \leftarrow\rightarrow 1$

$R \leftarrow A \circledast B$ (Logarithms) $A \circledast B$ is the base A logarithm of B i.e., the power to which A must be raised to give B. $A \circledast B \leftarrow\rightarrow (\circledast B) \div (\circledast A)$. A and B must be positive, and if $A=1$ then $B=1$.

$R \leftarrow A|B$ (Residue) $A|B$ is the remainder when B is divided by A. $0|B \leftarrow\rightarrow B$. If $A \neq 0$ then $R \leftarrow\rightarrow B-A \times \lfloor B \div A$. R will always lie between 0 (inclusive) and A (exclusive) regardless of whether A is positive or negative.

$R \leftarrow A \binom{}{} B$ (Binomial coefficient) This is often read 'A out of B'. One interpretation of it is the number of combinations of B things taken A at a time. A must be a non-negative integer. B may be any number.

$A \binom{}{} B \leftarrow\rightarrow 1$ if $A=0$

$B \times (B-1) \times \ldots \times (B+1-A)) \div (! A)$ if $A>0$

---Logical Functions

$R \leftarrow A \wedge B$ (And)
$R \leftarrow A \vee B$ (Or)
$R \leftarrow A \barwedge B$ (Nand)
$R \leftarrow A \barvee B$ (Nor)

In each case A and B must be 0 or 1. (See Table A.2)

---Relational Functions

$R \leftarrow A < B$ (Less)
$R \leftarrow A \leq B$ (Less or equal)
$R \leftarrow A = B$ (Equal)
$R \leftarrow A \geq B$ (Greater or equal)
$R \leftarrow A > B$ (Greater)
$R \leftarrow A \neq B$ (Not equal)

In each case R is 1 if the relation holds, 0 if it does not. A and/or B can be of type character only in the case of $=$ and $\neq$.

The relational functions on numeric arguments are all subject to comparison tolerance. A is considered ("tolerantly") equal to B if and only if $(|A-B) \leq$ (Comparison tolerance) $\times (|A) \lceil (|B)$.

The other five relational functions then use this version of equality in their definitions.

The comparison tolerance may be changed from its default value of about $1E^{-}8$ by means of the system variable $\square CT$.

The effect of comparison tolerance is to make $9=(3\star 2) \leftarrow\rightarrow 1$ for example even though $(3\star 2)-9 \leftarrow\rightarrow 3.725E^{-}09$

---Trigonometric Functions

$R\leftarrow A\circ B$ This is a family of related functions. The integer A selects the family member. See Table A.2 for details.

-Mixed Functions

The real power (and uniqueness) of *APL* is contained in the mixed primitive functions. The mixed functions deal with, and are defined on, arrays as a whole and not element by element. Their results have shapes which often differ from the shapes of their arguments. The mixed functions are not generally arithmetic in nature.

The mixed functions are shown in Table A.4.

$R\leftarrow\rho B$ (Shape) ρB is the shape vector of the array B.

$R\leftarrow ,B$ (Ravel) $,B$ is the vector whose elements are those of B taken by indexing sequence (that is, with the last index varying most rapidly). If B is a scalar, $,B$ is the vector whose sole element is B. If B is a vector, $,B$ is identical to B.

$R\leftarrow A\rho B$ (Reshape) $A\rho B$ is an array of shape $,A$ whose elements are taken sequentially from $,B$ repeated cyclically as required. A must be a nonnegative integer scalar or vector, or an empty vector. $(\iota 0)\rho B$ is the scalar $(,B)[1]$.

$R\leftarrow\phi B$
$R\leftarrow\ominus B$
$R\leftarrow\phi[V]B$
$R\leftarrow\ominus[V]B$ Reverse ϕB is an array identical to B except that the elements along the last axis are in reversed order. If B is a vector, then ϕB turns B end for end.

The function $\ominus$ is identical to ϕ except that the relevant axis is the first, not the last.

The axis operator (see Operators) can be applied to either ϕ or $\ominus$ to designate the relevant axis.

$R\leftarrow A\phi B$
$R\leftarrow A\ominus B$
$R\leftarrow A\phi[V]B$
$R\leftarrow A\ominus[V]B$ (Rotate) If A is an integer scalar or one-element vector and B is a vector, then $A\phi B$ is a vector identical to B except that if $A>0$ then the elements of B have been rotated cyclically left A places. If $A<0$ the rotation is to the right $|A$ places.

For higher dimensional arrays the shape of A must be $^{-}1\downarrow\rho B$ and then each element of A specifies the amount to rotate the corresponding vector along the last axis of B.

For example,

```
      M←→A B C D
         E F G H
         I J K L

1 0 ¯1 φ M ←→ B C D A
              E F G H
              L I J K
```

As in the monadic case, the function $\ominus$ is identical to ϕ except that the relevant axis is the first, not the last. The axis operator (see Operators) can be applied to either ϕ or $\ominus$ to designate the relevant axis.

$R\leftarrow A,B$
$R\leftarrow A,[V]B$ (Catenate) This function is used for gluing together two arrays to form a larger array.

If A and B are vectors (or scalars), A,B is the vector whose elements are those of A followed by those of B.

Matrices are catenated along the last axis of each by imagining them to be written side by side and then glued together along the adjacent sides. (The last axis is the one which is extended.) Obviously only the first dimension of each must match.

The same idea extends to higher dimensional arrays. For example, a 3×4×5 array may be catenated to a 3×4×2 array of the same type to form a 3×4×7 array.

Arguments which differ in rank by 1 are also permitted, so that for instance a vector may be catenated to a matrix by treating the vector as if it were an n×1 matrix. Again this concept extends to higher dimensional arrays, so that a 3×4 array may be catenated to a 3×4×5 array to form a 3×4×6 array.

The axis operator (see Operators) can be applied to catenate to designate some axis other than the last as the axis to be extended. The axis number *V* must designate one of the axes of the higher rank argument. For example, if the index origin is 1, and *A* and *B* are matrices, then *A*,[2]*B* is equivalent to *A*,*B*, and *A*,[1]*B* corresponds to gluing the bottom edge of *A* to the top edge of *B*.

Unless one of the arrays is empty they must be of the same type, i.e., both numeric or both character.

A scalar argument is extended as necessary.

R←*A*,[*V*]*B* (Laminate) Lamination is analogous to gluing two essentially 2-dimensional sheets of wood together to form a 3-dimensional board.

In *APL* we can join two identically shaped (say 5×7) matrices together to form a 3-dimensional result. The new axis of the result will have length 2, but we have a choice where we locate it in the new array. We can have the result of shape 2×5×7 or 5×2×7 or 5×7×2 depending on whether we put the new axis before the first of the original ones, between the first and second, or after the second.

If *V* is not an integer then *R*←*A*,[*V*]*B* specifies that *A* and *B* are to be laminated, not catenated. And the value of *V* relative to the original axis numbers specifies where the new axis is to go. Thus if the index origin is 1, the *A* and *B* are 5×7 matrices, then *A*,[0.5]*B* is 2×5×7, *A*,[1.5]*B* is 5×2×7 and R←*A*,[2.5]*B* is 5×7×2. In the last case, for example, *R*[;;1] ←→ *A* and *R*[;;2] ←→ *B*.

The exact value of *V* doesn't matter, only where it stands relative to the original axis numbers.

Both arguments must be of the same shape unless one is a scalar, in which case it is extended. Both arguments must be the same type (numeric or character) unless they are empty.

$R \leftarrow A \lozenge B$ (Dyadic Transpose) This function provides a way of permuting the axes of an array (and also of obtaining diagonal sections of an array).

Suppose B is a 3-dimensional array and we wish to form from it the 3-dimensional array R such that $R[I;J;K]=B[K;I;J]$ for all values of K, I and J that are valid subscripts for B. In *APL* this is expressed $R \leftarrow 3\ 1\ 2 \lozenge B$. The left argument of $\lozenge$ is found by inspecting the subscripts $K;I;J$ of B in the equation defining $R[I;J;K]$. The first subscript of B, i.e., K, is the 3rd subscript of R, the second, I, is the first of R and the third, J, is the second of R. Hence 3 1 2.

We can also take a "diagonal section" through an array. For example, we can derive from B a 2-dimensional array S such that $S[I;J]=B[J;I;J]$. In *APL* this is $S \leftarrow 2\ 1\ 2 \lozenge B$. The rule for finding the left argument is the same as above.

If B is a matrix then $2\ 1 \lozenge B$ is the conventional transpose of B and $1\ 1 \lozenge B$ is the main diagonal.

$R \leftarrow \lozenge B$ (Transpose) This function reverses the order of the axes of its argument. Formally, $\lozenge B \leftarrow\rightarrow (\phi \iota \rho \rho B) \lozenge B$. In particular, if B is a matrix then $\lozenge B$ is the conventional transpose of B.

$R \leftarrow A[B;C;\ldots;D]$

(Indexing) Elements may be selected from an array A to form a new array R by means of an index expression in square brackets. An index expression for an n-dimensional array A is a list of n expressions separated by semicolons. The value of each expression must be an array (e.g., B) each of those elements is a permissible index along the corresponding axis of A. Each indexing array may be of any rank, although scalars and vectors are the most common.

Any of the constituent expressions of an index expression may be omitted entirely; its value is taken to be the entire index vector for that axis of A.

The shape of R is the catenation of the shapes of the indexing arrays. In particular if A is a vector then the shape of $A[B]$ is the shape of B. Technically, for higher rank index arrays,

$$A[B;C;\ldots;D] \leftarrow\rightarrow ((\rho B),(\rho C),\ldots,(\rho D))\rho A[(,B);(,C);\ldots;(,D)]$$

The use of a pair of symbols, [and] and what amounts to a vector of arrays as one of its arguments, distinguishes indexing as an exception to the *APL* syntax rules. Nevertheless indexing is still conceptually a dyadic function of an array and an index.

$A[B;C;...;D]\leftarrow R$

(Indexed assignment) An indexed subset of the elements of an existing array A may be replaced by the elements of the array R.

$[B;C;...;D]$ must be a valid index expression for A and the shape of $A[B;C;...;D]$ must match the shape of R, except that axes of length 1 are ignored.

If R is a one element array of any rank, it is extended as necessary.

A and R must be of the same type unless the index expression selects no elements of A.

$R\leftarrow A\uparrow B$ (Take) This function selects elements from the beginning or end of a vector B and it can also be used to "pad" B out to a given length with zeros (if B is numeric) or blanks (if B is character). In general it can be thought of as selecting a "corner" of an array B.

A must be an integer scalar or vector, or an empty vector. B can be an array. Unless B is a scalar, there must be one element of A corresponding to each axis of B.

The shape of $A\uparrow B$ is the vector whose elements are the absolute values of those of A that is $|,A$.

If B is a vector (and therefore A is a scalar or a one element vector) then there are four cases.

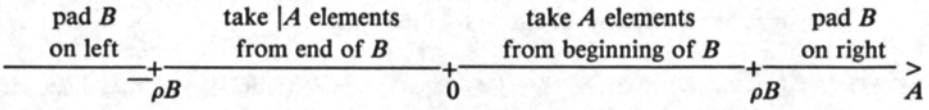
(a) $(\rho B)<A$. R is B catenated with $(\rho B)-A$ zeros (or blanks).

(b) $(0\leq A)\wedge(A<\rho B)$. R is a vector of the first A elements of B.

(c) $((-\rho B)\leq A)\wedge(A<0)$. R is a vector of the last $|A$ elements of B.

(d) $A<-\rho B$. R is $-(A+\rho B)$ zeros (or blanks) catenated with B.

The following diagram summarizes these cases.

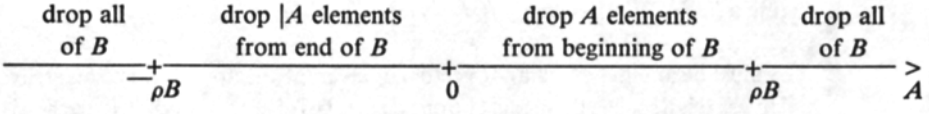


If B is of higher rank then each of the axes is treated as in the vector case, using the corresponding element of A.

$R \leftarrow A \downarrow B$ (Drop) This function is a variant of take. It also selects a "corner" of the array B but it does it by deleting rows and columns rather than by keeping them.

The conditions on A and B are the same as for take.

If B is a vector there are four cases as illustrated in the following diagram.



If B is of higher rank then each of the axes is treated as in the vector case, using the corresponding element of A.

$R \leftarrow A/B$
$R \leftarrow A \neq B$
$R \leftarrow A/[V]B$
$R \leftarrow A \neq [V]B$ (Compress) This function provides selection based on a boolean vector A of 1's and 0's.

B may be any array and A is a boolean scalar or vector, or an empty vector.

A scalar or one element vector A is extended to conform to B and a scalar B is extended to a vector conforming to A.

The number of elements of A (after extension) must equal the length of the last axis of B (after extension).

If B is a vector, the A/B is the vector consisting of those elements of B corresponding to the 1's in A. It follows that the length of A/B is the number of 1's in A.

If B is of higher rank, then the compression is applied to the vectors along its last axis.

$A⌿B$ is identical to A/B except that the compression is applied to the first axis, not the last.

The axis operator (see Operators) can be applied to either / or ⌿ to designate the relevant axis.

$R←A\B$

$R←A⍀$

$R←A\[V]B$

$R←A⍀[V]B$ (Expand) This function opens out the array B by inserting zeros (or blanks) based on the boolean vector A. It is a partial inverse to compression in the sense that $A/A\B \longleftrightarrow B$.

B may be any array and A is a boolean scalar or vector, or an empty vector.

A scalar B is extended to a vector of length equal to the number of 1's in A.

The number of 1's in A must equal the length of the last axis of B (after extension).

If B is a vector, then $A\B$ is the vector whose length is that of A and consisting of the elements of B placed in order wherever A has 1's and zeros (or blanks) wherever A has 0's.

If B is of higher rank then the expansion is applied to the vectors along its last axis.

$A⍀B$ is identical to $A\B$ except that the expansion is applied to the first axis instead of the last.

The axis operator (see Operators) can be applied to either \ or ⍀ to designate the relevant axis.

$R←⍳B$ (Index generator) $⍳B$ is a vector of B consecutive ascending integers, the first of which is the current index origin.

B must be a nonnegative integer scalar or other one element array.

$⍳0$ is a common expression yielding an empty numeric vector.

$R \leftarrow A \iota B$ (Index of) $A \iota B$ is a "search" function which finds the first occurrence in A of each element of B.

A can be any vector. B can be any array. $A \iota B$ has the same shape as B.

If B is a scalar then $A \iota B$ is the index (relative to the current index origin) of the first occurrence of B in A. If B doesn't occur in A at all, then $A \iota B$ is $\square IO + \rho A$ (i.e., the first index beyond the range of A).

If B is an array of higher rank then each element of $A \iota B$ is the least index in A of the corresponding element of B.

This function is obviously index origin dependent.

The equality test implied in this function uses the comparison tolerance for numeric arguments.

$R \leftarrow A \epsilon B$ (Member of) A and B can be any arrays. $A \epsilon B$ is a boolean array the same shape as A. Each element of $A \epsilon B$ is 1 if the corresponding element of A occurs anywhere in B, and 0 otherwise.

The equality test implied in this function uses the comparison tolerance for numeric arguments.

$R \leftarrow \grave{\Delta} B$ (Grade up) A "sorting" function. B may be any numeric vector. $\grave{\Delta} B$ has the same shape as B.

$\grave{\Delta} B$ *is the permutation of* $\iota \rho B$ such that $B[\grave{\Delta} B]$ is in nondecreasing order. The indices of any set of identical elements of B occur in $\grave{\Delta} B$ in ascending order.

Since $\iota \rho B$ is index origin dependent, so is $\grave{\Delta} B$. Comparison tolerance is not used in the comparisons.

$R \leftarrow \grave{\nabla} B$ (Grade down) B may be any numeric vector. $\grave{\nabla} B$ has the same shape as B.

$\grave{\nabla} B$ is the permutation of $\iota \rho B$ such that $B[\grave{\nabla} B]$ is in nonascending order. The indices of any set of identical elements of B occur in $\grave{\nabla} B$ in ascending order.

Since $\iota \rho B$ is index origin dependent, so is $\grave{\nabla} B$. Comparison tolerance is not used in the comparisons.

R←*A*?*B* (Deal) A random function. *A* and *B* are nonnegative integer scalars or one element arrays, and $A \leq B$.

A?*B* is a vector of length *A*, obtained by making *A* random selections, without replacement, from the population ⍳*B*.

Since ⍳*B* is index origin dependent, so is *A*?*B*. The system's random link parameter is used implicitly.

R←*A*⌹*B* (Matrix divide) This function is useful in numerical work for solving systems of linear equations, least-squares fitting problems and problems involving projections of vectors in n-space.

A and *B* are numeric arrays of rank 0, 1 or 2. Scalars and vectors are shaped into one column matrices. The (reshaped) arguments must both have the same number of rows, and the columns of (the reshaped) *B* must be linearly independent. (The latter implies in particular that *B* cannot have more columns than rows.)

The definition of *A*⌹*B* is deceptively brief: the shape of *A*⌹*B* is (1↓⍴*B*),(1↓⍴*A*) and the value of *A*⌹*B* is such that +/,(*A*−*B*+.×*A*⌹*B*)*2 is minimized.

In particular if *B* is a square, nonsingular matrix then the minimum referred to is zero and *B*+.×(*A*⌹*B*) ←→ *A*. If *A* is a vector then *A*⌹*B* is the vector solving the set of linear equations *B*+.×*X* ←→ *A*. If *A* is a matrix then *A*⌹*B* gives the solution of the set of equations for each column of *A*.

It can be seen from the definition that if *B* is a rectangular matrix then *A*⌹*B* gives the linear combination of the columns of *B* which most closely (in the least-squares sense) matches *A*. This can be used in linear regression for curve fitting, and in vector projection problems.

R←⌹*B* (Matrix inverse) *B* is a numeric array of rank 0, 1 or 2. A scalar or vector is reshaped into a one column matrix. The columns of the (reshaped) argument *B*′ must be linearly independent. (In particular, *B*′ cannot have more columns than rows.)

If *N* is the number of rows of the reshaped argument *B*′ and *I* is an *N*×*N* identity matrix then ⌹*B* ←→ *I*⌹*B*′, except that if *B* is a scalar or vector then the result is reshaped to a scalar or vector respectively.

$R\leftarrow A\perp B$ (Decode, or base value) This function is used to convert a coded representation, B, of a number into the number itself, according to the coding scheme or radix A. This is one of the essential operations in number system conversions.

If A and B are vectors of equal length then $A\perp B$ is a scalar equal to $+/W\times B$ where W is a weight vector corresponding to the radix A: $W\leftarrow\phi\times\backslash\phi 1\downarrow A,1$. For example if $A\leftarrow\rightarrow 3\ 5\ 7$ then $W\leftarrow\rightarrow 3\ 5\ 7\ 1$, and $3\ 5\ 7\perp 2\ 3\ 5\leftarrow\rightarrow +/3\ 5\ 7\ 1\times 2\ 3\ 5\leftarrow\rightarrow +/70\ 21\ 5\leftarrow\rightarrow 96$.

A scalar or one element vector argument is extended to conform to the other argument. Thus $2\perp 1\ 0\ 1\ 1\leftarrow\rightarrow 2\ 2\ 2\ 2\perp 1\ 0\quad 1\ 1\leftarrow\rightarrow +/8\ 4\ 2\ 1\times 1\ 0\ 1\quad 1\leftarrow\rightarrow 11$. ("The base 2 value of 1 0 1 1 is 11.")

In the case of arguments of higher rank, each radix vector along the last axis of A is applied to each representation vector along the first axis of B, in accordance with the above algorithm. The shape of $A\perp B$ is $(\overline{\ }1\downarrow\rho A),(1\downarrow\rho B)$.

$R\leftarrow A\top B$ (Encode, or representation) This function converts a number B into an encoded representation according to the coding scheme or radix A. This is one of the essential operations in number system conversions.

Encode is a partial inverse of decode. For example $3\ 5\ 7\top 96\leftarrow\rightarrow 2\ 3\ 5$ while $3\ 5\ 7\perp 2\ 3\ 5\leftarrow\rightarrow 96$.

If A is a vector and B is a scalar then $A\top B$ is equivalent to the following program. The index origin is assumed to be 1 and comparison tolerance is 0.

```
      ∇R←A ENCODE B;I
[1]   R←0×A                  ⍝0-VECTOR,SAME SHAPE AS A
[2]   I←ρA
[3]   REPEAT:
[4]       →(I=0)/0
[5]       R[I]←A[I]|B
[6]       →(A[I]=0)/0        ⍝REMEMBER: 0|B←→B
[7]       B←(B-R[I])÷A[I]
[8]       I←I-1
[9]       →REPEAT
          ∇
```

If A is a scalar then $A\top B\leftarrow\rightarrow A|B$.

In the case of higher rank arrays, each vector along the first axis of *A* is applied to obtain the representation of each element of *B*, the resulting representations being arrayed along the first axis of the result. That is,

$$A[;J;\ldots;K]\top B[L;\ldots;N]\leftarrow\rightarrow R[;J;\ldots;K;L;\ldots;N]$$

for all values of $J,\ldots K,L,\ldots,N$ which are valid indices for *A* and *B*.

The shape of $A\top B$ is $(\rho A),(\rho B)$.

$\blacktriangle A$

$R\leftarrow\blacktriangle A$ (Execute) This function executes the *APL* statement represented by the character vector (or scalar) *A*. The value of $\blacktriangle A$ is the value of the statement, if it has one. If the execution of *A* evokes an error report then the report of the type of error will be preceded by $\blacktriangle$. A system command is not an *APL* statement and cannot be executed with $\blacktriangle$.

Two common uses for execute are (a) passing object names, including function names, as arguments to functions, with later evaluation, and (b) converting character representations of numbers, obtained from a file, for example, to numeric form.

Execute can make programs difficult to analyze or understand. It should be used with restraint.

$R\leftarrow\overline{\blacktriangledown}B$ (Format) *B* may be any array. The result of this function is a character array which, if displayed, would have the same appearance as if *B* itself were displayed.

The system's printing precision parameter (see $\square PP$) is used implicitly for format, but the printing width (see $\square PW$) is not: format acts as though the printing width were infinite.

Format does not alter a character array argument.

For a numeric argument, the shape of $\overline{\blacktriangledown}B$ is the same as the shape of *B* except for the last dimension. A scalar *B* is an exception: it is treated as a one element vector.

Format is useful for mixing character and numeric data on one output display line, and for converting data to be sent to an external file or device such as a printer.

R←*A*⍕*B* (Dyadic format) This function is similar in purpose to monadic format, but it uses variations in the left argument to provide progressively more detailed control over the result.

B may be any numeric array. *A* is an integer scalar or vector.

In general a pair of numbers is used to control the result. The first determines the total width of a number field and the second controls the precision.

If the precision indicator is negative then E-format is used and the magnitude of the precision indicator is the number of digits in the multiplier. If the precision indicator is nonnegative then regular decimal form is used and the value of the indicator specifies the number of digits to the right of the decimal point.

If the width indicator is zero, a field width is chosen such that at least one space will be left between adjacent numbers.

If *A* is a scalar or a one element vector it is treated like a number pair with a width indicator of zero.

If *A* is a two element vector, it provides the width and precision for the entire array *B*.

Otherwise *A* must be a vector with a pair of elements (width, precision) for each index along the last axis of *B*.

-Operators

Operators in *APL* provide a means of modifying some of the primitive functions or of creating whole families of new functions.

Although in many contexts the terms "function" and "operator" are used more or less synonymously, in *APL* they have quite distinct meanings. "Function" is used for things such as + or ⍳ which take arrays as arguments and produce arrays as results. "Operator" is used for a special kind of function which takes functions and/or arrays as arguments and produces a derived function as a result.

Reduction, scan, inner product, outer product and axis are the five operators available in MicroAPL.

R←f/*B*
R←f⌿*B*
R←f/[*V*]*B*
R←f⌿[*V*]*B* (Reduction) NOTE: The symbol for the reduction operator is /. f/ is the mixed monadic function derived by applying reduction to any scalar dyadic primitive function f. *R*←f/*B* is the syntax of the derived function.

The definition of f/*B* is as follows.

(a) If *B* is a scalar, f/*B*←→*B*.

(b) If *B* is a 0-element vector, then f/*B* is the "identity element" for f, as shown in Table A.3. If no identity element exists a domain error is evoked.

(c) If *B* is a 1-element vector then f/*B* is the scalar (⍳0)⌷*B*.

(d) If *B* is a vector of length 2 or more, then f/*B* ←→ *B*[1] f *B*[2] f...f *B*[⌷*B*].

(e) If *B* is an array of higher rank, the reduction rules (b) - (d) are applied to the vectors along the last axis of *B*. The shape of f/*B* is ¯1↓⌷*B*.

f⌿ is identical to f/ except that the reduction is applied along the first axis instead of the last.

The axis operator can be applied to the derived function f/ or f⌿ to designate the relevant axis.

Some common reductions are

+/	sum of
×/	product of
⌈/	maximum of
⌊/	minimum of
∧/	"for every"
∨/	"there exists"
≠/	parity check

R←f⍀*B*
R←f⌿*B*
R←f⍀[*V*]*B*
R←f⌿[*V*]*B*

(Scan) NOTE: The symbol for the scan operator is ⍀. f⍀ is the mixed monadic function derived by applying scan to any scalar dyadic primitive function f. *R*←f⍀*B* is the syntax of the derived function.

The definition of f⍀*B* is as follows.

(a) If *B* is a scalar or a 0-element vector, f⍀*B* ←→ *B*.

(b) If *B* is a vector of length 1 or more then for every scalar $I \epsilon \iota \rho B$, (f⍀*B*)[*I*] ←→ f/*B*[⍳*I*].

(c) If *B* is an array of higher rank, the scan rules (a) and (b) are applied to the vectors along the last axis of *B*.

The shape of f⍀*B* is the shape of *B*. f⌿ is identical to f⍀ except that the scan is applied along the first axis instead of the last.

The axis operator can be applied to the derived function f⍀ or f⌿ to designate the relevant axis.

R←f/[*V*]*B*
R←f⌿[*V*]*B*
R←f⍀[*V*]*B*
R←f⌿[*V*]*B*
R←⊖[*V*]*B*
R←⌽[*V*]*B*
R←*A*⌽[*V*]*B*
R←*A*/[*V*]*B*
R←*A*⍀[*V*]*B*
R←*A*,[*V*]*B* (Axis) The axis operator, designated by the pair of symbols [and] , takes an axis value *V* and modifies the function to the left, usually by designating a relevant axis of one of its arguments.

In all cases *V* must be a numeric scalar or 1-element vector.

Since axes are numbered relative to the current index origin, the axis value *V* is origin dependent.

R←*A*f.g*B* (Inner product) Inner product derives a new dyadic mixed function f.g from any two dyadic scalar primitive functions f and g.

If *A* and *B* are vectors of the same length, or they are both scalars, then *A*f.g*B* ←→ f/*A*g*B*.

In general, if *A* and *B* are arrays other than scalars, and $(\overline{\ }1\uparrow\rho A) = (1\uparrow\rho B)$, then the shape of *A*f.g*B* is $(\overline{\ }1\downarrow\rho A),(1\downarrow\rho B)$ and (*A*f.g*B*) [*I*;...;*J*;*L*;...;*M*]←→ f /*A*[*I*;...;*J*;] g *B* [;*L*;...;*M*) for all valid sets of indices.

Finally if either one of *A* or *B* is a scalar, or if either is a 1-element vector, it is reshaped into a vector whose length satisfies the general case above.

The inner product +.× is the ordinary "matrix product." Other common inner products are ∧.= , +.= , ⌊.+ , ⌈.+ , and ×.*.

R←*A*∘.g*B* (Outer product) Outer product derives a new dyadic mixed function ∘.g from a dyadic scalar primitive function g.

The shape of *A* ∘.g*B* is $(\rho A),(\rho B)$ and (*A*∘.g*B*)[*I*;...;*J*;*L*;...;*M*] ←→ *A*[*I*;...;*J*] g B[L;...;M] for all valid sets of indices. (If *A* or *B* is a scalar, the index expression is omitted.)

Chapter 7

System Variables and System Functions

System variables and system functions provide facilities for communicating with the *APL* System. Unlike system commands they can be used within *APL* expressions.

System names are distinguished: they all start with □ or ⍰.

System variables and functions are always in your workspace, but they do not appear in)*FNS* and)*VARS* lists. They cannot (and need not) be copied or erased. Some of them can usefully be localized in function definitions: □*IO* is the most common example.

The difference between system variables and system functions is mainly a matter of their syntax.

System Variables

System variables are best viewed as variables which you share with the *APL* system. You can't do anything to or with a system variable without the system taking some notice.

When you use a system variable in an expression the system generates and supplies the value. For example □*WA* returns the working area available, but the system has to do a storage reorganization and cleanup to get that value.

Conversely, when the value of a system variable is specified, the system may automatically adjust one of its internal parameters. For example $\square IO \leftarrow 0$ results in the internal index origin parameter being set to 0. Often there is a limited set of acceptable values for an internal parameter, and the system will appear to ignore an attempt to give the corresponding system variable an unacceptable value.

$\square CT$ Comparison tolerance. Controls the internal comparison tolerance parameter. Acceptable values: 0 through $1E^{-}5$. Value in a clear workspace: $1E^{-}8$.

$\square IO$ Index origin. Controls the internal index-origin parameter, which is used in indexing, the axis operator, $\square FX$, ?, dyadic $\circledcirc$, $\uparrow$, $\downarrow$ and ι. It is the index of the first element of any nonempty vector. Acceptable values: 0 or 1. Value in a clear workspace: 1.

$\square LX$ Latent expression. When a workspace is loaded, $\square LX$ is executed. Value in a clear workspace: an empty vector.

$\square PP$ Printing precision. Controls the parameter which determines the number of significant digits in the output representations of numeric *APL* arrays. Acceptable values: integers 1 through 11. Value in a clear workspace: 8.

$\square PW$ Printing width. Controls the parameter which is the maximum width of a line. Affects all output except bare output ($\square$). Acceptable values: integers 24 through 80. Value in a clear workspace: 80.

$\square RL$ Random link. Used in roll and deal(?). Acceptable values: integers 1 through 32767.

$\square AV$ Atomic vector. A 256 element vector of the bytes whose hexadecimal representations are \$00,\$01,...,\$FF in ascending order. Thus, in origin 1, $\square AV[17]$ is equivalent to \$10. The elements of character variables are all elements of $\square AV$ and any element of $\square AV$ can be used in a character variable. The value of $\square AV$ cannot be altered.

$\square LC$ Line counter. A vector of the line numbers of active functions, most recently initiated first. $\square LC$ cannot be altered by assigning values to it.

□*TC* Terminal control. An 8 element character vector whose elements have the following effects when output to the screen.

□*TC*[1]	move cursor left
□*TC*[2]	move cursor right
□*TC*[3]	move cursor down
□*TC*[4]	move cursor up
□*TC*[5]	clear screen and home cursor
□*TC*[6]	home cursor to top left hand corner
□*TC*[7]	RETURN (" new line")
□*TC*[8]	Erase to end of line

□*TC* cannot be altered.

□*TS* Timestamp. Used for setting and reading the internal system clock. A six element integer vector representing the date and time as follows:

□*TS*[1]	Last two digits only of the year
□*TS*[2]	Month (1 through 12)
□*TS*[3]	Day (1 through 31, consistent with month and year)
□*TS*[4]	Hour (0 through 23)
□*TS*[5]	Minute (0 through 59)
□*TS*[6]	Second (0 through 59)

Acceptable values: representations of valid dates and times in the above format.

□*TS* may be set at any time. The system will then keep its clock up to date. The system clock is not kept in the workspace, so loading a new workspace does not affect it.

□*WA* Working area. Available space in the active workspace in bytes. □*WA* cannot be changed by assigning it a value.

□ Evaluated Input/Formatted Output. When □ is assigned a value, the system displays a representation of the value on the screen. □*PP* and □*PW* affect the display. A vector which cannot be displayed within the printing width is continued, indented on subsequent lines. Rows of a matrix are displayed as separate vectors. One line is skipped between the matrices of a 3-dimensional array, two lines between 3-dimensional subarrays and so on. *E* format is used automatically when necessary for numeric arrays.

When a value for □ is required, the system generates its value by obtaining it from the user: a prompt (□:) is displayed and the user must enter an expression to be evaluated, or → (see "Punt"). The value of the expression is the value of □.

⍞ Character Input/Bare Output. When ⍞ is assigned a character scalar or vector value, the system sends the characters to the screen in a continuous stream without gratuitous newline characters. This is useful and often necessary when you wish to display long strings containing cursor positioning characters (see □*TC*).

When a value for ⍞ is required, the system generates the value as it does for □ except that (a) there is no prompt: the cursor simply remains where it is, and (b) the input is taken as a vector of characters and is not evaluated. Trailing blanks are trimmed off and the result is always a (character) vector.

Program function key 2 (i.e., shifted "2" on the numeric keypad), as a response to ⍞ or □ input, is equivalent to a "punt."

NOTE: Because a machine awaiting character input looks just the same as a machine locked in a long calculation (the cursor is at the left margin), it is good practice to include your own prompt in your programs. ⍞ output followed directly by ⍞ input is a neat way of doing this on one screen line.

System Functions

□*CR F* Canonical representation. *F* is a character vector (or scalar) naming a function. The result is a character matrix containing a representation of the function. The representation is similar to that displayed by the ∇ − editor, but without ∇'s or line numbers. The result is of shape 0 0 if *F* does not denote an existing function.

□*DL S* Delay. *S* is a positive integer scalar. This function takes *S* seconds to complete. The result is the length of the delay, namely *S*.

□*EX A* Expunge. *A* is a character scalar, vector or matrix. Any variables or functions named by the rows of *A* are erased, if possible. The explicit result is a boolean vector whose *I*′ th element is 1 if the *I*′ th row of *A* denotes a name which is now available for use. whether or not an object by that name was erased.

□FX M Fix. *M* is a character matrix representing a function definition in the same format as the result of *□CR*. *□FX* establishes the definition if possible. The explicit result is a character vector naming the function established, or else the index (relative to the current index origin) of the first row of *M* containing a fault which prevented establishment. The name of the intended function cannot be in use, except as a function name.

□NC A Name classification. *A* is a character scalar, vector or matrix. The result is a vector of name classifications giving the usage of the character sequences in each row of *A*

0: a name available for usage
1: a label
2: a variable name
3: a defined function name
4: other (a distinguished name, or not a name)

□NL K Name list. *K* is a numeric scalar or vector with elements 1, 2 or 3. The result is a character matrix whose rows name the objects in the indicated classes which exist in the active workspace: 1, 2 or 3 for labels, variables or functions, respectively.

L □NL K Name list. *L* is a character vector or scalar. The result is like that of monadic *□NL* but the list contains only names beginning with one of the letters in *L*.

□STOP F *F* is a character (scalar or) vector naming a function. The result is an integer vector of line numbers on which "stops" have been placed.

N □STOP F

F is a character (scalar or) vector naming a function. *N* is a vector of line numbers. "Stops" are set on the lines of the named function whose numbers appear in *N* (See Stop Control for the effect of a stop). An empty vector *N* removes all stops. Editing a function removes all its stops.

N □TRACE F
□TRACE F Similar to *□STOP* but for "traces" instead of "stops."

□LOAD W *W* is a character vector naming a workspace. *□LOAD* is identical to *)LOAD* but it may be executed as an *APL* statement.

□PEEK V *V* is either (a), a scalar or vector of integers defining machine addresses, or (b), a 2-column matrix of integers whose rows define ranges of machine addresses. The ranges are inclusive.

The result is a character vector (i.e., of elements of *□AV*) containing the current contents of the machine addresses defined by V.

N.B. The machine addresses are interpreted as in origin 0, regardless of the current index origin.

Examples

(1) *□PEEK* 0 is the current contents of hexadecimal location 0000. The only reliable way to "see" the value is to look it up in *□AV*: *□AV*[*□PEEK* 0 produces an integer result in the current index origin.

(2) *□PEEK* 32768 32769 is the contents of the first 2 locations on the screen (Hexadecimal 8000-8001).

(3) *□PEEK* 1 2 *ρ*32768+0 1999 is the entire screen. (Hex 8000 - 87CF).

C □POKE V

V is a scalar, vector or range matrix, as in *□PEEK*. *C* is a character vector whose length is consistent with *V*. A scalar *C* is extended to be consistent with *V*.

Example *□AV □POKE* 1 2*ρ*32768+0 255 will show the result of storing all possible byte values into screen memory.

NOTE: It is sometimes important to know that the assignment of bytes to addresses occurs "from left to right" in *□POKE*.

□SYS C

A □SYS C Execute 6809 machine code. The machine code (i.e., instructions) contained in, or pointed to by, *C* is executed.

C is either
(1) a scalar or one element vector integer between 0 and 65535, representing the machine address at which execution is to begin, or

(2) a character (scalar or) vector of bytes to execute directly.

Control is returned to *APL* by executing a 6809 RTS instruction.

The explicit result of □*SYS* is an integer scalar representing the value of the 6809 D register at the time of returning.

A is an optional parameter list. If present it must be a (scalar or) vector of integers between 0 and 65535. The two-byte value represented by the first element of *A* is placed in the 6809 D register before execution begins, and the remaining elements are placed on the stack (two stack bytes for each element). The top pair of bytes on the stack contain the return address (i.e., of *APL's* □*SYS* handler). The second pair correspond to *A*[2], the third pair to *A*[3] and so on. In each pair the low order byte is stacked first.

The stack may be used by the machine language routine, but it must be left in this same condition again when the RTS instruction is executed to return to *APL*.

□*XR C* External representation. Translates *APL* characters to "*APL*-ASCII Overlay (Typewriter-Pairing)" representation. *C* is a (scalar or) vector of *APL* characters or elements of □*TC*. The result is a character vector formed by catenating the representations of the elements of *C* according to the *APL*-ASCII typewriter pairing convention (see appendix). Overstruck elements of *C* are expanded into the two constituent characters separated by a backspace character.

□*IR C* Internal representation. □*IR* and □*XR* are strict inverses except that □*IR* will handle either of the two possible permutations corresponding to the overstruck symbols.

There are more system functions in MicroAPL. These all concern files and are included in a subsequent chapter.

Chapter 8

Errors

If an error is detected in the execution of a statement, an error report is displayed. This report consists of an error message followed by a display of the statement. The point at which execution was interrupted is marked by a caret (∧). Any implicit results, such as variables having been assigned values, which occurred before the point of error, remain in effect.

Error Messages

SYNTAX ERROR

A line of *APL* characters is not a valid statement.

VALUE ERROR

An expression with no value occurs in a context requiring a value. The use of a variable before it has been assigned a value is a common cause of this error.

DOMAIN ERROR

The argument or arguments of a primitive function are not within its domain of definition. Generally speaking, *RANK* and *LENGTH* errors are recognized first if possible.

RANK ERROR

The rank of an argument of a function does not meet the requirements of the function, or the ranks of the left and right arguments do not conform.

LENGTH ERROR

The argument ranks conform, but the sizes of one or more axes do not.

INDEX ERROR

(1) The value of an index expression is an invalid index for the associated array. (2) An invalid axis number is specified in an axis operator.

CHAR ERROR

Certain keys are invalid in *APL* and evoke a character error report when struck.

SYMBOL TABLE FULL

Too many names have been used. Save the workspace; clear and copy the entire workspace. (The symbol table size may be changed in a clear workspace with the)*SYMBOLS* command.)

WS FULL

Out of memory. Clear the state indicator by executing → sufficiently many times. Erase unnecessary objects.

SYSTEM ERROR

Fault detected in the *APL* system.

DEFN ERROR

This error is dealt with in the chapter on defined functions.

Chapter 9

Files

A file in the *APL* system is viewed as a collection of items external to the APL workspace. These files can be created, added to, retrieved from, erased and renamed through the use of special *APL* system functions.

There are four types of files:

a) *APL*-sequential
b) BARE-sequential
c) Relative
d) Program

An *APL*-sequential file is simply a sequence of *APL* values of any shape, rank and type.

A BARE-sequential file is a sequence of 8-bit binary characters.

A Relative file is a sequence of 8-bit binary characters which are organized into fixed sized groups called 'records' which may be accessed in random order.

A Program file is a special kind of BARE-sequential file normally used to contain programs and workspaces. As such, it is not usually accessed with the techniques discussed in this chapter.

General Concepts:

Files reside on some external medium and are identified by a name. Filenames are constructed according to specific rules described below.

In order for the data in a file to be available to a workspace, it must be "tied" to the workspace with a tie-number. Information is transferred to and from the file using this number. When access is not required by the workspace, the file can be "untied."

Filenames:

The filenames of *APL*-sequential and BARE-sequential files are composed of any combination of letters and numbers (see Systems Overview Manual) (e.g., the name of a disk file can contain up to 15 characters). The following are valid filenames for sequential files:

GEORGE
INVENTORY.NOV
*SALES*004

The filenames of Relative files are somewhat more detailed than those of sequential files. The basic filename is formed by following the same rules as above. However, special additions are also required in the name. The meaning of the special sequences are described later in the section titled "Relative Files." The following are valid names for Relative files:

∨~(45∧*GEORGE,REL*
∨~(80∧*INVENTORY.NOV,REL*
∨~(200∧*SALES*004,*REL*

Replies:

The file operations described in the rest of this chapter frequently return replies which reflect the result of the operation. The reply is in the form of a character vector which contains text describing any abnormal condition encountered. If the operation was successful, the reply is an empty vector.

General File Manipulation Functions:

REPLY ← *FN* □*CREATE N*

FN is a character vector containing the name of a file which is to be created. If a file with the same name already exists, it is replaced. The argument *N* is an integer scalar specifying the "tie-number" to be used. Output operations such as □*WRITE* or □*PUT* are valid.

REPLY ← *FN* □*TIE N*

This function is used to access an existing file. The file specified by *FN* is located and attached to the workspace with the tie-number given by the integer scalar *N*. Only input operations such as □*READ* or □*GET* are valid on a file accessed in this manner.

REPLY ← *FN* □*APPEND N*

This function is used to add items of data to an existing sequential file. The file specified by *FN* is located and attached to the workspace with the tie-number defined by the integer scalar *N*. Only output operations such as □*WRITE* or □*PUT* are valid here.

REPLY ← *FN* □*UPDATE N*

This function is used to access existing relative files. *FN* specifies a file which is located and attached to the workspace with tie number *N*. Data can be sent to and retrieved from the file using the functions □*READ*, □*GET*, □*WRITE* and □*PUT*.

□*UNTIE N*

The argument *N* is an integer vector containing tie-numbers of files to be released from the workspace.

REPLY ← *FN* □*ERASE N*

The file named in *FN* is erased. *N* is an integer scalar representing the tie-number of the currently tied file to be erased.

REPLY ← *FN* □*RENAME N*

The argument *N* is an integer scalar specifying a currently tied file. This file is renamed to the name defined in the argument *FN*.

□*NUMS*

This function returns a vector of all currently active tie-numbers. Their order corresponds to that of the filenames returned by the function □*NAMES*.

□*NAMES*

This function returns a character array of the names of all currently tied files. Their order corresponds to that of the tie-numbers returned by the function □*NUMS*.

REPLY ← □*STATUS N*

The argument *N* is an integer scalar representing a file tie-number. The function response reflects the status of the most recent operation performed on the file with the specified tie-number.

Z←□*LIB L* *L* is a character vector designating a device (e.g., 'DISK/1'), or an empty vector. *Z* is a character matrix representing the directory for the device. (ASCII characters are not necessarily translated to *APL*.)

APL Sequential Files

APL-sequential files are a sequence of *APL* values of any shape, rank or type. Such files are accessed using □*CREATE*, □*TIE*, □*APPEND* and □*UNTIE*. Values are written into the file using □*WRITE* and read back into the workspace using □*READ*.

REPLY ← *Z* □*WRITE N*

Z is any *APL* variable and *N* is an integer scalar. The rank, shape and type of *Z* as well as all of its data are put on the file tied with number *N*.

Z ← □*READ N*

The 'next' item in the file tied with number *N* is read into the workspace and assigned to the variable *Z*. The variable assumes the shape, rank and type of the data value from the file.

BARE-Sequential Files

BARE-sequential files are a sequence of 8-bit binary values (called bytes). These files are accessed using □*CREATE*, □*TIE*, □*APPEND* and □*UNTIE*. Bytes are written to the file from *APL* character variables using □*PUT* and read back into the workspace using □*GET*.

REPLY ← *Z* □*PUT N*

The argument *Z* is a character vector and *N* is an integer scalar representing a tie number. Character data from the specified vector is written to the file. Each byte from the item is transmitted to the file and no additional bytes are added.

Z ← □*GET NL*

NL is a 2-element integer vector. The first element contains the tie-number of a currently tied file. Character data from the file is transmitted to the workspace and stored in *Z* as a character vector. The number of bytes transmitted is specified by the second element of *NL*. If there are too few bytes left in the file, only those available will be transmitted.

Relative Files

A relative file is composed of "records," each of a fixed size. The record size is defined as part of the filename when the file is created (using □*CREATE*). This is done by prefixing the name with the sequence ∨~(*NN*∧ where *NN* is the desired record length. The name must also be suffixed by the sequence ,*REL*. (The cryptic sequence is the *APL* equivalent of the ASCII codes required by the disk system.)

e.g., to create a relative file named *SAMPLES* with records of length 100, use the filename

∨~(100∧*SAMPLES,REL*

Relative files are accessed using □*CREATE*, □*TIE*, □*UPDATE* and □*UNTIE*. Operations on relative files are performed on individual records, so □*GET* and □*PUT* must be preceded by the □*SEEK* operation positioning the file-system to a specific record.

REPLY ← □*SEEK NL*

NL is a 2-element integer vector. The first element is the tie-number of a currently tied relative file. The file system positions its "current-record" pointer to the I'th record in the file where I is specified by the second element of the vector *NL*. If the record does not exist, a reply is returned to that effect. This reply can usually be ignored when the file is being written.

NOTES:

1 An attempt to write more characters than will fit on a record will result in an I/O ERROR.

2 An attempt to read more characters than are on one record will cause characters to be read from the next record.

3 The maximum record length in a relative file is 254 bytes.

Appendix A

Tables of Functions

Table A.1 - Primitive Monadic Scalar Functions

Symbol	Name	Definition or Example	Page
+	Conjugate or identity	$+B \leftarrow\rightarrow 0+B$	62
−	Negative	$-B \leftarrow\rightarrow 0-B$	62
×	Signum	$\times B \leftarrow\rightarrow$ ¯1 if $B<0$; 0 $B=0$; 1 $B>0$	62
÷	Reciprocal	$\div B \leftarrow\rightarrow 1\div B$	62
⌊	Floor	⌊3.14 ¯3.14 ←→ 3 ¯4	62
⌈	Ceiling	⌈3.14 ¯3.14 ←→ 4 ¯3	62
*	Exponential	$*B \leftarrow\rightarrow (2.7128...)*B \leftarrow\rightarrow e*B$	62
⍟	Natural logarithm	$⍟B \leftarrow\rightarrow e⍟B$	62
|	Magnitude or absolute value	|3.14 ¯3.14 ←→ 3.14 3.14	62
!	Factorial	$!0 \leftarrow\rightarrow 1$; $!B \leftarrow\rightarrow B\times !(B-1)$	62

Symbol	Name	Definition or Example	Page
?	Roll	?*B* ←→ Random choice from ⍳*B*	63
○	Pi times	○*B* ←→ *B*×3.14159...	62
~	Not	~0 1 ←→ 1 0	63

Table A.2 - Primitive Dyadic Scalar Functions

Symbol	Name	Definition or Example	Page
+	Plus	2+3.2 ←→ 5.2	63
−	Minus	2−3.2 ←→ ¯1.2	63
×	Times	2×3.2 ←→ 6.4	63
÷	Divide	2÷3.2 ←→ 0.625	63
⌊	Minimum	3⌊7 ←→ 3	63
⌈	Maximum	3⌈7 ←→ 7	63
*	Power	2*3 ←→ 8	64
⍟	Logarithm	*A*⍟B ←→ Log *B* (base *A*) *A*⍟B ←→ (⍟*B*)÷⍟*A*	64
|	Residue	*A*|*B* ←→ *B* if *A*=0 *A*|*B* ←→ *B*−*A*×⌊B÷A if *A*≠0	64
!	Binomial	*A*!B ←→ (!*B*)÷(!*A*)×!*B*−*A*	64
	coefficient	2!5 ←→ 10 3!5 ←→ 10	64

		A	*B*	*A*∧*B*	*A*∨*B*	*A*⍲B	A⍱B	
∧	And	0	0	0	0	1	1	64
∨	Or	0	1	0	1	1	0	64
⍲	Nand	1	0	0	1	1	0	64
⍱	Nor	1	1	1	1	0	0	64

<	Less			64
≤	Not greater	Result is 1 if the		64
=	Equal	relation holds, 0		64
≥	Not less	if it does not:		64
>	Greater	3≤7 ←→ 1	'A'≠3 ←→ 1	64
≠	Not equal	7≤3 ←→ 0	'B'='B' ←→ 1	64

○ *Trigonometric* *Restrictions*

A	*R*←*A*○*B*	Domain	Range
7	tanh *B*		
6	cosh *B*		

A	$R \leftarrow A \circ B$	Domain	Range
5	sinh B		
4	$(1+B*2)*.5$		$0 \leq R$
3	tan B		
2	cos B		
1	sin B		
0	$(1-B*2)*.5$	$B \leq 1$	$0 \leq R$
¯1	arcsin B	$(\lvert B) \leq 1$	$(\lvert R) \leq \circ \div 2$
¯2	arccos B	$(\lvert B) \leq 1$	$(0 \leq R) \wedge (R \leq \circ 1)$
¯3	arctan B		$(\lvert R) < \circ \div 2$
¯4	$(\bar{}1+B*2)*.5$	$1 \leq \lvert B$	$0 \leq R$
¯5	arcsinh B		
¯6	arccosh B	$1 \leq B$	$0 \leq R$
¯7	*arctanh B*	$(\lvert B) < 1$	

The angular measure is radians.

Table A.3 - Identity Elements of Dyadic Scalar Functions
(see Reduction)

DYADIC SCALAR FUNCTION	IDENTITY ELEMENT
+	0
−	0
×	1
÷	1
|	0
⌊	The largest representable number.
⌈	Greatest (in magnitude) negative no.
*	1
⍟	None
○	None
!	1
∧	1
∨	0
⍲	None
⍱	None
<	0
≤	1
=	1
≥	1
>	0
≠	0

A.4 - Table of Mixed Primitive Functions

In the following table the "syntax" column indicates the highest rank of the arguments. *S* : scalar; *V* : vector; *M* : matrix; *A* : any array. Generally, lower rank arguments are acceptable.

Arrays used in the examples:

```
N ←→ 3 4⍴⍳12 ←→ 1  2  3  4
                5  6  7  8
                9 10 11 12
C ←→ 3 2⍴'ABCDEF' ←→ AB
                     CD
                     EF
```

Syntax	Name	Examples	Notes	Page
⍴A	Shape	*⍴C* ←→ 3 2 *⍴*2 3 5 7 ←→ 4 *⍴*7 ←→ <empty vector>		65
V⍴A	Reshape	3 4⍴⍳12 ←→ *N* 4⍴*C* ←→ *'ABCD'* 0⍴2 3 7 ←→ <empty vector> (⍳0)⍴*C* ←→ *'A'*		65
,A	Ravel	*,N* ←→ ⍳12 *,*7 ←→ 1⍴7		65
⌽A	Reverse	*⌽C* ←→ *BA* *DC* *FE* *⌽*[1]*C* ←→ *EF* *CD* *AB* *⌽*2 3 5 7 ←→ 7 5 3 2	1	65
A⌽A	Rotate	¯1*⌽'WORDS'* ←→ *'SWORD'* 1 2*⌽*[1]*C* ←→ *CF* *EB* *AD*	1	66

Syntax	Name	Examples	Notes	Page
A,*A*	Catenate	3 4,7 2 1 ←→ 3 4 7 2 1 *C*,'*XYZ*' ←→ *ABX* *CDY* *EFZ*	1	66
A,[*V*]*A*	Laminate	'*CAT*',[0.5]'*DOG*' ←→ *CAT* *DOG* '*CAT*',[1.5]'*DOG*' ←→ *CD* *AO* *TG*	1	67
V⍉*A*	Dyadic Transpose	ρ3 1 2 ⍉(2 3 5ρι30) ←→3 5 2 1 1 ⍉ *N* ←→ 1 6 11	2	68
⍉*A*	Transpose	⍉*C* ←→ *ACE* *BDF*		*68*
V↑*A*	Take	¯2↑3 5 7 ←→ 5 7 4↑2 3 4 ←→ 2 3 4 0 2 1↑*C* ←→ *A* *C*		69
V↓*A*	Drop	2↓'*EXAMPLE*' ←→ '*AMPLE*' ¯1↓3 5 7 ←→ 3 5 ¯2 ¯1↓*C* ←→ 1 1ρ'*A*'		70
V/*A*	Compress	1 0 1/2 3 5 ←→ 2 5 0 1 1/[1]*C* ←→ *CD* *EF*	1	70
*V**A*	Expand	1 1 0\5 7 ←→ 5 7 0 1 0 1*C* ←→ *A B* *C D* *E F*	1	71
V[*A*] *M*[*A*;*A*] *A*[*A*;...;*A*]	Indexing	'*EXAMPLE*'[4 3 5] ←→ '*MAP*' '*ABCDEFGHIJKL*'[*N*]←→*ABCD* *EFGH* *IJKL* *C*[3 2;] ←→ *EF* *CD*	2	68

Syntax	Name	Examples	Notes	Page
A[*A*;...;*A*]←*A*	Indexed assignment	*C*[1;]←'*PQ*'←→ *C* ←→ *PQ* *CD* *EF*	2	69
⍳*S*	Index generator	⍳3←→ 1 2 3 ⍳0←→ <empty vector>	2	71
V⍳*A*	Index of	'ABCD'⍳'C' ←→ 3 'SOUS'⍳'CS' ←→ 5 1 'ABCD'⍳*C* ←→ 1 2 3 4 5 5	2,3	72
A∈*A*	Member of	'CAT'∈*C* ←→1 1 0 *C*∈'CABBAGE' ←→ 1 1 1 0 1 0	3	72
⍋*V*	Grade up	⍋6 8 6 2 ←→ 4 1 3 2 6 8 6 2[⍋6 8 6 2]←→2 6 6 8	2	72
⍒*V*	Grade down	⍒5 3 1 6←→4 1 2 3 5 3 1 6[⍒5 3 1 6]←→6 5 3 1	2	72
S?*S*	Deal	3?5 ←→ 3 4 1 3?5 ←→ 1 4 5		73
⌹*M*	Matrix inverse	⌹2 2⍴1 1 0 1 ←→ 1 ¯1 0 1 30×⌹1 2 5 ←→1 2 5 ⌹5 ←→ 0.2		73
M⌹*M*	Matrix divide	2 5⌹2 2⍴1 1 0 1←→ ¯3 5		73
A⊥*A*	Decode	10⊥1 8 6 7←→1867 24 60 60⊥15 50 10←→57010		74

Syntax	Name	Examples	Notes	Page
A⊤*A*	Encode	(8⍴2)⊤13←→0 0 0 0 1 1 0 1 24 60 60⊤56999←→15 49 59		74
⍎*V*	Execute	⍎'3+4' ←→ 7 ⍎'*C*' ←→ *AB* *CD* *EF*		75
⍕*A*	Format	⍴⍕*N*←→3 12 ⍕ 5.7 ¯1.2 ←→ '5.7 ¯1.2'		75
		⍕*C* ←→ *C*		
V⍕*A*	Dyadic format	5 2⍕(÷1 2 3)←→' 1.00 0.50 0.33'		76
		9 ¯4⍕(÷7)←→'1.429*E*¯01'		
		4 2⍕(÷3 1⍴⍳3)←→ 1.00 0.50 0.33		

NOTES:

1 Axis operator is allowed. (Axis is always index origin dependent.)

2 Index origin dependent.

3 Comparison tolerance dependent.

Appendix B

System Commands, Variables and Functions

B●1 - System Commands

B●2 - System Variables

B●3—System Functions (Non-files)

B●4 - System Functions (Files)

B●4 - System Functions (Files)

Appendix C

Character Code Tables

C●1 - *APL*-ASCII (Typewrite-Pairing) Overlay

	0-*	1—*	2—	3—	4—	5—	6—	7—
—0	*NULL*	⍒		0	—	*	◇	*P*
—1	*HOME*	⍋	¨	1	⍺	?	*A*	*Q*
—2	*RUN*	⍫	)	2	⊥	⍴	*B*	*R*
—3	*STOP*	⍉	<	3	∩	⌈	*C*	*S*
—4	*DEL*	⊖	≤	4	⌊	~	*D*	*T*
—5	*INST*	⍟	=	5	∈	↓	*E*	*U*
—6	*EEOL*	⍱	>	6	_	∪	*F*	*V*
—7	*CRFWD*	⍦	]	7	∇	⍵	*G*	*W*
—8	*CRBCK*	⍤	∨	8	∆	⊃	*H*	*X*
—9	*TAB*	⍀	∧	9	⍳	↑	*I*	*Y*
—A	*CRDWN*	⌿	≠	(	∘	⊂	*J*	*Z*
—B	*CRUP*	⍁	÷	[	'	←	*K*	{
—C	*CLEAR*	⍓	,	;	⍞	⊢	*L*	⊣
—D	*CR*	⍎	+	×	|	→	*M*	}
—E	⍲	⍈	.	:	⍕	≥	*N*	$
—F	⍀	⌶	/	\	○	—	*O*	*RUB*

*NOTE:
These columns contain extensions to the standard *APL*-ASCII typewriter-pairing convention.

C●2 - TERMINAL Character Set

	0—	1—	2—	3—	4—	5—	6—	7—
—0	NULL			0	@	P	‘	p
—1	HOME		!	1	A	Q	a	q
—2	RUN		"	2	B	R	b	r
—3	STOP		#	3	C	S	c	s
—4	DEL		$	4	D	T	d	t
—5	INST		%	5	E	U	e	u
—6	EEOL		&	6	F	V	f	v
—7	CRFWD		’	7	G	W	g	w
—8	CRBCK		(	8	H	X	h	x
—9	TAB		)	9	I	Y	i	y
—A	CRDWN		*	:	J	Z	j	z
—B	CRUP		+	;	K	[	k	{
—C	CLEAR		,	<	L	\	l	|
—D	CR*		—	=	M	]	m	}
—E			.	>	N	^	n	~
—F		μ	/	?	O	_	o	RUB

*NOTE:
On output, CR also causes an automatic skip to new-line.

C●3—ASCII Character Set

	0—	1—	2—	3—	4—	5—	6—	7—
—0	NULL	DLE		0	@	P	‘	p
—1	SOH	DC1	!	1	A	Q	a	q
—2	STX	DC2	"	2	B	R	b	r
—3	ETX	DC3	#	3	C	S	c	s
—4	EOT	DC4	$	4	D	T	d	t
—5	ENQ	NAK	%	5	E	U	e	u
—6	ACK	SYN	&	6	F	V	f	v
—7	BEL	ETB	’	7	G	W	g	w
—8	BS	CAN	(	8	H	X	h	x
—9	HT	EM	)	9	I	Y	i	y
—A	LF	SUB	*	:	J	Z	j	z
—B	VT	ESC	+	;	K	[	k	{
—C	FF	FS	,	<	L	\	l	|
—D	CR	GS	—	=	M	]	m	}
—E	SO	RS	.	>	N	^	n	~
—F	SI	US	/	?	O	_	o	RUB

C●4 - Internal Character Representation

	0—	1—	2—	3—	4—	5—	6—	7—
—1	*A*	*I*	*Q*	*Y*	5	→	⊖	↓
—3	*B*	*J*	*R*	*Z*	6	ρ	/	↑
—5	*C*	*K*	*S*	_	7	⊞	⊥	?
—7	*D*	*L*	*T*	0	8	⍫	\	~
—9	*E*	*M*	*U*	1	9	ι	⊀	+
—B	*F*	*N*	*V*	2	⍋	⍁	⊤	—
—D	*G*	*O*	*W*	3	⍋	,	⊥	×
—F	*H*	*P*	*X*	4	⍒	ϕ	ϵ	÷

...cont'd

	8—	9—	*A*—	*B*—	*C*—	*D*—	*E*—	*F*—
—1	⌈	∧	≠	;	⍒	∪	◇	*CRFWD*
—3	⌊	⍱		:	∘	⌶		*CRDWN*
—5	*	⍲	(	¨	□	$		*CRUP*
—7	⍟	<	)	¯	⍁	⊣		*CLEAR*
—9	|	≤	·	ω	⊂	⊢		*HOME*
—B	⍋	=	'	←	⊃	{		*CR*
—D	○	≥	[	α	∩	}		*EEOL*
—F	∨	>	]	∇	⍋	Δ	*CRBCK*	

NOTE:

1 Since only odd-numbered entries have meaning, only those are shown here.

APL is a powerful and concise programming language which is ideally suited for the analysis of financial, statistical and engineering data, database applications and data communications. One of its chief characteristics is the speed with which computer applications can be developed, and the ease with which existing programs may be modified. The language enjoys a high degree of standardization.

Waterloo microAPL follows closely IBM's internal standard of APL. All the standard language features consistent with a single-user environment are included.

Features and Extensions

- All the standard primitive functions and operators
- No limitations (other than workspace size) on array ranks or shapes
- Up to 80-character names
- Direct, fast screen access and cursor control
- Full-screen editing
- Blanks retained in defined functions for readability
- Ability to read and modify memory
- Ability to execute machine language functions
- Sequential files of APL arrays
- Arbitrary sequential files
- All system functions for function establishment, canonical representation, latent expression, etc.

This manual is presented in two parts: a tutorial introduction to microAPL and a comprehensive reference manual. Appendices are included which contain summaries of the language primitives and system features.

DISTRIBUTED BY

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$9.95/21907

ISBN: 0-672-21907-7