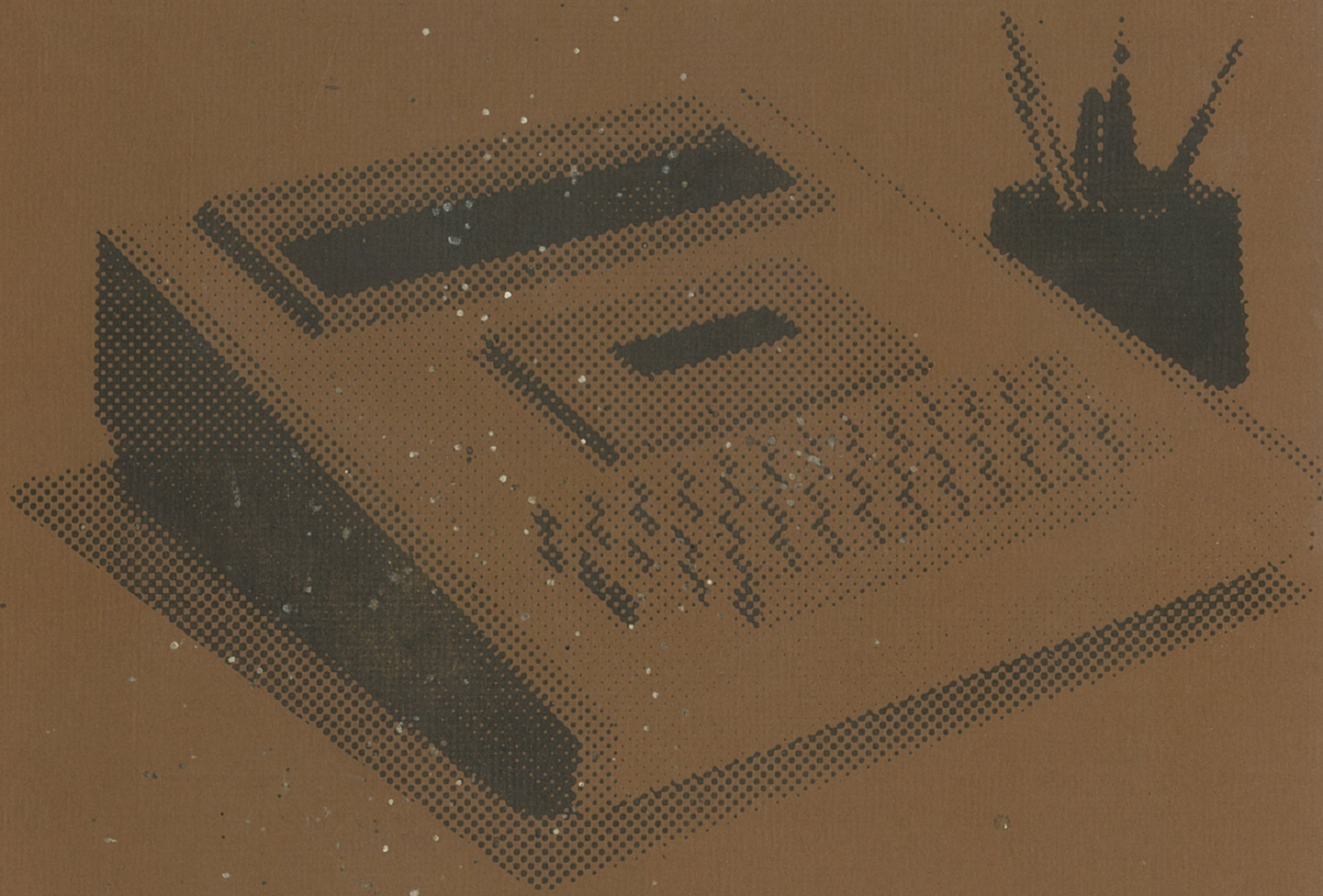
MCM/70



USER'S GUIDE

MICRO COMPUTER MACHINES INC.

TARNELL

MCM/70

USER'S GUIDE

MICRO COMPUTER MACHINES INC.

PREFACE

Thirty years ago, the computer was a virtually unheard of entity. Today, its presence is felt in almost every facet of our lives. Who would have believed in 1940 that, in less than thirty years, our weather would be monitored by space satellites; that our cataloque ordering would be done via direct communication with a machine; or that our children would not be relying solely on the teacher for education, but instead receiving assistance from an electronic device called a computer? Yes, in just a very short period of time, the computer has progressed from a laboratory curiosity to one of the most relied upon machines in history.

It has gained this status by reason of its two properties. They are, its ability to perform accurate calculations at speeds vastly superior to that of man, and its tremendous capacity to retain mountains of information in the form of facts and figures, retrievable at a moment's notice. Both of these properties are increasing all the time as new technological innovations are developed.

Speed and instant access to information have enabled the computer to become such a necessary requirement in the science, business and educational communities. But, at the same time, it has remained a mysterious "box" to the large segment of the population. Two factors are responsible for this paradoxical situation. The first one is availability. Very few people have even seen a computer, let alone used one. Because of its typical cost and the expenses involved in maintaining the people required to run it, the computer usually can be found only in universities and big businesses. With a purchase price of a few million dollars or a monthly rental fee of several thousands, not too many companies can afford the luxury of having one of these gadgets on their premises. Those that do are confronted with the second reason for its alien-like treatment.

Once a person has access to a computer, he must know its "language" in order to use it in any meaningful way. This language may be COBOL, or Fortran, or some other for which he has taken a three to six month course to learn. Most people who are using the computer, and almost all of those who should be using it, do not have this amount of time available to them for such an activity. Therefore they have to rely on "computer experts" to act as interfaces between themselves and the computer, thus creating an additional problem. Maybe you have already encountered this.

The computer experts are comprised of "analysts" and "programmers". The analysts analyze your problem, and the programmers convert the solution into computer readable form. Both are experts in their respective fields. Unfortunately, they are not experts in yours. Because of this, clear communication between you and the experts becomes a critical necessity, which, in practically every instance is an unattainable commodity. This means it may take the experts several attempts at finding a solution to your problem before they come up with one that you will accept. It may not be exactly the one you wanted, but, in order to start getting some tangible results, you take it. One can quickly see why a negative attitude exists among "non computer" people.

So, for these two reasons, availability and ease of use, Micro Computer Machines has developed the MCM/70. It is a desk-top computer that weighs approximately twenty pounds and offers the most powerful, yet easy to learn and use computer language available today. The name of this language is APL (an acronym taken from a book entitled "A Programming Language" written by Dr. Kenneth E. Iverson and published by John Wiley & Sons). You will find that, after only a brief exposure to the language and to the MCM/70, you will be performing computer applications which would normally take you weeks to do with any other language.

J. Morgan Smyth

Contents

Chapter 1: FEATURES OF THE MCM/70

Keyboard	1.1
Display Screen	1.2
Main Memory	1.4
Tape Cassette Option	1.4

Chapter 2: CHARACTERISTICS OF THE MCM/APL SYSTEM

Switching On	2.1
Switching Off	2.1
Modes Of Operation	2.1
Degree Of Accuracy	2.4
Error Reports	2.5
Order Of Execution	2.5
Input Format	2.7
Error Correction	2.8
Character Insertion And Deletion	2.9
Summary	2.10
Practice Exercises	2.10

Chapter 3: MORE OF THE SYSTEM'S CHARACTERISTICS

Variables	3.1
Valid Variable Names	3.3
Literals	3.4
Primitive Functions	3.5
Function Syntax	3.6
Arrays	3.7
Vectors	3.8
Matrices	3.8
Dimension And Rank	3.10
Complex Arrays	3.11
Summary	3.12
Practice Exercises	3.13

Chapter 4: SCALAR FUNCTIONS

Definition Of A Scalar Function	4.1
Signum	4.4
Identity	4.5
Exponentiation	4.5
Exponential	4.7
Logarithms	4.8
Natural Logarithm	4.8
Maximum	4.9
Ceiling	4.10
Minimum	4.10
Floor	4.11
Practice Exercises	4.12

Chapter 5: MORE SCALAR FUNCTIONS

Residue	5.1
Absolute Value	5.2
Factorial	5.2
Combinations	5.4
Trigonometric Functions	5.5
Pi Times	5.6
Monadic Random (Roll)	5.6
Practice Exercises	5.7

Chapter 6: RELATIONAL AND LOGICAL FUNCTIONS

Relational Functions	6.1
Logical Functions	6.2
Or	6.3
And	6.3
Nor	6.4
Nand	6.4
Not	6.5
Summary	6.6
Practice Exercises	6.7

Chapter 7: REDUCTION AND SCAN

Reduction	7.1
Averaging	7.2
Times Reduction	7.4
Order Of Execution	7.4
Scan	7.6
Practice Exercises	7.7

Chapter 8: DEFINED FUNCTIONS

Function Definition Mode	8.3
Displaying A Function	8.5
Function Editing	8.6
Line Addition	8.7
Line Insertion	8.7
Line Modification	8.9
Line Deletion	8.10
Practice Exercises	8.12

Chapter 9: TYPES OF USER DEFINED FUNCTIONS

Header Line	9.1
Function Syntax	9.1
Types Of Header Lines	9.3
Niladics	9.4
Monadics	9.5
Dyadics	9.6
Explicit vs No Explicit Result Functions	9.7
Summary	9.10
Practice Exercises	9.11

Chapter 10: TYPES OF VARIABLES

Local vs Global	10.1
Additional Local Variables	10.4
Suspended Functions	10.7
Summary	10.9
Practice Exercises	10.9

Chapter 11: SOME MIXED FUNCTIONS

Restructure	11.2
Dimension Of	11.5
Index Of	11.7
Index Generator	11.10
Indexing	11.13
Practice Exercises	11.18

Chapter 12: MORE MIXED FUNCTIONS

Membership	12.1
Grade Up	12.2
Grade Down	12.4
Catenate	12.5
Ravel	12.9
Take	12.10
Drop	12.11
Dyadic Random (Deal)	12.12
Practice Exercises	12.13

Chapter 13: REDUCTION AND SCAN WITH MULTIDIMENSIONAL ARRAYS

Reduction	13.1
Reduction Example	13.5
Scan	13.6
Scan Example	13.8
Practice Exercises	13.9

Chapter 14: COMPRESSION - EXPANSION

Compression	14.1
Expansion	14.5
Practice Exercises	14.6

Chapter 15: BRANCHING

Types Of Branches	15.1
Unconditional Branch	15.2
Conditional Branch	15.2
Labels	15.6
Positioning Of Branches In Statements	15.7
Summary	15.8
Practice Exercises	15.10

Chapter 16: INPUT - OUTPUT

Numeric Input	16.1
Output	16.4
Literal Input	16.5
Additional Features	16.7
Heterogeneous Output	16.9
Practice Exercises	16.9

Chapter 17: INNER AND OUTER PRODUCTS

Inner Product	17.1
Outer Product	17.5
Practice Exercises	17.7

Chapter 18: STILL MORE MIXED FUNCTIONS

Rotate	18.1
Reversal	18.5
Monadic Transpose	18.6
Dyadic Transpose	18.8
Practice Exercises	18.13

Chapter 19: THE REMAINING MIXED FUNCTIONS

Base Value (Decode)	19.1
Representation (Encode)	19.3
Null	19.3
Execute (Unquote)	19.5
Summary	19.7
Practice Exercises	19.7

Chapter 20: ERROR REPORTS

Domain Error	20.2
Index Error	20.2
Length Error	20.3
Range Error	20.3
Rank Error	20.4
Syntax Error	20.5
Tape Error	20.6
Value Error	20.6
Workspace Full	20.7

Chapter 21: SYSTEM FUNCTIONS AND VARIABLES

System Functions	21.1
System Variables	21.4
Comparison Tolerance	21.5
Index Origin	21.7
Line Counter And State Indicator	21.9
Print Precision	21.11
Print Time	21.11
Random Link	21.12
Workspace Available	21.13

Chapter 22: IDENTITY ELEMENTS

Appendix A: PRIMITIVES, SPECIAL SYMBOLS AND OTHERS

Primitives	A1
Special Symbols	A2
Numerics	A2
Alphabetics	A2
Symbols	A2
Overstrikes	A3
Interrupts	A3
Output Indicators	A4

Appendix B: AVS-EASY

Type Of Cassette Used	B1
Mounting And Dismounting A Cassette	B2
Activating AVS	B3
Tape Organization	B4
Tape Related Functions	B5
Initializing A Tape	B6
EASY	B7
Existing Groups	B8
Contents Of Existing Groups	B8
Reading Objects From Tape	B9
Writing Objects On Tape	B10
Deleting An Object From A Group	B11
Summary	B11
AVS	B12
Activating AVS And A Tape Group	B13
Local And Global Groups	B14
Resolution Of Group Synonyms	B15
Status Of AVS	B16
Appending An Object To The Global Group	B17
Creating An Entry In The Local Group Directory	B17
Closing A Tape	B18
Tape Related Errors	B19
Multi-Cassette Systems	B20

Chapter 1: FEATURES OF THE MCM/70

Before using the MCM/70 computer, you should become familiar with its three main features. On the basic unit, these include the keyboard, the display screen and the computer's memory. The two most noticeable features are the keyboard and the display screen shown in the illustration below. The third feature, the computer's memory, which plays a major role in every operation performed by the MCM/70, is not visable in figure 1.1 but is discussed in detail later on in this chapter.

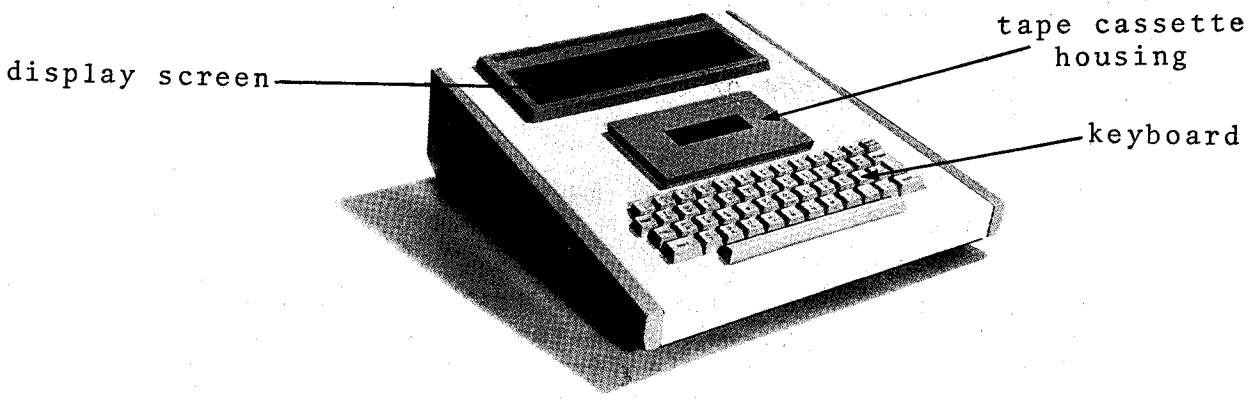


Figure 1.1: Typical MCM/70 Computer

One additional feature which is not standard on the basic unit but is offered as an option is the magnetic tape cassette. Since the vast majority of the existing MCM/70's are equipped with the cassette drive mechanism, a description of its characteristics has been included with the descriptions of the other three features. If your computer has this feature, you will be interested in Appendix C of this book, as it describes how the cassette is utilized.

Keyboard

Illustrated in figure 1.2 is a layout of the MCM/70 keyboard. It is like a regular typewriter keyboard except that most of the lower shift key positions contain all the capitalized letters of the alphabet, and all the upper shift positions are occupied by various symbols. Some

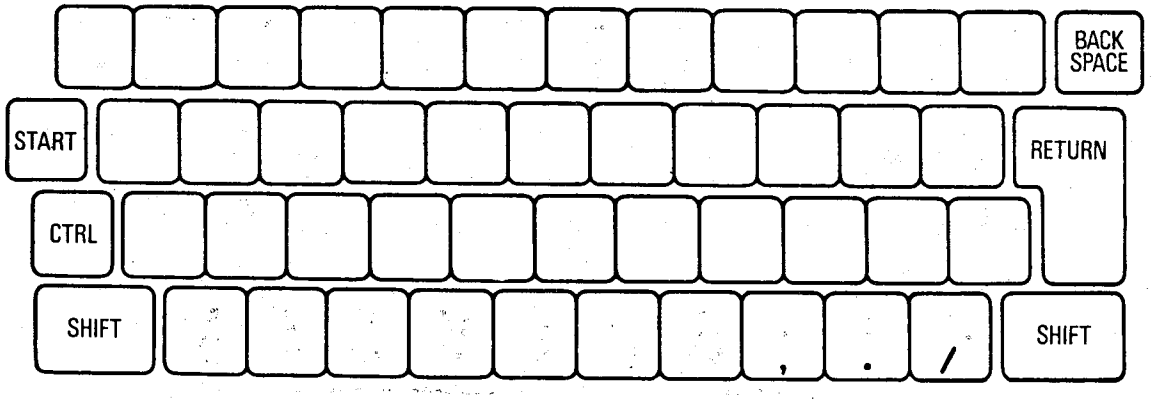


Figure 1.2: MCM/APL KEYBOARD

of these symbols may look familiar but most are probably quite foreign. As you read further, you will see that these symbols form the foundation of the MCM/70 system. In fact, these symbols play such a predominant role in the computer's utilization that the majority of this book is devoted to explaining their individual properties and how each one is used.

Display Screen

Figure 1.3 shows the MCM/70 character display screen. All input by the user and all output by the computer is displayed on this screen. The displayed characters are formed from a 5 × 7 illuminating dot matrix. In the example below, the solid block of illuminated dots

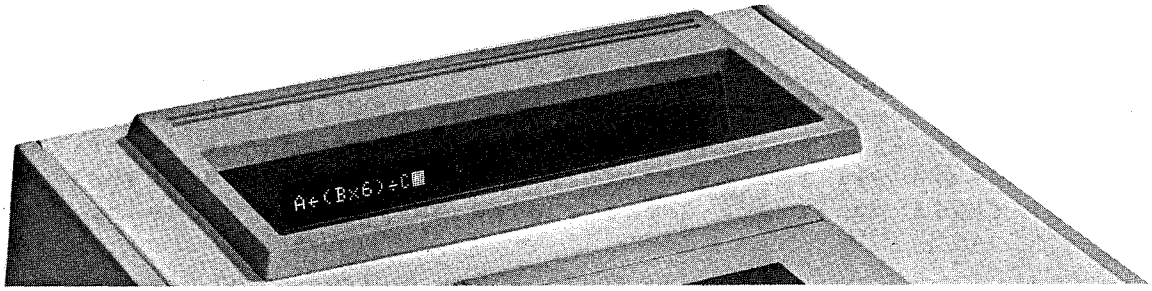


Figure 1.3: MCM/70 Display Screen

appearing to the right of the displayed characters is called the cursor. It indicates where the next character will be displayed when it is typed in by the user.

Up to 32 characters can be displayed on the screen at any one time. Below, figure 1.4 illustrates a "full" screen with all of its 32 display positions occupied by various characters. In cases where the

Figure 1.4: A "Full" Screen

input exceeds this limit, the computer will automatically shift all the characters it is currently displaying one position to the left as each additional character is entered. Figure 1.5 shows the above screen after one additional character has been entered. Notice the *A* disappeared when the 6 was typed. The *A* still exists in the

Figure 1.5: The Moving Window Effect

computer's memory and is still part of the input data, even though it does not physically appear on the screen. This technique is known as the moving window. The window moves one position to the right for each additional keystroke, allowing you to view the 31 most recent characters entered. If you wish to see part of the input entered but not currently being displayed on the screen, you press the BKSP (backspace) key once to get each character to display. By pressing the BKSP key once, the display screen in figure 1.5 is changed to look exactly like that in figure 1.4. But, unlike the input in figure 1.4, the added character 6 is now part of the input. Up to 85 characters can be entered as input at any one time.

Main Memory

Inside the MCM/70 there is a portion of its internal components that make up its main memory. This section, known as the user's workspace, serves both as a storage device and as a scratchpad. Any programs written and any data defined get stored here. Any computations performed are also done within this area.

At the beginning of each session, the workspace is completely empty. As new programs are written and new data defined, the workspace area available gradually begins to fill up. Approximately 1,500 bytes, which is equivalent to 1,500 characters, can be stored in the basic MCM/70. If the memory becomes full, you must perform a little "housekeeping" to make sure there is enough room for any additional data you wish to enter. This can be done by selectively erasing items, which is discussed in Chapter 2, or by clearing the memory entirely. One way to clear it is to terminate the session and start a new one. Problems of over loading the workspace can be avoided if your computer is equipped with a tape cassette attachment.

Tape Cassette Option

The purpose of the tape cassette is twofold. On one hand it acts as an extension to the size of the computer's memory, and on the other it serves as an excellent offline storage device and exchange medium. Each cassette has more than 100,000 bytes of usable storage space. This means a virtually unlimited number of reusable programs and

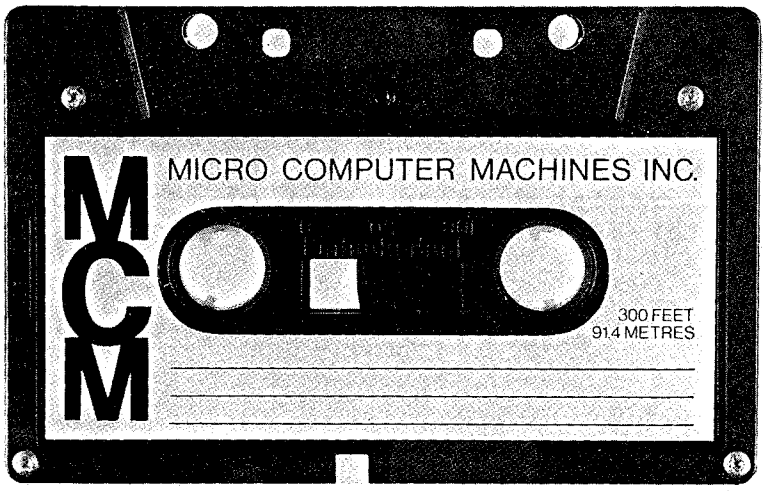


Figure 1.6: Tape Cassette

data can be written and saved on these cassettes. You can have several cassettes with each one containing routines that perform specific tasks. For instance, you could have an accounts receiveable tape comprised of your A/R file and the necessary programs to access and maintain it. You could have a tape containing nothing but statistical packages; or one that does simulation modelling; or even one for storing routines that perform computer assisted learning functions such as spelling and typing drills. You can quickly establish an entire library of these tapes, with each labelled according to the particular subject area it covers. Apart from the possible uses of the cassette when it is offline, online, it becomes part of the computer's main memory, immediately accessable by the computer whenever any of its contents are requested. The way it works is quite simple. Because all computations take place within the 1,500 byte workspace, the system must make sure there is enough space available to perform any immediate tasks before trying to do them. If there is not, the computer will automatically start transferring items from the workspace onto the tape until sufficient space is available.

The opposite is true when retrieving data from the cassette. If certain items are requested by the user, the system will first search for them in its main memory, then go to the cassette in order to carry out the request. Both reading from the cassette and writing onto it are done automatically without any user intervention. The only indication that the cassette is being referenced at all is by its visual movements. More information on this subject is covered in Appendix B.

Chapter 2: CHARACTERISTICS OF THE MCM/APL SYSTEM

Apart from its external features, the MCM/70 has one additional feature that makes it the most powerful microcomputer available today. This feature is called the MCM/APL system. Based on IBM's APL\360 system, MCM/APL is the computer language used by the MCM/70. It offers the user an easy to learn, easy to use, computer system to perform extremely complex tasks in a simple, concise and interactive manner. The power and flexibility of the system will become apparent as you read the rest of this chapter, plus those that follow, and do each set of practice exercises.

Switching On

The procedure for starting the MCM/70 is the same as for an electric typewriter or desk calculator. Simply plug it in and press the START key. To indicate it is ready for use, the computer displays *MCM/APL* on the screen. Press the RETURN key to erase this message.

Switching Off

Switching it off is just as easy. Type in □*OFF* only and press the RETURN key. The reason why this particular set of symbols is used rather than an OFF switch is explained fully in Chapter 21.

Modes of Operation

MCM/APL has two different "modes" of operation. One is called execution mode and the other is called definition mode. In execution mode, the MCM/70 immediately acts upon whatever you type in as soon as you press the RETURN key. In definition mode, it serves as an online

program writing and editing facility. Chapters 4 and 5 discuss in detail all the various characteristics of this mode.

When the computer is in execution mode, whatever is entered by the user is immediately evaluated by the computer. For instance, by entering the mathematical statement 2 + 3 , the computer immediately displays its result of 5 .

```
      2+3 <-------the input statement
5 <-------------------the computer's answer
```

Each subsequent statement you enter is computed with the same degree of promptness, much like that of a desk calculator. This is why execution mode is often referred to as calculator mode, as both descriptions are appropriate. Below are a few more examples of some rather elementary mathematical problems being solved by the MCM/70 in execution mode. For purposes of simplicity, those entered by the user are indented six spaces from the left margin.

```
      10+17
27

      17-10.5
6.5

      5×12
60

      54÷6
9
```

In these, and all ensuing examples, the computer's responses appear immediately beneath the inputs to which they refer, while in actual fact, the responses displayed on the screen by the computer replace the statement entered by the user. The dot matrix display has been substituted by the type font used in these examples to facilitate the printing of this book. The font chosen is the same as that used by other textbooks written on the subject of APL. Try each of the above examples. After you have entered each statement, press the RETURN key to signal to the computer that you have finished typing your input. Press the RETURN key again to erase the computer's response. You will notice some of the dots on the screen flash on and off after the RETURN key has been pressed. This is the computer computing.

Here are a few more examples:

```
      10-17
¯7
```

Notice that the ¯ symbol representing the negative 7 result is different from the one used in minus operations. The negative symbol ¯ resides on the upper shift position of the 2 key and the minus symbol - is on the plus key. To enter a negative value, we must include the negative sign.

```
      ¯7+10
3

      10+¯7
3

      10-¯7
17
```

Think of other examples and try them out to get a still better idea as to how the MCM/70 works in this mode. One thing you will notice is that while the computer is evaluating your input, you are not able to enter any other input until it has completed all of its calculations and has displayed the result. In this way, the computer is able to carry out its computations in an uninterrupted environment. If you do want to stop it at any time, you must press the CTRL (control) key until the lights on the screen quite flashing, and then press the → key (upper shift ←).

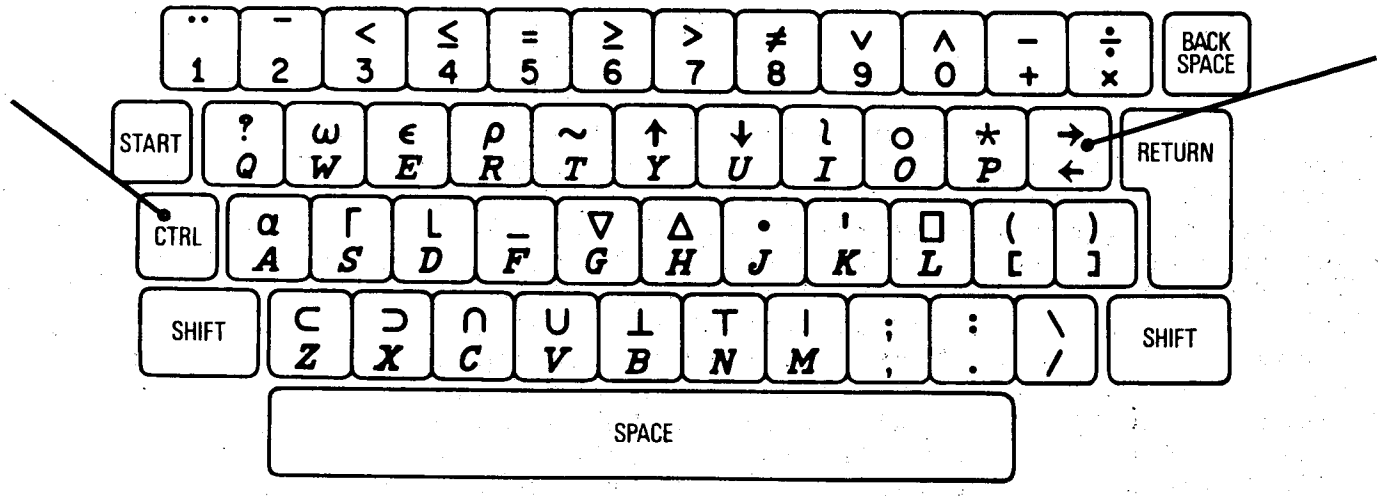


Figure 2.1: MCM/APL KEYBOARD

This will cause the computer to abandon whatever it is doing and wait for your next set of instructions. Consult Appendix A for more information on how to interrupt the computer while it is processing your statements.

Degree of Accuracy

All calculations performed by the MCM/70 are to an accuracy of sixteen decimal places. Usually the system displays only the first five though, with rounding being done on the fifth. This can be adjusted to display anywhere from the first two digits all the way to sixteen. For instance, in the following example, five significant digits are displayed.

```
      8÷3
2.6667
```

This is the preset limit to which the MCM/70 defaults unless otherwise instructed. To reset it to say 15 significant digits, the following expression is used:

```
      ⎕PP←15
```

The ⎕*PP* is called a system variable. It contains the current setting of the "print precision" for the MCM/APL system. Above, it was reset to 15. Now, by typing in the same calculation as above, the result exhibits the type of precision the MCM/70 uses:

```
      8÷3
2.66666666666667
```

All ensuing computer responses will be displayed to this accuracy until the setting is changed to something else. But if the computer is switched off and on again, or if the workspace is cleared of all its contents, the system will automatically set the printing precision back to five.

⎕*PP* is a predefined part of the system. It is just one of several system variables available which allow you to alter the prevailing conditions of the system at any time during the session. A complete list of all the system variables along with their respective meanings are contained in Chapter 21.

Error Reports

Occasionally you will enter statements which cannot be successfully executed by the computer. This could be caused by any number of reasons. But when it does, the computer will tell you exactly which character is causing the problem and what type of problem it is. For instance, if you try to do something that is mathematically impossible or you do not express yourself explicitly enough, the system will tell you so. Here is an example of just such an error:

```
      6÷0
DOMAIN ERROR
6÷0
```

The error report displayed indicates that division by zero is outside the "domain" of acceptable divisors. The first line of the report describes the type of error that has occured. The second line shows where it occured. The cursor is placed on the first character associated with the erroneous operation. In this case it is the 6. There are nine types of errors that can result within the MCM/70. Chapter 20 describes them all, along with reasons for their evocations.

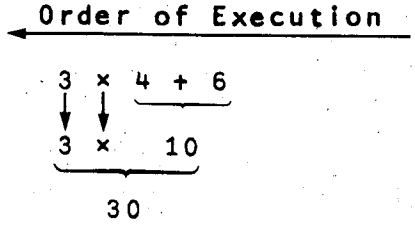
Order of Execution

One of the characteristics of the MCM/APL system (and APL systems in general) that deserves particular attention is the manner in which statements are evaluated. If asked to solve the problem 3 × 4 + 6, we would instinctively multiply the 4 to the 3 to produce a produce of 12, then add the 6 to the 12 and arrive at the final result of 18. Doing the multiplication before the addition is based on a rule taught at the elementary level in school. It states that multiplication and division are <u>always</u> done before addition and subtraction, no matter where they appear in the equation. This is why we got the answer of 18 above. But if the same expression were entered into the MCM/70 we would get a result quite different from this. It would return an answer of 30.

```
      3×4+6
30
```

The reason for this is quite simple. The MCM/70 does not designate priority to any of its operations. It simply evaluates all ex-

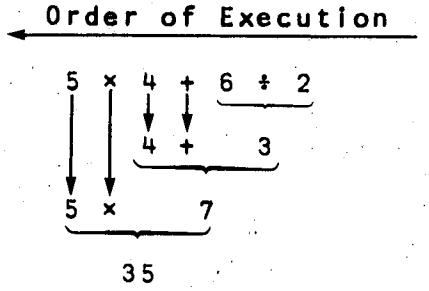
pressions from right to left. Therefore, to get its result of 30, the MCM/70 adds the 6 to the 4 to produce a sum of 10 and then multiplies the 3 by the 10. Here is a breakdown of the operation:



Here is another example:

```
      5×4+6÷2
35
```

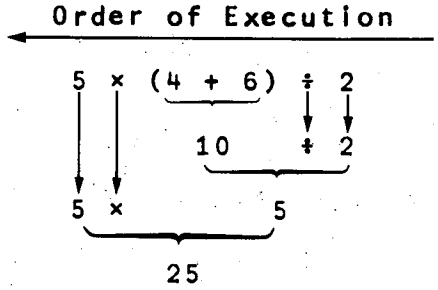
And here is a step by step description of its evaluation:



The right to left order of execution holds true in all cases, no matter what the operation may be. Priorities among the operators that exist in conventional mathematics are completely absent in the MCM/APL system. Thus the input process is simplified by the elimination of the need to worry about any hierarchy. But quite often it is desirable to perform certain operations before others. For instance, in the last example, if we wanted to add the 4 to the 6 and divide their sum by 2, we would probably do it in two steps; the addition on one line and the multiplication and division on another. This could become quite awkward, especially if the statement is fairly lengthy. Therefore, to get around this, parentheses have been employed to establish a priority system during statement execution. Here is how the statement could be executed with the addition being done before the division:

```
      5×(4+6)÷2
25
```

It is evaluated in the following sequence:



Operations within the parentheses are evaluated in the same sequence as those outside — from right to left.

Input Format

The computer accepts your input in the same form as it appears on the screen. This means the format you choose to enter your statements is entirely up to you. You can leave any number of blanks between the numbers and symbols (i.e., + and -) you like. For instance, here is the same operation entered two times, the first with no spaces separating the characters, and the second with several spaces inserted. The result in both cases is the same.

```
      3×4
12

      3     ×    4
12
```

You can also vary the order in which the characters are entered. In either example above, you could have spaced over and typed in the 4, then backspaced and typed in the 3 and the times sign, and still get the same result. Here are the steps involved to do this:

1. 4 cursor spaced over and the 4 typed.

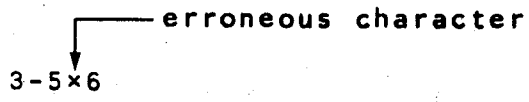
2. 4 cursor backspaced to where the 3 is to be typed.

3. 3×4 the rest of the statement is typed.

Notice the cursor is positioned over the 4 in step 3 resulting in the "negative image" of the 4 being displayed. If the RETURN key is pressed at this point, the result is still 12 because the cursor is not considered part of the input statement. It serves only to indicate where the next character is to appear if one is entered. By displaying the negative image of the character at the cursor position, we can tell what the character is and where the cursor is.

Error Correction

When typing in statements it is quite possible to press the wrong key by accident. A 7 may be typed instead of a 6 or an *A* instead of an *S*. When this happens, the computer's evaluation will result in either an error message, or worse still, the wrong answer. But usually errors are noticed before returning control back to the computer for its evaluation. Instead of pressing RETURN and letting the computer act on the input, we simply "backspace" the cursor to the character in error and "replace" it with the correct one. Here is an example of an expression where 6 is supposed to be added to 5 but the "times" key is pressed by mistake.



Before hitting the RETURN key, the error is noticed. To correct it, the BKSP (Backspace) key is pressed two times to position the cursor over the erroneous symbol and the plus sign is typed. The completed operation looks like this:

3-5+6

Now the RETURN key may be pressed and the computer will give you the desired result. Here are the steps involved in correcting the error:

1. 3-5×6 (original statement)
   cursor here

2. 3-5×6 Backspace key pressed until cursor here

3. 3-5+6 Plus sign typed, then RETURN key pressed

4. -8 (computer's response)

N.B. When the cursor is backspaced over an existing character, the illuminated dots representing the character are turned off and those within the 5 by 7 dot matrix which do not represent the character are turned on, thus displaying the negative image of the character.

Character Insertion and Deletion

Two other kinds of typing errors that can occur when entering input are the accidental omission of a character(s) and the accidental insertion of undesired character(s). For instance, suppose the statement (2×3)+6 was mistakenly entered (2×3+6. Assuming the RETURN key has not been pressed, here are the steps required to insert the missing character, allowing the line to be executed properly:

1. Backspace the cursor until it is positioned over the plus sign.

2. Press the CTRL key.

3. While holding the CTRL key down, press the SPACE bar once. This causes the characters +6 to shift one position to the right, leaving a blank position where the cursor is. The line now looks like this:

   (2×3 +6

4. Key in the missing right parenthesis and press RETURN.

   (2×3)+6

Character deletion is handled in the same manner, except instead of using the SPACE bar to do the eliminating, the BKSP key is used.

Summary

This chapter has given us the information needed to start up and operate the MCM/70. We have seen that this task is just as easy as the start up procedures of either a typewriter or a desk top calculator. The "mode" of operation is referred to as calculator or execution mode. We saw also that when operating in this mode, all calculations were performed to an accuracy of sixteen significant digits, the first five of which were displayed on the screen.

We saw that errors can occur and, when they do, the computer lets us know by stating the type and location of the problem encountered. If we are able to spot the errors before we return control back to the computer, we can easily replace the erroneous parts with the proper corrections, thus insuring a successful execution. When the computer does accept this input of evaluation, it takes it exactly as it appears on the screen, regardless of the sequence in which it was entered.

Some of the characteristics we have seen of the MCM/APL system are similar to those of many desk calculators, while still others go beyond this level of sophistication to offer an extremely powerful and flexible means of evaluating mathematical expressions. Chapter 3 describes even more features of this system that tend to take it out of the calculator family and draw it closer to that of a general purpose microcomputer.

Practice Exercises

1. Evaluate the following using the MCM/70:

(a)	7+5	(b)	6×18	(c)	17-3.5
(d)	12÷8	(e)	9.75+3	(f)	3-¯16.2

(g) 6×¯.2 (h) 10+10+10 (i) 10×10÷10

(j) 4×6-2 (k) 3.2÷6.25×10 (l) 7+7-7×7÷7

2. Write an APL expression to subtract negative four from positive eleven.

3. Write an APL expression to multiply six by seven and divide their product by three.

4. Write an APL expression to sum up the numbers 1, 2, 10, 5, 3, and 4.

5. Everyday a farmer collects, packages, and sells all the eggs his hens can lay. How many eggs does this involve if his daily collections for one week are 22, 23, 26, 23, 22, 27, and 25 eggs?

6. When packaged, how many dozens does this represent?

7. If he gets 83¢ a dozen, what is his weekly revenue?

Chapter 3: MORE OF THE SYSTEM'S CHARACTERISTICS

The previous chapter demonstrated the ease with which mathematical statements can be entered into the computer for evaluation. It also illustrated how the computer may be halted while in the process of carrying out its execution, and how to edit portions of an input statement before control is returned to the computer. It showed how the MCM/70 can be operated with the simplicity of a calculator, while at the same time being utilized in ways far beyond the capabilities possessed by most calculators. This chapter further expands on this by introducing variables, literals, and data structures known as arrays, which can be defined within the MCM/APL system.

Variables

In all of our examples so far the computer's responses have been displayed on the screen. This is the one chance we have to see them, as they are erased from the computer's memory as soon as the RETURN key is pressed. This characteristic aids in keeping the memory space free of unnecessary items, yet at the same time it forces us to remember those results that may be needed in future calculations. Instead of resorting to this method of data retention, we can instruct the computer to store its answers in its own memory. To do this, we must give the storage location a name. Here is an example of two numbers being added together and their total being stored in memory under an arbitrarily chosen name:

```
      A←8+7
```

Notice the result 15 is not displayed. Only the cursor symbol appears. To display the contents of *A*, we simply type in the letter.

```
      A
15
```

We can also use *A* in calculations just as we would a value.

```
      A×3
45
```

```
      6↑A-13
3
```

A is called a variable, which means it can be reassigned different data depending on the user's instructions.

Several variables can reside in the workspace at any one time and all of them can be used in calculations.

```
      B←10
■
      A×B
150
      C←A×B
■
      C
150
```

To list the names of the variables presently in the workspace, the following system function is used:

```
      ⎕VA
```

to which the response in this case is

```
A
B
C
```

If, at any time, we wish to erase any of these variables from memory, we use the system function ⎕*EX* and enclose the name(s) of the variable(s) to be erased, in quotes. Here is the variable *A* being erased from memory:

```
      ⎕EX 'A'
■
```

If we again execute ⎕*VA*, we get the following:

```
      ⎕VA
B
C
```

If we now try to use *A* in any way, the following occurs:

```
      A×10
VALUE ERROR
A×10
```

(To find out more about system functions, see Chapter 21.)

Valid Variable Names

The variable names above were chosen on an arbitrary basis. When defining a variable, the name selected must fit the following criteria.

1. It must begin with a letter of the alphabet or the delta symbol Δ.
2. It may consist of the letters *A* to *Z*, the delta, and the numbers from 0 to 9 as long as the latter are not the first characters in the name.
3. It may contain any number of characters, but the first three must be unique as the system looks at the first three only.
4. It cannot contain any spaces or blank characters.
5. It cannot be the same name as any programs that may already exist in the workspace.

Here are some typical valid and invalid variable names:

Valid	Invalid
A	1*K*
TOTAL	*S-M*
C12	*X Y*
DΔR	*AαD*
XXX	*G/L*
ST	∇*N*
Δ	□*TT*

Since there is a wide range of acceptable variable names available, you should try to make each one meaningful, so that you can easily relate them to the data they represent. For instance, it makes more sense to call the accumulation of a set of numbers something like *SUM* or *TOT* rather than giving it a name such as *A* or *B*. This way, as you define more and more variables in your workspace, it is much easier to remember which names represent which values.

Literals

All references to data, up to this point, have been to numbers. In each instance, calculations were performed on input "data" and the computer responded with its output "data" results. But the term "data" does not refer just to numbers; it applies to textual material as well. For instance, the word "nature" is classified as data. If we wished to enter this data into the computer's memory, we would do so in the following manner:

```
      S←'NATURE'
```

Just as numeric data is stored, so too is textual data. By surrounding it in single quotation marks, the word "nature" is interpreted by the system as being literal as opposed to numeric. The quotes are not part of the data but serve only to categorize it as being literal.

```
      S
NATURE
```

In cases where a quote is supposed to be part of the text, a double quote is used to represent this.

```
      Q←'ROBBIE''S'
```

```
      Q
ROBBIE'S
```

If a quote is missing or there is one too many entered, the system responds accordingly.

```
      P←'DAVID'S'
VALUE ERROR
P←'AVID'S'
```

When the literal is a number, it takes on an entirely different property than that of its numeric counterpart.

```
      '2'+2
DOMAIN ERROR
'2'+2
```

In the above example, the character '2' is distinguishable by the system from the value 2.

Literals are used quite often during a session. They can serve as headings for other numeric and literal output and they can also be used as both input and output statements in programs, as we will see later.

Primitive Functions

So far we have been referring to the symbols + - × ÷ as operators. More specifically, they are predefined symbols which the computer recognizes as representing various primitive functions. The word "primitive" means they already exist within the MCM/APL system and need not be defined by the user. The word "function" indicates that each symbol performs a specific task. The plus sign + represents addition; the minus sign - means subtraction, and the times × and divide ÷ signs stand for multiplication and division respectively. The routines or functions of each symbol are fixed, but the data each is supplied is not. For instance, to add the numbers 3 and 4 together, we would type:

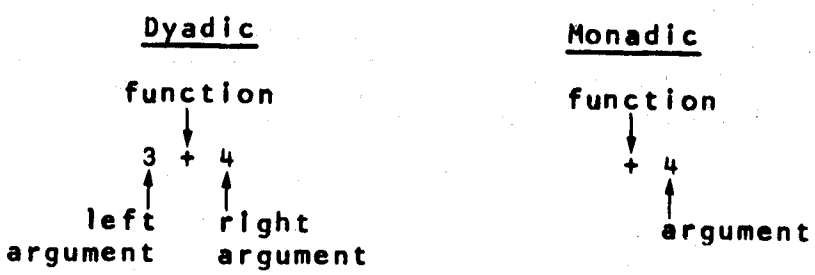
```
      3+4
```

The two numbers could have been 6.2 and ¯7 or 10000 and .001, and the function would perform the same, adding the one number to the other. These numbers are said to be arguments of the function. Together, with the function symbol, they form a statement or expression. The

expression 3+4 above is comprised of two arguments and one function symbol. Its task is to add the value 4 to the value 3.

Function Syntax

There are two kinds of primitive functions in MCM/APL. One is called dyadic and the other is called monadic. The expression 3+4 is a dyadic function. This means it has two arguments, one on each side of the function symbol. A monadic function has only one argument. The argument in this case always appears to the right of the function symbol.



Most functions can be used both monadically and dyadically. Below are a few examples of monadic uses of some of the functions we have seen so far.

```
      -7
¯7
```

When the minus sign is used monadically, it changes the sign of its argument.

```
      -¯7
7
```

If the argument is positive, the result becomes its negative equivalent, and vice versa. This can be compared to the dyadic use of the minus sign when the left argument is zero.

```
      0-¯7
7
```

Using the divide sign monadically produces the reciprocal of its argument.

```
      ÷4
.25
```

Finding the reciprocal of a value is the same as dividing that value into one.

```
      1÷4
.25
```

Arrays

The arguments of a function are not only categorized by their numeric or literal properties, but also according to their rank and dimension. So far, all our examples have involved one number performing an operation on another. For instance, 3+4 resulted in the 4 being added to the 3. Each of these arguments is referred to by the computer as being a scalar, or a single value. But often our calculations involve several numbers. They could be figures representing various inventory quantities or observations from an experiment. We may want to tally up our current inventory levels or measure the frequency distribution of our experiment findings. To do either of these things requires the input of many numbers. When this happens, the numbers are no longer referred to as scalars or individual quantities, but are collectively called an array of numbers.

Arrays can take on many sizes and shapes and can vary from a single item all the way to several hundred. But basically all arrays fall into four categories. They are:

1. Scalar
2. Vector
3 Matrix
4. Complex array

Vectors

We saw that a single datum is called a scalar. A vector can be thought of as several scalars placed in a line to form a list or chain. For instance, the following series of numbers is a vector:

```
      5 6 2 3 9
```

So too is the literal

```
      'RHINOCEROS'
```

You will remember in the previous section on literals, we saw how to classify the words *NATURE* and *ROBBIE'S* as literals. But what you may not have noticed was that we actually defined literal vectors.

Each scalar within a vector is said to be a component or element of the vector. The numeric vector 5 6 2 3 9 has five components and the literal vector *RHINOCEROS* has ten components. To create a numeric vector, we simply type in all the components, each separated from the others by at least one space.

```
      OBS←2.6 ¯7 2 4 1 0
```

■

```
      OBS
2.6 ¯7 2 4 1 0
```

You will see in Chapter 6 a primitive function for generating vectors which is useful in certain applications where particular numbering series are desired.

Matrices

Apart from having just vectors, or strings of numbers and characters, we can also have matrices. A matrix is any array which is arranged into rows and columns. In figure 3.1, the block of numbers forms a matrix consisting of 4 rows and 5 columns.

Dow-Jones U.S. Exchange Index

	30 Ind.	20 Rails	15 Util.	65 Stocks
Mar. 11	888.45	197.08	93.72	281.82
Mar. 8	878.05	195.45	93.69	279.27
Mar. 7	869.06	194.83	93.56	277.30
Mar. 6	879.85	197.46	93.85	280.43
Mar. 5	872.42	197.62	93.53	278.96
Mo. Ago	803.90	180.01	93.37	259.54
Yr. Ago	969.75	194.13	110.46	303.53
1973-74 High	1,051.70	228.10	120.72	334.08
1973-74 Low	788.31	151.97	84.42	247.67

Figure 3.1: Dow-Jones Index

The statistics for the industrials reside in column 1, those for rails are in column 2 and so on, while the closing figures for the various indices on March 11 are in row 1, and those for March 8 are in row 2.

This next example in figure 3.2 is an illustration of a matrix with 8 rows and 7 columns.

HOCKEY RECORD

Eastern Division

	G	W	L	T	F	A	P
Boston	78	52	17	9	347	215	113
Montreal	78	45	24	9	293	240	99
N.Y. Rangers	78	40	24	14	300	251	94
Toronto	78	35	27	16	274	230	86
Buffalo	78	32	34	12	242	250	76
Detroit	78	29	39	10	255	319	68
Vancouver	78	24	43	11	224	296	59
N.Y. Islanders	78	19	41	18	182	247	56

Figure 3.2: National Hockey League Standings

When matrices are defined in the computer, they appear in the same format as in the two illustrations above. Here is an example involving a variable called *NHL* whose contents are being displayed:

```
      NHL
78    52    17     9    347    215    113
78    45    24     9    293    240     99
78    40    24    14    300    251     94
78    35    27    16    274    230     86
78    32    34    12    242    250     76
78    29    39    10    255    319     68
78    24    43    11    224    296     59
78    19    41    18    182    247     56
```

As you can see, the contents of *NHL* appear in the same type of configuration as those in figure 3.3. When the system is asked to display a matrix, it does so a row at a time. After displaying the first row, it waits for you to press the RETURN key before displaying the next row. It indicates this by also displaying a small rectangle of illuminated dots at the extreme right side of the screen. There is a way to have the computer automatically display each row after a given amount of time, as you will see in Chapter 12.

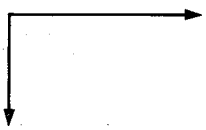
Dimension And Rank

Both vectors and matrices are said to have both magnitude and direction. Their magnitude is depicted by the values they represent and their direction is symbolized by their structure. Because a vector is a single string of numbers or characters, it is defined as having one direction.

Vector:

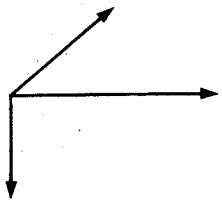
Matrices on the other hand have 2 directions.

Matrix:



You can have arrays with 3 directions, which are called 3-dimensional, or complex arrays.

3-dimensional Array:



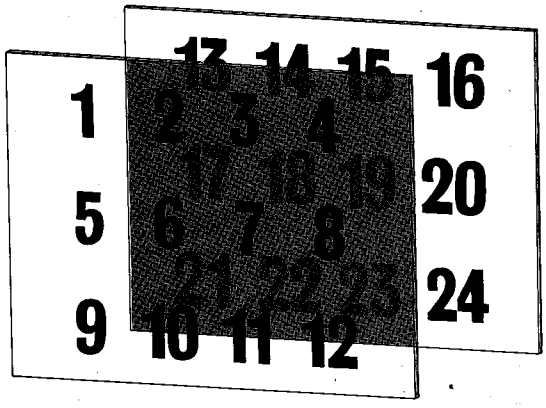
The number of directions can be from 1 up to a possible 32. The number of directions an array has is referred to as its rank. A vector has a rank of 1, a matrix has a rank of 2, and a 3-dimensional array has a rank of 3. A scalar, which is also classified as an array, has no rank at all, as it has no direction. Its single property is magnitude only.

Array Type	Rank
Scalar	0
Vector	1
Matrix	2
Complex array	3-32

The direction of an array is called its dimension. This way not only is the direction of the array accounted for, but also the extent to which it goes. To determine the dimensions of an array, we use the dimension of function, which is discussed in Chapter 11. Dimensions and ranks of arrays are referred to later when we discuss indexing individual array components. Their ranks and dimensions must be known before indexing can occur.

Complex Arrays

Any array whose rank exceeds 2 is called a multidimensional or complex array. Although an array that has any more than 3 dimensions is difficult to envision, it can have up to 32 dimensions. Like a matrix, a multidimensional array has rows and columns, plus additional dimensions referred to as planes. Here is an example of a 3-dimensional array which has 2 planes, each with 3 rows and 4 columns and contains the numbers 1 to 24.



Obviously the display screen cannot show any more than one row at a time, but for clarification sake, all matrices and multidimensional arrays will appear in 2-dimensional form. The above (2×3×4) array takes on the following form:

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

13   14   15   16 }
17   18   19   20 } plane 2
21   22   23   24 }
```

Arrays of various ranks appearing in the following examples will be shown in a way to facilitate you in recognizing their dimensions. Each plane will be separated by a blank line.

Summary

This chapter has introduced you to variables and the various properties your data can possess. You saw that data can be either literal or numeric, and that it need not be just individual characters and numbers. It can be organized into lists (vectors), tables (matrices), cubes (3-dimensional arrays), and even structures of greater

complexity. You saw how vectors were created, but you did not see how arrays of greater ranks were built. This has been left for a later chapter as it involves a little more knowledge of the APL language than has been covered to this point. But, by knowing how vectors are formed, you may try experimenting, using them as arguments to the primitives you have learned. Remember, the functions can be used both monadically and dyadically. Of the four primitives you have seen, (+ - × ÷), only two were illustrated in the section dealing with the "monadic" concept. See if you can determine the purpose of the times and plus functions when they are used monadically. Both are discussed in the next chapter along with several new functions.

Practice Exercises

1. Evaluate the following expression, given

```
A←6
B←¯10
C←.5
```

(a)	`A×7`	(b)	`420÷A×14`	(c)	`A÷0`
(d)	`A×C`	(e)	`A÷C+B`	(f)	`A÷C+B÷20`
(g)	`A+B+C`	(h)	`C-B`	(i)	`-B`
(j)	`--B`	(k)	`6--B`	(l)	`¯C`
(m)	`A×B×C`	(n)	`30+A×B×C`	(o)	`B÷÷C`

2. Enter the following statement and then display the contents of all three variables.

```
A←B←C←10
```

3. Which of the following variable names are invalid?

(a)	`A`	(b)	`ΔΔΔ`	(c)	`F_D`
(d)	`MAXIMUM`	(e)	`6F`	(f)	`Z∇Y`
(g)	`PAL`	(h)	`Δ2Δ`	(i)	`Y2684`

4. Assign the numeric vector 1 2 3 4 5 to a variable called *VEC* and assign the literal vector *VECTOR IS* to the variable *HD*.

5. Using the vector *VEC* created in number 4, execute the following statements:

   (a) *VEC*+6 (b) *VEC*-10

   (c) 2×*VEC* (d) ¯3+*VEC*

6. Using the formula

   $$\text{Centigrade} = \left(\frac{5}{9} \times \text{Fahrenheit} - 32\right)$$

   convert 67 degrees fahrenheit into centigrade.

7. What is the total area of a 15 by 12 foot room?

   Area = length × width

Chapter 4: SCALAR FUNCTIONS

The previous chapters have demonstrated the ease with which arithmetic calculations can be performed on the MCM/70. The functions performed were addition, subtraction, multiplication and division. This chapter introduces a few more functions that go beyond the arithmetic level of mathematics to that of algebra. These functions deal with exponents and logarithms and magnitudinal comparisons. Although their descriptions make them sound complicated, they are extremely easy to use and understand. This chapter also employs vectors in many of its examples to illustrate that the functions discussed here can accept arrays of all sizes and shapes as their arguments, just as they do with scalars.

Definition of a Scalar Function

All the primitive functions in MCM/APL are divided into two categories. One is called scalar functions and the other is called mixed functions. The basic differences between the two are the scalar functions operate on a "one-for-one" basis and the sizes and shapes of their results are directly related to those of their arguments.

As stated earlier, a scalar is defined as being a single datum. The value 67 and the letter *C* are both scalars. The term "scalar function" also refers to the same "unit" type definition, but in a slightly different context. When the addition function is used dyadically with two scalars as arguments, it computes their total by adding one to the other.

```
      3+4
7
```

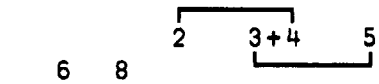
But when the arguments are two vectors, the result looks quite different.

```
      2 3+4 5
6  8
```

Since it works on a "one-for-one" basis, this function performs the addition by the following method:

```
  2   3
+4    5
------
  6   8
```

Or you could think of it this way:



The first element in the right argument is added to the first element in the left; and the second element in the left argument is added to the second in the left. When used dyadically, the addition function is called a scalar dyadic function. Here are a few more examples of scalar dyadic functions performing calculations on various vector arguments:

```
      16 25 64÷2 5 16
8  5  4

      4 7 10 2-2 10 7 4
2  ¯3  3  ¯2

      2 4×100 100
200  400
```

When all the values in one of the arguments are identical, we need to key in only one of the values. This last example could also have been expressed in the following manner:

```
      2 4×100
200  400
```

The system automatically repeats the value 100 until both arguments are of equal lengths. But this only happens if one of the arguments contains one number. Look at what happens when both arguments have more than one value each, but are of unequal length.

```
      3 7×5 9 8
LENGTH ERROR
      7×5 9 8
```

The system is unable to determine which number is supposed to be multiplied by which other number. So, instead of making an assumption about the intention, it terminates with the above message to indicate its problem.

When these functions are used monadically, they are referred to as scalar monadic functions.

```
      ÷5 4 2
.2  .25  .5

      -6 ¯4.2 ¯10
¯6  4.2  10
```

One of the prominent characteristics of scalar functions is the direct relationship between their arguments and results. Apart from correct, the results take on the same sizes and shapes as their respective arguments. Scalar arguments produce scalar results, vector arguments give vector results and so on for all arrays. If one argument is a scalar, the result assumes the size and shape of the other argument.

Result		Lt. Arg.		Rt. Arg.
scalar	←	scalar	+	scalar
vector	←	vector	+	scalar
vector	←	scalar	+	vector
vector	←	vector	+	vector
matrix	←	matrix	+	scalar
matrix	←	scalar	+	matrix
		.		
		.		
		.		
		etc.		

In our example of 4 5×10 6 the two 2-element arguments produced a 2-element result. Here are some more illustrations to further show this correlation:

```
      10+12
22

      1 2 3×4 5 6
4  10  18

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

There are several such scalar functions in the MCM/APL system, many of which perform monadically and dyadically. The functions we have seen so far (+ - × ÷) have all been scalar functions. The rest, which include the symbols

* ● ⌈ ⌊ | ! ○ < ≤ = ≥ > ≠ ∨ ∧ ⍲ ⍱ ~ and ?

are discussed in this, and other chapters. This chapter will describe the set of functions characterized by the symbols * ● ⌈ and ⌊. But before we start to discuss them, there are two scalar functions whose dyadic uses have been explained but we have still to cover their monadic properties.

Signum

When used monadically, the multiplication function indicates the sign of its argument.

```
      ×4 ¯4
1  ¯1

      ×67 14 0 ¯7.2
1  1  0  ¯1
```

If the value is positive, the system indicates this by returning a result of 1. If it is zero or a negative value, the response is 0 or ¯1 respectively.

Identity

Just to round out the series, the plus function can also be used monadically.

```
      +7 2.3 ¯6
7 2.3 ¯6
```

The system assumes a zero to be the left argument. In the case where one argument is a scalar and the other is not, you will recall that the scalar is expanded in dimension and rank until it is of the same structure as the non-scalar argument.

```
      4÷8 2 16
.5 2 .25
```

The dimension and rank of the result of arguments like this are always equal to that of the non-scalar argument.

Exponentiation

When writing an expression to illustrate that a number is to be raised to a power of some other number, we typically symbolize this operation as X^Y, where X is to be raised to the power Y. Since this can be a rather confusing way of expressing it and, since power operations are very similar to add, subtract, multiply and divide type operations, the power function is written as 2*3. For instance, if we want to raise the value 2 to the power of 3, here is how we would do it:

```
      2*3
8
```

And if we want to raise 2 to several powers,

```
      2*2 3 4 5
4 8 16 32
```

or raise several values to one or more powers, we would express it like this:

```
      2 3 4 5*2
4 9 16 25
```

```
      2 3 4*4 3 2
16  27  16
```

Apart from finding the squares and cubes of numbers, the power function is also able to find their roots. For example, in order to find the square root of a number, you simply raise it to half its power. Here is the expression to find the square root of 16:

```
      16*.5
4
```

and here is the cube root of 125:

```
      125*1÷3
5
```

or just

```
      125*÷3
5
```

since the monadic divide function assumes a 1 to be the left argument.

As we get into very large numbers, the system alters its output format slightly to eliminate the need for displaying all the repeating zeros that can occur. For instance, if we raised the number 20 to the power of 9 the operation would look like this:

```
      20*9
5.12E11
```

which really means 51,200,000,000 is the correct answer. The letter *E* in the result is an exponential notation meaning "times 10 to the power of."

We can also use *E* format for entering data.

```
      10E1+2
102

      10E3+7E4
80000

      5.12E11*÷9
20
```

If the result happened to be very small, the same kind of notation would be displayed.

```
      .01*6
1E¯12
```

Of course there are limits to the range of numbers that can be created, and, when we try to exceed this limit, the system will tell us so.

```
      100*100
RANGE ERROR
100*100
```

The acceptable range is anywhere from 7.237E75 to ¯7.237E75.

Exponential

If the power function is used monadically the system assumes the left argument to be *e*, the base of the system of natural logarithms, or 2.7182818.....

```
      *1
2.7183

      *.2 ¯10
1.2214  4.5E¯5
```

Logarithms

The logarithm function is closely connected to the exponential function. To raise a value to a certain power, we write the expression as *X***Y* with *X* representing the value to be raised and *Y* being the exponent. The logarithm function is written as *X*●*Y* and means "to what power must *X* be raised before it is equal to *Y*?" The result is the exponent.

```
      2●8
3
```

The above expression is read as "log 8, base 2" and traditionally written as "$\log_2 8$."

Here are some more examples:

```
      9●81
2

      10●10 100 1000
1  2  3
```

The symbol ● does not appear on the keyboard. It is a combination of the characters * and o (upper shift *O*). The procedure for combining the two is either * backspace o or o backspace *. The sequence is not important.

Natural Logarithm

Monadically the logarithm symbol finds the "natural log" of its argument. It uses the base of the natural logarithm, (*e* or 2.71828....) as its left argument.

```
      ●10
2.3026
```

It is usually written as "$\log_e 10$" and read as "log 10, base *e*." Here are of few more examples:

```
      ●100 200
4.6052  5.2983

      ●2.4 50
.87547  3.912
```

Since the natural log function is the inverse of the exponential function, they negate each other. Here is a way to prove it:

```
      *●2.4 50
2.4  50
```

Maximum

Imagine you are a teacher who has just given his students a fifty question "true-false" test. A perfect paper is worth fifty marks, with each right answer worth one mark and each wrong answer costing two marks. This means if someone got every question wrong, he would end up with a mark of ¯50, which is not too impressive looking on report cards. So you establish a limit of zero as being the lowest possible mark anyone can receive. After grading the first ten tests and adding up their final marks, you discover them to be the following:

```
23  42  6  ¯10  15  ¯6  25  33  17  ¯2
```

Since you plan on storing these marks in your student record file, you would like to enter the marks, as they are, and have the computer handle the ones that fall below the zero limit. One way this could be done is by means of the maximum function. Its role as a dyadic function is to determine which of its arguments is the greater. The symbol used to represent this function is the ⌈ which is located on the *S* key.

```
      4⌈6
6

      3⌈4 3.1 2
4  3.1  3
```

And so to solve our hypothetical problem, the function would look like this:

```
      0⌈23 42 6 ¯10 15 ¯6 25 33 17 ¯2
23  42  6  0  15  0  25  33  17  0
```

thus simplifying our marking task and also making some of the students' parents a little less sad.

Ceiling

In its monadic syntax, the ⌈ function "rounds up" its argument to the closest integer. For example,

```
      ⌈8.8
9
```

and

```
      ⌈7.2 ¯2.3 6
8  ¯2  6
```

In the case of ¯2.3 the result of ¯2 is actually higher in value. If the argument is already an integer, its result remains the same.

Minimum

Just as the maximum function determined the greater of its two arguments, the minimum function ⌊ (upper shift *D*) calculates the lesser of the two.

```
      4 6⌊7 3
4  3
```

```
      ¯2.6⌊¯2.7
¯2.7
```

Floor

The counterpart of the monadic ⌈ , which rounds its argument up to the closest integer(s), is the monadic ⌊ . It rounds its argument down to the closest integer(s).

```
      ⌊4.2 6.7 ¯2.3 20
4  6  ¯3  20
```

Instances arise however, where instead of just rounding up or rounding down, it would be preferable to round to the closest integer, which neither the ceiling nor the floor functions do. For instance, when rounding a number such as 4.2 to its closest integer, the ⌊ function could be used to return a result of 4. But what about a number like 6.7? We can not do both of these operations by using the same function. That is, we cannot unless we add .5 to the argument first. Here is the operation performed without the .5 being added:

```
      ⌊6.7 4.2
6  4

      ⌈6.7 4.2
7  5
```

and here is the floor function again, but with the .5 added first:

```
      ⌊.5+6.7 4.2
7  4
```

and the ceiling function with the .5 subtracted first:

```
      ⌈6.7 4.2-.5
7  4
```

It appears, that by either adding or subtracting the .5 and using the appropriate function, the same results can be obtained. Not quite however. If one of the arguments' values happens to end in .5 already, the results will be different.

```
      ⌊10.5+.5
11

      ⌈10.5-.5
10
```

Therefore, depending on whether we want the ".5" values rounded up or down, dictates the method we choose.

Practice Exercises

1. Using the MCM/70, evaluate the following:

   (a) `3 6+2`

   (b) `4 7+10 12`

   (c) `8 2 5-4 1 3`

   (d) `2 3+10×4 5`

   (e) `5⌊3 4 5 6 7`

   (f) `2 3*2`

   (g) `2 3*3 2`

   (h) `2*0 1 2 3 4`

   (i) `*⍟10`

   (j) `¯10*.3`

   (k) `2⍟512`

   (l) `3 4⍟81 1024`

   (m) `×¯.01 5 0`

   (n) `¯.56⌈¯.57`

   (o) `(3+2)⌈4`

   (p) `6E34⌊6E35`

   (q) `6⌊25*.5`

   (r) `(2*3)⌈(3*2)`

2. Write APL expressions equivalent to the following:

   (a) 3^2

   (b) $3^3 + 2^4$

   (c) 4^{-2}

   (d) $\sqrt{25}$

   (e) $X^2 + Y^2$

   (f) $3X^2+2X-1$

3. Find the squares, cubes and fourth powers of the numbers 4 and 5.

4. Find the square, cube and fourth roots of the numbers 4 and 5.

5. What are the natural logarithms of the numbers 2, 10, and 27.5?

6. What are the logarithms of 27 and 243 to the base 3?

7. One store is selling oranges at "2 for 13 cents" and another at "69 cents a dozen." Which is the better buy and what would the difference in price be if you bought 2 dozen?

Chapter 5: MORE SCALAR FUNCTIONS

This chapter introduces a few more functions associated with algebra, plus a set pertaining to trigonometry. The algebra functions calculate the remainders of division operations, absolute values, factorials and combinations; and the trigonometric ones deal with sines, cosines, tangents, etc. One more function included here is the monadic random, or, as it is often called, the roll function. It does not fall into either one of these two categories, but its output may provide good input to others.

Residue

The residue function, employing the symbol | (upper shift *M*), does just what its name implies. It produces the residue or remainder left after one number is divided by another. For instance, 8 divided by 3 leaves a remainder of 2.

```
      3|8
2
```

Notice the divisor is the left argument and the dividend is the right.

```
      4|7 8 9 10
3 0 1 2

      2.6 3.2|6.7
1.5 .3
```

If we have to determine what the nonintegral part of a number is, this function can be used in the following manner:

```
      1|10.5 6.234
.5 .234
```

If the left argument is negative, so too is the result.

```
      ¯2|3 ¯5 ¯4
¯1  ¯2  0
```

But if the left argument is positive and the right argument is negative the result is quite different.

```
      4|¯11
1
```

What happens here is that the 4 is added to the ¯11 as many times as is required to make the result a positive value. And it is this value that gets printed. Above, the 4 is added to ¯11 to give an initial result of ¯7. This ¯7 is then increased by 4 to give ¯3. But still the result is negative. So the 4 is again added to the ¯3 to give the final result of 1.

Absolute Value

Monadically, the | function is used to produce the absolute value of its argument.

```
      |¯7 0 6 ¯2.3
7  0  6  2.3
```

Conventional mathematics places a vertical bar on both sides of its argument in the form $|X|$. But APL eliminates the right bar to make it more consistent with the rest of the notation.

Factorial

In how many different ways can you arrange 4 items? The answer is 24 different ways. It is determined by the mathematical equation 4! which means "4 times 3 times 2 times 1." In APL, the equation is !4 with the factorial function symbol ! being a combination of ' (upper shift K) and the period or decimal point.

```
      !4
24

      !5
120

      !6 3
720   6
```

Knowing how to use the factorial function means we can also do permutations. When calculating the number of permutations of "n" different things, taking "r" at a time, without repetition, we use the formula

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

which could also be expressed as

$$(n!)\div(n-r)!$$

The APL equivalent to this is

```
(!N)÷!N-R
```

If we were given the problem of finding out how many 6 letter words could be formed from the letters of the word "computer", we would solve it in the following manner:

```
      N←8     (there are 8 letters in the word "computer".)

      R←6

      (!N)÷!N-R
20160
```

Obviously most of these 20160 "words" are not part of the English language as we know it though.

Combinations

Now that we have found out how permutations are handled, the next step is calculating combinations. The fundamental difference between a permutation and a combination is that, in a permutation, order is taken into account, while in a combination, it is not. The equation used for calculating the number of combinations of ways in which objects can be selected from a group, without regard to their order is:

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

which could be written in APL as:

```
(!N)÷(!R)×!N-R
```

But, by employing the ! function dyadically, the same statement can be expressed as

```
R!N
```

For instance, to solve the problem, "how many ways can 2 marbles be selected from a population of 6?", the algorithm would look like this:

```
      2!6
```

and the answer would be 15.

Here are both expressions used:

```
      (!6)÷(!2)×!6-2
15

      2!6
15
```

Clearly, the second method takes much less of the computer's time to evaluate than the first.

Trigonometric Functions

The upper shift *O* symbol ○ has some interesting characteristics. In its dyadic form, the large circular symbol performs various trigonometric functions, depending on the value of its left argument. Here is a table of all the trigonometric operations that are possible with this symbol:

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

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

What is the sine of 3 radians?

```
      1○3
.14112
```

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

```
      (((1○2)*2)+(2○2)*2)
1
```

Pi Times

When used monadically, in the form ○*A*, the expression means "Pi times *A*". (Pi meaning π or 3.14159...)

```
      ○1
3.1416
```

Although this function expects its right argument to be expressed in terms of radians, most applications which use it deal in degrees. Therefore the conversion equations for degrees to radians and vice versa become constant necessities. With the information we know now about APL, we should be able to do these conversions ourselves if given the conversion formulae. The one for changing degrees to radians is:

$$1 \text{ radian} = \frac{180 \text{ degrees}}{\text{Pi}}$$

To find the number of radians in 30 degrees, the solution could be expressed in APL as:

```
      (30×○1)÷180
.5236
```

or just

```
      ○30÷180
.5236
```

See if you can convert a radian value to its degree equivalent.

Monadic Random (Roll)

The monadic random, or roll function ? (upper shift *Q*) differs somewhat from all the other scalar functions we have seen so far. For instance, here it is being executed three times, each time with the same argument, but the results it produces are all different.

```
      ?6
3

      ?6
4

      ?6
1
```

Can you guess what its function is? If you think its a number series generator, you are wrong. Its purpose is to generate numbers alright, but in a totally random sequence. The above three examples asked the system to select an integer between 1 and 6. In this particular case, the 6 could represent the six different sides of a die, and the roll function could be thought of as the simulation of the rolling of this die to see which side comes up. This logically leads us to a dice game involving two dice. All we have to do to get this is to type another 6 into the argument.

```
      ?6 6
2  6

      ?6 6
5  5

      ?6 6
3  4
```

and so on.

Practice Exercises

1. Using the MCM/70, evaluate the following:

   (a) `2|8` (b) `2.7|8.6`

   (c) `|¯7 26.2 0` (d) `2|¯7`

   (e) `!4` (f) `!3 2 1`

   (g) `3!9 8 7` (h) `o1 2 3`

(i) 1000 .5 1 1.5 (j) ¯2 ¯101

(k) ?10 (l) ??100

2. Three hundred and thirty-seven apples were picked from a tree and sorted into dozen quantities. How many apples were left over?

3. Using the formulae

Circumference = πR

Area = πR^2

find the circumference and area of a circle having a radius of 3 inches.

4. Prove the following:

$\sin^2 30^\circ + \cos^2 30^\circ = 1$

$\sin 60^\circ \cos 30^\circ + \cos 60^\circ \sin 30^\circ = 1$

$$\frac{\tan 180^\circ - \tan 120^\circ}{1 + \tan 180^\circ \tan 120^\circ} = \tan 60^\circ$$

5. What is the APL expression to find the cosecant of .5 radians, knowing that the cosecant is the inverse of sine.

6. Use the monadic random function to simulate the flipping of a coin 5 times.

Chapter 6: RELATIONAL AND LOGICAL FUNCTIONS

There are two sets of scalar functions that deal in relational and logical comparisons of data. They are used to answer such questions as, "Is *A* greater than *B*?" and "are both of these values equal to zero?". In every case, their responses are of a "yes-no" nature. For example, the two questions above could be answered with "yes, *A* is greater than *B*" and "no, they are not both equal to zero". Because the data being compared is typically numeric, it is only fitting that the results be numeric also. Therefore the system has denoted the value 1 to represent a "yes" or "true" condition and 0 as meaning "no" or "false".

Relational Functions

Within the APL system there are six symbols used to denote the relational functions. These six symbols are < ≤ = ≥ > ≠. They occupy the upper shift positions on the keys 3 through 8. They are used to compare the value of the left argument to that of the right. The various comparisons that can be performed are as follows:

Function	Meaning
$X<Y$	X less than Y
$X\leq Y$	X less than or equal to Y
$X=Y$	X equal to Y
$X\geq Y$	X greater than or equal to Y
$X>Y$	X greater than Y
$X\neq Y$	X not equal to Y

Here are a few examples of all of them.

```
      3>4
0

      3<4
1

      3=4 3
0 1

      3≠4 3
1 0
```

```
      ¯7≥¯6 ¯8 ¯7.9
0  1  1

      2.6≤2.6 2.7 2.5
1  1  0
```

The only two scalar functions that allow literals as arguments are the = and ≠.

```
      '6'='567'
0  1  0

      'A'≠'ABC'
0  1  1

      'M'>'N'
DOMAIN ERROR
'M'>'N'
```

Logical Functions

The logical functions in APL are the following:

Function	Meaning
$X \vee Y$	X or Y
$X \wedge Y$	X and Y
$X \nor Y$	neither X nor Y
$X \nand Y$	not both X and Y
$\sim Y$	not Y

If you are familiar with Boolean algebra, you will probably recognize the uses of these functions right away. Their results are 1's and 0's just like those of the relational functions, but their arguments must be 1's and 0's also.

Or

The or function determines if at least one of its arguments is equal to 1.

```
      0∨1
1
```

If it is, the computer will return a 1 response.

```
      1∨0
1

      1∨1
1
```

If both arguments are 0, the response is zero.

```
      0∨0
0
```

In the case of vector arguments, the corresponding pairs are scanned for ones.

```
      0 1∨1 0
1  1

      0∨1 0 1
1  0  1
```

And

The and function expects both of its corresponding arguments to be equal to 1 before it returns a 1 result.

```
      1∧0
0
```

```
      1∧1
1

      0 1∧1 1
0  1
```

Nor

The nor function produces the opposite result to that of or. It is created by overstriking the ∨ and the tilde symbol ~ (upper shift *T*).

```
      1∨0
0

      0∨0
1
```

Both of the corresponding elements in its arguments must be equal to 0 before it will return a 1 result.

```
      0 1∨1 0
0  0

      0∨1 0 1
0  1  0
```

Nand

The nand function is the inverse of the and. It is produced by overstriking the ∧ and the ~ symbols.

```
      1⍲1
0
```

It states that both corresponding values in both arguments must not be equal to one in order for it to return a 1 response.

```
      0*0
1

      1*1 0 1
0  1  0
```

Not

The one scalar function that may only be used monadically is the not function.

```
      ~1
0

      ~0
1

      ~0 1 0
1  0  1
```

It produces the logical negation of its argument. If its argument is a 1, the result is a 0, and vice versa.

Summary

This completes the roster of scalar functions existing in the MCM/70. Just to recap what they are, here is a brief summary containing the symbols each is represented by, and the responsibility each one assumes:

Scalar Monadic

Function	Meaning
+Y	Y
-Y	0-Y
×Y	Signum Y
÷Y	Reciprocal of Y
*Y	e to the Yth power
⌈Y	Ceiling of Y
⌊Y	Floor of Y
|Y	Absolute value of Y
⍟Y	Natural logarithm of Y
!Y	Factorial Y
○Y	Pi times Y
?Y	A random number from 1 to Y
~Y	Not Y

Scalar Dyadic

Function	Meaning
X+Y	X plus
X-Y	X minus Y
X×Y	X times Y
X÷Y	X divided by Y
X*Y	X to the Yth power
X⌈Y	Maximum of X and Y
X⌊Y	Minimum of X and Y
X|Y	X-residue of Y
X⍟Y	Base-X logarithm of Y
X!Y	Binomial coefficient: Y taken X at a time
X○Y	Trigonometric functions
X<Y	X less than Y
X≤Y	X less than or equal to Y
X=Y	X equal to Y
X≥Y	X greater than or equal to Y
X>Y	X greater than Y
X≠Y	X not equal Y
X∨Y	X or Y
X∧Y	X and Y
X⍱Y	Neither X nor Y
X⍲Y	Not both X and Y (X nand Y)

We have not seen the last of these scalar functions yet. The next chapter, and others show additional ways in which they can be employed. But, throughout it all, their prescribed tasks remain the same.

Practice Exercises

1. Using the MCM/70, evaluate the following:

   (a) `1∧1`

   (b) `1∧1 0 1 1`

   (c) `0∨1 0 1 1`

   (d) `1⍱~1`

   (e) `~1 1 1`

   (f) `1 0⍲0 1`

2. Write an APL expression for each of the following:

   (a) Assign *A* the value 8 if *B* is greater than *C*, otherwise, assign *A* the value 0.

   (b) Assuming *X* represents a vector of values, reset all those values which equal 3 back to zero, and leave the other values as they are.

Chapter 7: REDUCTION AND SCAN

When we first started performing calculations with the MCM/70, we used scalars as arguments.

```
      3+4
7
```

Then we saw the computer could also handle series of numbers as well, which greatly facilitated many of our computing tasks. When we wanted to multiply a group of numbers by 100, we did so by one simple operation.

```
      100×6 7 8 5 4
600  700  800  500  400
```

We found also that both arguments could be vectors, and the same functions that worked with scalars could be extended to perform the same way with vectors, by treating the corresponding pairs of elements of each vector as a set of scalars.

```
      6 4 3+7 2 8
13  6  11
```

Again our ability to solve more problems was increased. The only area we have not yet addressed is performing calculations on the individual components within an array. An example is the summing of the elements contained in the right argument of this last example above. We can mentally calculate the sum of 7 2 8 as being 16, but how do we express this in APL? This chapter shows us how. It also illustrates how to perform "cumulative summing" on arrays. The two functions used to carry out these tasks are called reduction and scan respectively.

Reduction

Our above problem of summing the values 7 2 8 has two solutions. We could insert plus signs between each of the numbers in the form 7+2+8

or we could use the reduction function to arrive at the same answer.

```
      +/7 2 8
17
```

A reduction function is denoted by the solidus or slash symbol / located in the lower right-hand corner of the keyboard. Together with the plus sign, the above expression is read as "plus reduce 7 2 8" or "plus over 7 2 8".

Here are two more examples:

```
      +/1 2 3 4 5 6
21

      +/100 1000 10000
11100
```

Averaging

With the aid of the plus reduction function, we can now calculate the average of a set of values. Averaging is simply a matter of finding the total of a group of numbers and dividing it by the sum of numbers involved. The vector 7 2 8 has three numbers in it. Therefore, the averaging statement is

```
      (+/7 2 8)÷3
```

Imagine these numbers represented in-stock inventory items. This means the average stock on hand is

```
      (+/7 2 8)÷3
5.6667
```

items.

We are able to divide by 3 because we know the vector contains three elements. But what if we do not know the number of elements? There is an APL function which we will cover later on which will tell us this. To improvize in the mean-time, let us develop a method of getting around this.

We know that if we perform the comparison equal on two arguments, the result consists of 1's and 0's to indicate "yes" and "no" replies.

```
      3 2 6=4 2 5
0  1  0
```

If we compare a vector of numbers to itself, we would receive all ones in the reply.

```
      7 2 8=7 2 8
1  1  1
```

Notice there are three 1's equalling the length of either argument. If we were to add up these ones, what would you get?

```
      +/1 1 1
3
```

Did you hear a bell just ring?

If we now apply what we have developed to our average statement, we will arrive at the correct answer.

```
      (+/7 2 8)÷(+/7 2 8=7 2 8)
5.6667
```

And we do.

This principal frees us from having to know the length of the vector argument before we try to average it.

```
      A←4 2 3 9 5 7
■

      (+/A)÷(+/A=A)
5
```

Times Reduction

The function performed by the reduction is determined by the accompanying scalar primitive. Here is an example of finding the product of a set of numbers:

```
      ×/7 2 8
112
```

Here is another:

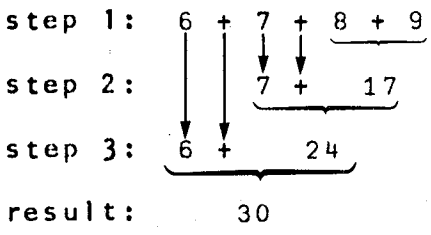
```
      A←1 2 3 4 5 6 7 8 9 10
      ×/A
3628800
```

Order of Execution

Like all calculations in APL, the order of execution of the reduction functions is from right to left. In the cases of plus and times, this is not too apparent because evaluations either way yield the same results.

```
      +/6 7 8 9
30
```

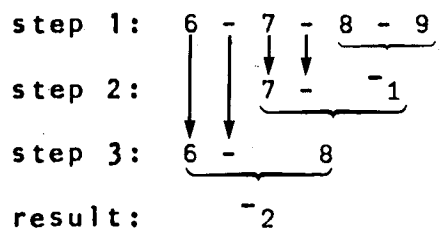
The above statement was evaluated in the following manner:



But when the operation is a subtraction or division, the sequence in which the numbers are taken has a definite affect on the result. The following function is a good illustration of this:

```
      -/6 7 8 9
¯2
```

The way its answer was arrived at was by the following steps:



Reductions can be done with all the scalar dyadic functions.

Function Symbol	Name	Function Symbol	Name
+	plus	<	less than
-	minus	≤	less than or equal to
×	times	=	equal to
÷	divide	≥	greater than or equal to
*	power	>	greater than
⍟	logarithm	≠	not equal
⌈	maximum	∨	or
⌊	minimum	∧	and
|	residue	⍱	nor
!	binomial coefficient	⍲	nand
○	circular		

Here are examples of some of the other scalar primitives being employed with the reduction function:

```
      ⌈/2 ¯6 4 0
4

      ⌊/2 ¯6 4 0
¯6

      */3 2 1
9

      ∨/1 0 0
1

      ∧/1 0 0
0

      ÷/12 ¯6 ¯4 3 6
16
```

Scan

The scan function looks very similar to the reduction function. They both work with only scalar dyadic primitives, and they both are denoted by the slash symbol. The only difference in appearance is the scan uses the reverse slash.

Reduction Syntax	Scan Syntax
f/*A*	f*A*

In spite of the fact that they look alike, they perform quite differently from one another. While the reduction function "reduces" its argument to a single total, or product, or what have you, the scan function produces a "cumulative" total or product, etc.

```
      +\1 2 3 4
10  9  7  4
```

It works from the right of its argument to the left. This last example was evaluated by the following procedure:

1. The last value of 4 in the result is the last value in the argument.

2. The 4 and the 3 in the argument are added together to produce the 7 in the result.

3. The 4, the 3 and the 2 in the argument are added together to produce the 9 in the result.

4. Then the entire argument is summed to arrive at the 10 in the result.

Each of these steps implies the use of the plus reduction function.

```
step 1:  +/4        or   4

step 2:  +/3 4      or   7

step 3:  +/2 3 4    or   9

step 4:  +/1 2 3 4  or  10
```

Here is another example:

```
      ×\5 2 4 3
120  24  12  3
```

And here is how it was evaluated:

step 1: ×/3 or 3

step 2: ×/4 3 or 12

step 3: ×/2 4 3 or 24

step 4: ×/5 2 4 3 or 120

The following are a few more illustrations involving other scalar primitives with this function:

```
      ⌈\1 2 3 4 5
5  5  5  5  5

      ⌊\1 2 3 4 5
1  2  3  4  5

      ●\3 2 512
2  9  512
```

Practice Exercises

1. Employ the MCM/70 to evaluate the following:

(a) +/3 6 7 2 (b) -/3 6 7 2

(c) +/¯20 10.2 6 (d) ×/¯20 10.2 6

(e) +\3 6 7 2 (f) -\3 6 7 2

(g) +\¯20 10.2 6 (h) ×\¯20 10.2 6

2. Each weekend for five weekends, a boy goes out collecting discarded pop bottles. His findings each week are 32, 45, 27, 36, and 24 bottles. How many did he collect altogether? If they are worth 2 cents each, how much money did he earn?

3. A climber is ascending a mountain. If he can cover enough ground each day, equivalent to the vertical gains of 9200, 7100, 5900, 3600, and 2100 yards, how high up is the mountain peak? What will be his total vertical elevation at the end of each day? (hint: Reverse the order of his yardage gains.)

4. How many dollars and cents are there in 6 quarters, 35 nickels, and 57 pennies?

5. For the following series of numbers, write the APL expression to find:

   (a) the largest value

   (b) the smallest value

   (c) the difference between the largest and the smallest

   (d) the average value

```
A←7 ¯2 6.3 13 0 ¯.1 4 9 1
```

Chapter 8: DEFINED FUNCTIONS

There are approximately eighty "primitive" functions in MCM/APL. Forty-eight of these were described in the last few chapters. These functions each perform unique tasks and most of them are able to accomodate arguments of varying shapes and sizes. But, like most computers, there is a limit to just how much can be "pre-programmed" into it. This restriction is imposed by the physical constraints of the unit itself. Even so, the MCM/70 has considerably more pre-programmed routines in it than any other computer its size. But because the constraint is present, MCM/APL has been designed in such a way to enable you to get around it by allowing you to create your own functions which you may need for your particular applications.

Apart from having its own repertoire of functions, it is still able to accept others. Remember in Chapter 1 we mentioned there were two different modes of operation for the MCM/70? One is called execution mode and the other is called definition mode. Well, it is by using the MCM/70 in definition mode that we are able to extend its capabilities.

So far we have been utilizing the computer in execution mode only, employing several of the primitives available. Each time we entered an expression it was immediately evaluated by the computer and the result was displayed on the screen. This meant anything entered that could be successfully computed was done so and anything that could not was rejected with an appropriate error message. This type of environment is fine for calculations that are done on a "one-time" basis only. But in cases where the same routines will be used more than once, they should be stored in the computer's memory so that they do not have to be re-entered from the key board each time they are used. We defined variables for very much the same reason. It saves time and reduces the possibility of typing errors. But when writing our own user defined functions, unlike storing variables, the mode of the system has to be changed in order for it to accept them. Here is a simple problem for which we will define a function to solve:

> After taking the attendance readings in a school classroom over a two week period, we would like to determine the following:
>
> 1: the average attendance,
>
> 2: the highest attendance reading, and
>
> 3: the lowest reading.

Our record attendance levels were the following:

Mon.	Tues.	Wed.	Thurs.	Fri.	Mon.	Tues.	Wed.	Thurs.	Fri.
23	25	24	26	28	23	27	28	27	26

In order to carry out the calculations with a minimal amount of typing, let us assign these numbers to a variable.

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

Now the first thing we want to find is the average attendance. To do this, we must first add up all the values and then divide by the number of attendance readings taken. But if we did not know that number, one quick way to find out is with the following statement:

```
      +/X=X
10
```

Obviously *X* equals itself, so the result of *X*=*X* is ten 1's. And the plus reduction of these ten ones yields a result of 10. This little trick was covered in Chapter 7. Here is the count calculated again, but this time the result is assigned to the variable *N*:

```
      N←+/X=X

      N
10
```

And to find the average of *X*, the following is typed:

```
      (+/X)÷N
25.7
```

Now that we have determined the average attendance, the next step is to find the highest attendance reading. By using the maximum symbol in conjunction with the reduction symbol, this task can be carried out in the following way:

```
      ⌈/X
28
```

And the lowest reading can be obtained just as easily.

```
      ⌊/X
23
```

Now that we know the APL expression required to solve our problems, let us define a function that will automatically do them for us.

Function Definition Mode

The first thing we have to do when defining a function is to change the mode of the computer. Above the letter *G* on the keyboard, there is a symbol called del (∇) which is used to change the system from execution mode to definition mode and vice versa. To get the system into definition mode, we key in the del followed by the name of the function being defined. Let us call this one *ATTEND*.

```
      ∇ATTEND
[1]
```

Notice the system responds with [1]. This signifies it is in definition mode and ready to accept the first line of the function. In our case it is *N←+/X=X* to count up how many attendance readings were taken.

```
[1]N←+/X=X
[2]
```

After entering this statement and pressing the RETURN key, the system responds with [2], asking for the next line. We reply with *(+/X)÷N* which is the expression to find the average attendance.

```
[1]N←+/X=X
[2](+/X)÷N
[3]
```

The system continues to accept our statements, a line at a time.

```
[3]⌈/X
[4]⌊/X
[5]
```

When all our algorithms have been entered, we signal this to the computer by entering another del.

```
[3]⌈/X
[4]⌊/X
[5]∇
```

This causes the computer to switch from definition mode back to execution mode. Notice only the cursor appears on the screen after the del has been typed and the RETURN key pressed.

Just to make sure we are in fact in execution mode, try a simple calculation like the one below.

```
      2+2
4
```

Now that our function *ATTEND* has been defined and stored in the computer's memory, it can be executed at any time during the session by simply typing in its name.

```
      ATTEND
25.7
```

The first line displayed by the computer is 25.7, representing the average for the set of values contained in *X*. There are two columns of illuminated dots at the extreme right end of the screen. They are to let you know that *ATTEND* has not yet finished executing all of its statements. At this point, it is through with line 2 and ready to execute line 3. It is just waiting for you to press the RETURN key, signalling it to continue with its calculations. This rectangle of illuminated dots appears every time output is displayed while the system is still executing a user defined function. To facilitate illustrating the displayed output of *ATTEND*, assume the RETURN key has been pressed so the output can appear immediately, one after the other. Here are the rest of the results.

```
28
23
```

These two figures represent the highest and the lowest attendance readings respectively.

Once a function has been defined, you do not have to do it again, assuming of course the unit is not shut off without first saving the function on a tape cassette. This means *ATTEND* can be used at any time, with any values in *X*.

```
      X←6 26 ¯2.5 7.2
■

      ATTEND
9.175
26
¯2.5

      X←1 2 3 4 5 6 7 8 9 10
■

      ATTEND
5.5
10
1
```

Displaying a Function

If, at any time, we wish to view any or all of the lines in a defined function, we simply go back into definition mode again by typing the del and the name of the function we wish to display.

```
      ∇ATTEND
[5] ■
```

Notice the system displays the line number which is one greater than the number of lines the function contains. It is assuming we wish to add more statements to it. To get it to display one of its existing lines, we have to redirect it to the appropriate statement by keying in the statement's line number (in brackets). Below, the system is requested to display line 1.

```
      ∇ATTEND
[5][1]
```

Upon pressing the RETURN key, the computer responds with

```
[1] N←+/X=X
```

and it places the cursor over the *N*, thinking we are going to change something on this line. (We will discuss how function statements are edited in a few more pages).

By pressing the RETURN key only, line 2 will be displayed and, by repeating this process, we get to see all the lines in the function, which look like this:

```
[1]N←+/X=X
[2](+/X)÷N
[3]⌈/X
[4]⌊/X
```

Once at line 5 we can either close the function by typing in the del again or go back to some other line, or even add more lines. We can even redirect the computer to display a different line while it still has more lines to display by typing in [*L*], where *L* is the line number we wish. We can also close a function in the middle of its line display, just as long as the del is either the very first or very last character entered on the line.

Function Editing

Functions usually undergo several changes before a satisfactory version is obtained. This could be due to many reasons. We may just want to add more lines, modify existing ones, or even delete some that are no longer needed. But whatever the reason, it is best to know the four basic changes that can be made to an existing defined function to modify its structure and/or contents. These are:

1) add one or more lines into the function;

2) insert one or more lines between those already existing within the function;

3) modify the contents of existing lines;

4) erase or drop lines from the function.

Line Addition

In our example of evaluating classroom attendance, we found the average, the highest, and the lowest recordings registered. Now that we have found these statistics, let us also find the range between the highest and the lowest. To do this we have only to subtract the high reading from the low in the following manner:

```
      (⌈/X)-L/X
```

To add this expression to *ATTEND*, we simply go back into definition mode

```
      ∇ATTEND
[5]
```

and type it in.

```
[5](⌈/X)-L/X
[6]∇
```

Now when *ATTEND* is executed, the following results are produced:

```
      ATTEND
5.5         average
10          highest value
1           lowest value
9           range
```

The reason the values produced by *ATTEND* look so different from those of our class attendance statistics, is that we redefined *X* with the values 1 to 10, and have not changed it since.

Line Insertion

Notice that each line of a function is an integral value beginning at 1 and numbering up to the last line of the function. If we wish to insert an additional statement between lines 3 and 4, we would give this new statement a line number of 3.5 or 3.01 or even 3.99, just as long as it was greater than 3 but less than 4. (The limit on any line number is 127.99). In our function *ATTEND*, let us insert a line to

display the contents of *X* everytime *ATTEND* is executed. Below, *ATTEND* is opened up again and redirected to line number .5.

```
      ∇ATTEND
[6][.5]
[.5]
```

This is the line number we have chosen to place our statement in to display the values in *X*. Because .5 is smaller than 1, the contents of this new statement will be the first ones evaluated whenever *ATTEND* is executed.

```
      ∇ATTEND
[6][.5]
[.5]X
[.6]∇
```

Notice that, after the new statement is entered, the line counter is incremented by .1. The system assumes we are going to add more than one line. But this time we are not, so we terminate definition. Now if *ATTEND* is executed, we would get the following results:

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

And if *ATTEND* is displayed, it would look like this:

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

Notice the line numbers have been renumbered. The line that was entered as [.5] is now line 1 and all the other line numbers have been incremented by one. This is done automatically by the system as soon as the function is closed.

Line Modification

Lines 2 and 3 are used to find the average attendance. Line 2 determines the number of attendance recordings and line 3 uses this figure to complete the averaging calculation. The whole process could have been done in one line, which would mean the overall execution time would be decreased and the variable N would not have to be defined. The expression would look like this:

```
      (+/X)÷+/X=X
```

In order to incorporate this new expression into *ATTEND*, either line 2 or line 3 would have to be modified. The obvious choice would be line 3 since the first part of the expression is already there and besides, this is the line that displays the result, assuming of course we only want the average displayed once.

The procedure for modifying an existing function line is very much the same as that for inserting a new line. First we have to open the function,

```
      ∇ATTEND
[7]
```

and then redirect it to the appropriate line.

```
      ∇ATTEND
[7][3]
```

After pressing the RETURN key, notice the contents of line 3 are displayed with the cursor being placed over the first character.

```
[3](+/X)÷N
```

All we have to do now is space over to the *N* and replace it with +/X=X to change the statement to

```
[3](+/X)÷+/X=X
```

We can either press the RETURN key now and close the function on line 4 by typing the del or we could just type in the del as the last character of line 3. Both achieve the same results, it's just that the latter is quicker.

```
[3](+/X)÷+/X=X∇
```

And now we are back in execution mode. Just to make sure *ATTEND* still works, here it is again:

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

Upon displaying the function, we see that it now looks like this:

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

The new contents of *ATTEND* now make line 2 an unnecessary item which is serving no other purpose other than consuming usable space in memory. This means we can get rid of it, which leads us nicely into how existing lines are dropped from functions.

Line Deletion

Both deletion and modification of lines in a defined function are performed the same way as deletion and modification of statements entered while in execution mode. For instance, to delete line 2 from *ATTEND*, simply open *ATTEND*,

```
      ∇ATTEND
[7]
```

redirect it back to the line to be erased,

```
[7][2]
```

press the RETURN key, and when it displays the contents of that line,

```
[2]N←+/X=X
```

hold the CTRL key down and press the BKSP key to delete all the characters,

```
[2] ■
```

then type in the del

```
[2]∇
```

and press the RETURN key. After this is done and the function is displayed, you will see it looks like this:

```
[1]X
[2](+/X)÷+/X=X
[3]⌈/X
[4]⌊/X
[5](⌈/X)-⌊/X
```

Notice again the lines have been renumbered into a consecutive sequence. And upon executing *ATTEND*,

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

it still returns the same results.

There is another way to delete entire lines of functions which is much quicker than the method just described. Instead of manually erasing every character in a line, we can simply press the CTRL key and then the SHIFT and ← keys at the first available position of the line to be deleted and press the RETURN key. Assume the contents of the "old" line 2 was still present in *ATTEND*. To erase it using this second technique, you would follow these steps:

1. Go back into function definition mode with *ATTEND*.

```
      ∇ATTEND
[7]■
```

2. Redirect the system back to line 2

```
      ∇ATTEND
[7][2]
```

and press the RETURN key.

3. After the system has displayed the contents of line 2 and placed the cursor over the letter *N*,

   [2]*N*←+/*X*=*X*

   press the CTRL key and then the SHIFT and ← keys.

4. The system will then display line 3. (This response is different from the one given by the other method.) Type in the del to end function definition mode

   [3]∇←/*X*)÷+/*X*=*X*

   and press the RETURN key.

If we now display *ATTEND*, we will see that the contents of the "old" line 2 has been erased and the remaining lines have been renumbered, just as they were after the other line deletion method was employed.

Practice Exercises

1. What are the two modes of operation in which the MCM/70 can function?

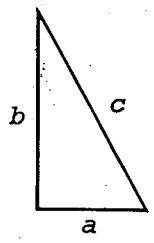
2. How can you switch from one mode to the other?

3. After each line of a function is entered and the RETURN key is pressed, what is the computer's response?

4. The formula for finding the length of the hypotenuse of a right-angled triangle is:

$$c = \sqrt{a^2 + b^2}$$

where the triangle is of the following shape:



Given that *a* is 3 inches long and *b* is 4 inches long define a function called HYP that will calculate the length of *c*.

5. Key this function into the MCM/70:

```
 ∇STATS
[1]'NUMBER OF OBSERVATIONS:'
[2]+/X=X
[3]'LARGEST VALUE:'
[4]⌈/X
[5]'SMALLEST VALUE:'
[6]⌊/X
 ∇
```

(a) Assign X ten numbers at random.

(b) Execute the function.

(c) Add these two statements to the function:

```
[7]'RANGE:'
[8](⌈/X)-⌊/X
```

(d) Close the function and execute it again.

(e) Insert these two lines between the existing line numbers 2 and 3:

```
[2.1]'OBSERVATIONS ARE:'
[2.2]X
```

(f) Close the function and re-execute it.

(g) Delete lines 5 and 6.

(h) Close the function and re-execute it.

(i) Insert two lines in their place; one to state the word "average" and the other to find the average of X.

(j) Close the function and re-execute it.

Chapter 9: TYPES OF USER DEFINED FUNCTIONS

With what we learned in Chapter 8 on user defined functions, we can now experiment with the various "types" that can be created. The word types refers to the number of arguments a defined function can have. We will see in this chapter there are three possible choices available - no arguments, one argument or two arguments. We will also see how some functions can be used as arguments themselves, while others can not. This again is determined by the user as he is defining his function in the computer. The number of arguments a function can have and its ability to become an argument of another function (primitive or defined) are both contained in a part of a function known as its header line.

Header Line

A defined function is made up of basically two parts, the body and the header line. The body runs from line 1 to the last line of the function. The header line is the line which contains the name of the function.

```
      ∇ATTEND  ◄──── header line
[1]  (+/X)÷+/X=X  ┐
[2]  ⌈/X          │
[3]  ⌊/X          ├── body
[4]  (⌈/X)-⌊/X    ┘
```

Even though it is not assigned a line number as far as we can see, it is referred to by the system as line 0. This enables us to perform editing on it if ever we choose to do so.

Function Syntax

Apart from containing the function's name, the header line also contains the function's syntax. The syntax dictates the rules by which a function can be executed. Just as it does for primitive functions, this syntax indicator tells the system how many arguments to expect before beginning its execution of a defined function. For primitive functions the syntax indicators are a predetermined part of the system. They state whether their respective functions are monadic or dyadic in use. The functions ÷2 and 6÷2 are two examples of this.

A comparison is drawn between the primitives and the defined functions by referring to the symbol used to represent the primitive as the name of the primitive.

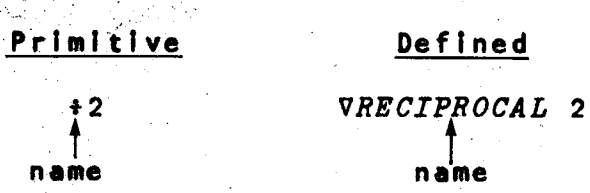


Figure 9.1

The value 2 in each case is considered to be an argument to the function. Therefore each function must be monadic. Notice the 2 is separated from the name of the defined function by a space, while the 2 associated with the primitive is not. The divide symbol ÷ cannot be construed as being a valid variable or function name, therefore there is no need to separate it from its argument. But the defined function's name and argument must be divided from each other by at least one space to distinguish to the system which is the name and which is the argument. If no space was provided, the header line would appear as:

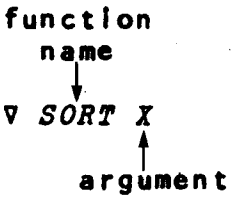
```
∇RECIPROCAL2
```

inferring the name to be *RECIPROCAL2* and that no argument is expected.

As with primitive functions, we can also have monadic and dyadic user defined functions. The header line of *RECIPROCAL* states that it is a monadic function. Another header line stating the same thing is the following.

```
∇SORT X
```

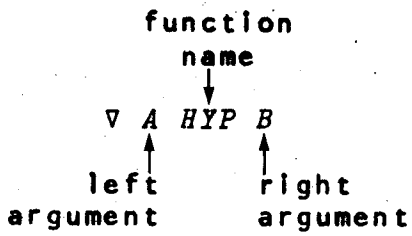
The function is called *SORT* and its argument is *X*.



The name of a monadic function always preceeds its argument, just as the primitive function's symbol always preceeds its argument. The header line of a dyadic function looks like this:

```
∇ A HYP B
```

The name of the function is *HYP* and its two arguments are *A* and *B*.



Types of Header Lines

Having seen the syntactical relationship between primitive functions and user defined functions, you may be asking yourself, "well then, what kind of function is *ATTEND*?" Granted it does not have any arguments and yet it can still be executed. The reason why lies in the need for functions of this type. We will encounter functions later that require no imput in order to perform their required tasks. Drills and descriptive type functions fall into this category. Functions expecting no arguments are called niladic. This is the third and last type of header line a defined function can have. Just to review them again, they are:

niladic - no arguments
monadic - one argument
dyadic - two arguments

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

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

Figure 9.2: User Defined Function Types

Ignoring any distinctions between "no explicit result" and "explicit result" for the moment, let us concentrate on the functions mentioned in the table above.

Suppose for the time being that all the functions listed above have already been defined in the workspace. Some of them contain primitives that are not discussed until later in the book, but the primary objective right now is to understand all the different kinds of functions that can be created. So please ignore for now any unknown primitives that appear in them but when you do learn about them later on, return to these pages and see how they are used.

Niladics

One function that does not contain any new primitives is our old friend *ATTEND*.

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

It is pretty straight forward and needs no further discussion, other than to note that it needs no argument in order to be executed.

Monadics

The function *SORT* holds something new for us though.

```
      ∇SORT
[1]X[⍋X]
 ∇
```

As its name implies, it is a monadic function that sorts numbers. And since it is monadic, it expects a right argument. If one is not supplied, here is what happens:

```
      SORT
SYNTAX ERROR
SORT
```

This is the same type of response we would get if we tried to execute any of the primitive functions in the same manner.

```
      +
SYNTAX ERROR
:
```

Therefore *SORT* must always be accompanied by a right argument.

```
      SORT 2 6 3 1 ¯4
¯4  1  2  3  6
```

Displaying its argument in ascending sequence, *SORT* can accept any vector of numbers, whether they are in the form above or in a predefined variable.

```
      A←100 2 6.6 33 ¯10 7
```

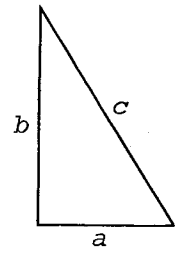
```
      SORT A
¯10  2  6.6  7  33  100
```

Dyadics

The function *HYP* is defined as being dyadic. Upon displaying its contents we can see it is used to find the length of the hypotenuse of a right-angled triangle, using the formula $c = \sqrt{a^2 + b^2}$.

```
     ∇A HYP B
[1]  ((A*2)+B*2)*.5
     ∇
```

The two arguments *A* and *B* represent the lengths of the sides of the same name in the triangle.



If lengths of sides *a* and *b* were 3 and 4 inches respectively, the length of *c* would be determined by employing *HYP* in the following manner:

```
      3 HYP 4
5
```

We can use the same function to solve for two or more triangles at once.

```
      2 6 HYP 5 7
5.3852  9.2195
```

That takes care of the top row of functions in figure 9.2, which were all described as being "no explicit result" function. Each one did produce a result though; so what does this phrase imply? The distinction between explicit and no explicit result producing functions is explained in the next section along with examples of the uses for the functions listed in the second row.

Explicit vs. No Explicit Result Functions

The only difference between an explicit result function and a no explicit result function is that the explicit result, at the end of its computations, may be used as an argument of another user defined or primitive function. For instance an expression such as *F*+6 where *F* is the name of a niladic explicit result function is a perfectly valid statement in MCM/APL. We can easily tell an explicit result function by the presence of the specification arrow (←) in the header line. Notice in the second row of function names in fig. 9.2 that all the function header lines have this left pointing arrow in them. The function *ROLL* is a typical example of an explicit result, niladic function. It selects, at random, two numbers from one to six by means of the primitive function.

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

Here is how it works:

```
      ROLL
3 4

      ROLL
1 1
```

The purpose of *ROLL* is to simulate the rolling of two dice.

```
      ROLL
6 1

      ROLL
3 2
```

If we wanted to add up the result of the 2 random numbers generated to get their total count, we would simply place a +/ before the name *ROLL*.

```
      +/ROLL
9

      +/ROLL
7
```

```
      +/ROLL
4
```

Notice we can use *ROLL* as if it were a variable argument for some other operation, namely the +/ function. This is possible because *ROLL* is defined as a function which produces an explicit result. Remember *SORT* does not produce an explicit result. All it does is display its argument in ascending sequence. Therefore it cannot be used as an argument to some other operation.

```
      +/SORT 2 1 3
1  2  3
VALUE ERROR
+/SORT 2 1 3
```

In the above example the function *SORT* rearranged its argument into ascending sequence and printed it out just as it did before, but, because it is not an explicit result function, the expression caused the value error to occur. (The system knows *SORT* is a no explicit result function, so upon completing its execution of *SORT*, it looked for a variable of the same name, and found none.) Therefore, even though *SORT* can accept an argument as input, it itself cannot be used as an argument to another function. The remaining two functions to be explained exemplify this distinction further.

The next function in the table is *SQRT*, a monadic, explicit result function which finds the square root(s) of its arguments.

```
      SQRT 25
5

      SQRT 4 16 64
2  4  8

      (SQRT 9 36)*2
9  36
```

SQRT looks like this:

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

The last function in the table, *RND*, rounds off the value(s) of the right argument according to the specification in the left. For in-

stance, to round the value 76.826 to two significant digits, the appropriate statement would be:

```
      2 RND 76.826
76.83
```

The function either rounds up or down to its closest significant digit.

```
      10+1 RND .0162 5 10.269
10.1  15  20.3
```

Look at the line in *RND* that does the actual rounding and see if you can determine how it works.

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

The technique is really quite simple.

We can combine *RND* and *SQRT* together for computing square roots. Assume we are interested in only the first three decimal places of the roots. If the root exceeds this limit, as it does here,

```
      SQRT 7
2.6458
```

we can employ *RND* to present the answer in the form we want.

```
      3 RND SQRT 7
2.646
```

Before leaving function types, there is one more point to note. User defined functions can be used by other defined functions in much the same way primitives are. In the last example, we saw how one function could be used as an argument to another. Instead of having to key in the names of both functions, we could have had *SQRT* "call" *RND* automatically. Let us change *SQRT* so that it does this. It should look like this:

```
      ∇Z←SQRT N
      [1]Z←3 RND N*0.5
      ∇
```

Upon executing *SQRT*, we see it does in fact use *RND* to perform the

perform the prescribed rounding.

```
      SQRT 7
2.646
```

Functions can call any number of other functions, which in turn can call still others. The called functions can even call the function that did the initial calling. Or, a function can call itself. Functions of this type are known as recursive functions. Here is an example of one:

```
      ∇R←FAC N
[1]→4×⍳N=0
[2]Z←N×FAC N-1
[3]→0
[4]Z←1
      ∇
```

(The right pointing arrows, → , in lines 1 and 3 are known as branch arrows, which are discussed in detail in Chapter 15.)

It performs the same task as the factorial primitive.

```
      !4
24

      FAC 4
24
```

It invokes itself on line 2.

Summary

You will probably do more function writing, editing and executing on the MCM70 than anything else. Therefore it is wise to spend a considerable amount of time on these three things. Before you write a function, you should know how you want to use it. Do you want its result to be input of further calculations? Do you want it to accept arguments? If so, how many? A complete understanding of the header line is required before these questions can be answered adequately.

This chapter has shown you there are three types of functions than can be created - niladic, monadic and dyadic. The type or syntax is determined by the number of arguments the header line has. One point that deserves mentioning while we are here is that the names of the functions must adhere to the same rules as applying to variable names. The same is also true for argument names.

When determining the type or result the function is to produce, remember a specification arrow in the header line indicates an explicit result is expected when the function completes its execution. The name you chose in the header line to represent this result must be assigned some data within the body of the function while it is executing. In other words, you cannot indicate to the system, that it is to expect data to be temporarily stored in its memory under a certain name and then never assign that name any data. The end result would be a *VALUE ERROR* if you tried to refer to the function's result as an argument to some other function. The names you choose in the header line to represent your data (both input and output) take on special characteristics within the MCM/APL system. These characteristics plus others are mentioned in this next chapter.

Practice Exercises

1. What is the maximum number of arguments a defined function can have? What is the minimum?

2. How many different types of defined functions are there?

3. The system refers to the header line as line _____.

4. Develop a monadic, explicit result function that produces the natural logarithm for any valid number.

5. Develop a monadic, explicit result function which finds the average of any set of values.

6. Develop a dyadic, no explicit result function to print the perimeter and area of a rectangle.

7. Develop a function having the header line

```
∇TOT←UNITS TIMES COST
```

that takes as input the invoice listing of the number of units sold and their unit costs, and produces the total cost of the invoice.

Chapter 10: TYPES OF VARIABLES

Within MCM/APL there are two types of variables that can be defined. One is called global and the other is called local. All the variables we have defined to this point have been global variables. This chapter explains what each type is, the differences that exist between them and where and when each applies.

Local vs. Global

To assist in explaining local and global variables, let us assume the six functions described in Chapter 9 are still in the workspace. Their header lines again are:

```
∇ATTEND
∇SORT X
∇A HYP B
∇R←ROLL
∇R←SQRT N
∇C←A RND B
```

The only variable referred to in *ATTEND* is one called *X*.

```
 ∇ATTEND
[1](+/X)≠+/X=X
[2]⌈/X
[3]⌊/X
[4](⌈/X)-⌊/X
 ∇
```

It has to be assigned values before *ATTEND* can be executed successfully. If it is not, the following occurs:

```
VALUE ERROR
ATT[1] (+/X)≠+/X=
```

We will get back to this *X* in a moment.

The next function on the list is *SORT*. It also uses a variable called *X*. But its *X* performs two tasks at the same time. It not only helps specify the function's syntax, but it also represents the values to be

sorted whenever *SORT* is executed. Just to refresh our memories a little, here is the function *SORT*.

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

And here it is in use.

```
      SORT 6 ¯2 4 7 3 1
¯2  1  3  4  6  7
```

The only variable referred to here is *X*, and it appears to always take on the values that appear to the right of *SORT*. But what if *X* were assigned values before *SORT* was executed? Let us do that. We will assign *X* a literal vector.

```
      X←'THIS IS A TEST'
```

If we again execute *SORT*, look what happens.

```
      SORT 3 2 1
1  2  3
```

The function seems to have ignored the variable *X* containing the literal and instead directed all references to *X* to the one in the header line. To confirm our variable *X* still exists and remains unchanged, let us display its contents.

```
      X
THIS IS A TEST
```

The only conclusion we can draw from this is that there are two *X*'s present; one representing the literal vector and the other representing the numeric vector. But if we execute the system function ⎕*VA*, which lists all the variables currently residing in memory, we see there is only one,

```
      ⎕VA
X
```

and it represents the literal vector. We have just seen a local variable at work. By definition, a local variable becomes "active" only while the function, in which it resides. When the function is finished, it is automatically erased from the computer's memory. The corollary to this is that, when the local variable becomes "active",

any global variable of the same name becomes "inactive" until the function employing the local variable has finished executing. At that time the inactive global resumes the same role it did before the function was executed. Globals are not associated with specific functions as locals are. Their sole domain is the workspace itself, even though they can be used by functions, as *X* was in *ATTEND*.

For a variable to be "localized" to a particular function, it must appear in the function's header line. In *SORT*, the variable *X* is in the header line. In the function *HYP* there are two local variables, *A* and *B*. Again, they serve to indicate the function's syntax plus they assume the values supplied to the function when it is executing.

```
      ∇A HYP B
[1]   ((A*2)+B*2)*.5
      ∇

      3 HYP 4
5

      (7-4) HYP (2+2)
5
```

This applies even when the arguments are global variables.

```
      S1←2
■

      S2←5
■

      S1 HYP S2
5.3816
```

And our global variable listing confirms it.

```
      ⎕VA
X
S1
S2
```

In those functions that produce explicit results, the variable used to represent the result is also classified as a local variable. It too disappears upon function completion, although the values it represents are either displayed on the screen or passed on as an argument to another function (primitive or defined).

```
      SQRT 7
2.6458          (result displayed)

      10+SQRT 7
12.646                (result passed on to a primitive)

      2 RND SQRT 7
2.65                      (result passed on to a defined function)

      ⎕VA
X
S1
S2
```

The variable *R* used by *SQRT* is erased after each execution of the function.

Additional Local Variables

Most user defined functions require more than one line in order to complete all their calculations and any intermediate results obtained as each line is executed are stored in variables for later use. But once the function has finished executing all its lines, these intermediate values are no longer needed. This means, that if they are not stored in local variables, they will begin to clutter up the workspace and cut down on the area available for other calculations. They also tend to make it difficult to remember which variables are useful and which are not. Therefore, to aid in the general housekeeping of the workspace, MCM/APL allows us to define several variables as being local to particular functions. This means that instead of having just the arguments of a function automatically erased by the system when it completes its computations, we can localize many more variables and have them all disappear at the end of function execution. Obviously not all of them are needed to determine the function's syntax, so those that are not are separated from those that are by means of semicolons. For instance, below is the header line of a dyadic, explicit result function that has 5 local variables - *R*, *A*, *B*, *EXT*, and *TOT*.

```
      ∇R←A TIMES B;EXT;TOT
```

The first three describe the function type and the remaining two (*EXT* and *TOT*) indicate that the system will erase them and their contents from memory each time *TIMES* has completed its execution. Had they not been included in the header line as illustrated, they would have been stored in memory as global variables and remained there until we requested that they specifically be erased, or until we terminated the session. Here is the entire function *TIMES*:

```
     ∇R←A TIMES B;EXT;TOT
    [1]EXT←A×B
    [2]TOT←+/EXT
    [3]R←TOT+TOT×0.07
     ∇
```

It is used to calculate the total amount of an invoice, given the number of goods ordered and their unit prices.

Here are some typical data it might use:

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

Notice in line 3 of the function that a 7% sales tax is added on to the total invoice price to make it more realistic. It does not matter whether the figures indicating the quantity sold are the left or right argument; the function excutes just the same.

```
      6 23 16 9 TIMES 5.95 .98 1.59 2.25
111.2051
```

The only place where care must be taken is in making sure the corresponding values of the quantity and unit price columns hold the same position within their respective arguments. Otherwise, the wrong unit prices would be multiplied to the quantity figure, giving a completely erroneous result.

A quick check will tell us whether any new variables were defined and left in memory as a result of executing *TIMES*.

```
      ⎕VA
X
S1
S2
```

There is not. In order to make both *EXT* and *TOT* global again so that we can see the total amount before tax and the price extensions, we would have to change the header line itself. At first glance you may think this would be a difficult task, as there seems to be no way of directing the system to the header line because it has no line number. You will recall from Chapter 9 that even though one is not displayed, it has an associated line number of 0. Therefore, changing the header line is the same as changing any other line. Here is how we would get back to line zero.

```
      ∇TIMES
[4][0]
```

Notice that, whenever we edit any existing function, only the name of the function has to be entered, not the entire header line. In response to the command above the system would respond with

```
      [0] R←A TIMES B;EXT;TOT
```

And, just as we modify any other line, we would space across to the characters we plan on changing, (in this case, to the first semicolon,) press and hold the CTRL key down while we repeatedly press the BKSP key until all the necessary characters have been erased. Then press the RETURN and enter the closing del to get back into execution mode.

```
      [0]∇R←A TIMES B
      [1]∇ T←A×B
```

One point to note here is we cannot close a function while line 0 or the header line is being displayed. We have to close on some other line.

To see if our modification did work, let us execute *TIMES* again,

```
      6 23 16 9 TIMES 5.95 .98 1.59 2.25
111.2051
```

and then display the global variable list,

```
      ⎕VA
X
S1
S2
EXT
TOT
```

we see that *EXT* and *TOT* have in fact been made global.

```
      EXT
35.7  22.54  25.44  20.25

      TOT
103.93
```

Therefore, by adding and deleting variable names within a function header line, we can dictate to the system whether they are to be treated as locals or globals.

Suspended Functions

Local variables become active only while a function is executing. When finished, they are automatically erased from memory. But what happens when a function does not successfully complete its operations? If an undefined variable is referenced, or a mathematical error encountered, the execution of the function is halted immediately and the line containing the error is displayed along with the type of error that has occured. Suppose our function *BIL* had an error in it. Say we added to it a fourth line containing an erroneous statement.

```
      ∇BIL
[4]6+
[5]∇
```

That should do it. Now if we execute it the following will occur:

```
      5 BIL 7
SYNTAX ERROR
BIL[4] 6+
```

At this point *BIL* is considered to be still executing. It has become suspended because of some erroneous statement and will remain so until it is told either to resume again or abandon its efforts. We can determine that *BIL* is suspended by looking at the output of both the state indicator ⎕*SI* and the line counter ⎕*LC*.

```
      ⎕SI
BIL*

      ⎕LC
4
```

They indicate that *BIL* is indeed suspended at line 4. But what about the local variables created by *BIL*?

```
      ⎕VA
X
S1
S2
A
B
EXT
TOT
R
```

They are still present in the system and can be referenced.

```
      EXT
35

      A
5

      B
7
```

They will remain in memory as long as *BIL* remains suspended, assuming of course we do not erase them ourselves with the ⎕*EX* function.

By typing in →0 we can get *BIL* to terminate its execution and clean up the mess it has left in the workspace.

```
      →0
37.45 ←——our result

      ⎕SI
  (blank screen)

      ⎕VA
X
S1
S2
```

More on suspended functions is covered in Chapter 21.

Summary

There are two advantages to having local variables. First, any data name redundancies that may develop between items you have defined earlier in the session and those that are used within defined functions are automatically resolved. This serves as a protective device for any data you have created that is external to any function. Secondly, the "housekeeping" chores of your workspace are greatly facilitated. You do not have to manually erase unwanted variables created by a function each time it is executed. The system does this for you.

The one advantage of global variables is that they are there whenever you need them. They can either reside in memory or in cassette and can only be erased by a $\square EX$ or $\square WC$ command. They are used to represent data you wish to retain.

Practice Exercises

1. Name the two kinds of variables that exist in MCM/APL.

2. What is the major feature that distinguishes a local variable from a global?

3. If two variables with the same name are in the workspace at the same time, which one is considered "active" and which one is considered "inactive"?

4. Under what conditions would the situation described in question 4 exist?

5. In the following header line, make the variable names *A*, *B* and *C* local to the function.

```
∇Z←X PBS Y
```

6. Make the variable *M1* global to the following function.

```
∇PERT;A;Q;M1;R
```

Chapter 11: SOME MIXED FUNCTIONS

Chapters 4, 5 and 6 described over forty scalar functions available on the MCM/APL system. These chapters demonstrated that the main characteristic of all the scalar functions is the direct relationship between the lengths of the arguments and the lengths of their results. Scalar arguments produced scalar results, three element vector arguments returned three element vector results.

```
      2 4 6+7 8 9
9  12  15
```

There was also a similarity in the arguments themselves. If at least one argument was not a single value,

```
      3×4.5 20
13.5  60
```

then both arguments had to be of equal length and dimension.

```
      2 4+6 3 5
LENGTH ERROR
2 4+6 3 5
```

This restriction is quite reasonable as the arguments of most of these functions require this conformability in traditional algebra anyway. But there are other functions in MCM/APL where this rule is not necessary, and therefore has not been applied. This group of functions, called mixed functions, has been reserved for this chapter in order that we could get a better understanding of how the APL system works. Many of the mixed functions do not perform typical algebraic calculations, but rather offer a means by which to structure data into varying array forms and numbering systems. There are 22 mixed functions in MCM/APL and, like the scalar functions, they too perform monadically and/or dyadically. For the complete list of their names, the symbols they use and a brief description of what they do, see Appendix A. This chapter deals with five of them and the remainder are discussed in Chapters 11, 12, 18 and 19.

Restructure

The function used to create arrays of varying dimensions is called rho. The symbol used to represent it is ρ (upper shift *R*). Used dyadically, the entire function is written as *X*ρ*Y*, where *X* represents the values of the dimensions the result is to assume, and the *Y* represents the actual contents of the resultant array. For instance, to define a 4 element vector containing the value 3, we would do it this way:

```
      4ρ3
```

to which the response would be

```
3 3 3 3
```

or, if we wanted a 5 element vector of alternating 3's and 4's:

```
      5ρ3 4
3 4 3 4 3
```

The system repeats the right argument as often as necessary until it equals the value of the left argument. If the number of elements in the right already exceed the amount specified by the left, the system will take only the prescribed amount.

```
      2ρ'ABCD'
AB
```

Notice in each of the above examples the left argument was always a nonnegative integer. Obviously anything other than that would not be acceptable. For instance, we could ask for three and a half numbers,

```
      3.5ρ10
DOMAIN ERROR
3.5ρ10
```

or for some negative amount,

```
      ¯3ρ10
DOMAIN ERROR
¯3ρ10
```

The system responds accordingly. This last example is roughly equivalent to trying to give some numbers back to the system.

When the left argument is a single value, the result is always a vector.

```
      4⍴'ABCDEFG'
ABCD

      10⍴'□'
□□□□□□□□□□
```

When the left argument is two values, the result is in the form of a 2-dimensional array.

```
      2 3⍴5
5  5  5
5  5  5
```

The number of elements contained in the left argument represents the rank of the result. The actual values of the elements of the left argument represent the dimensions or size the result is to assume. Above, the result produced is a 2 row, 3 column matrix. Here is another:

```
 rows      columns
      3  4⍴1 2 3 4 5 6 7 8 9 10 11 12
1   2   3   4
5   6   7   8
9  10  11  12
```

The result above is a matrix consisting of 3 rows and 4 columns. The left argument format is always the same.

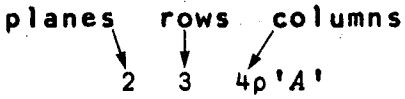
(rows, columns) ⍴ data

The right argument is used to develop the result, a row at a time.

For 3-dimensional arrays, the format is

(planes, rows, columns) ⍴ data

For example, the array generated below has 3 dimensions - 2 planes, each with 3 rows and 4 columns.




```
      2  3  4ρ'A'
AAAA
AAAA
AAAA

AAAA
AAAA
AAAA
```

Here is one with 4 planes, each with 2 rows and 3 columns:

```
      4 2 3ρ'∇*Δ'
∇*Δ
∇*Δ

∇*Δ
∇*Δ

∇*Δ
∇*Δ

∇*Δ
∇*Δ
```

This last one is another example of the 3-dimensional arrays. Actually these arrays can have up to 32 dimensions with each dimension holding up to 256 elements. But it is doubtful that you will be able to create such arrays, as the memory is not able to store them.

One array still to be covered is the scalar. Remember we said that if the left argument of the reshape function is a single value, the result is always a vector? This means even if we ask for only one number, it is still classified as a vector.

```
      1ρ7 8 9
7
```

To get scalar results the left argument must be two quotes placed side by side.

```
      ''ρ7 8 9
7
```

The result is still 7, but this time it is a scalar 7, even though there seems to be no visible difference between this 7 and the one in the previous example. To determine the difference. we must employ the rho symbol monadically.

Dimension Of

To create arrays of various shapes and sizes, we use the rho symbol monadically in the form *X*ρ*Y*. Once these arrays are created we can use this same symbol monadically to find out what these shapes and sizes are. For instance, to develop a four element vector, we could do the following:

```
      D←4ρ6 2
```

Then to determine its dimension, we would do this:

```
      ρD
```

to which the computer would respond with

```
4
```

indicating *D* consists of four elements. The result of this last example equals the left argument of the previous example.

Let us define a matrix and take its dimensions. First the creation,

```
      M←3 4ρ6
```

then the dimension taking.

```
      ρM
3 4
```

Again the result is the same as the left argument above. Notice also this result is a 2-element vector, just as the matrix *M* is a 2-dimensional array. When the dimension of *D* was found, the result was a 1-element vector because *D* is a 1-dimensional array. Therefore, we can easily determine the number of dimensions, or rank, of an array by

counting up the number of elements contained in the result of a dimension of function. In the example below, what is the rank of *P*?

```
      ρP
2  3  4
```

If you said 3, you are right, because the result has three elements. We could count these elements up each time, or we could employ the rho symbol to do it for us.

```
      ρ2 3 4
3
```

An easier way is

```
      V←ρP
■

      ρV
3
```

And an easier way yet is

```
      ρρP
3
```

This means performing two consecutive "rhos" on a variable determines its rank.

```
      ρM
2  3

      ρρM
2

      ρD
4

      ρρD
1
```

This seems to work fine for vectors and matrices, but what about scalars? Remember they have no dimensions. Therefore, the only log-

ical conclusion is that the system will return a vector result, but a vector that has no elements.

```
      ρ6
  (blank screen)
```

The system displayed nothing, indicating the result of ρ6 is an empty vector, or a vector containing no elements corresponding to an argument which has no dimensions. One way to determine if this is actually the case is to take the dimension of the result. If it has no elements, what do you think the result of the operation will be?

```
      ρρ6
0
```

Right. This is why in Chapter 3 we were able to say that scalars have a rank of 0.

We have not seen the last of the empty vector. It is referred to several times again, particularly in Chapter 15 concerning the branching ability within defined functions.

Index Of

One of the very useful mixed functions in the MCM/APL system is *X*⍳*Y*. The iota symbol ⍳ (upper shift *I*) is used to determine "where in the vector *X* is the first occurance of *Y*?".

```
      7 8 9⍳8
2
```

The result above gives the relative location of the 8 within the vector 7 8 9. We can see this is position number 2.

```
      'ABCDEF'⍳'DA'
4  1
```

Above *D* is the fourth element in the left argument and *A* is the first. In the case where the right argument is not in the left argument, the system returns a value which is one integer greater than the length of the left argument.

```
      'XYZ'⍳'A'
4

      'XYZ'⍳'AX'
4  1

      'THE BIG RED BARN'⍳'THREAD'
1  2  9  3  14  11
```

This function gives us the capability of comparing literal data just as we have with numerics. We saw that relational functions such as greater than (>) and less than (<) work with numeric data only. They tend to rule out any chance of testing the similarities between literals. But, by employing the index of function, we can get around this restriction. Our first step in doing this is to assign all the letters of the alphabet to a variable.

```
      ALF←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Then by using this as the left argument, we can easily determine the relationship between individual characters in the right argument.

```
      ALF⍳'AB'
1  2

      ALF⍳'ABA'
1  2  1

      ALF⍳'PAYROLL'
16  1  25  18  15  12  12
```

When we get into the actual indexing of arrays, we will see several applications where this function can be employed.

In the examples used so far, the arguments have been either scalars or vectors. This function can also accept arrays of greater rank, but, in all cases, the left argument must always be a vector. Here are a few examples where it is not.

```
      3⍳3
RANK ERROR
3⍳3

      A←2 2⍴⍳4
```

```
      A⍳2 4
RANK ERROR
A⍳2  4
```

The reason for the *RANK ERROR*'s is quite simple. In the first example, 3⍳3 , the left argument is a scalar and, as stated earlier, scalars have no magnitude or direction. Therefore, it is impossible to produce an index value for it. You will see further examples of this when we get to the section on indexing of arrays.

The second example failed because the system has no adequate means of expressing the various coordinates of each dimension. It could not return a result of

```
1;2  2;2
```

as this is not in keeping with the consistency of an all numeric result for this function. As a consequence, the left argument must always be of rank 1. This of course does not imply that the right must also adhere to the same rule. Quite the contrary. It can be of any rank and shape it likes. You will notice from the following examples that the rank and shape of the result is always the same as that of the right argument.

```
      2 4⍳A
3  1
3  2

      3 6 9⍳2 2 2⍴⍳8
4  4
1  4

4  2
4  4
```

Index Generator

In the examples so far that have involved arrays with ranks greater than zero, we have had to generate our own numbers. For instance, to create this variable,

```
      X←1 2 3 4 5 6 7 8 9 10
```

we have manually keyed in all the numbers and the required spaces. There are many instances like this one where the numbers involved are pretty straight forward, yet the task of entering them is fairly arduous. Now that we have come to this section, we should be able to simplify the job, for, as the title says, the function described here generates numbers, beginning at the index origin. Here is its format:

```
      ⍳Y
```

It generates a string of all the integers from one to *Y*.

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

      ⍳3
1  2  3
```

The index origin or starting point for both the monadic and the dyadic uses of the ⍳ symbol is preset to 1.

```
      2×⍳4
2  4  6  8
```

This can be changed to 0 by means of the system variable ⎕*IO* (index origin).

```
      ⎕IO←0
```

■

With it set to zero, here are a few more examples of both the monadic and dyadic iota functions:

```
      ⍳5
0  1  2  3  4

      77 78 79⍳77
0
```

Let us change it back to 1 as it is easier to understand the rest of the examples when they are executed in an origin 1 environment.

```
      ⎕IO←1
```

In the section Dimension Of, remember we discussed the empty vector. We were able to create it by taking the dimensions of a scalar. We could also have created one by using the dyadic rho and telling it to take none of the elements from its right argument.

```
      0⍴2 3 4
(blank screen)
```

Due to the fact that it can be created, we should have some means of representing it to the system and to other users. So MCM's definition of an empty vector is

```
      ⍳0
```

That's it. If you take the dimension of this expression you will see it is in fact equal to zero.

```
      ⍴⍳0
0
```

More about this little creature later.

So now we have learned what the rho (⍴) and the iota (⍳) symbols do. We can quickly generate numbers with the monadic iota

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

and we can reshape these numbers into a variety of structures.

```
      M←3 4⍴⍳12
```

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

```
      A←3 2 2⍴⍳12

      A
 1  2
 3  4

 5  6
 7  8

 9 10
11 12
```

We can determine the dimensions and rank of any variable by means of the monadic rho function.

```
      ⍴M
3 4

      ⍴A
3 2 2

      ⍴⍴M
2

      ⍴⍴A
3
```

And we can even pick out where, in a certain vector, lies the first occurance of some particular data.

```
      6.7 2.5 8.4⍳8.4
3
```

By knowing just these four functions, the number of applications possible increase tremendously. And once you have covered the next function described in this chapter, your scope for problem solving will have increased even further.

Indexing

Selecting specific data from an array is the function of the square brackets [ ]. To illustrate how this is done, assume we have a vector called *P* and in it there are 5 numbers.

```
      P
100   86   57   63   74
```

Assume too that, when *P* was defined, the third element was mistyped. It should have been 67 instead of 57. Instead of assigning all the numbers to *P* again and running the risk of another typing error, it would be preferable to just replace the one in error. To do this, here is what to type:

```
      P[3]←67
```

Upon displaying the contents of *P*, we see this was in fact done.

```
      P
100   86   67   63   74
```

If we want to select specific members of *P*, all we have to do is indicate their relative locations, or indices as they are called, and the system will get them for us.

```
      P[2 4]
86   63

      P[1 3 1]
100   67   100
```

The order in which the indices appear within the brackets dictates the order in which the values are indexed.

```
      P[5 1]
74   100
```

The indexing operation can even be a calculation.

```
      P[2+3]
74
```

And indexing variables can be used as arguments to other functions.

```
      6+P[5]
80
```

```
      P[1]+P[5]
174
```

They can even be used as arguments to other indexing operations.

```
      P[P[5]-73]
100
```

just as long as the indices do not exceed the dimensions of the arrays they are indexing.

```
      P[10]
INDEX ERROR
P[10]
```

If the array to be indexed is a matrix or a multidimensional array, we must distinguish to the system which planes, rows and columns we wish to index. For example, in the matrix below, suppose we want to index the third element in the second row, which is the letter *L*.

```
      MAT
MCM
APL
```

We must have some way of telling the system where this letter is. To do this, we state its coordinates, or points of relative location, in the following manner:

```
      [2;3]
```

The above example points to the element situated in row 2, column 3.

The sequence is always the same - the value representing the row is stated first and then the column's indice.

```
matrix[row;column]
```

Here is the entire function:

```
      MAT[2;3]
L
```

The coordinate value representing the row is always separated from the value representing the column by means of a semicolon.

To select the element in row 1, column 2, we type

```
      MAT[1;2]
C
```

To obtain all the letters in row 1, we could type

```
      MAT[1;1 2 3]
MCM
```

or, a much simpler expression is just

```
      MAT[1;]
MCM
```

When there are no values supplied for the specific coordinates of a dimension, the system assumes we mean all the coordinates for that dimension. In the above case, where no column coordinates were specified, the system interpreted this as meaning "extract the entire contents of row 1". To pick out the third element in the first row and the second element in the second row, the expression would be

```
      MAT[1 2;3 2]
MP
```

Now that we have found out how to index specific points within a vector and a matrix, can you determine how multidimensional arrays are indexed? Remember, they do not have just rows and columns, but also planes. There has to be some way of telling the system which plane, row and column you mean. For instance, the array *A1* has 3 dimensions.

```
      A1
101   102   103   104
105   106   107   108
109   110   111   112

201   202   203   204
205   206   207   208
209   210   211   212
```

It has 2 planes, each containing 3 rows and 4 columns, which can quickly be verified by the following function:

```
      ρA1
2  3  4
```

We want to find out what the contents are in the second row, third column of the first plane. How do you think it should be stated? If you said *A*1[2;3;1] you are almost right. The correct answer is *A*1[1;2;3] since the rows and columns figures are always the last two in any indexing operation. The sequence for specifying the indices of a 3-dimensional array is this:

```
array[plane;row;column]
```

For a matrix, it is simply

```
matrix[row;column]
```

and a vector is just

```
vector[element]
```

In the above array *A*1, the value 107 is situated in the second row, third column of the first plane.

```
      A1[1;2;3]
107
```

The entire second row of the first plane is

```
      A1[1;2;]
105  106  107  108
```

and the entire third column of the first plane is

```
      A1[1;;3]
103  107  111
```

The entire first plane is

```
      A1[1;;]
101   102   103   104
105   106   107   108
109   110   111   112
```

What would be the result of this expression?

```
      A1[;2;3]
```

Try it and find out.

To illustrate how new values are re-assigned to any of these indexed points, here are a few examples:

1: replacing the value 107 with a zero

```
      A1[1;2;3]←0
■

      A1
101   102   103   104
105   106     0   108
109   110   111   112

201   202   203   204
205   206   207   208
209   210   211   212
```

2: respecifying row 2 of the first plane to contain all 1's

```
      A1[1;2;]←1
■

      A1
101   102   103   104
  1     1     1     1
109   110   111   112

201   202   203   204
205   206   207   208
209   210   211   212
```

3: respecifying column 3 of plane 1 to contain ¯1 ¯2 ¯3

```
      A1[1;;3]←¯1 ¯2 ¯3
■

      A1
101   102    ¯1   104
105   106    ¯2   108
109   110    ¯3   112

201   202   203   204
205   206   207   208
209   210   211   212
```

4: respecifying all of plane 1 to contain the numbers 1 to 12

```
      A1[1;;]←3 4⍴⍳12
■

      A1
 1  2  3  4
 5  6  7  8
 9 10 11 12

201 202 203 204
205 206 207 208
209 210 211 212
```

You will find that by structuring data into arrays and indexing just those portions required, problems are not only simplified, but also the number of variables required is reduced. As you become more adept with MCM/APL, you will find yourself using arrays more and more in your applications because anything you can do with individual variables, you can also do with larger arrays, with the aid of indexing in many cases.

We have by no means covered all the applications that can be done by using indexing. They are almost as numerous as you care to make them. You will see a few of them in examples contained in descriptions of some primitive mixed functions still to be discussed in the next chapter and in the practice exercises.

Practice Exercises

1. Using the MCM/70, evaluate the following:

(a)	`⍳10`	(b)	`2×⍳6`	(c)	`¯3+⍳7`
(d)	`⍴⍳10`	(e)	`⍴1`	(f)	`⍴⍳1`
(g)	`(2×⍳6)[3]`	(h)	`(⍳10)⍳10`	(i)	`(2+⍳10)[5 10 5]`
(j)	`⍴3 4⍴⍳12`	(k)	`⍳0`	(l)	`⍴⍳0`

2. If $M \leftarrow 3\ 4\rho 10+\iota 12$, evaluate the following:

(a) $M[1;2]$	(b) $M[2;1]$	(c) $M[1;2\ 3]$
(d) $M[3;]$	(e) $M[1\ 2;1\ 2]$	(f) $\rho M[1\ 2;1\ 2]$
(g) $M[;3]$	(h) $M[3\ 2;]$	(i) $\rho M[3\ 2;]$
(j) $M[12-M[1;1];]$	(k) $M[;10\mid M[1;]]$	(l) $M[;]$

3. (a) Create an array having 2 planes, each with 3 rows and 4 columns, and containing the numbers 201 to 224 consecutively.

   (b) Replace the value in the second plane, third row, first column with the number 36.

   (c) Multiply the entire first plane by 10.

   (d) Place zeros in the first row of the second plane.

4. (a) Find the indices of the negative numbers in the vector V.

```
          V
6  ¯2  2.3  7  ¯5.2  0
```

   (b) Using a single APL statement, replace all negative values in V with the number 10.

Chapter 12: MORE MIXED FUNCTIONS

This chapter describes eight more functions belonging to the mixed function group. They bear little relationship to one another, but can collectively be used to perform some rather sophisticated operations. The remaining mixed functions yet to be discussed are contained in Chapters 18 and 19.

Membership

Earlier we saw the index of function, which states where, in the left argument, the first occurance of the right argument is.

```
      2 6 7 5⍳7
3
```

There is also a function that answers the question "Is the following data to be found within the argument?". The syntax for this function is

A∈*B*

and is read as "Is *A* a member of *B*?". But unlike the index of function which returns the right argument's relative location within the left argument, the membership function ∈ (upper shift *E*) returns a "yes-no" response of 1 or 0, just as the relational and logical functions did.

```
      7∈2 6 7 5
1
```

Because the result is 1, we know the value 7 is a member of the right argument.

```
      3 6∈2 6 7 5
0 1
```

The 3 is not a member, but the 6 is.

Notice the size and shape of the result is the same as that of the left argument, which is not the case for the index of function.

```
      (2 2⍴⍳4)∈4 1
1   0
0   1
```

Its left argument must be a vector, whereas the membership's argument can be of any rank.

```
      (2 3⍴⍳6)∈3 3⍴¯2 6 3 0 7 12 9 2.5 1
1   0   1
0   0   1
```

The arguments can even be literals.

```
      'THAT'∈'HELLO THERE'
1  1  0  1
```

Literals and numerics can also appear together, although

```
      '3'∈3 4
0
```

the result will always be 0's.

Grade Up

If an occasion arises when you would like to have a certain set of numbers re-arranged into ascending or descending sequence, you could write a 4 or 5 line function, or you could employ either the grade up or the grade down, in conjunction with the indexing function. For instance, assume you wanted to display the values 7, 4, 1, 9 and 6 in ascending sequence, you would first assign them to a variable,

```
      X←7 4 1 9 6
```

and then perform the following operation to get the displayed result:

```
      X[⍋X]
1  4  6  7  9
```

(The ⍋ symbol is a combination of the upper shift *H* symbol Δ overstruck with the upper shift *M* symbol |)

Let us take a closer look at the expression above. Because two functions are involved, it is obviously a 2-step operation. The first operation performed is the ⍋*X*. Here it is by itself. See if you can determine what it does.

```
      ⍋X
3  2  5  1  4
```

Can you quess? Notice the numbers displayed range from 1 to 5 but are arranged in a rather random order. Can you see anything common between the values and their respective positions in the displayed vector with those of *X*? Really it is quite simple. The result of 3 2 5 1 4 is saying that the third element of *X* is the lowest value, the second is the second lowest, the fifth is the middle value, the first is the second highest and the fourth is the highest. If we indexed *X* with this vector, what do you think the result would be?

```
      X[3 2 5 1 4]
1  4  6  7  9
```

That's right. It is the values of *X* displayed in ascending sequence. Therefore, the function ⍋*X* produces the indices of the values of *X* in the order dictated by the values of the elements in *X*.

Here are a few more examples:

```
      ⍋¯2 ¯3 ¯4
3  2  1

      ⍋3 3 3 3
1  2  3  4
```

Notice that if there are any elements with the same value, the system ranks them according to their relative positions within the argument.

```
      ⍋'ABC'
DOMAIN ERROR
⍋'ABC'

      ⍋2 2⍴⍳4
DOMAIN ERROR
⍋2 2⍴⍳4
```

The arguments of both the grade up and the grade down functions are limited to numeric vectors, but there are several user defined functions available to allow you to overcome these restrictions if you need to.

Grade Down

As you have probably quessed, the grade down function ⍖ (⍒ overstruck with |) does essentially the same thing as the grade up. However, its result contains the indices arranged to denote the values of the argument according to their respective magnitudes, with the index of the largest value appearing first.

```
      X
7  4  1  9  6

      ⍖X
4  1  5  2  3

      X[⍖X]
9  7  6  4  1

      ⍖¯2 ¯3 ¯4
1  2  3

      ⍖3 3 3 3
1  2  3  4
```

It also treats elements with the same values identical to the way the grade up function does. Their respective locations become the deciding factor.

Catenate

One of the properties of most of the mixed functions is the ability to manipulate the sizes and shapes of arrays. Some generate arrays, others simply alter their existing dimensions, and still others expand or diminish the defined arrays. One that does expand the sizes of arrays is the catenate function , . Its syntax is

```
      X,Y
```

where *Y* is appended to *X* to produce a result which contains both the elements of *X* and those of *Y*. For example, if we wished to take two vectors and join them together, we would do it like this:

```
      2 4 6,3 5 7
2 4 6 3 5 7
```

The two vector arguments above each have 3 elements. Therefore the result is a 6-element vector.

```
      A←2 4 6,3 5 7
■
      ⍴A
6
```

If we had forgotten to include a number in the above statement, say it was a 10, we could just do

```
      A←A,10
■
```

which would append 10 to the values already represented by *A*, thus avoiding the problem of having to retype all the numbers in again.

```
      A
2 4 6 3 5 7 10
```

What if we missed a number somewhere in the middle? Perhaps there should be an 8 between the 6 and the 3. Any ideas? Well, how about this?

```
      A←A[⍳3],8,A[3+⍳4]
■
      A
2 4 6 8 3 5 7 10
```

The same operation could have been written as

```
      A←A[1 2 3],8,A[4 5 6 7]
■
```

It is just that the former one saves a few key strokes. Here are a few more examples:

```
      2,4 6.7 11
2  4  6.7  11

      2,3,4,5,6
2  3  4  5  6

      ¯4 2,8,7 21 3,9
¯4  2  8  7  21  3  9

      'M','I','C','R','O'
MICRO

      'MONTHLY',' ','STATEMENT'
MONTHLY STATEMENT
```

This function comes in handy whenever entering a string of data that is too long to be entered on one line. Say there were 100 numbers to enter. They could not all fit on the same input line. But we could enter as many as possible at a time, then tack on the rest on the next line. Here is a small sample to show how this is done:

```
      N←2 4 6 8 10 12
■

      N←N,14 16 18 20
■
```

After the second line has been entered, *N* contains all ten numbers, just as if they had all been typed in together.

```
      N
2  4  6  8  10  12  14  16  18  20
```

Besides catenating scalars and vectors, we can also join matrices and multidimensional arrays together. Here is an example of two matrices, *X* and *Y* being joined:

```
      X←2 2ρ3
```

```
      Y←2 3ρ6
```

```
      X,Y
3  3  6  6  6
3  3  6  6  6
```

Here is how we can add one more column to a matrix:

```
      X,9
3  3  9
3  3  9
```

or one more row:

```
      X,[1]9
3  3
3  3
9  9
```

The [1] indicates to the system the catenation is to be performed along the first coordinate of the left argument (in this case, the rows).

If neither of the arguments is a scalar, the arrays involved must be of equal rank and the corresponding dimensions of the coordinates not being joined together must be equal. Here are a few more examples:

```
      A←2 3 4ρ'A'
```

```
      B←2 3 5ρ'B'
```

```
      A,B
AAAABBBBB
AAAABBBBB
AAAABBBBB

AAAABBBBB
AAAABBBBB
AAAABBBBB

      ρA,B
2  3  9
```

```
      C←1 3 4⍴'C'
■

      A,[1]C
AAAA
AAAA
AAAA

AAAA
AAAA
AAAA

CCCC
CCCC
CCCC
      ⍴A,[1]C
3  3  4

      D←2 1 4⍴'D'
■

      A,[2]D
AAAA
AAAA
AAAA
DDDD

AAAA
AAAA
AAAA
DDDD

      ⍴A,[2]D
2  4  4
```

If catenation is going to occur along the first coordinate only, we can use the expression `A,[1]C` or we can use this:

```
      A⍪C
AAAA
AAAA
AAAA

AAAA
AAAA
AAAA

CCCC
CCCC
CCCC
```

The symbol above is a combination of the comma and the minus sign. When they are used together, catenation occurs along the first coordinate only. The minus sign used here is also used with several other functions to represent first coordinate operations. One of these is covered in the next chapter.

Ravel

When the comma is used monadically, it is called the ravel function. Its purpose is to turn its argument into a vector result.

```
      ,2 2⍴⍳4
1  2  3  4

      ⍴,2 2⍴⍳4
4
```

The length of the result is always the product reduce (×/⍴arg.) of the dimensions of the argument.

```
      M
BIG
BAD
JOE

      ⍴M
3  3

      ×/⍴M
9

      ,M
BIGBADJOE

      ⍴,M
9
```

Notice ravelling of the matrix is done a row at a time. Arrays of greater rank are ravelled a plane at a time. This is the same sequence used by the dimensioning function ⍴.

One other point here is this function also offers an easy way of converting a scalar into a one element vector.

```
      ρ5
 (blank screen)

      ρ,5
1
```

This little fact may prove of assistance when performing calculations requiring vector arguments. One of these is the dyadic ⍳ function. Its left argument must be a vector.

Take

The take function ↑ (upper shift *Y*) literally takes data from its right argument according the amount(s) specified in the left argument.

```
      3↑2 6 ¯4 7 5
2  6  ¯4

      1↑27 26 15
27
```

In the first example above, the system was asked to take the 3 leading elements of the right argument, and the second example asked it to take only the first element of its right argument. The number of elements in the left argument must always equal the rank of the right.

```
      M
101   102   103
104   105   106
107   108   109
110   111   112

      2 2↑M
101   102
104   105
```

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

In the above, the system was asked for only the first two rows of the first two columns of *M*.

It also works with literal data.

```
      3↑'HISTORY'
HIS

      L
HADDOCK
HAMBURG

      2 3↑L
HAD
HAM
```

Apart from "taking" from the front of the right argument, we can also take from the end. This is done by negating the left argument.

```
      ¯3↑2 6 ¯4 7 5
¯4  7  5

      ¯1↑27 26 15
15

      ¯5↑'HISTORY'
STORY

      ¯2 ¯4↑L
DOCK
BURG
```

Drop

The drop function ↓ (upper shift *U*) is the reverse of the take.

```
      2↓'ABCDE'
CDE

      2↓'HISTORY'
STORY
```

If the left argument is positive the result is any data that remains in the left argument after the prescribed amount has been dropped off.

```
      3↓10 21 16 32 27
32  27
```

If the system is asked to drop off more data than is provided, the result is an empty array.

```
      4↓7 8
 (blank screen)

      ⍴4↓7 8
0
```

This function also works with larger arrays.

```
      M
101   102   103
104   105   106
107   108   109
110   111   112

      2 2↓M
109  112
```

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

If this last answer is confusing, look at the accompanying schematic.

Here are two more:

```
      0 3↓L
DOCK                    HADDOCK
BURG                    HAMBURG
```

and

```
      0 ¯4↓L
HAD                     HADDOCK
HAM                     HAMBURG
```

Dyadic Random (deal)

Remember earlier we discussed the properties of the monadic random function? Its use is to select an integer, at random, from 1 to *N* where *N* represents the right argument. When this same symbol, *?*, is used to form a dyadic function, it is called the dyadic random and performs in a similar fashion to the monadic random. However, instead

of selecting just one integer, it can select several. The quantity to be selected is specified by the left argument and the population range is represented by the right.

```
      3?10
1  8  6

      3?10
2  5  3

      3?10
4  7  8
```

Notice there are never any integers the same in each of the results. Selection is always done without replacement.

Practice Exercises

1. Generate ten random numbers between 1 and 50 and find out which ones fall between 10 and 20. (use the membership function.)

2. Write a function to sort alphabetic strings into ascending sequence.

3. Use the take and drop functions to extract the word *LIKE* from the sentence

   THE LAKE WAS LIKE A SHEET OF GLASS

4. Using the following matrix, re-arrange the rows by sorting the last column into ascending sequence.

```
      A
SCREWS
JIGSAW
PLANER
PLIERS
WRENCH
```

The result should be

WRENCH
PLANER
SCREWS
PLIERS
JIGSAW

5. Repeat the process in question 4, working from the last column to the first. Once you have covered "branching", you should be able to develop a function to do this.

Chapter 13: REDUCTION AND SCAN WITH MULTIDIMENSIONAL ARRAYS

In Chapter 7 we saw how the reduction and scan functions work. The arguments used in every example were vectors. But, like most of the MCM/APL functions, these two can also accept arguments of greater ranks. Multidimensional arrays were not included in Chapter 7. Instead left for this chapter to allow you to become more familiar with higher ranking arrays and how they are formed.

Reduction

Just to refresh your memory, here is a plus reduction of a vector:

```
      +/⍳9
45
```

If this vector is reshaped into a matrix and the same function performed, we get the following:

```
      A←3 3⍴⍳9
```

```
      +/A
6  15  24
```

Our matrix looks like this:

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

and the result is calculated like this:

[1 + 2 + 3] = 6

[4 + 5 + 6] = 15

[7 + 8 + 9] = 24

Here is another:

```
      ×/A
6  120  504
```

which is calculated as

```
[1 × 2 × 3] =   6
[4 × 5 × 6] = 120
[7 × 8 × 9] = 504
```

As with vectors, the order of execution is still from right to left.

```
      +/A
1.5  4.8  7.875
```

which is

```
[1 + 2 + 3] = 1.5
[4 + 5 + 6] = 4.8
[7 + 8 + 9] = 7.875
```

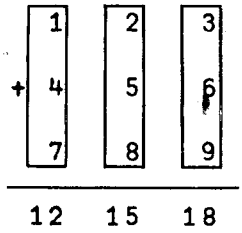
It appears as if the reduction is performed along the rows, or first dimension of the array. This is true in a way. The reduction is actually done along the corresponding elements of *A*'s columns, or second dimension. The first element in column one is divided by the first element in column two, after it has been divided by the first element in column three. Even though it is defined this way, you should think of it in which ever way is easiest to remember. The confusion occurs when reduction is performed along the rows. For instance, to sum along the first dimension, the expression is

```
      +/[1]A
```

and the result is

```
12  15  18
```

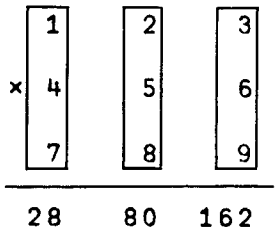
The [1] means the reduction is to be performed along the corresponding elements in each of the rows of *A*.



Here is another:

```
      ×/[1]A
28  80  162
```

which is



Since we can specify the first coordinate with this function, we should also be able to specify the second.

```
      +/[2]A
6  15  25
```

which is the same as

```
      +/A
6  15  25
```

or columns coordinates, as *A* has only two dimensions.

If the array is of greater rank, this same feature applies.

```
      P←2 3 4⍴⍳24
```

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

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

      +/P
10  26  42
58  74  90
```

or

```
 1 + 2 + 3 + 4 = 10
 5 + 6 + 7 + 8 = 26
 9 + 10 + 11 + 12 = 42

13 + 14 + 15 + 16 = 58
17 + 18 + 19 + 20 = 74
21 + 22 + 23 + 24 = 90
```

The shape of the result above is a 2 by 3 matrix. This shape is determined by the coordinate along which the reduction takes place. The result has all the dimensions of the argument except the one which is being "reduced". Above, this is the columns, which has a dimension of 4.

This last example could also have been written as

```
      +/[3]P
```

since the third coordinate represents the columns.

Reduction along *P*'s rows is done by referring to the second coordinate.

```
      +/[2]P
15  18  21  24
51  54  57  60
```

and reduction along P's first coordinate is

```
      +/[1]P
14   16   18   20
22   24   26   28
30   32   34   36
```

or we could use the symbol ⌿ (/ overstruck with the minus sign -) to do the same thing.

```
      +⌿P
14   16   18   20
22   24   26   28
30   32   34   36
```

The ⌿ symbol is identical in meaning to the symbols /[1]. Reduction always occurs along the first coordinate.

```
      ×⌿A
28  80  162
```

Reduction Example

For five days, three boys harvested the following number of bushels of apples:

	Mon.	Tues.	Wed.	Thurs.	Fri.
Greg	10	12	11	12	9
Mike	9	13	14	11	10
Jim	11	11	13	14	9

To evaluate the boys' performance, the man who owns the orchard wants to know the following three things:

1. How many bushels did each boy pick?
2. How many bushels were picked each day?
3. What was the total number of bushels picked?

The first step in finding the solutions is to assign the bushel counts to a variable.

```
      B←10 12 11 12 9 9 13 14 11 10 11 11 13 14 9
```

The next step is to reshape this variable into a matrix similar to the one above.

```
      B←3 5⍴B
```

Then solve the three problems.

```
      +/B                solution to question 1
54  57  58

      +⌿B                solution to question 2
30  36  38  37  28

      +/+/B              solution to question 3
169
```

Scan

The scan function works in an identical manner to the reduction. Using the same arrays *A* and *P*, here are a few scan examples. First, a display of the contents of *A* and *P*:

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

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

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

```
      +\A
 6     5     3
15    11     6    along the second coordinate
24    17     9

      +⍀A
12    15    18
11    13    15    along the first coordinate
 7     8     9

      -\P
¯2     3    ¯1     4
¯2     7    ¯1     8
¯2    11    ¯1    12
                      along the last coordinate
¯2    15    ¯1    16
¯2    19    ¯1    20
¯2    23    ¯1    24

      -\[2]P
 5     6     7     8
¯4    ¯4    ¯4    ¯4
 9    10    11    12
                      along the second coordinate
17    18    19    20
¯4    ¯4    ¯4    ¯4
21    22    23    24

      -⍀P
¯12   ¯12   ¯12   ¯12
¯12   ¯12   ¯12   ¯12
¯12   ¯12   ¯12   ¯12
                           along the first coordinate
 13    14    15    16
 17    18    19    20
 21    22    23    24
```

Scan Example

The ABC Company has 5 branches spread across Canada. At the end of the fiscal year, head office receives balance reports from each of the branches indicating their net income for each of the 12 past months. Here are their figures, including their respective balances at the end of the previous year:

	Montreal	Toronto	Winnipeg	Edmonton	Vancouver
Last Year's Bal.	1,190	1,850	755	925	1,025
Month end Bal.'s					
January	75	92	47	55	72
February	85	99	52	59	81
March	93	116	56	64	89
April	115	132	62	77	100
May	105	147	68	89	114
June	127	152	71	101	112
July	128	155	79	113	115
August	103	135	72	110	107
September	96	127	61	102	102
October	88	112	63	91	97
November	86	101	63	89	92
December	72	98	52	82	83

(figures are in thousands)

Head office wants to know the balance figures at the end of each month for each of the branches, using last year's closing balance figures as a base.

The first step in solving this is to store the figures into a matrix.

BAL

72	98	52	82	83
86	101	63	89	92
88	112	63	91	97
96	127	61	102	102
103	135	72	110	107
128	155	79	113	115
127	152	71	101	112
105	147	68	89	114
115	132	62	77	100
93	116	56	64	89
85	99	52	59	81
75	92	47	55	72
1190	1850	755	925	1025

Notice the contents of *BAL* form an inverted matrix to that of the previous figures. The reason for this will become apparent shortly.

The next step is the scanning.

```
      +\BAL
2363   3316   1501   1957   2189
2291   3218   1449   1875   2106
2205   3117   1386   1786   2014
2117   3005   1323   1695   1917
2021   2878   1262   1593   1815
1918   2743   1190   1483   1708
1790   2588   1111   1370   1593
1663   2436   1040   1269   1481
1558   2289    972   1180   1367
1443   2157    910   1103   1267
1350   2041    854   1039   1178
1265   1942    802    980   1097
1190   1850    755    925   1025
```

Scanning is conducted from the bottom to the top of the matrix. This is consistent with our right-to-left execution rule.

Practice Exercises

1. In June 1970, the population of the ten provinces and two territories of Canada were as follows:

Nfld.	P.E.I.	N.S.	N.B.	Que.	Ont.	Man.	Sask.	Alta.	B.C.	Yukon	N.W.T.
517	110	782	627	6013	7551	983	941	1595	2128	17	33

(figures are in thousands)

In 1971 and 1972, the population increased by the following amounts:

	1972	1971
Nfld.	10	5
P.E.I.	1	2
N.S.	5	7
N.B.	7	8
Que.	31	15
Ont.	122	152
Man.	4	5
Sask.	-10	-15
Alta.	27	33
B.C.	62	57
Yukon	1	1
N.W.T.	1	2

Construct a 12 by 3 matrix like this:

10	5	517
1	2	110
5	7	782
7	8	627
31	15	6013
122	152	7551
4	5	983
-10	-15	941
27	33	1595
62	57	2128
1	1	17
1	2	33

and find the population amounts for each of the provinces and two territories for June of 1971 and 1972. Calculate the total population of Canada for each of the three years.

Chapter 14: COMPRESSION - EXPANSION

MCM/APL offers several functions that allow you to select specific elements from existing arrays. These functions include take, drop, restructure, and indexing. Each performs quite differently, but they all do essentially the same thing. There are two more functions that also fall into this category. They are compression and expansion. Their formats are the following:

Compression	Expansion
V/[*N*]*A*	*V*\[*N*]*A*

Notice the compression function uses the solidus symbol /, used also by the reduction function, and the expansion employs the reverse solidus \, used by the scan. The in both cases is a logical vector of 1's and 0's; the *N* indicates along which coordinate the function is being applied; and the *A* is the array to be compressed or expanded.

Compression

Here is a typical example showing how the compression function works:

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

The left argument 1 0 1 above directs the system to select only the first and third elements of the right argument. The left argument's first element, 1, tells the system to select its counterpart in the right. The second element, 0, tells the system to ignore its right argument counterpart, and the third element, 1, asks for its counterpart be selected, giving us a result of 6 4. Here are a few more examples:

```
      1 1 0 1/2 ¯4 7 5.2
2  ¯4  5.2

      0 0 1/3 27 81
81

      0 0/2.2 6
 (blank screen)
```

The result of this last example is an empty vector since the function asked for none of the contents of the right argument to be selected.

When dealing with arrays of rank greater than 1, entire planes, rows and columns can be omitted.

```
      M←3 4⍴⍳12
```

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

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

or

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

The [2] indicates the compression is to occur along *M*'s second coordinate, which in this case is the columns or last dimension, as *M* has only two. When this pointer is omitted, the system automatically defaults to the last dimension. This is why the above two compressions produced identical results.

In order to eliminate one or two of the rows from this matrix, we would use [1] to indicate this to the system.

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

Seeing that we are dealing with the first dimension, we could use the symbol ⌿ as we did with the reduction function.

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

When ⌿ is used, the system always assumes the compression is along the first dimension. If the right argument is a 3-dimensional array, it

would be the planes that are compressed. If it is a matrix argument, the rows get compressed. If the argument is a vector, the ⌿ performs identically to the /, since vectors have only one dimension anyway.

```
      0 1 0⌿2 3 4
3

      0 1 0/2 3 4
3
```

Here is a hypothetical problem which could be solved by using the compression function:

> At the end of a semester, a teacher wanted to find out which students in his class attained honours standings. He had given three tests during the term. Their marks were as follows:
>
> | Name | Test 1 | Test 2 | Test 3 |
> |---|---|---|---|
> | Mannen | 25 | 26 | 20 |
> | Burden | 17 | 24 | 18 |
> | Phelps | 24 | 33 | 21 |
> | Duncan | 20 | 25 | 17 |
>
> (He had a very small class)
>
> The first and the third tests were out of 30 and the second test was out of 40, which meant a perfect score would be 100. He was interested in those with 75 and over.

With this data, he created two matrices - one called *MRK* and the other called *NMS*. *MRK* contained the students' marks while *NMS* contained the corresponding students' names.

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

      NMS
MANNEN
BURDEN
PHELPS
DUNCAN
```

He then summed up the marks for each student and stored them in a variable called *SUM*.

```
      SUM←+/MRK
```

```
      SUM
81  59  78  62
```

The next thing he did was to compare the values contained in *SUM* to the honour indicator 75 to see which ones were equal to or above this level.

```
      75≤SUM
1  0  1  0
```

Since there are so few students involved, he could have easily determined this just by looking at the contents of *SUM*. However, as a class would normally contain a few dozen students, a "manual" method of comparison would not pnly be lengthy, but also suseptible to errors. So it is always best to perform a "computer comparison" anyway.

Here is the same operation again, but with the result being assigned to a variable:

```
      TOP←75≤SUM
```

Now, all he has to do is use *TOP* to perform a compression on *NMS* to get the names of his honour students.

```
      TOP/NMS
MANNEN
PHELPS
```

He could have done this entire calculation using only one statement.

```
      (75≤+/MRK)/NMS
MANNEN
PHELPS
```

Besides performing logical compressions on data, this function is also used to perform "branching" within user defined functions, as we will see in the next chapter.

Expansion

The expansion function is very similar to compression. It has the same properties, except that instead of compressing its right argument, it increases its size. Here is an example:

```
      1 0 0 1\6 7
6  0  0  7
```

The result is an expansion of the right argument. The system uses zeros to indicate where expansion took place.

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

If the expanded array is a literal, blanks or spaces are used.

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

The expanded results retain the same properties as their respective arguments. Only their sizes are different. If the argument is numeric, so too is the result.

```
      1 0 1\2 3
2  0  3
```

```
      10+1 0 1\2 3
20  10  30
```

The size of the result is determined by the length of the left argument.

```
      X←3 4⍴⍳12
```

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

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

```
      1 1 0 1 1\'INTO'
IN TO
```

```
      Y
MCM
APL

      1 0 1 0 1\Y
M C M
A P L
```

To increase the number of rows in *X* and *Y*, the expansion must occur along the first coordinate. Either the [1] or the ⍀ can be used to accomplish this.

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

      1 0 0 1⍀Y
MCM

APL
```

Arrays of greater rank can be expanded in the same manner.

One final note. The number of 1's in the left argument must equal the dimension along which the right argument is to be expanded.

Practice Exercises

1. Evaluate the following:

   (a) `1 0 1/2 3 4`

   (b) `1 0 1\2 3`

   (c) `0/3`

   (d) `0\3`

   (e) `(A<6)/A←⍳10`

   (f) `(0,(×|A),0)\A←¯2 6 1.1 ¯7`

2. Try these two expressions:

```
      0\⍳0
```

and

```
      0\''
```

The first one returns a result of 0 and the second one returns a single space, or blank character. To prove it, try

```
      ' '=0\''
```

The result is 1. If the same comparison is conducted on 0\⍳0 the result is 0.

Knowing that 0⍴ of a literal returns an empty literal vector, '', and 0⍴ of a numeric returns an empty numeric vector ⍳0, write a function which has the header line

```
      ∇R←LORN X
```

to determine if *X* is a literal or a numeric.

3. Compress the first and the third rows out of the matrix

```
      A←3 4⍴⍳12
```

Expand the result back up to the same dimensions as *A*, but with the compressed result occupying the first row and the other two rows being filled with zeros.

4. In the following matrix, select all the rows that begin with the letter *P*.

```
      F
PEACHES
APPLES
PEARS
CHERRIES
ORANGES
PLUMS
```

5. "Blank out" those rows in *F* which do not begin with *P*. The result should look like this.

```
PEACHES

PEARS


PLUMS
```

Chapter 15:

BRANCHING

In Chapter 8 we created a function called *ATTEND* which found the average, lowest value, highest value, and the range between the lowest and highest for any set of numbers. Chapter 9 explained all the various types of functions that can be defined within the MCM/APL system. The examples used in both of these chapters were functions that contained only one or two lines each. They proved adequate enough to explain how functions could be defined and what the six different function types are, but the number of lines each of them contained was hardly representative of the length of functions you will be writing. Yours will probably contain anywhere from 5 to 20 or more statements, depending on their complexity and purpose. Initially, their lengths may be due to your limited exposure to the MCM/APL system, but as you gain experience, you will find short cuts in expressing your solutions and thus be able to reduce the number of lines required. One of these short cut techniques is the ability to branch to various lines within the function while it is executing.

Types of Branches

In all the multilined defined functions seen so far, the statements contained in each have been executed in a straightforward fashion. The contents of line 1 were computed before those of line 2, which were done before line 3, and so on to the end of the function. But, in many computer applications, things do not always run this smoothly. You may want to execute a certain set of statements depending on the prevailing conditions at the time. Or you may want to "loop through" a specific set of instructions several times, changing the data and/or the parameters slightly each time. But whatever the reason, the branch feature of the MCM/APL system allows you to redirect the computer's execution anywhere throughout the body of your function. The symbol used to indicate a branch operation is the right pointing arrow →. There are two types of branches possible. One is called the conditional branch and the other is the unconditional branch.

Unconditional Branch

The term "unconditional branch" means the system is to perform the branch everytime it is encountered. The format is

```
→LOCATION
```

The → symbol signifies a branch is to take place and the line location or number to which the branch is being made appears to its right. This target line is either a line number or a label. To begin with, we will use line numbers in all examples. Labels are discussed later on. Here is a typical branch instruction:

```
      .
      .
      .
[6]→3
      .
      .
      .
```

If the above statement was an actual line within a function, the computer would always redirect its execution back to line 3 whenever encounters line 6. Below, if this line is executed by the system, computation would be automatically rerouted to line 10, as the value, which ultimately becomes the argument of the branch function, is the value 10 (i.e., 5×*A*←2).

```
[4]→5×A←2
```

This means you can include other calculations on the same line as a branch instruction.

Conditional Branch

A "conditional branch" is one that may or may not take place, depending on the circumstances. Below, the function *RANK* has 3 conditional branches in it; one on line 1, another on line 2, and still another on line 3. The other branches you will recognize as being unconditional. But, within these unconditional branch statements there is something we have not seen before. This is a branch to a non-existent line number. In each case it is line 0.

```
 ∇RANK A
[1]→(0=ρρA)/6
[2]→(1=ρρA)/8
[3]→(2=ρρA)/10
[4]'MULTIDIMENSIONAL ARRAY'
[5]→0
[6]'SCALAR'
[7]→0
[8]'VECTOR'
[9]→0
[10]'MATRIX'
 ∇
```

Why then are we branching to this line? The reason is quite simple. The system has been programed to interpret any branches to non-existent lines as meaning "terminate execution of the function immediately." Line 0 was chosen because it will never appear in the body, whereas some other number may. Actually, any integer which is not a valid line number will do. This rule applies to both kinds of branches. We did see earlier the system referred to line 0 as being the header line whenever we performed any function editing. But the header line is not classified as part of the function's body. Therefore, branching to it does in fact cause termination of the function's execution.

Here is how *RANK* works:

```
      RANK 86
SCALAR
```

This is what happened. When the first line of *RANK* was encountered, the system first evaluated the contents between the parentheses, which are

```
      0=ρρA
```

When *A* represents the number 86, the result of this operation is a 1. (A scalar has a rank of 0, therefore ρρA results in 0, and 0=0 results in 1). Once this calculation is completed our statement looks like this:

```
      [1]→1/6
```

Notice the expression 1/6 is a compression function which returns a result of 6.

Here it is outside the statement:

```
      1/6
6
```

This means line 1 is reduced to 6 which is the same as our unconditional branch. Just to retrace the execution of this line, here is a breakdown of the various steps:

```
      [1]→(0=ρρA)/6
      [1]→   1   /6
      [1]→      6
```

After evaluating line 1, the system then jumped down to line 6, displayed the statement *SCALAR*, and then went onto line 7, which instructed the system to conclude its execution of *RANK* (branch to line 0).

Below *RANK* is executed again, this time using a 3 element vector as its argument.

```
      RANK 23 64 17
VECTOR
```

Looking back at the function *RANK* itself, it is line 2 that directs the computer down to line 8 where the statement '*VECTOR*' is located. This meant when *RANK* was executed, the branch on line 1 did not occur, but the one on line 2 did. Why? Well the comparison operation 0=ρρ*A* on line 1 resulted in a 0 as *A* represented a 3-element vector which has a rank of 1. This meant the statement on line 1 was reduced to

```
      [1]→0/6
```

We know 0/6 always returns an empty vector result. Therefore, whenever the system is asked to branch to an empty vector, it merely ignores the request. For instance,

```
      [1]→⍳0
```

causes no effect on the order in which the lines of the function are executed. The system simply goes on to line 2 and continues its evaluation of *RANK*. Here is a breakdown of the operations as they occured on line 1:

```
      [1]→(0=ρρA)/6
      [1]→   0   /6
      [1]→      ⍳0
```

Line 2 is evaluated the same way line 1 was when *A* represented a scalar. Because a compression operation is used in all three conditional branches and all three are comparing the rank of *A*, the function ***RANK*** could have been written more concisely. Below is a revised definition of ***RANK***.

```
     ∇RANK A
[1] →(0 1 2=ρρA)/4 6 8
[2] 'MULTIDIMENSIONAL ARRAY'
[3] →0
[4] 'SCALAR'
[5] →0
[6] 'VECTOR'
[7] →0
[8] 'MATRIX'
     ∇
```

Here is an example of it in use:

```
      RANK 2 2ρ⍳4
MATRIX
```

Because *A* represents a 2 by 2 matrix, it has a rank of 2. This causes the comparison operation within the parentheses

```
0 1 2=ρρA
```

to produce a result that looks like this:

```
0  0  1
```

So the expression gets reduced to this:

```
[1] →0 0 1/4 6 8
```

We know that

```
0 0 1/4 6 8
```

gets compressed to 8, therefore the result of this statement is

```
[1] →8
```

and the branch is completed.

Just make sure the numbers 4 6 8 representing the target lines of the different branches are in the same sequence as the values 0 1 2 representing the various ranks of the arrays entered. This insures branching is to the appropriate line.

Conditional branches can be expressed in various ways. Here are just a few of them.

```
→(X op Y)↑N

→(X op Y)⍴N

→N×⍳(X op Y)
```

The character *N* represents the line number or receiving target of the successful branch, and "op" stands for any one of the symbols

< ≤ = ≥ > ≠ ∨ ∧ ⍱ ⍲ ∈

"op" could also be a user defined function which returns an explicit result of 1 or 0.

Labels

You have just seen how the computer can be directed to any line in a function by means of the branch arrow → . By saying →6 the system automatically goes to line 6 to resume its execution of the function. But what happens when you do some function editing and the contents of the original line 6 now reside on some other line? When new lines are inserted and old ones deleted, the system renumbers them again to reflect any changes. Therefore, branching to specific line numbers may not be such a good idea if there is a chance the function may undergo some alterations in the future. So, to avoid this possibility, MCM/APL has incorporated line labels which, in effect, operate the same as variables. This means that instead of saying

```
→6
```

we could say

```
→MST
```

where *MST* is the label name for line 6.

To associate the receiving line with a particular label, the label must be the first item to appear in that line, and it must be separated from the statement by a colon. Here is a typical function line containing a label:

```
      [6]MST:TOT←TOT×.07-A÷Q
```

Whenever this line is executed, the system will act as if *MST* is not present. But, whenever *MST* is referred to while the function is executing, it will take on the value 6 or whatever line number it resides on at the time.

Here is a function employing a label:

```
       ∇R←PAS N
      [1] R←1
      [2]L2:R←(R,0)+0,R
      [3] →L2×N>R[2]
       ∇

       PAS 4
1  4  6  4  1
```

Mathematicians will recognize that *PAS* produces the coefficients equivalent to (0,⍳*N*)!*N*. Although labels act like local variables in that they both become activated and take on specific values only while the function is executing, they are referred to as being local constants. Their values never change throughout the execution of the function. To check this, write a function that uses labels and try to change their values.

Positioning Of Branches In Statements

The branch symbol can appear anywhere within an APL statement. We have seen examples where it was the first character, now here is one where it is positioned near the middle of a statement.

```
[7]R←(⍳0)=X←→(0=A)/11
```

If the above branch is successful, execution of the statement on line 7 is halted at the branch arrow, and is resumed again at line 11. If the branch is not successful, the branch function returns an empty numeric vector result which gets assigned to the variable *X*. Therefore, if the comparison (0=*A*) above produces a 0 result, *X* is replaced by ⍳0 and *R* is specified with a 1. If (0=*A*) produces a result of 1 , both *X* and *R* retain their present values and execution gets rerouted down to line 11.

Only when the branch instruction is in the middle of the statement does it return a numeric empty vector result. If it is the first character in the statement, it does not produce any result.

Summary

Branching within a defined function can be conditional or unconditional. Unconditional branches are performed everytime they are encountered, while conditional ones are dependent on the prevailing circumstances. Below are some typical branch instructions:

1. Unconditional

```
→6
→L1
→A×B
```

2. Conditional

   (a) multiple choice

```
→(L1,L2)[1+(X=Y)]
→NϕL1,L2,L3
→(N,~N)/L1,L2
```

(b) branch or fall through

```
→(N>M)/L1
→(N>M)ρL1
→(N>M)↑L1
→((N>M)∧(N<0))/L1
→L1×⍳N>M
→L1⌈⍳N>M
```

3. Function exit

```
→0
→
```

If the argument of the branch function is a vector, the branch applies to the first element only.

```
[3] →6 2 10
```

The branch above is to line 6. If the right argument is an empty vector, no branching occurs

```
[3] →⍳0
```

Practice Exercises

1. Define a function to repeatedly flash the word '*HELLO*' on the screen. Set ⎕*PT* to 1 beforehand.

2. Define a function to simulate the +/ primitive for vector arguments.

3. Define a function to generate the Fibinacci series

   1 1 2 3 5 8 13 21 ....

   where any number in the series, except the first two, is the sum of the previous two.

Chapter 16: INPUT - OUTPUT

In order to be completely interactive with its user, the MCM/70 must have the ability to accept input from the keyboard and display data on the screen while in the process of carrying out calculations. The user should be able to supply data at various intervals of computation and the computer should be able to display its results at various stages also. To accomodate these features, two symbols have been employed. Both are used to accept input from the keyboard and to display output on the screen. But the manner in which each performs, and the results each produces are completely different from the other. This chapter describes these two, plus a technique for combining both literal statements and numeric values together to form a single output line.

Numeric Input

To accept numeric input from the screen, the quad function ⎕ (upper shift *L*) is used. The function *SORT* illustrates.

```
      SORT
ENTER DATA.
 ⎕: ■
```

When *SORT* is executed, it first displays the statement *ENTER DATA*. The next line it displays is

```
 ⎕: ■
```

Displaying the ⎕: is the system's way of indicating it is expecting numeric data to be entered. The cursor accompanying this request is the input promt. It shows where the first number is to appear on the screen when we key it in. Upon entering our data, the screen looks like this:

```
 ⎕: 20 16 21 15 17 22 18
```

Even though ⎕: is still on the screen, it is not considered to be part of the input.

Let us press the RETURN key and view the result.

```
15  16  17  18  19  20  21
```

SORT looks like this:

```
      ∇R←SORT
[1]'ENTER DATA.'
[2]R←⍞
[3]R←R[⍋R]
      ∇
```

Line 2 of *SORT* is the one containing the numeric input request.

The quad function can also be used outside of user defined functions.

```
      6+⍞
 ⍞: 
```

When we supply a value, it gets added to the 6.

```
 ⍞: 7
13
```

Here is another:

```
      ⍞-3
 ⍞: 10
7
```

Each time numeric input is requested, the system signals this to the user by displaying the ⍞: symbols.

Input to a quad can also be in the form of an APL statement. Instead of entering the value 10 as we did above, we could have entered 2×5 .

```
      ⍞-3
 ⍞: 2×5
7
```

or 8+2

```
      ⍞-3
 ⍞: 8+2
7
```

or even 6+20-16

```
      ⍞-3
 ⍞: 6+20-16
7
```

The quad evaluates its input as if it were an APL statement. This means variable names and even function names can be used as input to ⎕.

Here is another function using the quad to obtain input from the user while it is executing. The function *TST* below selects 2 numbers at random, displays them and expects the user to key in their product.

```
      TST
6×5=
 ⎕: 30
RIGHT
4×9=
 ⎕: 28
WRONG. TRY AGAIN.
4×9=
 ⎕: 36
RIGHT
2×7=
 ⎕: STOP
```

Typing in *STOP* terminates the exercise. The function looks like this:

```
      ∇TST;Q;STOP;A
     [1]STOP←○1
     [2]Q←2?12
     [3]Q[1];'×';Q[2];'='
     [4]→0×⍳STOP=A←⎕
     [5]→(A≠×/Q)/L8
     [6]'RIGHT'
     [7]→2
     [8]L8:'WRONG. TRY AGAIN.'
     [9]→3
      ∇
```

Line 4 contains the numeric prompt.

Instead of typing *STOP* above, the right pointing arrow → could have been entered to do the same thing. When used this way, it is recognized by the system to mean "terminate the execution of this function."

Output

Besides being used to accept numeric input, the ⍞ is also used to display both numeric and literal output. When used in this way, the specification arrow ← must appear to the right of the quad.

```
      ⍞←A←6+7
13

      A
13
```

It may also be used within calculations without having any affect on the final result.

```
      A←7+⍞←6+7
13

      A
20
```

In the function *MEAN* below, it is used to display the literal string *ENTER DATA*.

```
       ∇R←MEAN
[1]    R←(+/R)÷⍴R←⍞⊣⍞←'ENTER DATA'
       ∇
```

By employing this output primitive, the entire function above was able to be written on one line.

```
      MEAN
ENTER DATA
 ⍞: ⍳10
5.5
```

One symbol on line 1 that is new to you is the ⍴ (upper shift *J*). To find out more about it, consult Chapter 19.

Literal Input

To accept numeric input from the keyboard, we use the quad symbol ⎕. To accept literal data, we use the quote-quad symbol ⍞ (upper shift *L* and upper shift *K*). Unlike the quad symbol which has no arguments, the quote-quad can be used both monadically and dyadically. An example of it being used monadically is this:

```
      N←⍞'ENTER NAME. '
```

The system displays the argument and places the cursor at the end.

```
ENTER NAME. ■
```

The system is waiting for input, and we comply.

```
ENTER NAME. ERIN
■
```

By displaying the contents of *N*, we see that not only our input data has been assigned, but also the displayed output.

```
      N
ENTER NAME. ERIN
```

If we wanted just the input, we could have expressed our initial statement as

```
      N←12+⍞'ENTER NAME. '
```

Here is the entire process again:

```
      N←12+⍞'ENTER NAME. '
ENTER NAME. ■
```

When we enter our input,

```
ENTER NAME. ERIN
```

and press the RETURN, *N* will be the following:

```
      N
ERIN
```

Just what we want.

```
      ⍴N
4
```

If there is no line to be displayed with the ⍞ function, the way to avoid having to supply one is to key in a null or empty literal vector as its argument.

```
      A←⍞''
```

The only thing displayed is the cursor, prompting for input to be assigned to *A*.

Using the ⍞ monadically always places the cursor immediately after its displayed argument to indicate were the first character of our response is to appear. One thing to note is the cursor can be backspaced over the output section and editing can be performed on the output.

```
      V←⍞'3+?=7'
3+?=7
```

We can, at this stage, backspace the cursor and change the ? to the character 4 by doing the following:

1. 3+?=7 original statement
2. 3+?=7 cursor backspaced to ?
3. 3+4=7 the ? replaced with a 4
4. press the RETURN key

After this sequence of steps, *V* will contain the literal string 3+4=7.

```
      V
3+4=7

      ⍴V
5
```

Instead of backspacing to do the replacement, an easier way is to use the ⍞ dyadically. When it is used dyadically, the left argument is a value representing where on the display screen the cursor is to be positioned. For instance,

```
      A←3⍞'3+ =7'
```

causes the '3+ =7' to be displayed and the cursor to appear between the + sign and the = sign in the form

```
3+■=7
```

We can now enter our input and press the RETURN.

```
3+4■7
```

What the computer has assigned to *A* is the enter literal, including our input.

```
      A
3+4=7
```

The left argument can be any integer from 1 to 32. Needless to say the applications for this function are quite extensive.

Additional Features

Remember back in Chapter 3 we saw that whenever data was supposed to include one or more quotation marks, two adjacent quotes had to be used to represent this? The following is a typical example:

```
      Y←'SHE''S FOUR YEARS OLD.'
■

      Y
SHE'S FOUR YEARS OLD.
```

Although this rule is required above for obvious reasons, it does not apply to the ⍞ function.

```
      Y←⍞''
SHE'S FOUR YEARS OLD.

      Y
SHE'S FOUR YEARS OLD.
```

Below, the quote-quad function in line 1 of *QUES* is used to accept responses to the accompanying question.

```
     ∇QUES
[1]  Q1:A←¯6↑⍞'WHAT IS THE CAPITAL OF CANADA? '
[2]  →Q2×⍳∧/'OTTAWA'=A
[3]  'WRONG. TRY AGAIN.'
[4]  →Q1
[5]  Q2:'RIGHT'
     ∇
```

Here it is in use:

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

If, when executing a function that is prompting for literal input, you decide you would rather not respond, there is you can cause the system to abandon its request. The following function, when executed, will continually ask for input by constantly looping around lines 1 and 2.

```
     ∇LOOP;A
[1]  A←⍞''
[2]  →1
     ∇

      LOOP
STOP
HELP
```

Even after entering such statements as *STOP* and *HELP*, the function still keeps asking for more input. The only way this loop can be broken, other than turning off the computer, is to press the CTRL, SHIFT, and ← keys. When they are pressed in this order, and held down until the ← key is pressed, the computer will stop its execution and

display the word *INTERRUPT*, followed by the point of interruption on the next line.

```
INTERRUPT
LOO[1] A←■''
```

The execution of the function has now been suspended at this line. See □*SI* in Chapter 21 for more details on what to do with suspended functions.

Heterogeneous Output

To facilitate the displaying of both numeric and literal data on the same line, we use the semicolon.

```
      'THE DATA IS ';1 2 3 4 5
THE DATA IS 1 2 3 4 5
```

This feature applies mostly to output formatting that includes both headings and results together.

```
      [7]'TOTAL IS ';+/X
      [8]'AVERAGE IS ';(+/X)÷⍴X
```

Practice Exercises

1. Define a function that will display an integer from zero to fifteen and ask the user to spell the word. For example,

```
      SPELL
SPELL 10
TEN
SPELL 4
FOR
WRONG. TRY AGAIN.
SPELL 4
FOUR
```

2. Write a function to accept names of varying lengths and create a matrix result. When all the names have been entered, signal this to the computer by pressing the RETURN key only when it asks for more input.

Example:

```
      N←ENTER
ENTER NAMES.
BUTLER G.
ATKIN B.
BRADSHAW C.
 (RETURN key only)
```

} user input

```
      N
BUTLER G.
ATKIN B.
BRADSHAW C.

      ρN
3  1
```

3. Write a function that generates two different integers between 1 and 10, and asks the user what they are. Give him three tries before telling him what they are.

Example:

```
      GUESS
 □: 2 6
NO.
 □: 3 7
RIGHT. I HAVE TWO MORE.
 □: 4 7
NO.
 □: 9 10
NO.
 □:1 2
THEY ARE 3 5
 □: 0
```

A response of 0 terminates the quiz.

Chapter 17: INNER AND OUTER PRODUCTS

As you become more proficient in expressing your data in array form, you will discover there are several different functions and expressions which can take these arrays and produce the same results. Some may be written more concisely and perform more efficiently than others. But you may find that those which look sophisticated may in fact prove to be impeditive to the system's performance. Alternative methods are always great assets. The inner and outer product functions are two such alternatives which you should consider whenever dealing with arrays.

Inner Product

How many elements are in the vector *X*? We determined this earlier by employing two methods. One was simply ρ*X* and the other was +/*X*=*X* . Both expressions produced the same results as long as *X* was a vector. But if *X* was a scalar, the first method returned an empty vector and the other returned a 1.

```
      X
1  2  3  4  5  6

      ρX
6                          } vector argument
      +/X=X
6

      ρ3
 (blank screen)            }
                           } scalar argument
      +/3=3
1                          }
```

If we plan on using the result in any other calculations, we have to decide which one is best suited for the task. For instance, if the data could be scalars as well as vectors, the +/*X*=*X* is the one to choose. It assures us the result will always be a numeric value. But if the data will be in vector form only, then ρ*X* is probably more

efficient to use. There is also a third method to consider in this evaluation. To find out how many elements are in X we could use either of the above two expressions, or this one:

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

This expression is a typical inner produce statement. Read as "X plus dot equal X", it performs exactly the same way as $+/X=X$ does when X is a scalar or a vector. The primitives you use on both sides of the dot, or decimal point must be scalar dyadics. Just to refresh your memory, the symbols representing scalar dyadic functions are:

```
+ - × ÷ * ⌈ ⌊ | ⍟ ! ○ < ≤ = ≥ > ≠ ∨ ∧ ⍱ ⍲
```

Expressions such as $X⍳.\in X$ and $X\sim.\rho X$ will not work. Below is an example of another inner product.

```
      M←3 5 1
      N←4 2 3
    N+.×M
25
```

This last one is read as "N plus dot times M".

Let us go through it a step at a time to see how it arrived at the answer. The first operation is

```
      N×M
```

which is essentially

```
      4 2 3×3 5 1
```

The products of the multiplications are

```
  4   2   3
× 3   5   1
-----------
 12  10   3
```

These products are then summed to produce the final result.

```
      +/12 10 3
25
```

Here are a few more computations using *M* and *N* as arguments:

```
      N×.+M
196

      N⌈.⌈M
5

      N⌊.⌊M
1

      N+.+M
18
```

When both arguments are scalars or vectors, the result is always a scalar. But, if at least one of them has any higher rank, certain rules are imposed. For example, if the left argument is a 3-element vector and the right argument is a matrix, the matrix must have three rows. The general rule states that the dimension of the last coordinate of the left argument must equal the first coordinate of the right argument.

```
      K
10  11
12  13
14  15

      L
1 2 3

      L+.×K
76 82
```

The solution above was arrived at in the following manner.

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

or

```
      (+/L × K[;1]) , (+/L × K[;2])
```

When a matrix is the left argument and a vector is the right, the number of columns in the matrix must equal the length of the vector.

```
      ρK
3  2

      K+.×2 3
53  63  73
```

which is

```
      (+/K[1;] × 2  3) , (+/K[2;] × 2  3) , (+/K[3;] × 2  3)
```

If they are not equal, the following occurs:

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

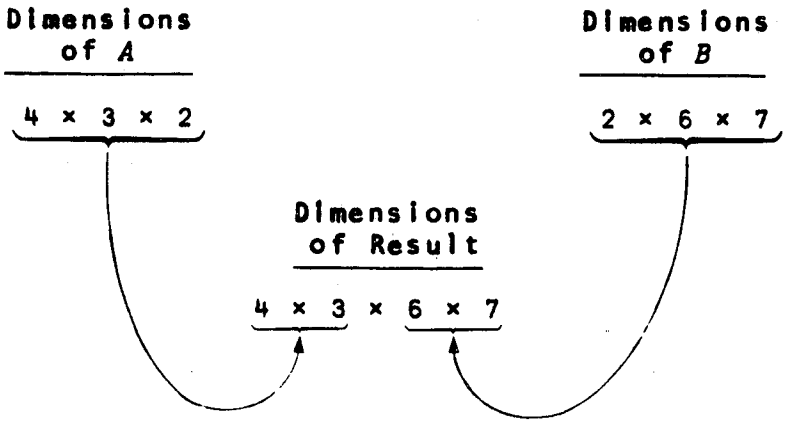
The dimensions of the result are a combination of all but the last coordinate value of the left argument and all but the first coordinate of the right argument. In the example `K+.×2 3` above, `K` is a 3 by 2 matrix operating on a 2-element vector. Therefore, the result is a 3-element vector. If it were a 4 by 2 matrix on the left, and a 2 by 3 matrix on the right, the computer would return a 4 by 3 matrix result.

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

      B
10   15   12
17   13   16

      A+.×B
 44    41    44
 98    97   100
152   153   156
206   209   212
```

This result was obtained by the following calculations:

	column 1	column 2	column 3
row 1	+/1 2×10 17	+/1 2×15 13	+/1 2×12 16
row 2	+/3 4×10 17	+/3 4×15 13	+/3 4×12 16
row 3	+/5 6×10 17	+/5 6×15 13	+/5 6×12 16
row 4	+/7 8×10 17	+/7 8×15 13	+/7 8×12 16

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



Outer Product

The purpose of the generalized outer product is to allow every element in the right argument to perform a specific operation on every element in the left argument.

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

Above, every element in the left vector was multiplied by every element in the right. The expression ∘.× is read as "null dot times." And, like the inner product, the only primitives that can be used by

the outer product are scalar dyadics. Below is an illustration showing how the computer evaluated this last example:

×	3	5
2	6	10
4	12	20

Unlike the inner product, only one scalar primitive can be employed at a time, and there are no dimension restrictions placed on the arguments.

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

The coordinates of the result are a combination of the coordinates of both arguments. The result's rank is the sum of the ranks of the two arguments. If both arguments are 3-element vectors, which are of rank 1, the result is a 3 by 3 two dimensional array.

```
      C
¯3  0  2  1
 6  4  3  7

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

11   5
 9   3
 8   2
12   6

      1 2 3 4 5∘.≤1 2 3 4 5
1 1 1 1 1
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
0 0 0 0 1
```

The null symbol, ∘, when used in this context, does not perform any real task other than to indicate to the system that it is to do an outer product operation. But by itself, the null symbol is a primitive whose use is discussed in Chapter 19.

Practice Exercises

1. For $A \leftarrow 2\ 6\ 4\ 5$ and $B \leftarrow 3\ 4\ 5\ 1$, evaluate the following:

   (a) $A+.\times B$ (b) $A+.+B$ (c) $A-.*B$

   (d) $B+.\times A$ (e) $A\lceil.\lfloor B$ (f) $A\lfloor.\lceil B$

   (g) $A\circ.+B$ (h) $A\circ.\times B$ (i) $A\circ.+B$

2. Restructure A and B into 2 by 2 matrices and perform question 1 again.

3.(a) Joe works a regular 40 hours shift each week. Periodically, he works a few hours overtime during the week and on Sundays. His hourly pay is rated into the following categories:

regular pay	3.50
overtime	4.75
Sundays	6.90

During one week, he logged the following hours:

regular hours	40
overtime	7
Sunday	8

His weekly pay could either be calculated by

```
      +/40 7 8×3.5 4.75 6.9
```

or

```
      40 7 8+.×3.5 4.75 6.9
```

For the entire month, Joe's time sheet looked like this:

	week 1	week 2	week 3	week 4
regular	40	40	35	40
overtime	7	8	4	6
Sunday	8	2	0	10

Calculate his wages for each week.

(b) Frank and John work in the same shop as Joe. Their pay rates, along with Joe's, are listed below.

	Joe	Frank	John
regular pay	3.50	3.75	3.25
overtime	4.75	5.05	4.50
Sunday	6.90	7.25	6.35

They too worked the same number of hours as Joe did during the four weeks. Calculate their weekly pay using the inner product method.

4. Use the outer product function to develop the times table for the integers 1 to 6.

Chapter 18: STILL MORE MIXED FUNCTIONS

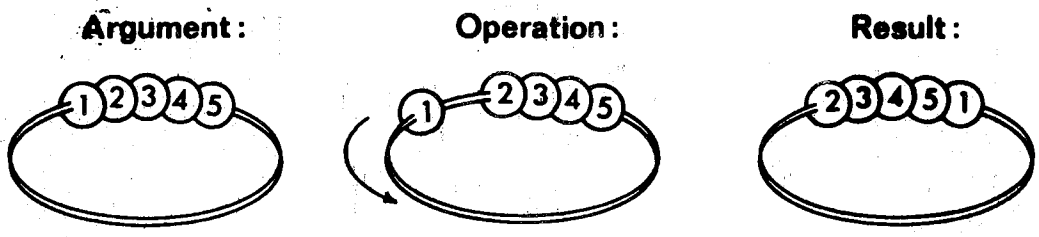
All the primitive functions in MCM/APL are divided into two groups - scalar and mixed. The scalar functions perform operations that are predominantly algebraically oriented. Such computations as addition, multiplication and division are all tasks of scalars. So too are the trigonometric calculations, logarithms, factorials, Boolean functions, and numerical comparisons (less than, equal to, etc.). With the exceptions of the equal and not equal functions, they all require their arguments to be numeric and they all produce numerical results. Mixed functions, on the other hand, are mostly concerned with the actual management of data. They can determine the size, shape and rank of any array, plus retrieve any of its elements on request. They can generate data, drop data and even join two sets of data together. All the various chores these functions perform help to increase the power and flexibility of the MCM/70 with an ease of use unknown in any other language. In total, there are 22 mixed functions available, 14 of which have already been discussed and the remaining 8 are covered in this chapter and the next.

Rotate

The symbol to represent the rotate function is ϕ (upper shift *O* overstruck with upper shift *M*). The function's entire syntax is written as *X*ϕ*Y*. If *X* is a single integer value and *Y* is a vector, then *X*ϕ*Y* performs a cyclic rotation of *Y*. For example:

```
      1ϕ1 2 3 4 5
2 3 4 5 1
```

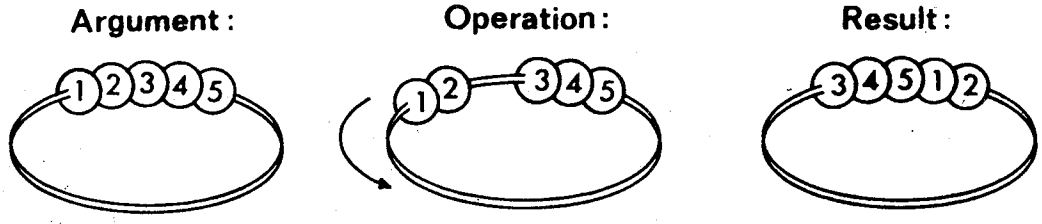
Above, the function caused the right argument to be rotated by one position from front to back, with the very first number being shifted to the back. Here is a schematic illustrating what happened:



To rotate the first two, the expression would look like this:

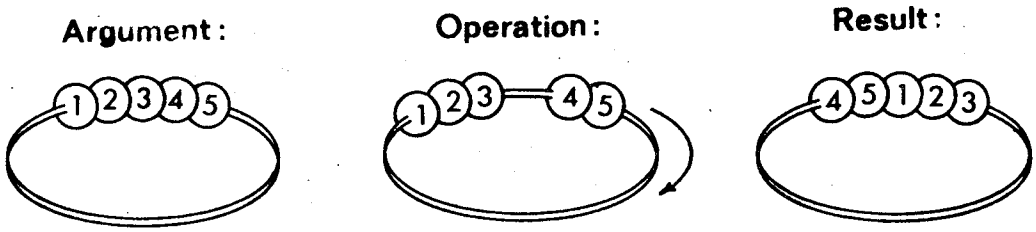
```
      2⍉1 2 3 4 5
3 4 5 1 2
```

And here is its schematic:



In the above two examples the system rotated the elements in the right argument, one at a time, in a front-to-back direction according to the number stated in the left argument. If the left argument is a negative amount, the direction of rotation is reversed.

```
      ¯2⍉1 2 3 4 5
4 5 1 2 3
```



Here are two more:

```
      ¯1⍉1 2 3 4 5
5 1 2 3 4

      ¯3⍉1 2 3 4 5
3 4 5 1 2
```

Literals may also be rotated.

```
      ¯5⍉'NOONAFTER'
AFTERNOON

      4⍉'NOONAFTER'
AFTERNOON
```

If the right argument is an array of rank >1, and the left argument is a single value, the columns of the array are rotated.

```
      A←3 4⍴⍳12
```

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

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

This same operation could have been expressed as

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

The [2] causes the rotation to be done along the second coordinate which is the columns in this case anyway. To produce a cyclic rotation of the rows of *A*, we direct the system to the first coordinate.

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

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

We have run across this coordinate pointing feature before. First with catenate, then with reduction and scan, and then with compression and expansion. In all five instances, there were additional symbols to refer specifically to the first coordinate, namely ⍪ ⌿ and ⍀. So too is there one to represent coordinate rotation. It is the large circle ○ overstruck with the minus sign.

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

      M←3 5⍴'EATBLACKQUUNTGR'
■

      M
EATBL
ACKQU
UNTGR

      3⌽M
BLEAT
QUACK
GRUNT

      2⊖¯2⌽M
GRUNT
BLEAT
QUACK
```

Instances arise when it may not be desirous to have all the planes, rows, and columns of an array rotated by the same amount. You may want to rotate specific rows or columns, while the others remain as they are. Or there may be applications requiring rotation to be done by varying degrees. For instance, to rotate the first elements in all the columns of *A* by one position and rotate the second elements by two positions while the third set of elements remain stationary, the expression would look like this:

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

The same is true for rows.

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

Reversal

The symbol ⌽ when used monadically reverses the order of its right argument if it is a vector. (Scalars are not affected)

```
      ⌽4 3 2 1
1 2 3 4

      ⌽'DEW'
WED

      ⌽'EDIT'
TIDE
```

When the argument is of greater rank, the columns are reversed.

```
      Z←2 3 4⍴⍳24
```



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

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

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

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

To reverse other coordinates, simply direct the system to the proper dimension. Reversal of the rows of *Z*:

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

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

Reversal of Z's planes:

```
      ⌽[1]Z
13   14   15   16
17   18   19   20
21   22   23   24

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

or just,

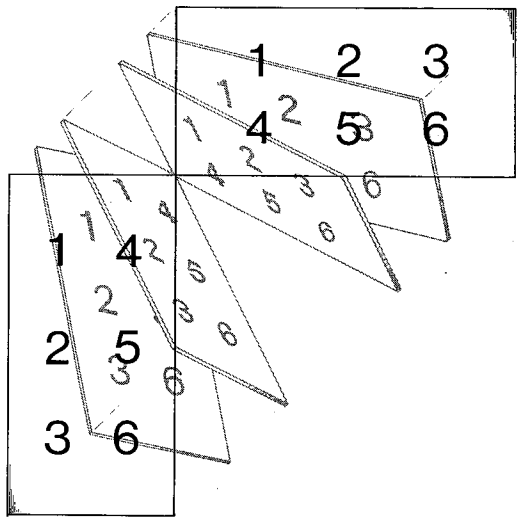
```
      ⊖Z
13   14   15   16
17   18   19   20
21   22   23   24

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

Monadic Transpose

The last function to be discussed that is associated with array restructuring, is the transpose. Its purpose is to transpose or interchange the elements within an array. This involves switching the rows and columns around and, as we will see later, switching the elements between various planes. But for now we will deal with rows and columns.

The symbol used by the transpose function is ⍉ (upper shift *O* overstruck with the reverse solidus). When this function is used monadically with a matrix argument, it reshuffles the rows and columns around so that row 1 of the argument becomes column 1 of the result, row 2 becomes colmun 2 and so on. Here is an example:

```
      M
1   2   3
4   5   6

      ⌽M
1   4
2   5
3   6
```



The contents of the 2 by 3 array *M* are rearranged into a 3 by 2 result.

Here is another.

```
      N
OIL
GNU
RUN
ERG
SEE

      ⌽N
OGRES
INURE
LUNGE
```

If the arrays are of any greater rank, the function is extended to exchange the elements in all the dimensions.

```
      X←2 3 4⍴⍳24
```

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

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

```
      □←T←⍉X
 1   13
 5   17
 9   21

 2   14
 6   18
10   22

 4   16
 8   20
12   24

      ⍴T
4  3  2
```

The dimensions of the result are the reversal of those of the argument. Notice column 1 (1 5 9) in plane 1 of *A* is column 1 in plane 1 of *T* and column 1 in plane 2 of *A* becomes column 2 in plane 1 of *T*. This procedure is repeated for every column of *A*. Array restructuring of this type is described in greater detail in the next section on dyadic transpose. So before you think of rejecting this function as being too complicated for your applications, read on a little further.

Dyadic Transpose

As we have just seen, the monadic transpose reverses the coordinates of its argument when it exchanges the elements. This can also be accomplished by using the transpose symbol ⍉ in its dyadic syntax.

```
      M←2 3⍴⍳6
```

```
      M
1  2  3
4  5  6

      2 1⍉M
1  4
2  5
3  6
```

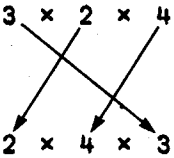
The left argument above, 2 1, refers to the order in which the dimensions are to appear in the result - the second dimension (columns) first, and the first dimension (rows) second. The elements within the result are also arranged accordingly. This is identical to the monadic transpose, as all we did was reverse the coordiates. But when dealing with multidimensional arrays, the interchange of their contents in this set pattern may not be desired. In instances where this is the case, you will find the dyadic transpose can accomplish this by stating, in its left argument, the specific positions within the result the argument's original coordinates are to be placed. Examine the following example:

```
      A←3 2 4⍴⍳24

      A
 1  2  3  4
 5  6 77  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24

      ⎕←R←3 1 2⍉A
1  9 17
2 10 18
3 11 19
4 12 20

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

      ⍴R
2 4 3
```

The left argument 3 1 2 directed the system to construct an array out of the coordinates and values of *A* such that

coordinate 1 of *A* becomes coordinate 3 of *R*
coordinate 2 of *A* becomes coordinate 1 of *R*
coordinate 3 of *A* becomes coordinate 2 of *R*

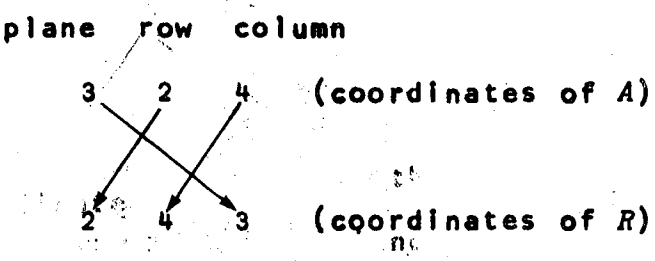
This means the 3 × 2 × 4 argument



becomes a 2 × 4 × 3 result

Along with the redimensioning, there was also the relocation of the elements. For example, the value 7 which appears in plane 1, row 2, column 3 (or [1;2;3]) of *A* was inserted into plane 2, row 3, column 1 (or [2;3;1]) of *R*. The general algorithm for this transpose is *A*[*X*;*Y*;*Z*] with the coordinates of *A* being relocated in *R*[*Y*;*Z*;*X*] where the function is (*Y*,*Z*,*X*)⍉*A*.

Another way to interpret this is to take the coordinates of both *A* and *R*, as was done above, and determine from them how the elements were rearranged.



The arrows indicate where the elements were moved to. The arrow running from plane coordinate of *A* to the columns coordinate of *R* indicates the contents of plane 1 of *A* are the same ones appearing in column 1 of *R*. Those in plane 2 of *A* are also in column 2 of *R*. Plane 3 of *A* holds the same items as column 3 of *R*.

The arrow going from the rows coordinate of *A* to the planes coordinate of *R* shows that the contents of each of the rows in *A* are identical to the respective planes of *R*. All the numbers that appear in row 1 of each of the planes of *A* also appear in plane 1 of *R*. All the contents of the three row 2's of *A* are in plane 2 of *R*. The contents of column 1 (i.e., 1 5 9 13 17 21) in *A* are the same as those in row 1 of *R* because, as the "4-to-4" arrow above indicates, the contents of all the columns of *A* appear in the respective rows of *R*.

Here are some more examples. See if you can determine how each element in the argument was assigned its relative location within the result.

```
      ⎕←S←3 2 1⍉A
1     9    17
5    13    21

2    10    18
6    14    22

3    11    19
7    15    23

4    12    20
8    16    24

      ⍴S
4  2  3
```

One hint on this last one. The rows are not included in the interchange as the 2 in the left argument 3 2 1 is the second element.

Here is another:

```
      ⎕←T←2 3 1⍉A
 1     5
 9    13
17    21

 2     6
10    14
18    22

 3     7
11    15
19    23

 4     8
12    16
20    24

      ⍴T
4  3  2
```

The result of a dyadic transpose need not always be of the same rank as the argument. For example, a matrix result can be created from *A* by the following means:

```
      □←M←2 2 1⍉A
1   13
2   14
3   15
4   16

      ⍴M
4  2

      □←N←1 1 2⍉A
 1    2    3    4
13   14   15   16

      ⍴N
2  4
```

The rank of the result is equal to the largest value contained in the left argument.

```
      ⌈/1 1 2
2
```

To determine the major diagonal of an array, the left argument is set to all 1's. Below is a matrix whose major diagonal is 6 10 16.

```
      P
 6   20   17    4
 2   10   34   12
19   13   18    3
```

Here is how it is found:

```
      1 1⍉P
6  10  16
```

This last result is a vector because the largest value in the left argument is 1.

One final note on the left argument. It must always be a permutation vector. This means it can be 1 1 2 or 1 2 2 or 1 2 3 or 2 1 2 , but not 3 3 3 or 2 3 3 or 1 3 3. And the number of elements it contains must equal the rank of the right argument.

Practice Exercises

1. For $A \leftarrow \iota 8$, $B \leftarrow 3\ 2\rho\iota 6$, and $C \leftarrow 2\ 3\ 2\rho 10+\iota 12$ evaluate the following:

(a) ϕA	(b) $\phi\phi A$	(c) ϕB
(d) $\phi[1]B$	(e) $\ominus B$	(f) $2\phi A$
(g) $^{-}2\phi A$	(h) $2\phi B$	(i) $2\ominus B$
(j) $^{-}2\phi B$	(k) $^{-}2\ominus B$	(l) $^{-}2\ominus{}^{-}2\phi B$
(m) ϕC	(n) $\ominus C$	(o) $\phi[2]C$
(p) $1\ {}^{-}1\ominus B$	(q) $0\ 1\ 2\phi B$	(r) $(2\ 3\rho\iota 6)\phi C$

2. Using the same variables defined in question 1, execute these statements:

(a) $\circledcirc A$	(b) $1\circledcirc A$	(c) $2\ 1\circledcirc B$
(d) $\circledcirc B$	(e) $3\ 2\ 1\circledcirc C$	(f) $1\ 1\ 1\circledcirc C$
(g) $1\ 1\circledcirc B$	(h) $\rho 1\ 1\circledcirc B$	(i) $1\ 2\ 2\circledcirc C$
(j) $2\ 2\ 1\circledcirc C$	(k) $1\ 1\ 2\circledcirc C$	(l) $1\ 2\ 3\circledcirc C$

3. The scan function performs from right to left.

```
      +\1 2 3 4
10  9  7  4
```

This means if we want to evaluate a series of numbers by using the scan, we have to remember to enter them in reverse order,

```
      +\4 3 2 1
10  6  3  1
```

and read the result in reverse to what we are normally used to. An easy way to avoid all this is to employ the ϕ function to do both of these operations for us.

```
      ⍎+\⍕1 2 3 4
1  3  6  10
```

What would be the expression for sum scanning the rows of a matrix?

4. How can we multiply a vector of numbers to each of the rows in a matrix? With a vector of

```
      5 10 20
```

and a matrix of

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

how can we produce a result of

```
      5  20   60
     20  50  120
     35  80  180
```

There is no one function in the system that can do this. Therefore, we must manufacture one. By using both the outer product and the transpose functions, we can complete out task. First, let us assign the two arguments to variable names.

```
      V←5 10 20
```

```
      M←3 3⍴⍳9
```

Here is the statement required to multiply *V* to each of the row of *M*:

```
      1 2 2⍉M∘.×V
 5   20   60
20   50  120
35   80  180
```

Here it is for the columns:

```
      1 2 1⍉M∘.×V
  5   10   15
 40   50   60
140  160  180
```

If *V* is the left argument, and *M* is the right, row multiplication is

```
      2 1 2⍉V∘.×M
 5   20   60
20   50  120
35   80  180
```

Columns multiplication is

```
      1 1 2⍉V∘.×M
  5    10    15
 40    50    60
140   160   180
```

Try other outer product functions such as + - * and other left arguments to the transpose to see the results that are obtained.

Chapter 19: THE REMAINING MIXED FUNCTIONS

To complete the list of mixed functions available on the MCM/70, this chapter describes the remaining four. The first two deal with numbering systems and how to convert from one to the other. The third function, called the null, shows how two or more unrelated data types can appear in the same statement without one interfering with the other. The last function, the execute, illustrates how literal data can be converted to numerics and how characters can become valid variable names. At the end of the chapter, there is a brief summary listing of all the mixed functions found within MCM/APL.

Base Value (Decode)

The base value or decode function ⊥ (upper shift *B*) has a syntax of *X*⊥*Y* and is used to convert a vector of values which are contained in *Y* from the numbering system expressed by *X* to another. An example would be changing binary numbers (base 2) to decimals (base 10).

```
      2⊥0 1 0 0 1
9

      2⊥1 0 1
5

      2⊥1 0 0
4
```

In cases where the numbering system has "weighted" measurements, such as yards, feet, and inches, these weights are specified in the left argument.

```
      1 3 12⊥14 2 7
535
```

The above example determined how many inches there are in 14 yards, 2 feet, and 7 inches. The left argument is called a radix vector. It states the relationship between each of its elements (there are 12 inches in a foot and 3 feet in a yard).

How many seconds are there in 8 hours, 45 minutes and 16 seconds?

```
      1 60 60⊥8 45 16
31516
```

The radix vector indicates to the system that there are 60 seconds in a minute, 60 minutes in an hour. If days were involved, the 1 would be changed to 24 to reflect that there are 24 hours in a day.

How many pints are there in 2 gallons, 3 quarts, and 1 pint?

```
      1 4 2⊥2 3 1
23
```

This last function was solved by the following process:

Step 1

the number of pints in one gallon: ×/4 2 or 8
the number of pints in one quart: ×/2 or 2
the number of pints in one pint: ×/ or 1

The first element in the left argument is not used by the system for anything other than to balance out the number of elements in each argument.

Step 2

```
      +/8 2 1×2 3 1
23
```

For those interested in polynomial evaluations, the decode function will be of some help. For instance, to solve the equation

$$4X^2 + 2X + 1$$

where X equals 2, the solution is be expressed in the following manner:

```
      2⊥4 2 1
21
```

The value for X is the left argument and the coefficients of the polynomial are the right. Here is the same polynomial evaluated again, this time with X taking on the value 3.

```
      3⊥4 2 1
43
```

Representation (Encode)

The representation or encode function ⍴ (upper shift *N*) is the inverse of the decode. It "breaks up" its right argument according to the values contained in the left.

How many days, hours, minutes and seconds are there in 32056 seconds?

```
      1 24 60 60⍴32056
0  8  54  16
```

What is the binary notation for the decimal value 7?

```
      (5⍴2)⍴7
0  0  1  1  1
```

How many yards, feet and inches are there in 436 inches?

```
      1760 3 12⍴436
12  0  4
```

The answer to this last one was arrived at by the following process:

1. The last element of the left argument (12) was divided into 436 to produce a quotient of 36 and a remainder of 4 . This 4 became the last element in the answer.

2. The quotient 36 was then divided by the next element in the left argument (3) yielding a quotient of 12 and a remainder of 0, the second element in the answer.

3. The system then divided the 12 by 1760 to produce a quotient of 0 and a remainder of 12, the first element in the answer.

Null

We saw earlier where the null symbol ∘ (upper shift *J*) was part of the outer product function. It performed no real task other than to indicate to the system that an outer product operation was being requested. But when used by itself, it does do a unique operation. The syntax of the null function is

X∘*Y*

and its duty is to ignore *Y*. This may seem strange at first glance, so let us have a closer look. In the following expression, two assignments are taking place. *A* is assigned a numeric matrix and *B* is assigned a literal vector.

```
      A←2 2⍴⍳4⋄B←'BOOK'
```

The null function makes this possible on one line.

```
      A
1  2
3  4

      B
BOOK

      ⍴A
2  2

      ⍴B
4
```

By using the null, the number of lines a function may have can be reduced. This can result in a decrease in execution time whenever the function is evoked. Here is a portion of a function without the null.

```
        .
        .
        .
      [4]I←0 0
      [5]J←'ENTER DATA.'
      [6]K←2 3⍴⍳6
        .
        .
        .
```

And here it is with the null:

```
        .
        .
        .
      [4]I←0 0⋄J←'ENTER DATA.'⋄K←2 3⍴⍳6
        .
        .
        .
```

The time required to execute the entire function is reduced, due to the elimination of certain "housekeeping" duties that would normally be performed as the system completes its evaluation of a line. Such things as checking to see if any intermediate results have to be erased and the updating of the $\square LC$ variable must always be done before the system is able to proceed to the next line. Therefore, by reducing the number of lines, we can also reduce execution time. One additional side affect is saving on memory space, as each line number takes up approximately 9 bytes of available space.

Execute (Unquote)

The symbol used to denote the execute or unquote function is ⍎ (upper shift *B* overstruck with upper shift *J*). Defined as a monadic function, the execute's main purpose is to unquote and execute the literal contents of its argument.

```
      ⍎'2+2'
4

      A←⍎'10×6'

      A
60
```

It treats its argument as if it were as APL statement itself.

```
      C←1 2 3

      D←⍎'C'

      D
1  2  3
```

This primitive is used mostly in defined functions that are interactive with the user. Data that is normally numeric is instead accepted as literal text so that the defined function can analyse it before turning it into its numerical equivalent. A mathematical drill is a good application area for this function. If the drill uses the

quad function ⎕ to accept answers to its questions, the user can simply retype the question as his response to obtain the correct result. For instance, suppose there is a function called *TST* which selects numbers at random and asks for their products.

```
      TST
WHAT IS 2×3
 ⎕: ■
```

Instead of typing in 6, we could just as easily enter 2×3 because, as you will recall, the quad function evaluates the input as if it were an APL statement before it releases it for further use.

```
      A←⎕
 ⎕: 2×3
■

      A
6
```

Therefore, it is best to accept all input in literal form, scan for things like 2×3, and if satisfied that the data entered is in suitable form, use the execute function to transform it into its numerical equivalent. You will discover many areas where this function can be used.

Summary

This draws to an end the descriptions of all the primitive functions that exist within the MCM/70 system. As a brief review, here is a list of all the mixed functions we have seen:

Function	Meaning
ρY	Dimension of Y
$X\rho Y$	Reshape Y to have dimension(s) X
ιY	First Y consecutive integers from origin (1 or 0)
$X\iota Y$	First location(s) of Y within vector X
$X[Y]$	Yth element(s) of X
$X\epsilon Y$	Membership of X in Y
$X\top Y$	Representation of Y in number system X
$X\perp Y$	Value of the representation Y in number system X
$X?Y$	X integers selected randomly without replacement from 1 to Y
$\phi[N]Y$	Reversal along the Nth dimension of Y
$\ominus Y$	Reversal along the first dimension of Y
$X\phi[N]Y$	Rotation by X along the Nth dimension of Y
$X\ominus Y$	Rotation by X along the first dimension of Y
$\ominus Y$	Transpose Y
$X\ominus Y$	Transpose Y according to X
$,Y$	Ravel Y (make Y a vector)
$X,[N]Y$	Catenate Y to the Nth dimension of X
X,Y	Catenate Y to the first dimension of X
$X\uparrow Y$	Take first or last $\lvert X$ elements of Y as X is + or
$X\downarrow Y$	Drop first or last $\lvert X$ elements of Y as X is + or
$\grave{\Delta}Y$	Indices of values of the vector Y sorted in ascending sequence
$\grave{\nabla}Y$	Indices of values of the vector Y sorted in descending sequence
$X\circ Y$	Ignore Y
$\ast Y$	Execute the literal vector Y as an APL statement

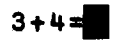
Practice Exercises

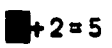
1. Using the decode function, solve the expression $5X^2+10X+7$ where X equals 2.

2. How many inches are there in 2 yards, 2 feet, 6 inches?

3. How many yards, feet, and inches are there in 102 inches?

4. Write an APL function that will ask addition and subtraction questions and except the answer in its literal form. Vary the position of the unknown in each question. For example, the following expressions should be typical displays of the function:





Chapter 20: ERROR REPORTS

Whenever the system is asked to execute an APL statement, it does so immediately. This is assuming of course that it can. Things such as misspelled variable or function names, forgotten numbers, or even insufficient memory space can block the system from carrying out its tasks. When this happens, the system suspends its execution and displays for you the type of problem it is encountering. This display is known as an error report. Here is a typical error report:

```
DOMAIN ERROR      error type
A÷6×9.5           erroneous text
```

On the second line, the cursor indicates which function is experiencing the problem. Here is another:

```
LENGTH ERROR
  3+6 5 7
```

The cursor is positioned on the first character associated with the faulty function.

The report is usually two lines long, but it can be more. Its length depends on the type of function in error. If it is the execute, the report is three lines long. Here is what it looks like if the expression ⍎'2+' is entered:

```
SYNTAX ERROR
⍎2
⍎'2+'
```

If the error occurs within a defined function, the system includes the function's name and the line number where suspension occured. Here is a function called *TIC* which has encountered an undefined variable name on line 4:

```
VALUE ERROR
TIC[4] A×6.2
```

If the line containing the error exceeds thirty-two characters in length, the system displays the portion with the error only.

There are nine different types of errors that can occur. The complete list, along with some of the reasons for their occurances, follows.

Domain Error

A *DOMAIN ERROR* occurs if at least one of the arguments used is outside the predefined limits of the function. Dividing by zero, catenating a literal to a numeric, and using a noninteger value where one is required, are all examples of function domains being exceeded.

```
      6÷0
DOMAIN ERROR
6÷0
      'SUM = ',2
DOMAIN ERROR
'SUM = ',2
      ⌊6.5
DOMAIN ERROR
 6.5
```

Index Error

If an attempt is made to index a non-existent coordinate of an array, the system will respond with an *INDEX ERROR*.

```
      A←7 8 9
      A[12]
INDEX ERROR
A[12]
```

Length Error

Trying to perform an operation on two arrays which have the same rank but nonconformable dimensions, causes a *LENGTH ERROR*. Examples are:

```
1:        5 3+6 2 4
LENGTH ERROR
5 3+6 2 4

2:        A←2 3⍴'BOXTOP'

          A
BOX
TOP

          B←4 2⍴'SENDTHIS'

          B
SE
ND
TH
IS

          B,A
LENGTH ERROR
B,A
```

Range Error

A *RANGE ERROR* is displayed whenever an attempt is made to create a number that cannot be represented by the MCM/70. The concept of infinity and imaginary numbers are subject to this message. Included here are also those numbers which lie outside the span of numbers from 7.237005577*E*75 to 7.237005577*E*75.

```
      10E100
RANGE ERROR
10E100
```

```
      23E16×17E60
RANGE ERROR
23E16×17E60
```

```
      ○/⍳0
RANGE ERROR
○/⍳0
```

This last example is an attempt to determine the identity element of the trigonometric function, while one does not exist.

Rank Error

If a certain function expects its arguments to be of conformable ranks, but they are not, a *RANK ERROR* will occur. Trying to add a vector to a matrix evokes this error.

```
      3 4+2 2⍴⍳4
RANK ERROR
  4+2 2⍴⍳4
```

Specifying the wrong number of coordinates when performing an indexing function will terminate with the same result.

```
      A←1 2 3
```

```
      A[2;3]
RANK ERROR
A[2;3]
```

Syntax Error

A statement which is grammatically incorrect, in APL terms, invokes a *SYNTAX ERROR* response. Some of the more common causes are:

1: Missing arguments

```
      2+
SYNTAX ERROR
2+
```

2: Missing operators

```
      8(6)
SYNTAX ERROR
 (6)
```

In traditional mathematics, this expression could mean the 8 and the 6 are to be multiplied together. But all operations must be stated explicitly in MCM/APL.

3: Unbalanced parentheses

```
      ((2+B)⌈B*X
SYNTAX ERROR
 (2+B)⌈B*X
```

4: Two juxtaposed variables

```
      A←1

      B←2

      A B
SYNTAX ERROR
A B
```

5: Invalid header line

```
      ∇I 10
SYNTAX ERROR
 I 10
```

If the header looks syntactically correct, check for the presence of a variable of the same name as that in the header.

Tape Error

If the system is physically unable to read from or write onto the tape cassette, it will indicate this by displaying a *TAPE ERROR* message.

Some of the probable causes are:

1. Cassette not mounted properly
2. The write enable cover is missing
3. Faulty tape

See Appendix B for more details on how tapes should be mounted and used, plus additional reasons for *TAPE ERROR* conditions.

Value Error

A *VALUE ERROR* indicates reference is being made to a variable that does not exist.

```
      6×B
VALUE ERROR
6×
```

The above error condition resulted due to one of the three following reasons:

1: *B* is the name of a variable that is not currently residing in the workspace

2: the name of the variable is misspelled

3: *B* is the name of a defined function which has not produced an explicit result

Workspace Full

When the system runs out of available workspace in which to store its data, or perform its tasks, it displays a *WS FULL* error message. The amount of workspace available in the basic MCM/70 is approximately 1,500 bytes. This is referred to as the "2K model." The K stands for kilo. Therefore, 2K means 2,000 bytes. (The missing 500 some odd bytes is used by the system functions and variables.)

Options of 4K and 8K byte workspaces are offered as well. The unit can also have one or two built-in tape cassette drives, with each cassette mounted holding approximately 110,000 bytes of information. Therefore, the limit that must be reached before this error condition occurs is dependent on the amount of workspace available. Basically, every operation must take place within the "main" memory, i.e., that portion which is not on tape.

```
      A←100 100 100 100 100 100⍴7.8
WS FULL
A←▮00 100 100 100 100 100⍴7.8
```

To estimate the amount of storage area your data needs, the byte requirements for all the different data types that can be entered are listed below.

any literal character	1 byte
any integer from 0 to 127	1 byte
any integer from 128 to 32,767	2 bytes
any integer from 32,768 to 2,147,483,647	4 bytes
any integer above 2,147,483,647	8 bytes
and real or floating-point numbers	8 bytes

Negative values require the same amount of storage space as their positive equivalents.

Chapter 21: SYSTEM FUNCTIONS AND VARIABLES

Once you begin defining variables and writing your own functions, you will have to learn how to manage your workspace properly to insure optimum usage of the memory space available. To do this, you need a means of communicating with the system, not only to find out what is in the workspace but also to instruct it in organizing its contents in some orderly fashion. You may also want to change some of the basic assumptions the system has made in regards to its environment. These include such things as the number of significant digits shown for each value displayed on the screen; the length of time each line is displayed; and the origin used by many of the APL primitive functions.

Two types of vehicles have been developed to enable you to both communicate with, and to manage the contents of the workand some of the aspects of the MCM/APL system itself. One is called system functions and the other is called system variables. Both of these rather abstract objects are distinguishable from other defined functions and variables we are used to, by the fact that each of their names begins with the quad symbol □. These are the only instances where this symbol may be used to form the name of any of the workspace contents. In total there are five system functions and eight system variables, all of which are described in this chapter.

System Functions

Here is a list of the five system functions predefined in the MCM/APL system:

Name	Meaning
□*EX*	Expunge or erase one or more specific objects from the workspace.
□*FN*	List the names of the functions currently residing in the workspace.
□*OF*	Terminate the session.

⎕VA List the names of the variables currently residing in the workspace.

⎕WC Clear the entire contents of the workspace.

All of these functions produce explicit results and all, with the exception of *⎕EX*, are defined as being niladic functions - meaning they require no arguments. For the one that does, the *⎕EX* function, its syntax is

```
⎕EX A
```

where *A* is a literal containing the name, or names of the item(s) to be erased. For instance, assume we have three defined functions called *AGE*, *FND*, and *T*, and four variables, called *TOT*, *PI*, *A*, and *Q1* presently defined in the workspace. We can quickly verify this by executing both the *⎕FN* and *⎕VA* functions.

```
      ⎕FN
AGE
FND
T

      ⎕VA
A
PI
Q1
TOT
```

Notice the result obtained by executing each function is a 2-dimensional array, with each row containing the name of a function or, in the case of *⎕VA*, the name of a variable. The results will always have 4 columns, but the number of rows is dependent on the number of names involved.

To erase one of these objects, we must enclose its name in quotes.

```
      ⎕EX 'FND'
```

The above expression just erased *FND* from memory. Now when we display the names of the functions present, we get the following:

```
      ⎕FN
AGE
T
```

If we wish to erase more than one item, their names must be separated from one another in the right argument by at least one space.

```
      Z←'AGE TOT'
■
      ⎕EX Z
■
      ⎕FN
T
      ⎕VA
A
Q1
PI
```

They can even be in matrix form.

```
      ⎕EX 2 4⍴'A   PI  '
■
```

Which means the output from both ⎕FN and ⎕VA can be used as input to ⎕EX.

Actually, the argument can be of any rank up to 32, with each row containing an object's name.

If we want to erase all the contents of the workspace, we could type either

```
      ⎕EX ⎕FN    and    ⎕EX ⎕VA
```

or

```
      ⎕EX ⎕FN,[1]⎕VA
```

or simply

```
      ⎕WC
```

The *⎕WC* function clears everything from the workspace, plus it reinitializes all the system functions and variables to their original settings. Parameters such as index origin, state indicator and random link are all set back to the values they contained at the beginning of the session.

If we execute either the *⎕FN* or the *⎕VA* function in a clear workspace, the results will always be empty 0 by 4 arrays.

```
      ⎕WC
MCM/APL      indicates a clear workspace

      ρ⎕FN
0  4

      ρ⎕VA
0  4
```

The last system function to be discussed is the *⎕OF* function. We saw its purpose in Chapter 1. One thing to remember when using it is that everything presently defined in memory vanishes immediately after this function is evoked. If your computer has a tape cassette, the system will remain active long enough to store all the contents onto the tape before it complies with the *⎕OF* request. This is to guard against any accidental termination and to assure that all of your data has been safely stored on the cassette.

When the system has completed its "housekeeping" and has in fact "signed off", all electrical power flow within the MCM/70 is turned off. It remains off until the START key is pressed.

System Variables

System variables provide us with a means of altering certain aspects of the MCM/APL system which are preset by the system at start up time. Here is a list of the ones available, along with a brief description of their uses:

Name	Meaning
$\square CT$	Comparison tolerance
$\square IO$	Index origin
$\square LC$	Line counter
$\square PP$	Print precision
$\square PT$	Print time
$\square RL$	Random link
$\square SI$	State indicator
$\square WA$	Workspace available

Comparison Tolerance

The first one on the list, $\square CT$, is used in conjunction with the relational primitives $< \leq = \geq > \neq$ and the maximum $\lceil$ and minimum $\lfloor$ functions. When you are comparing two numbers according to their respective magnitudes, the computer must, at some point, set a limit on the accuracy to which the comparison is being made. For instance, the following statement returns a 0 result if it is conducted in a clear workspace.

```
      2=2.000000001
0
```

The value 2 is not equal to 2.000000001, but it is very close. If we inserted a few more zeros, the system would eventually say they were equal. But instead of doing that, we could increase the comparison tolerance level of the system to a point high enough that it will assume that values such as the two above are in fact equal. Before we do this, let us find out what its present setting is.

```
      ⎕CT
4
```

To better understand what this number represents, we had better look again at our definition of a byte. We saw earlier that the amount of workspace area is measured in bytes. A byte is like an inch. It is used in the computer industry as a unit of measurement to determine memory size. Instead of saying the computer has so many square inches of memory, we say it has so many bytes of memory. And the number of bytes that can be contained in 1 square inch is increasing all the time. Even though a byte is used as a unit of measurement, it, like the inch, can be broken down even further.

One byte in the MCM/70 is composed of 8 bits. A bit is the lowest possible measurement unit. Therefore, a character which requires 1 byte of storage space could also be said to require 8 bits. And numbers which require 8 bytes of memory could be thought of as needing 64 bits (8 bytes × 8 bits each). But how does all this relate to the ⎕CT variable? Well, when calculations (including comparisons) are performed by the MCM/70, they are done to a degree of accuracy equal to sixteen significant digits. In the case of the relational functions, and ceiling ⌈ and floor ⌊, the system allows for a little "fuzz factor" or approximation tolerance to be permitted in its evaluation. Remember the ⎕CT variable contained the value 4? This 4 represents the number of groups of 4 bits it will ignore in its evaluation. Therefore, in a clear workspace, the system neglects to include the last 16 bits (4 groups of 4 bit quantities) in all of its comparisons. If you wanted to eliminate the "fuzz factor" entirely and obtain the most precise comparison possible, you would respecify ⎕CT to have the value 0.

```
      ⎕CT←0
      
      1234567890123456=1234567890123457
0
```

By increasing the tolerance a little, the system can be made to believe these two numbers above are equal.

```
      ⎕CT←2
      
      1234567890123456=1234567890123457
1
```

As we further increase the tolerance, so too are we further increasing our level of acceptance of various comparisons. The highest tolerance level we can have is 13. Here it is set to this level and, as you can see from the ensuing calculations, its effects are expressed by the results.

```
      ⎕CT←13
■

      2=2.1
1

      2=2.01
1

      2=2.0001
1

      ⌈2.0001
2
```

So, depending on the precision required, we can adjust the comparison tolerance level to suit the need.

Index Origin

For those who prefer the origin of their numbering systems to be 0 instead of the assumed setting of 1, the ⎕*IO* variable has been implemented. The affects of changing this parameter are experienced by all the functions that refer to it when calculating their results. Such functions as ⍳*Y* and *?Y* fall into this category.

```
      ⎕IO←0
■

      ⍳5
0 1 2 3 4

      ?1
0
```

All indexing calculations are also included. The coordinate values of all rectangular arrays are decremented by one in an origin zero environment.

```
      A←6 10 ¯2 3.4 ¯7
■

      A[0]
6

      A[3]
3.4
```

This is also carried over to other primitives which perform their computations along specific coordinates of arrays.

```
      M←3 4⍴⍳12
■

      M
0  1  2  3
4  5  6  7
8  9 10 11

      +/[0]M
12 15 18 21

      +/[1]M
6 22 38

      0 1⌽M
0  4  8
1  5  9
2  6 10
3  7 11

      1⌽[0]M
4  5  6  7
8  9 10 11
0  1  2  3
```

Line Counter and State Indicator

There are three system variables to which you are unable to specify values. Two of them are the line counter ⎕*LC* and the state indicator ⎕*SI*. (⎕*WA* is the third)

```
      ⎕LC←2
SYNTAX ERROR
⎕LC←2
```

It would appear as though the system is responsible for these two, but in actual fact, it is the user who indirectly determines their values.

The ⎕*LC* variable contains an empty numeric vector when the system has a clear workspace. But when a user defined function is executing, it becomes a vector consisting of one or more values. Assume it contains the value 2.

```
      ⎕LC
2
```

The 2 represents the line number of a user defined function which is currently being executed. The system was in the process of executing line 2 when ⎕*LC* was displayed. If the function has been suspended for some reason, the suspension occured while line 2 was being evaluated.

In conjunction with the ⎕*LC* variable, the ⎕*SI* variable contains the name of the function currently executing. If the function has become suspended, an asterisk accompanies the function name.

```
      ⎕SI
DET*

      ⎕LC
2
```

It appears above as if the execution of the function *DET* has been suspended at line 2. If *DET* had been called by some other function, the name of the calling function, and the line number from which the call was made, would also be contained in ⎕*SI* and ⎕*LC* respectively. Suppose *DET* was called by a function named *MST* on line 6. The contents of the two system variables would look like this:

```
      ⎕SI
DET*
MST

      ⎕LC
2 6
```

If *MST* were again executed and *DET* became suspended, but this time on line 4, the two variables would contain the following:

```
      □SI
DET*
MST
DET*
MST

      □LC
4  6  2  6
```

The latest suspension appears in the first row of the *N* by 4 matrix of *□SI* and as the first element in the *N*-length vector represented by *□LC*. Only suspended functions are flagged with asterisks. The "pendant" ones, like *MST*, are not.

As more and more functions become suspended and both the state indicator *□SI* and the line counter *□LC* grow in size, less and less memory space becomes available for other uses. After a while, you could eventually use up all the workspace area by simply filling up the state indicator with the names of suspended and pendant functions. Therefore, you should always keep these two variables clear of data whenever possible. This is accomplished by keying in the branch arrow → only, and pressing the RETURN key. One branch arrow on a line by itself is needed for every <u>suspended</u> function name listed.

```
      □SI
DET*
MST
DET*
MST

      →
■

      □SI
DET*
MST

      →
■

      □SI
 (blank screen)

      ρ□SI
0  4

      ρ□LC
0
```

Print Precision

We saw in Chapter 2 that the ⎕*PP* variable was needed to vary the number of significant digits of a number displayed on the screen. The system default is five. For instance, the expression 2÷3 creates a repeating decimal result.

```
      2÷3
.66667
```

Even though the calculation is carried out to sixteen decimal places, only the first five significant figures are displayed, with rounding done on the last figure.

We can easily determine the print precision setting.

```
      ⎕PP
5
```

And altering it is just as easy.

```
      ⎕PP←2
■

      2÷3
.67
```

The values assigned to the variable can be any integer from 2 to 16.

Print Time

Normally when the computer displays something on its screen, it leaves it there indefinitely. Removing it requires you to press the RETURN key. But sometimes you will want various pieces of data to be displayed automatically, a piece at a time, without your intervention required each time. To do this, you must reset the print time variable ⎕*PT* from its assumed setting of infinity to some reasonably measurable quantity. But before we do this, let us see how infinity is denoted for this variable.

```
      ⎕PT
0
```

Zero is used to represent infinity to the system for this variable only.

Now to change it, we simply assign it any integer value between 1 and 255. The value of the integer assigned represents the number of tenths of seconds the system is to display its output. For instance, the computer will display its output for 3 tenths of a second if $\square PT$ is set to 3. Five second displays would require $\square PT$ to be set to 50. This ability to vary the display time is most useful when displaying multidimensional arrays and when executing a defined function which is interactive with the user.

If at any time you wish to stop the system from automatically displaying lines, press the SPACE bar. This causes the system to leave the line currently on the screen for an indefinite period of time, or until you signal to it to resume its automatic display. This is done by pressing the RETURN key.

Random Link

When using either the roll or the deal function to select numbers at random, the system tries to normalize the distribution of its selection as much as possible. This is to insure that we get the most evenly distributed random sample possible. To do this, it must continually change the starting point used by its selection algorithm after each ? function is executed. This starting point is a link number in the chain of random numbers the system can generate. Each time a roll or deal function is done, a different link is designated as the new starting point. You will notice that whenever you start a session or clear memory, the random link is always 16807 (i.e., 7*5).

```
      □RL
16807
```

And after executing either of the query functions, $\square RL$ is reset to 6874.

```
      ?10
7

      □RL
6874
```

By executing several query functions and displaying the ⎕*RL* variable after each one, you will notice a definite pattern in the sequence in which the random links are determined. The reason for this is in case you wish to generate a certain set of observations (numbers) again to re-establish an exact replica of some previously generated observations. But in other instances you may want your data to be generated, using an entirely different link, thus eliminating any chance of coincidence or redundancy in the generated data. To do this, you simply reset the random link variable yourself to and integer value from 1 to 32,767.

```
      ⎕RL←160
■
      ?200
31
```

Workspace Available

To determine the amount of memory space available for your use, you use the system variable ⎕*WA*.

```
      ⎕WA
1548
```

Depending on whether you have a "2, 4, or 8K" computer, the value represented by this variable when the workspace is clear, is the number of bytes the computer has. This space is used to contain both functions and variables, plus for performing whatever calculations you request it to do. By using this variable, you can determine how much space is required by your programs and data, and which is the best way to load up memory in order to optimize the space that is available.

Chapter 22: IDENTITY ELEMENTS

When we covered the reduction function earlier, there was one type of argument that was not mentioned - the empty vector. If this data type is used as input, the computer's response may look a little strange. Here are a few examples:

```
      +/⍳0
0

      ×/⍳0
1

      ●/⍳0
RANGE ERROR
●/⍳0
```

You may find the pattern is a bit hard to detect. This is mainly due to the fact that there is none. What the empty vector does here is cause the system to return the identity element of the primitive function used, if it has one, that is. You can see the function ● does not. But what is an identity element?

Mathematics defines identity elements as being those unique quantities that, whenever used as input to some function, have no effect on the result. For instance, when adding two numbers together, if one of the numbers is 0, the result will always be equal to the other number.

```
      4+0
4

      0+6.7
6.7
```

It does not matter if the 0 appears as the left or the right argument, the result is still the same. Therefore, the value 0 is defined as being the identity element of the addition function. In the case of the multiplication function, the identity element is 1.

```
      1×¯5
¯5

      6×1
6
```

There are actually three kinds of identity elements. There are left identities, right identities, and the identity elements. When a unique quantity such as the 0 and 1 we saw above can be either the left or right argument of the function, it is called "the identity element." When it can be the left argument only, it is called the left identity. If it can appear as the right argument only, it is called the right identity element. A typical right identity element is the 0 when used by the minus function.

```
      6-0
6

      ¯10-0
¯10
```

Another is the value 1 when used as a divisor.

```
      3÷1
3

      ¯10.45÷1
¯10.45
```

A much easier way to determine the identity element of each primitive is to simply employ the primitive to do a reduction along an empty vector argument. (Note: Only scalar dyadics can be used.)

```
      +/⍳0
1

      -/⍳0
0
```

Notice MCM/APL does not distinguish between the various types of identity elements, but rather indicates only that one exists and what it is. For those primitives that do not have any, the responses are *RANGE ERROR*'s.

Function	Identity Element	Which Argument (right or left)
$X+Y$	0	both
$X\times Y$	1	both
$X-Y$	0	R
$X\div Y$	1	R
$X*Y$	1	R
$X\bullet Y$	both	
$X\lceil Y$	$^{-}7.237...E75$	both
$X\lfloor Y$	$7.237...E75$	both
$X\mid Y$	0	L
$X\circ Y$	none	
$X!Y$	1	L
$X\vee Y$	0	both
$X\wedge Y$	1	both
$X\overset{\star}{\vee}Y$	none	
$X\overset{\star}{\wedge}Y$	none	
$X<Y$	0	L
$X\leq Y$	1	L
$X=Y$	1	both
$X\geq Y$	1	R
$X>Y$	0	R
$X\neq Y$	0	both

Note: The identity elements for the relational functions < ≤ = ≥ > ≠ apply for logical arguments only.

Bibliography

Berry, P.C., APL\360 Primer, White Plains, N.Y., IBM Corp., Form No. GH20-0689, 1969

Day, M., MCM/70 Introductory Manual, Willowdale, Ontario, Micro Computer Machines, Inc., 1974

Dept. of Education, Audio Course in APL, Los Angeles, Calif., IBM Corp., 1971

Falkoff, A.D., and K.E. Iverson, APL\360 User's Manual, Yorktown Heights, N.Y., IBM Corp., Form No. GH20-0683, 1968

Gilman, L., and A.J. Rose, APL\360 An Interactive Approach, New York, John Wiley & Sons, Inc., 1972

Grey, D.L., A Course In APL\360 With Applications, Reading, Mass., Addison-Wesley Publishing Co., 1973

Hanson, J.C., and W.F. Manry, APL: an intro, Atlanta, Georgia, Atlanta Public Schools, 1971

Iverson, K.E., ALGEBRA: An Algorithmic Treatment, Menlo Park, Calif., Addison-Wesley Publishing Co., 1972

Katzan, H., APL User's Guide, New York, Van Nostrand Reinhold Co., 1971

Pakin, S., APL\360 Reference Manual, Chicago, Science Research Assoc., Inc., 1972

Smyth, J.M., York APL, Toronto, Ontario, Ryerson Polytechnical Institute, 1972

Appendix A: PRIMITIVES, SPECIAL SYMBOLS AND OTHERS

Primitives

Meanings

Symbol	Monadic	Page	Dyadic	Page
+	Identity	4.5	Addition	2.2
-	Negation	3.6	Subtraction	2.2
×	Signum	4.4	Multiplication	2.7
÷	Reciprocal	3.7	Division	2.4
*	Exponential	4.7	Exponentiation	4.5
⍟	Natural logarithm	4.8	Logarithm	4.8
⌈	Ceiling	4.10	Maximum	4.9
⌊	Floor	4.11	Minimum	4.10
|	Absolute value	5.2	Residue	5.1
!	Factorial	5.2	Combination	5.3
○	Pi times	5.6	Circular	5.5
?	Monadic random	5.6	Dyadic random	12.12
<			Less than	6.1
≤			Less than or equal to	6.1
=			Equal to	6.1
≥			Greater than or equal to	6.1
>			Greater than	6.1
≠			Not equal	6.1
∨			Or	6.3
∧			And	6.3
⍱			Nor	6.4
⍲			Nand	6.4
~	Not	6.5		
⍴	Dimension of	11.5	Restructure	11.2
⍳	Index generator	11.10	Index of	11.7
[ ]			Indexing	11.13
∈			Membership	12.1
⍷			Encode	19.3
⊥ ⍎			Decode	19.1
⌽ and ⊖	Reversal	18.5	Rotate	18.1
⍉	Monadic transpose	18.6	Dyadic transpose	18.8
, and ⍪	Ravel	12.9	Catenate	12.5
↑			Take	12.10
↓			Drop	12.11
⍋	Grade up	12.2		
⍒	Grade down	12.4		
⍬			Null	19.3
⍎	Execute	19.5		
/ and ⌿			Compression	14.1
\ and ⍀			Expansion	14.5
f/ and f⌿	Reduction	7.1		
f\ and f⍀	Scan	7.6		
f.F			Inner product	17.1
∘.f			Outer product	17.5

Special Symbols

Symbol	Meaning	Page
■	Cursor	1.3
( )	Parentheses for nesting	2.7
→	Branch	15.1
←	Assign data to a variable	3.1
⍞	Quad for input and output	16.1
⎕	Quote-quad for literal input	16.5
'	Quote for literal data	3.4
∇	Del for mode change	8.3
¯	Negative sign	2.2
E	Exponential notation	4.6
.	Period or decimal point	2.2
:	Label separator	15.7

Valid APL Characters

Numerics

0 1 2 3 4 5 6 7 8 9

Alphabetics

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbols

¨ ¯ < ≤ = ≥ > ≠ ∨ ∧ − ÷ + × ? ⍵ ∈ ⍴ ~ ↑ ↓ ⍳ ○ * → ←

⍺ ⌈ ⌊ _ ∇ ∆ ∘ ' ⍞ ( ) [ ] ⊂ ⊃ ∩ ∪ ⊥ ⊤ | ; , : . / \

Overstrikes

ϕ (o and |)
● (o and *)
ᴪ (v and ~)
✱ (^ and ~)
⌀ (o and \)
¡ (' and .)
ŧ (- and ,)
⍋ (Δ and |)
⍒ (∇ and |)
⍞ (□ and ')
ᵑ (n and °)
⁄ (/ and -)
↚ (\ and -)
⊖ (o and -)
⊞ (□ and +)
⍿ (⊥ and °)
⍑ (T and °)
$ (*S* and |)

Interrupts

When the MCM/70 is computing, it is in complete control. It does not allow input from the user at this time and it only returns control back to the user when it needs more input, or after it has finished its calculations. If you instruct it to perform a specific task, then decide differently, there are two ways you can interrupt its computations. They are called the "hard" and "soft" interrupts.

The soft interrupt can be invoked by pressing the CTRL key then the ← key. This causes the computer to abandon its execution as soon as it has finished with the statement it is currently on. The hard interrupt causes the computer to stop much sooner. By pressing the CTRL key and the → symbol (upper shift ←), the system abandons its execution as soon as it can. It also displays a two line report. The first line contains the word *INTERRUPT*, and the second is the statement it was executing, with the cursor placed over the precise character where the interruption occured.

```
INTERRUPT
A←6+[redacted]50
```

One thing to note is that when you press the CTRL key down, hold it down until the lights on the screen stop flashing. Otherwise, the computer will not accept your interruption request.

Output Indicators

When output from the computer is being displayed on the screen, it is accompanied by one or more vertical bars at the right end of the screen. There are three types of these output indicators that can occur. They are as follows:

- Output line exceeds 31 characters in length. Press RETURN key to receive the rest.
- An array of rank greater than one is being displayed, a row at a time. Press the RETURN to view the next row.
- Either a single line of output, or the last row of an array is being displayed. When the RETURN key is pressed, the system is ready to either accept input or resume its execution.

Appendix B:

AVS-EASY

Almost every MCM/70 is equipped with the tape cassette option. Your unit may house either one or two tape drive mechanisms. Each drive can handle any number of cassettes, and each cassette can accommodate two major software features. One feature, called EASY (External Allocation SYstem), serves as a permanent filing system for both data and user defined functions. The other feature is called AVS (A Virtual System). It enables the cassette to act as an extension to the amount of main memory space available. Both of these features are included with every unit having at least one tape drive. Together they expand the power and capability of the MCM/70 far beyond its computational abilities described previously. This appendix discusses both features in detail, plus describes some of the characteristics of the MCM/70 tape cassettes.

Type of Cassette Used

The cassette used by the MCM/70 looks very similar to that used by audio tape recorders.

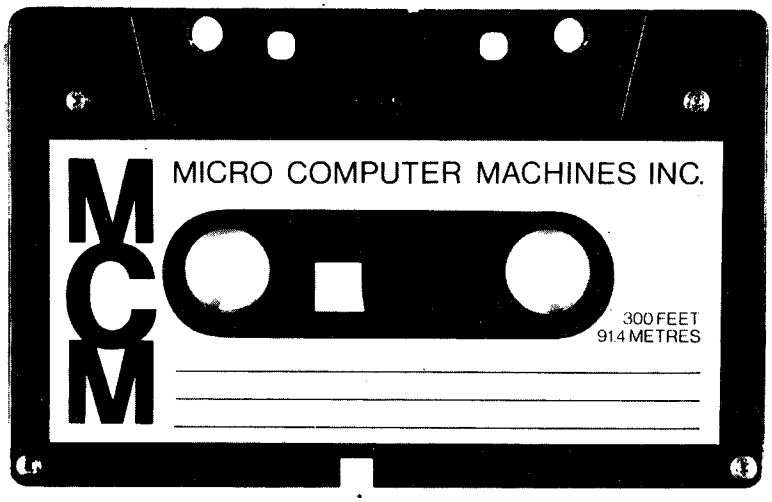


Figure B.1: MCM/70 Tape Cassette

It has the same dimensions as an audio cassette and in fact can be used by an audio recorder. But the opposite is not true. An audio cassette is unacceptable for use by the MCM/70 to record its electronic impulses as it is not of digital grade quality. It is also required to be equipped with what is known as a pre-recorded clock track. Tapes possessing these characteristics are supplied with each unit having the cassette option, plus additional cassettes are available from MCM/70 distributors on request.

Before attempting to use the cassette, it is advisable to read all of the following information, as rather dire consequences can occur if you are unfamiliar with either AVS or EASY prior to their use. Probably the most important thing to remember when using a cassette is that it must be closed before it is removed from the unit. The second most important is that each cassette must be initialized before it can be used. The procedures for doing both of these activities are discussed in detail later in this appendix.

Mounting and Dismounting a Cassette

Mounting the cassette in the MCM/70 follows the same procedure as that of the tape recorder. The exposed tape portion is inserted first with the side containing the statement DO NOT RECORD ON THIS SIDE being turned face down. Notice in Figure B.2 the offset notch at the bottom of the cassette makes it impossible for you to mount it any other way. If you mount the tape before beginning the session, it will wind back and forth for a few seconds as soon as you press the START key. Its activities during this time relate to the AVS feature and are discussed under that section.

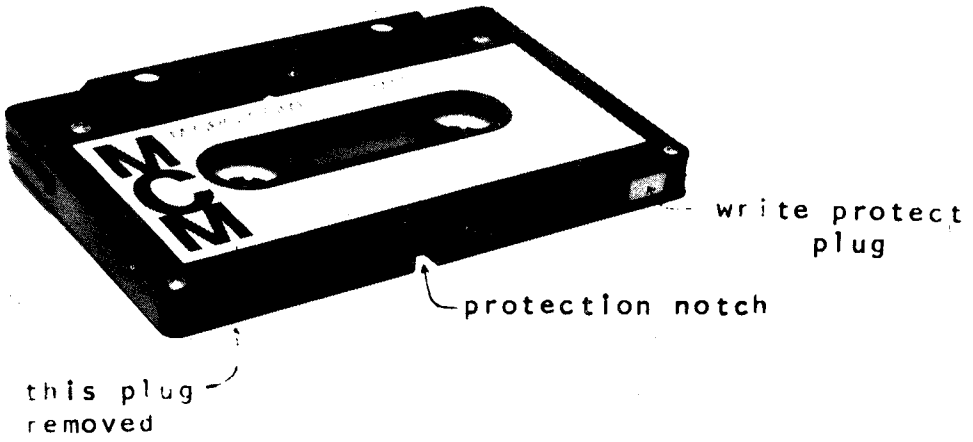


Figure B.2: MCM/70 Tape Cassette

Activating AVS

When the tape does move back and forth after the START key is pressed, it does not necessarily mean that AVS is being activated. What the system is doing is checking to see if AVS was active while this particular tape was mounted at the end of a session. If it was not, the system simply rewinds the tape. But if it was, the system restores whatever was in the computer's memory immediately prior to the session's termination back to its original status just prior to the ending of the session. This means that if there were three functions and two variables residing in the computer's memory at the time the session concluded, these same three functions and two variables would be automatically restored back into memory by the system as soon as the START key is pressed. How this is done is covered later, after you have learned a bit more about AVS and how the cassette is used as a storage medium. However, before we leave this topic, one point to note is that if AVS is active at the end of the session, the tape mounted in drive 1, or the system drive, must be mounted prior to the start of a session in which it will be used. When it is, AVS will also be automatically reactivated by the system. Tapes falling into this category are referred to as AVS tapes.

Since we have seen that AVS can be active, we may also assume it can be inactive as well. If a session is initiated without an AVS tape mounted, AVS is considered inactive. Therefore, the system recognizes that the size of the main memory is the only usable space it has at its disposal for storing functions and variables. Even if a cassette is mounted later on in the session, the system retains its previous understanding of the amount available storage space it has. It is not until it is explicitly stated otherwise by the user that the system acknowledges the tape as representing additional usable space for potential storage purposes. And the function used to accomplish this is the following:

```
□XS 0
```

This immediately makes whatever unused space there is on the mounted tape available to the system. In the event that its main memory should ever approach total capacity, the system can now call on the cassette to act as additional storage space, just as if it were part of main memory itself. More is mentioned in the following section about the activating and deactivating of AVS, and why the value 0 is used above.

Tape Organization

A cassette may be used to store either EASY or AVS related objects only, or it may be used to store both types simultaneously. With the AVS feature, there is no format restrictions imposed on the objects involved. A tape can contain either one or one hundred objects on it at any one time and in any order. The system is responsible for both writing them out to the cassette and reading them back again. But EASY works slightly different.

This difference exists because the two features are used for two different purposes. While AVS provides a means of extending the computer's memory, EASY enables objects to be retained indefinitely. The objects associated with AVS exist on tape only as long as AVS is active. Once AVS is made inactive, they are immediately removed from the tape. But the tape objects belonging to EASY are perminently filed on the tape. The only way they can be removed is by the user instructing the system to do so.

EASY acts as a perminent filing system. It can store objects on tape, read them back again, and erase them entirely. In order to do this, EASY organizes the tape's contents into groups. Each group can contain any number of defined functions and variables with no restrictions placed on either their conformability or content. The members in each group usually share some common bond, such as pertaining to the same application, so at least they are logically related. But this is not something that is required. It just makes it easier for the user to remember which objects refer to which application. For instance, one group could contain a set of routines for performing mortgage calculations, while another holds a surveying package, and still another has statistical analysis functions in it. The content of each group is strictly up to the user to define. And the size of each group is dependent on the amount of available space on the tape. Since each tape can hold over 100,000 bytes of information, a single group could theoretically be quite large. We say theoretically because one other restriction could come into play, depending on the memory size of the computer, which imposes a upper limit on the number of members possible within one group. Not only is the size of each group limited, but so too is the number of groups that can be established on a single cassette. The maximum number is 256, but we will see in the section entitled "Initializing a Tape" that this number is physically impossible to attain on some units. Although some units cannot handle 256 groups, all can easily

accommodate well over one hundred, which is quite adequate for most users. The number of groups each of your tapes contain will probably never exceed thirty.

Each group on a tape is designated by a number. The number used is any integer from 0 to 255 and is determined by the user when the group is created. This means group 0 could be the one containing the mortgage routines, group 27 the surveying package, and group 200 the statistical functions. Their creation can be in any sequence, independent of the number used, and their members may be located anywhere on the cassette.

Tape Related Functions

Apart from the system functions relating directly to AVS and EASY, there are 4 other functions that deal with the cassette in general. These five are the following:

Function Name	Meaning
□*XI*	initialize tape
□*XN*	list group numbers and their respective contents
□*XF*	close tape and deactivate AVS
□*WC*	close tape and clear workspace
□*OFF*	close tape and terminate session

The first function on the list, □*XI*, is probably the least used one of them all, yet its role is most important. Without it, none of the cassettes could even be accessed. Therefore, its detailed description follows immediately. The others, with the exception of □*XN*, are left until near the end of the appendix to permit some exposure to EASY and AVS first. In this way their purposes take on more relevance.

Initializing a Tape

Every cassette has over 100K bytes of usable storage space for user defined functions and variables. We have just seen that this space can be logically subdivided into smaller amounts called groups, enabling functions and variables which share some common relationship to be linked together by means of a group number. The membership in these groups does not have to be predetermined when the group is created. New members can be added to existing groups as well as old ones deleted or replaced. Both size and content of each group is continually changing as modifications, additions, and deletions of group contents take place. In order to make this possible, the system has to know two things. It must at all times know the numbers of the groups that have been created, and the names and locations of each of their respective contents. To do this, it establishes tape directories containing all the relevant information regarding each group and their individual members. These directories are automatically maintained by the system, with each group having its own directory.

One of the pieces of information stored in the group's directory is a list of the names of the items associated with the group. Another is their respective storage locations on the tape. Since the members of a group may be scattered throughout the tape, each of their locations must be contained in the directory to provide the system with a means of quickly accessing them. The system obtains its location address by reading some pre-recorded information from the tape at the time it writes the member. This information is put there during an initializing process the tape has gone through beforehand.

All tapes must be initialized before they can be used. New cassettes are initialized by the following process:

1. Mount the cassette in the tape drive. For those units having two drives, mount it in the left one.

2. Type in (↑0) □XI 0 and press RETURN. This system function instructs the computer to establish its addressing scheme on the cassette and to create a master group dirctory which will later contain the numbers of all the groups whenever they are formed in the future.

The entire operation takes approximately 10 minutes.

To re-initialize tapes that have already been initialized, type:

```
(10) □XI 1
```

This second function has the same effect as clearing out all the contents from every group on the tape. But instead of physically going through the entire tape to do this though, it simply destroys the master directory which points to all the group directories, and creates a new one. Even though data may still be on the cassette after this function is finished, they are irretrievable by the user and have no effect on future items recorded. Both functions provide a "clear" cassette suitable for use by EASY and AVS.

One point mentioned earlier referred to certain limits being imposed on both the number of groups a tape can hold, and the number of members that can be included in any particular group. This limitation is present only on the units having 2K memories. The reason it exists here is due to the amount of memory space consumed by the directories whenever a tape group is accessed. For instance, within the main directory (the directory of all the group directories), each group number listed in it takes up 3 bytes. And each member listed in a group directory uses 5 bytes. Since both the master directory and the directory of the group being referenced are brought into memory whenever the group is mentioned by the user, a sufficient amount of memory can be consumed by the system for its use only. Therefore, it's obvious that the more members in a group, and the more groups existing on a single cassette tend to impose on the amount of memory space available. But since this problem is apparent only on the 2K units, and only under unusual circumstances, you should not encounter it, if you keep your groups and their contents down to a reasonable number.

EASY

The primary purpose of EASY is to provide a convenient permanent storage medium for user defined functions and variables. As you know, at the end of every session, whatever is residing in main memory disappears and is no longer retrievable at a later date. In some cases this is advantageous, but in most it is not. One obvious disadvantage is the necessity for us to redefine the same functions and variables very time we wish to use them. EASY eliminates this.

It also provides us with the capability of retaining many different items on one cassette by dividing it into various groups. The entire EASY feature is comprised of the following three functions:

Function Name	Meaning
$\square XR$	read object(s) from a group
$\square XW$	write object(s) in a group
$\square XD$	drop object(s) from a group

In order to see how each of these functions works, let us assume we have a cassette mounted in the cassette drive, and the tape contains five groups of items.

Existing Groups

The groups currently residing on the cassette are each represented by a single value. To determine which values have already been assigned to groups, the following function is used:

```
      □XN ⍳0
0  1  2  10  200
```

This vector result indicates the tape has five groups numbered 0, 1, 2, 10 and 200 on it.

Contents of Existing Groups

To determine the names of the items in each particular group we use the same $\square XN$ function. But instead of issuing an ⍳0 as its argument, we use the group number.

```
      □XN 10
NIM
A
BIN
B1
```

The result is an N by 4 matrix with each row containing an object's name. The fourth column is added to facilitate readability when ravelled.

```
      ,⎕XN 10
NIM A   BIN B1
```

The ⎕XN function returns the names of the group's objects only. It does not indicate whether they are names of functions or variables. This can only be answered by reading them into main memory first.

Reading Objects From Tape

In the last section the system indicated there are four objects in group 10. To retrieve a copy of one of them, we execute the following:

```
      10 ⎕XR 'NIM'
```

The left argument represents the group number and the right is the name of the item being read. It must always be enclosed in quotes. The system function ⎕XR (external read) instructs the system to create a copy of *NIM* and place it in main memory. In this way the "original" *NIM* is left intact on the cassette to ensure availability for future use.

If copies of several objects are to be read, the right argument is expanded accordingly:

```
      10 ⎕XR 'NIM A B1'
```

It may be in the form of a vector, or it may also be a matrix. The advantage here being the use of the ⎕XN function to produce a right argument for the ⎕XR.

```
      ⎕←A←⎕XN 10
NIM
A
BIN
B1

      10 ⎕XR A
```

or simply

```
      10 □XR □XN 10
```

Actually the □XR function as well as the other EASY Functions can accept right arguments of any rank.

When reading objects into main memory you must be careful not to accidently replace any of the existing members. If there is something in memory with the same name as something to be read from tape, the system will first erase the main memory one and then read in the tape copy. This could be disasterous in some cases. Just remember it whenever you are using this function.

Writing Objects On Tape

Creating a cassette copy of an object in main memory follows the same procedure as that used by the □XR function. Below the item *AVG* is written into group 10.

```
      10 □XW 'AVG'
```

```
      ,□XN 10
NIM A   BIN B1  AVG
```

If group 10 did not exist beforehand, it would have been created by the system in the process. For instance, if *AVG* was also stored in the previously non-existent group 36,

```
      36 □XW 'AVG'
```

the new group would be created on the cassette and its associated member would written out.

```
      □XN 10
0  1  2  10  36  200

      □XN 36
AVG
```

Objects with identical names can reside in different groups on the same cassette, but they cannot both be affiliated to the same group. This is consistent with the principle applying to duplicate names in main memory.

Deleting An Object From A Group

Once objects are stored on a cassette, they remain there indefinitely, or until they are removed by the user. When it is decided that they should be removed, there are two ways to do it. One way is to re-initialize the tape as we saw earlier. When this is done, the system merely erases the main directory pointing to all the group directories on the cassette. It does not go through and clear off every individual item. But in most cases this type is not necessary since only specific objects are involved.

To delete tape objects selectively we use the □*XD* function. Here is an example showing two members of group 10 being deleted:

```
      10 □XD 'NIM AVG'
■

      ,□XN 10
A    BIN B1
```

Summary

This complete the description of EASY. Its principal use is to store both functions and variables on a relatively permanent basis, thus providing a means for not only establishing a library of personalized MCM/70 packages, but also furnishing MCM/70 users with a convenient mechanism for exchanging packages. The grouping of logically related items on the tape facilitates the organization of each cassette, plus it allows for several unrelated packages to be contained on a common device which greatly increases the efficiency of the tape space available. This grouping feature also forms a fundamental part of AVS as we shall see in the following sections.

AVS

In every computer there is a finite amount of memory space available in which to perform calculations and keep the programs and data involved. Some computers have more than others. For those having a somewhat lesser amount of memory, anything that can be done to optimize this space is certainly welcomed. One way optimization is achieved in the MCM/70 is by the computer language itself. APL is the most logical choice of languages for a small computer due to its extremely efficient coding properties. Programs that can be written in one line in APL take roughly 10 to 20 line in most other languages. Two prime examples of this are the computer languages known as Basic and Fortran. Most of the memory of the machines having these languages is consumed by the programs themselves. Very little is left over for data and calculation space.

Another advantage the MCM/70 has over most other small computers is AVS. The effect of this feature is to logically increase the memory size from its present 2, 4 or 8K capacity to an amount just over 100K bytes.

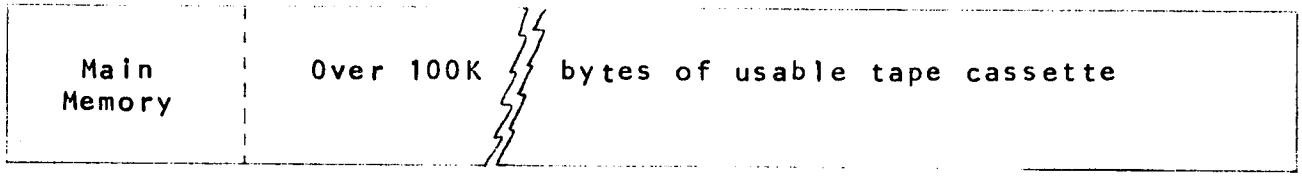


Figure B.3: Memory Space Available Under AVS

This does not mean that an array having dimensions 100 by 100, or a two-thousand line function can be defined. But it does mean that a vast quantity of items, each consuming less than the total number of main memory bytes available, can be maintained simultaneously by the system. To the user it appears as if they are all in main memory at the same time. These items include global variables, and inactive and pendant functions. For instance, in the expression

```
B+6
```

the *B* could be either in main memory or on the cassette. It is up to the system to determine this and take whatever action is necessary. If *B* is in main memory, it simply evaluates the statement. But if it

is on the tape, the system moves *B* back into memory and then executes the statement. The only indication to the user that the latter is taking place is if he sees the tape move.

The only time objects are moved to the cassette is if the system needs more memory space in order to carry out its activities. It swaps one or more items to and from main memory depending on when they are needed and what the space requirements are at the time. On top of this it also treats at least one of the groups on the tape as if it too were a part of main memory. The group selected is determined by the user. As an example, suppose the following functions are in main memory.

```
      □FN
AH
ASC
HIL
```

and the objects below are contained in group 6:

```
      □XN 6
SIG
ΔX
```

If group 6 is selected to be the active group, then its contents are just as accessable to the user as those in main memory. The activation of both AVS and a tape group are achieved at the same time and by the same function.

Activating AVS And A Tape Group

AVS can be "turned on and off" at anytime. During most sessions it will probably be "on" as the only reason for it not to be is if there is no tape to mount. The function used to activate it is

```
      □XS group number
```

The right argument can be any valid group number from 0 to 255.

Local And Global Groups

Under EASY all groups are regarded equally. They are all independent of one another and all are accessable to the user through the EASY functions. But under AVS they are treated slightly different. In this system, group 0 is designated as being the "global" group and all others are called "local" groups. The main distinction between these two classifications is that the global group is always active whenever AVS is active, whereas the local groups are only active if the user requests them to be. And only one local group may be active at a time.

Assuming no local groups are active, a schematic of the workspace would look like that in figure B.4.

Figure B.4: AVS and only the global group active

The function to obtain this condition is

```
□XS 0
```

Even though all the items associated with group 0 can be referenced in the same manner as those in main memory, they are not all placed into main memory when the group is made active. Only the directory of group 0 is read in. And it's only when the individual members in group 0 are referenced by the user that they get selectively copied in. The same rule applies to the contents of active local groups.

The following function activates group 6:

```
□XS 6
```

After executing this last function, the condition of the system appears to the user as it is represented in Figure B.5.

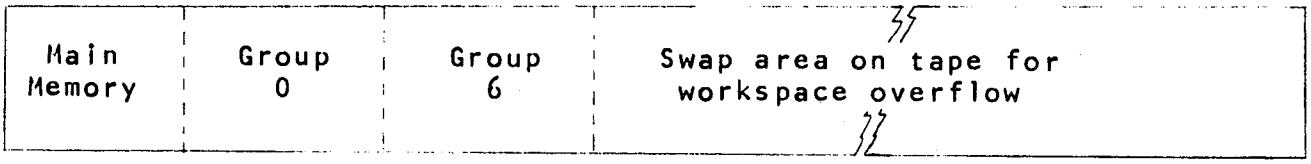


Figure B.5: AVS with both the global and local groups active

Since only one local group can be active at any one time, Figure B.5 illustrates the typical configuration of the workspace that exists in AVS. When the system is configured into this form, all the objects defined in the workspace section, group 0, and the active local group 6 may be referenced as if they all were contained in main memory at the same time. But one key point to remember is that any modifications made to any of the objects belonging to either the global or the local group are also reflected on the tape. In other words, if we change a function in group 6, we will find that the system will "update" group 6 to reflect this change once group 6 is de-activated. This also applies to the □*EX* function. If we expunge any members from these two groups with this function, the members are automatically dropped from the group as well as being erased from memory.

Resolution Of Group Synonyms

Periodically an activated group contains an object having the same name as another object presently in the workspace. Since no two items can have the first three characters in their names identical to another's, the system must take appropriate steps to prevent this from occuring. Figure B.6 shows only AVS and the global group as being active.

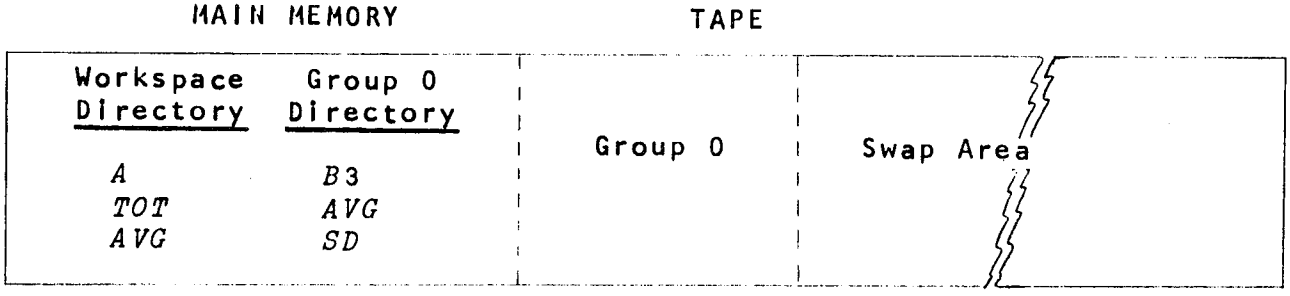


Figure B.6: AVS and group 0 active

Notice that an object named *AVG* appears in both directories. The system resolves this by deleting the workspace copy, thus producing a configuration like the one shown in Figure B.7.

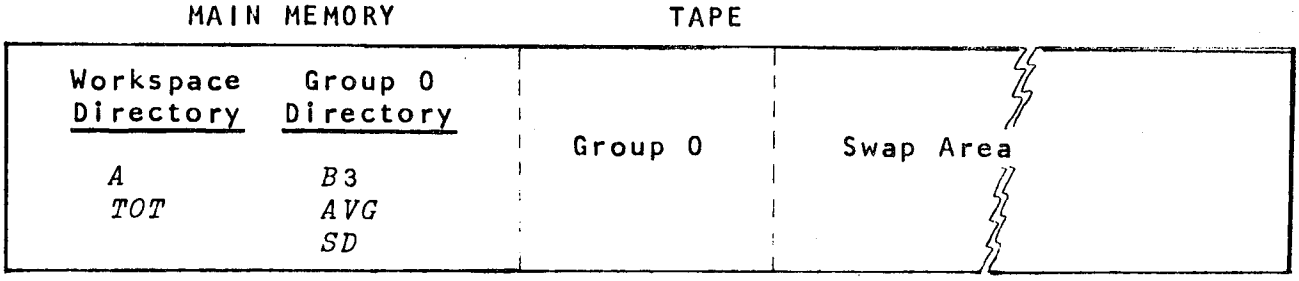


Figure B.7: Resolved AVS Workspace

The general rule for name redundancies between the objects in the workspace and those in any group is that the system will always resolve it by deleting the workspace ones. If the redundancy occurs between the global group and the active local group, the global group item is "concealed" within its directory while the redundancy exists.

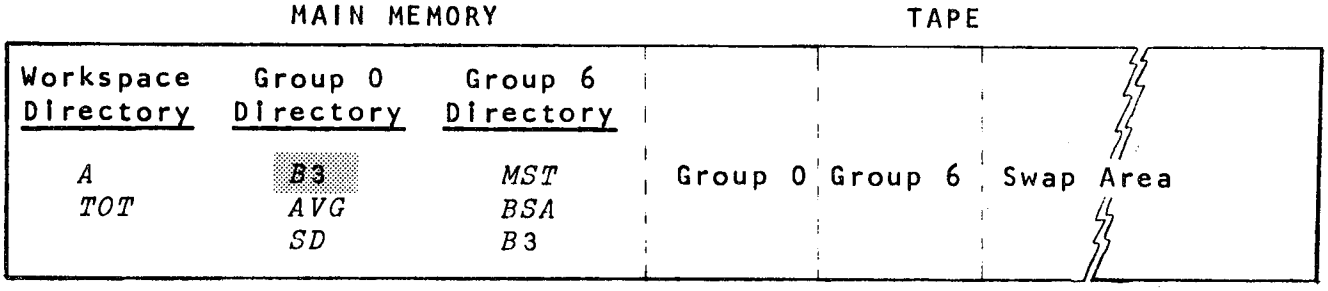


Figure B.8: "Concealed" object

The concealment exists until either the *B3* in group 6 is erased from memory or group 6 is deactivated.

Status Of AVS

At any time it is possible to determine whether AVS is "on or off." If on, it is also possible to find out which local group is active. Both pieces of information are derived from the function □*XV*. If AVS

is not active, the result is an empty vector. If AVS is active, the result is the number of the active local group.

```
      ⎕XV
10
```

If the result is 0, no local group is active at the time, but AVS is. (The 0 refers to the global group.)

Appending An Object To The Global Group

Before removing a cassette for the MCM/70 it is imperative that it be closed first. There are three ways in which this can be accomplished, as we shall see after the next section. But whatever method is used, the tape gets updated to reflect any changes that have occured to the contents of the global and active local groups. The contents of the workspace may, or may not, be written out to the tape at the same time, depending on the closing procedure used. If there are certain items in the workspace that are to be retained, the best way to insure that this is done is to assign them to the global group. This is accomplished in the following manner:

```
      ⎕XA 'object name(s)'
```

By doing this, the system will automatically write them out on tape before it does the close. And by assigning them to group 0, they will be made available immediately upon inserting the cassette again and activating AVS.

Creating An Entry In The Local Group Directory

The function ⎕XA gives us the capability of appending an object to the AVS global group only. To do this to any of the local groups, we have to either issue a ⎕XW or use the function ⎕XC. The difference between these two functions is that ⎕XW immediately writes the object involved on the cassette and updates the directory of the group indicated. The ⎕XC function does not write the object out. It simply places the name of the object in the group's directory. For instance, here is a typical example:

```
      11 ⎕XC 'BEB'
```

BEB may or may not be the name of an existing object in the workspace, since all the system is doing is adding the name to group 11's directory on the assumption that an object with this same name will be added to the group some time in the future. Both functions require that the local group receiving the item not be active at the time.

The convenience of this feature becomes apparent when closing the tape. If group 11 were made active after the ⎕*XC* added the name *BEB* to group 11's directory, and if an object called *BEB* gets created subsequent to the activation, and remains there until the tape was closed, the system automatically takes *BEB* from memory and places it into group 11 on the tape.

Closing A Tape

The most important thing to remember when using a tape is to close it before removing it from the unit. If this is not done, at least some, if not all of the data it contains may be lost. This is caused by the directories containing the addresses of all the data on the tape not being given an opportunity to be rewritten back on the cassette from where they reside in main memory. Therefore this step is a necessity whenever the cassette is used. To instruct the system to close the tape, there are three functions available. They are as follows:

```
⎕XF ⍳0

⎕WC

⎕OFF
```

The second two we have seen before but they are mentioned again here because of their association with AVS.

All three functions cause the system to update the tape copies of the global group and the active local group, if their workspace copies have been altered in any way. The function ⎕*XF* deactivates AVS but leaves the contents of the workspace as they are. The opposite is

true for the □*WC* function. It clears out the workspace but leaves AVS active. And the □*OFF*'s main responsibility is to terminate the session. But before it does, it takes all the contents of the workspace and the workspace status and preserves them in the present state on the cassette. This is to insure that nothing is accidently destroyed. In order to have the system restore them back in the computer at some later date, the cassette must be mounted in the tape drive before the START key is pressed. If this is done the computer will automatically reconstruct the saved items in memory to appear as though the □*OFF* function had never been executed. If it is not done, any attempt to access the tape will result in a *TAPE ERROR*. This, and other errors can occur with the tape as we shall see in this next section.

Tape Related Errors

Tape errors can occur for several reasons, and the messages displayed vary depending on the cause. Three error messages relate directly to tape access attempts. They are the following:

Message	Probable Causes
TAPE ERROR	This message is due to either a physical problem or a logical one. It's best to check for the former first, which are: - the tape is not mounted properly in the tape drive - the write enable plug is missing, prohibiting the system from writing on the tape - there is not enough available space on the tape to receive new data - due to excessive tape wear, the system is no longer able to read it The logical reasons are the following: - the tape has not yet been initialized - an attempt is being made to initialize a tape

that has not yet been fully rewound, or is still "open"

- reference is being made to an invalid tape drive number

- as mentioned in the previous section, if a tape mounted in drive 1 is closed by the □*OFF* function, while AVS is active, it must be mounted in the same drive before the START key is pressed

RANGE ERROR

- an attempt is being made to issue a □*XW*, a □*XD*, or a □*XC* to an active group. The group must be inactive before it can receive any of these requests.

- an attempt is being made to activate or deactivate a group which contains an object having the same name as one listed in the □*SI* matrix.

WS FULL

- a □*XF* has been issued, but there is not enough space available with main memory to contain all the items that presently reside there, plus those that have been written out on tape by AVS. Some of them must be expunged before the □*XF* function can be performed.

Multi-Cassette Systems

In a multi-cassette system, drive 1 (the tape drive on the left side of the unit) is called the system drive. This means that AVS uses this drive only when performing any tape operations. The EASY functions default to this drive, but it is able to access the other drives by the indexing method employed below.

10 □*XR*[2]'*TOT*' drive 2 (the one on the right side of the unit)

```
      □XN[4] ⍳0    drive 4 (an auxillary tape drive)

   99 □XW[1]'TOT' drive 1 (the system drive)
```

Above, we assumed the index origin to be 1. If it is 0, all three indices used would be decremented by 1.

ANSWERS TO PRACTICE EXERCISES

Chapter 2

1.

(a)	12	(b)	108	(c)	13.5
(d)	¯1.5	(e)	12.75	(f)	19.2
(g)	¯1.2	(h)	30	(i)	10
(j)	16	(k)	.0512	(l)	7

2.
```
      11-¯4
```

3.
```
      (6×7)÷3
```

4.
```
      1+2+10+5+3+4
```

5.
```
      22+23+26+23+22+27+25
168
```

6.
```
      (22+23+26+23+22+27+25)÷12
14
```

7.
```
      14×.83
11.62
```

Chapter 3

1.

(a)	42	(b)	5	(c)	*DOMAIN ERROR*
(d)	¯3	(e)	¯.63158	(f)	*DOMAIN ERROR*
(g)	¯3.5	(h)	10.5	(i)	10
(j)	¯10	(k)	¯4	(l)	*SYNTAX ERROR*
(m)	¯30	(n)	0	(o)	¯5

2.
```
      A←B←C←10
      A
1 2 3 4 5 6 7 8 9 10
      B
1 2 3 4 5 6 7 8 9 10
      C
1 2 3 4 5 6 7 8 9 10
```

3.

Valid	Invalid
A	*F_D*
ΔΔΔ	6*F*
MAXIMUM	*Z∨Y*
PAL	
Δ2Δ	
*Y*2684	

4.
```
      VEC←1 2 3 4 5
      HD←'VECTOR IS'
```

5. (a) 7 8 9 10 11 (b) ¯9 ¯8 ¯7 ¯6 ¯5
(c) 2 4 6 8 10 (d) ¯2 ¯1 0 1 2

6.
```
      (5÷9)×67-32      or      (5×67-32)÷9
19.444
```

7.
```
      15×12
180
```

Chapter 4

1. (a) 5 8 (b) 10 19 (c) 4 1 2
(d) 42 53 (e) 3 4 5 5 5 (f) 4 9
(g) 8 27 (h) 1 2 4 8 16 (i) 10
(j) *RANGE ERROR* (k) 9 (l) 4 5
(m) ¯1 1 0 (n) ¯.56 (o) 5
(p) 6*E*34 (q) 5 (r) 9

2. (a) 3*2 (b) (3*3)+2*4 (c) 4*¯2
(d) 25*.5 (e) (*X*2)+*Y*2 (f) (3×*X*2)+(2×*X*)-1

3.
```
      4*2 3 4
16  64  256
      5*2 3 4
25  125  625
```

4.
```
      4*÷2 3 4
2  1.5874  1.4142
      5*÷2 3 4
2.2361  1.17  1.4953
```

5.
```
      ●2 10 27.5
.69315  2.3026  3.3142
```

6.
```
      3●27 243
3  5
```

7.
```
      13÷2
7.5
      69÷12
5.75
```
Therefore the 12 for 69 is the cheaper of the two.
```
      ((13÷2)-69÷12)×24
18
```

Chapter 5

1. (a) 0 (b) .5 (c) 7 26.2 0
   (d) 1 (e) 24 (f) 6 2 1
   (g) 84 56 35 (h) 3.1416 6.2832 9.4248
   (i) 0 1 0 ¯1 (j) 0 1.5708 (k) any integer from 1 to 10
   (l) any integer from 1 to 100

2.
```
      12|337
1
```

3. Circumference
```
      ○3
9.4248
```
   Area
```
      ○3*2
28.274
```

4.
```
      A←○30÷180
      B←○60÷180
      C←○120÷180
      D←○1
      ((1○A)*2+(2○A)*2
1
      ((1○B)×2○A)+(2○B)×1○A
1
      ((3○D)-3○C)÷1+(3○D)×3○C
1.7321
      3○B
1.7321
```

5.
```
      ÷10.5
```

6.
```
      ?5⍴2
```

Chapter 6

1. (a) 1 (b) 1 0 1 1 (c) 1 0 1 1
   (d) 0 (e) 0 0 0 (f) 1 1

2. (a) $A \leftarrow 8 \times B > C$ (b) $X \leftarrow X \times X \neq 3$

Chapter 7

1. (a) 18 (b) 2 (c) ¯3.8
   (d) ¯1224 (e) 18 15 9 2 (f) 2 1 5 2
   (g) ¯3.8 16.2 6 (h) ¯1224 61.2 6

2. (a)
```
      +/32 45 27 36 24
164
```
(b)
```
      .02×+/32 45 27 36 24
3.28
```

3.
```
      H←2100 3600 5900 7100 9200
```
(a)
```
      +/H
27900
```
(b)
```
      +\H
27900  25800  22200  16300  9200
```

4.
```
      +/6 35 57×.25 .05 .01
3.82
```

5. (a) `⌈/A`
(b) `⌊/A`
(c) `(⌈/A)-⌊/A`
(d) `(+/A)÷-/A=A`

Chapter 8

1. Execution or Calculator mode and Definition mode

2. By keying in the ∇ symbol (del)

3. It displays the number of the next line available in the function to receive input

4.
```
      ∇C←A HYP B
[1]   C←((A*2)+B*2)*.5
      ∇
```

5. (a)
```
      X←10?50
```

(b)
```
      STATS
NUMBER OF OBSERVATIONS:
10
LARGEST VALUE:
33
SMALLEST VALUE:
3
```

(c)
```
      ∇STATS
[7]   'RANGE:'
[8]   (⌈/X)-⌊/X
```

(i)
```
∇STATS
[8][4.1]
[4.1]'AVERAGE:'
[4.2](+/X)÷⍴X=X
```

(j)
```
[4.2](+/X)÷⍴X=X∇

      STAT
NUMBER OF OBSERVATIONS:
10
OBSERVATIONS ARE:
36  10  3  19  7  25  22  33  21  20
AVERAGE:
19.6
SMALLEST VALUE:
3
RANGE:
33
```

Chapter 9

```
Maximum is 2.
Minimum is 0.

3

0

 ∇R←LOG X
[1]R←⍟X
 ∇

 ∇A←AVG N
[1]A←(+/N)÷⍴N=N
 ∇

 ∇L REC W
[1]'PERIMETER IS:'
[2]2×L+W
[3]'AREA IS:'
[4]L×W
 ∇

 ∇TOT←UNITS TIMES COST
[1]TOT←+/UNITS× COST
 ∇
```

(d)
```
      ]∇
```

```
      STATS
NUMBER OF OBSERVATIONS:
10
LARGEST VALUE:
36
SMALLEST VALUE:
3
RANGE:
33
```

(e)
```
      ∇STATS
[9][2.1]
[2.1]'OBSERVATIONS ARE:'
[2.2]X
```

(f)
```
      [2.2]X∇
```

```
      STATS
NUMBER OF OBSERVATIONS:
10
OBSERVATIONS ARE:
36  10  3  19  7  25  22  33  21  20
LARGEST VALUE:
36
SMALLEST VALUE:
3
RANGE:
33
```

(g)
```
      ∇STATS
[11][5]
[5] (press CTRL, SHIFT, and then the ← key)
[6] (press CTRL, SHIFT, and then the ← key)
[7]
```

(h)
```
      [7]∇
```

```
      STATS
NUMBER OF OBSERVATIONS:
10
OBSERVATIONS ARE:
36  10  3  19  7  25  22  33  21  20
SMALLEST VALUE:
3
RANGE:
33
```

Chapter 10

1. global and local

2. The name of a local variable appears in the function header line whereas the name of a global variable does not.

3. Local variables are active and any global variables with the same names as the locals are inactive. All other global variables are still active.

4. When the function whose header line contains the local variable names is executing.

5.
```
      ∇Z←X PBS Y
[6][0]
[0]∇Z←X PBS Y;A;B;C
[1]∇
```

6.
```
      ∇PERT
[6][0]
[0]∇PERT;A;Q;R
[1]∇
```

Chapter 11

1. (a) 1 2 3 4 5 6 7 8 9 10
   (b) 2 4 6 8 10 12
   (c) ¯2 ¯1 0 1 2 3 4
   (d) 10
   (e) (blank screen)
   (f) 1
   (g) 6
   (h) 1 2 3 4 5 6 7 8 9 10
   (i) 10 20 10
   (j) 1 2 3 4
   5 6 7 8
   9 10 11 12
   (k) (blank screen)
   (l) 0

2. (a) 12
   (b) 15
   (c) 12 13
   (d) 19 20 21 22
   (e) 11 12
   15 16
   (f) 2 2
   (g) 13 17 21
   (h) 19 20 21 22
   15 16 17 18
   (i) 2 4
   (j) 11 12 13 14
   (k) 11 12 13 14
   15 16 17 18
   19 20 21 22
   (l) 11 12 13 14
   15 16 17 18
   19 20 21 22

3. (a) `A←2 3 4⍴200+⍳24`

   (b) `A[2;3;1]←36`

   (c) `10×A[1;;]`

   (d) `A[2;1;]←0`

4. (a)
```
      (V<0)/⍳⍴V
2 5
```

(b)
```
      V[(V<0)/⍳⍴V]←10
      V
6 10 2.3 7 10 0
```

Chapter 12

1.
```
(A∈10+⍳10)/⍳⍴A←10?50
```

2.
```
 ∇R←SORT A
[1]R←A['ABCDEFGHIJKLMNOPQRSTUVWXYZ'⍳A]
 ∇
```

3.
```
      13↓17↑'THE LAKE WAS LIKE A SHEET OF GLASS'
LIKE
```

4.
```
A[⍋'ABCDEFGHIJKLMNOPQRSTUVWXYZ'⍳A[;6];]
```

Chapter 13

1.
```
      M
 10   5   517
  1   2   110
  5   7   782
  7   8   627
 31  15  6013
122 152  7551
  4   5   981
-10 -15   941
 27  33  1595
 62  57  2128
  1   1    17
  1   2    33

      +\M
 532   522   517
 113   112   110
 794   789   782
 642   635   627
6059  6028  6013
7825  7705  7551
 990   985   981
 916   926   941
1655  1628  1595
  19    18    17
  36    35    33

      +/+/M
21828
```

Chapter 14

1. (a) 2 4 (b) 2 0 3 (c) (blank screen)

   (d) 0 (e) 1 2 3 4 5 (f) 0 ¯2 6 1.1 ¯7 0

2.
```
    ∇R←LORN X
[1]R←''=0\0⍴X
    ∇
```

3. (a)
```
B←0 1 0⌿A

      B
5 6 7 8
```

   (b)
```
      0 1 0⍀B
0 0 0 0
5 6 7 8
0 0 0 0
```

4.
```
R←('P'=F[1;])⌿F
```

5.
```
('P'=F[1;])⍀R
```

Chapter 15

1.
```
    ∇FLASH
[1]'HELLO'
[2]→1
    ∇
```

2.
```
    ∇R←SUM X;I
[1]R←X[1]
[2]I←2
[3]→0×⍳I>⍴X
[4]R←R+X[I]
[5]I←I+1
[6]→3
    ∇
```

3.
```
    ∇Z←FIB N;T
[1]Z←1 1
[2]→0×⍳N<T←+/Z[¯1 0+⍴Z]
[3]Z←Z,T
[4]→2
    ∇
```

```
      FIB 35
1 1 2 3 5 8 13 21 34
```

Chapter 16

1.
```
      NUM
ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN
ELEVEN
TWELVE
THIRTEEN
FOURTEEN
FIFTEEN

      ⍴NUM
15  8

 ∇SPELL;A
[1]A←?15
[2]'SPELL ';A
[3]R←8⍴(⍞'')⍴8⍴' '
[4]→1×⍳∧/R=NUM[A;]
[5]'WRONG. TRY AGAIN.'
[6]→2
 ∇
```

2.
```
 ∇N←ENTER;L;M;I;A;J
[1]L←⍴M←''
[2]'ENTER NAMES.'
[3]→(0=⍴A←⍞'')/L7
[4]M←M,A
[5]L←L,⍴A
[6]→3
[7]L7:N←((¯1+⍴L),J←⌈/L)⍴' '
[8]I←1⋄L←0+\0⌽L
[9]N[I;⍳⍴A]←A←L[I]↓L[I+1]↑M
[10]→9×⍳(1⍴⍴N)≥I←I+1
 ∇
```

3.
```
∇GUESS
[1]A←2?10
[2]I←1
[3]→0×ι0=B←□
[4]→1×ι∧/A∈2ρB
[5]→(3=I←I+1)/8
[6]'NO.'
[7]→3
[8]'THEY ARE ';A
[9]→1
∇
```

Chapter 17

1. (a)
```
55
```

(b)
```
30
```

(c)
```
¯269
```

(d)
```
55
```

(e)
```
4
```

(f)
```
3
```

(g)
```
5   6   7   3
9  10  11   7
7   8   9   5
8   9  10   6
```

(h)
```
 6   8  10   2
18  24  30   6
12  16  20   4
15  20  25   5
```

(i)
```
.66667   .5   .4   2
     2      1.5  1.2  6
1.3333      1   .8   4
1.6667   1.25   1   5
```

2. (a)
```
36  14
37  21
```

(b)
```
16  13
17  14
```

(c)
```
¯7768   10
 ¯3061  251
```

(d)
```
22  38
14  35
```

(e)
```
 5   2
 5   4
```

(f)
```
3  4
4  4
```

(g)
```
 5   6
 7   3

 9  10
11   7

 7   8
 9   5

 8   9
10   6
```

(h)
```
 6   8
10   2

18  24
30   6

12  16
20   4

15  20
25   5
```

(i)
```
 .66667    .5
  .4         2

    2       1.5
  1.2         6

1.3333      1
  .8        4

1.6667    1.25
  1         5
```

3. (a)
```
P←3.5 4.75 6.9
H←3 4ρ40 40 35 40 7 8 4 6 8 2 0 10
P+.×H
228.45  191.8  141.5  237.5
```

(b)
```
P1←3 2ρ3.75 3.25 5.05 4.5 7.25 6.35
H1←4 3ρ40 7 8 40 8 2 35 4 0 40 6 10
H1+.×P1
243.35   212.3
204.9    178.7
151.45   131.75
252.8    220.5
```

4.
```
      (⍳6)∘.×⍳6
 1  2  3  4  5  6
 2  4  6  8 10 12
 3  6  9 12 15 18
 4  8 12 16 20 24
 5 10 15 20 25 30
 6 12 18 24 30 36
```

Chapter 18

1. (a) 8 7 6 5 4 3 2 1 (b) 1 2 3 4 5 6 7 8

(c)
```
2 1
4 3
6 5
```

(d)
```
5 6
3 4
1 2
```

(e)
```
5 6
3 4
1 2
```

(f) 3 4 5 6 7 8 1 2

(g) 7 8 1 2 3 4 5 6

(h)
```
1 2
3 4
5 6
```

(i)
```
5 6
1 2
3 4
```

(j)
```
1 2
3 4
5 6
```

(k)
```
3 4
5 6
1 2
```

(l)
```
3 4
5 6
1 2
```

(m)
```
12 11
14 13
16 15

18 17
20 19
22 21
```

(n)
```
17 18
19 20
21 22

11 12
13 14
15 16
```

(o)
```
15 16
13 14
11 12

21 22
19 20
17 18
```

(p)
```
3 6
5 2
1 4
```

(q)
```
1 2
4 3
5 6
```

(r)
```
12 11
13 14
16 15

17 18
20 19
21 22
```

2. (a) 1 2 3 4 5 6 7 8 (b) 1 2 3 4 5 6 7 8 (c)
```
1 3 5
2 4 6
```

(d)
```
1 3 5
2 4 6
```

(e)
```
11 17
13 19
15 21

12 18
14 20
16 22
```

(f) 11 20 (g) 1 4 (h) 2

```
(i) 11   14    (j) 11   19    (k) 11   12    (l) 11   12
    17   20        12   20        19   20        13   14
                                                 15   16

                                                 17   18
                                                 19   20
                                                 21   22
```

3. `⊖+\⊖M`

Chapter 19

1.
```
      2⊥5 10 7
47
```

2.
```
      1 3 12⊥2 2 6
102
```

3.
```
      1760 3 12⊥102
2  2  6
```

4.
```
∇DRILL;A;B;C;S
[1]A←'0123456789'∘⎕PT←6∘S←0 0
[2]K:B←(' ',A[⌈/?2 2ρρA])[3?3]
[3]B←B[1],'=',B[2],'+-'[?2],B[3]
[4]I:→E×⍳'S'∈C←(B⍳' ')↑B
[5]→I∘⎕←'WRONG'∘S←S+0 1∘→J×⍳≢C
[6]J:→K∘⎕←'RIGHT'∘S←S+1 1
[7]E:'YOUR SCORE IS:'
[8]⌊.5+100×+/S⌈0 1;' PERCENT'
∇
```

Index

Absolute value, |, 5.2
Adding a line to a function, 8.7
Addition, +, 2.2, 2.3
Additive identity, +, 4.5
And, ∧, 6.3
Arccos, ¯2○, 5.5
Arccosh, ¯6○, 5.5
Arcsin, ¯1○, 5.5
Arcsinh, ¯5○, 5.5
Arctan, ¯3○, 5.5
Arctanh, ¯7○, 5.5
Arguments, 3.6, 9.3
  explicit and implicit, 9.7
Arithmetic negation, -, 2.2
Arrays, 3.7
  dimension of, 3.10
  rank of, 3.10
  restructuring of, 11.2
Assignment, ←, 3.1
Asterisk, *, 4.5
Averaging, 7.2, 8.2

Backspace key (BKSP), 2.8
Base value (decode), ⊥, 19.1
Bit, 21.6
Body, 9.1
Boolean algebra, 6.2
Brackets, [ ], 11.13
Branching, → , 15.1-15.9
  summary, 15.9
Byte, 1.4, 21.6

Calculator mode, 2.2
Cassette, tape, 1.4, B1
Catenate, ,, 12.5
Ceiling, ⌈, 4.10
Character, 2.8, 3.5
Circular functions, ○, 5.5
Conditional branch, 15.2
Colon, :, 15.6
Combinations, !, 5.4
Comparison tolerance, □*CT*, 21.5
Complex array, 3.11
Component, 3.8
Compression, / or ⌿, 14.1-14.4
Constant, 15.6
Control key (CTRL), 2.9
Coordinates of an array, 11.14
Corrections, 2.8
Cosh, 6○, 5.5
Cosine, 2○, 5.5
Cursor, 1.3

Deal, ?, 12.12
Decimal point, ., 2.4
Decode (base value), ⊥,19.1
Degree of accuracy, 2.4
Degrees, 5.6
Defined functions. See Functions.
Definition mode. See Mode.
Del, ∇, 8.3
Deleting a line in a function, 8.10
Delta, Δ, 3.3
Dimension, 3.10
Dimension of, ⍴, 11.5
Direction, 3.10
Display of
  a result, 2.2
  a variable's content, 3.1
Display screen, 1.2, 2.2
Division, ÷, 2.2, 2.4
Domain error, 20.2
Drop, ↓, 12.11
Dyadic functions, 3.6, 9.6
Dyadic random (deal), ?, 12.12
Dyadic transpose, ⍉, 18.8

E-notation, 4.6
Editing of functions, 8.6-8.12
Element, 3.8, 11.16
Empty vector
  numeric, 11.11
  literal, 14.7
Encode (representation), ⊤, 19.3
Equal, =, 6.1
Erase function, □*EX*, 3.2
Error report, 2.5, 20.1-20.7
Errors
  domain, 20.2
  index, 20.2
  length, 20.3
  range, 20.3
  rank, 20.4
  syntax, 20.5

tape, 20.6
value, 20.6
workspace full, 20.7
Escape from input loop, 16.8
Evaluated input, 16.2
Execute (unquote), ⍎, 19.5
Execution mode, 2.1
Expansion, \ or ⍀, 14.5
Explicit results, 9.7
Exponential, *, 4.7
Exponential notation, *E*, 4.6
Exponentiation, *, 4.5
Expunge function, □*EX*, 3.2

Factorial, !, 5.2
Floor, ⌊, 4.11
Function definition, 8.1
Function definition mode, 8.3
Functions, display of, 8.5
Functions, editing of, 8.6-8.12
Function headers, 9.1-9.11
Function syntax, 2.6, 9.2
Functions, primitive, 3.5
Absolute value, |, 5.2
Addition, +, 2.2, 2.3
Additive identity, +, 4.5
And, ∧, 6.3
Base value, ⊥, 19.1
Catenate, , or ⍪, 12.5
Ceiling, ⌈, 4.10
Circular, ○, 5.5
Combinations, !, 5.4
Compression, / or ⌿, 14.1-14.4
Deal, ?, 12.12
Decode, ⊥, 19.1
Dimension of, ⍴, 11.5
Division, ÷, 2.2, 2.4
Drop, ↓, 12.11
Dyadic random (deal), ?, 12.12
Dyadic transpose, ⍉, 18.8
Encode, ⊤, 19.3
Equal, =, 6.1
Exclusive or, ≠, 6.1
Execute, ⍎, 19.5
Expansion, \ or ⍀, 14.5
Exponential, *, 4.7
Exponentiation, *, 4.5
Factorial, !, 5.2
Floor, ⌊, 4.11
Grade down, ⍒, 12.4
Grade up, ⍋, 12.2
Greater than, >, 6.1
Greater than or equal to, ≥, 6.1
Index generator, ⍳, 11.10
Indexing, [ ], 11.13
Index of (ranking), ⍳, 11.7
Inner product, f.F, 17.1
Less than, <, 6.1
Less than or equal to, ≤, 6.1
Logarithm to a base, ⍟, 4.8
Maximum, ⌈, 4.9
Membership, ∈, 12.1
Minimum, ⌊, 4.10
Monadic random (roll), ?, 5.6
Monadic transpose, ⍉, 18.6, 18.8
Multiplication, ×, 2.2
Nand, ⍲, 6.4
Natural logarithm, ⍟, 4.8
Negation, -, 2.2
Nor, ⍱, 6.4
Not, ~, 6.5
Not equal, ≠, 6.1
Null, ⍬, 17.5, 19.3
Or, ∨, 6.3
Outer product, ∘.f, 17.5
PI times, ○, 5.6
Power, *, 4.5
Random, ?, 5.6, 12.12
Ranking (index of), ⍳, 11.7
Ravel, ,, 12.9
Reciprocal, ÷, 4.3
Reduction, f/ or f⌿, 7.1-7.5
Representation, ⊤, 19.3
Residue, |, 5.1
Restructure (reshape), ⍴, 11.2
Reversal, ⌽ or ⊖, 18.5
Roll, ?, 5.6
Rotate, ⌽ or ⊖, 18.1
Signum, ×, 4.4
Subtraction, -, 2.2
Take, ↑, 12.10
Transpose, ⍉, 18.6, 18.8
Trigonometric functions, 5.5
Unquote (execute), ⍎, 19.5
Functions, suspended, 10.7
Fuzz, 21.5

Global variables, 10.1-10.9
Grade down, ⍒, 12.4
Grade up, ⍋, 12.2

Greater than, >,6.11
Greater than or equal to, ≥, 6.1

Header line, 9.1-9.11
  editing of, 10.6
Heterogeneous output, 16.9
Hyperbolic functions, 5.5

Identities
  additive, +, 4.5
  hyperbolic, 5.5
  trigonometric, 5.5
Identity elements, 22.1
Index error, 20.2
Index generator, ⍳, 11.10
Index of (ranking), ⍳, 11.7
Index origin, ⎕*IO*, 21.7
Indexing, [ ], 11.13
Inner product, f.F, 17.1
Input
  evaluated, 2.7
  literal, 16.5
  numeric, 16.1
Input loop, escape from, 16.8
Inserting a line in a function, 8.7
Interrupt procedure, 2.3, 16.8, A3
Iteration, 16.8

K, 20.7
Keyboard, 1.1
Kilo, 20.7

Labels, 15.6
Length error, 20.3
Less than, <, 6.1
Less than or equal to, ≤, 6.1
Line counter, ⎕*LC*, 10.7, 21.9
Line editing, 2.8, 8.9
Literal input, 16.5
Literal output, 16.4, 16.5
Literals, 3.4
Local variables, 10.1-10.9
Logarithm, natural, ⍟, 4.8
Logarithm to a base, ⍟, 4.8
Logical functions, 6.2
Logical negation (not), ~, 6.5
Looping, 16.8

Main memory, 1.4
Matrix, 3.8

Maximum, ⌈, 4.9
Membership, ∈, 12.1
Memory, 1.4
Minimum, ⌊, 4.10
Mixed functions, 4.1, 9.5, 19.7
Mixed output, 16.9
Mode
  definition, 2.1, 8.3
  execution, 2.1
Monadic functions, 3.6
Monadic random (roll), ?, 5.6
Monadic transpose, ⍉, 18.6
Multidimensional arrays, 3.11, 13.1
  construction of, 11.2
Multiplication, ×, 2.2

Names, restrictions on, 3.3
Nand, ⍲, 6.4
Natural logarithm, ⍟, 4.8
Negative, ¯, 2.3
Negative image, 2.8
Negation, arithmetic, -, 2.2, 4.3
Niladic functions, 9.4
Nor, ⍱, 6.4
Not, ~, 6.5
Not equal, ≠, 6.1

Or, ∨, 6.3
Order of execution, 2.5, 7.4
Origin setting, 2.5, 7.4
Outer product, ∘.f, 17.5
Output, ⎕, 16.4
Output, heterogeneous, 16.9
Overstruck characters, 4.8

Parallel processing, 4.2
Parentheses, 2.7
Pendant functions, 21.10
Permutations, 5.3
Pi times, ○, 5.6
Planes, 11.3
Polynomials, 19.2
Power, *, 4.5
Primitive functions, 3.5, A1
Print precision, ⎕*PP*, 2.4, 21.11
Print time, ⎕*PT*, 21.11
Pythagorean theorem, 9.6

Quad, ⎕
  input, 16.1

  output, 16.4
  system functions, 21.2
  system variables, 21.5
Query (roll, deal), ?, 5.6, 12.12
Quote, ', 3.4, 16.7
Quote-quad, ⍞, 16.5-16.9

Radian, 5.5
Radix vector, 19.1
Random (roll, deal), ?, 5.6, 12.12
Random link, ⎕*RL*, 21.12
Range error, 20.3
Rank, 3.10
Rank error, 20.3
Ravel, ,, 12.9
Reciprocal, ÷, 4.3
Recursive function, 9.10
Reduction, f/ or f⌿, 7.1, 13.1-13.5
Relational functions, 6.1
Representation, ⊤, 19.3
Residue, |, 5.1
Respecification, 3.2
Restructure (reshape), ⍴, 11.2
RETURN key, 2.2
Reversal, ⌽ or ⊖, 18.5
Rho, ⍴, 11.2
Roll, ?, 5.6
Roots, 4.6
Rotate, ⌽ or ⊖, 18.1
Rounding, 4.11
Rows, 11.3

Scalar, 3.7
Scalar function, 4.1
Scanning, 7.6, 13.6
Screen, display, 1.2, 2.2
Semicolon, ;, 10.4, 11.14
Sign-off function, ⎕*OF*, 2.1, 21.4
Sign-on procedure, 2.1
Signum, ×, 4.4
Sine, 1○, 5.5
Sinh, 5○, 5.5
Size. See Dimension.
Sorting, 12.2, 12.4
Space available, ⎕*WA*, 21.13
Specification, ←, 3.1
Standard scalar functions, 4.1, 6.6
START key, 2.1
State indicator, 10.7, 21.9
Stop execution, 2.3, 16.8, A3
Subtraction, -, 2.2
Suspended functions, 10.7
  listing of, 21.9
  removal of, 21.10
Syntax error, 20.5
System functions
  ⎕*EX*, 21.2
  ⎕*FN*, 21.2
  ⎕*OF*, 2.1, 21.4
  ⎕*VA*, 21.2
  ⎕*WC*, 21.3
System variables
  ⎕*CT*, 21.5
  ⎕*IO*, 21.7
  ⎕*LC*, 21.9
  ⎕*PP*, 21.11
  ⎕*PT*, 21.11
  ⎕*RL*, 21.12
  ⎕*SI*, 21.9
  ⎕*WA*, 21.13

Tape cassette, 1.4, B1
Tape error, 20.6, B3
Take, ↑, 12.10
Tangent, 3○, 5.5
Tanh, 7○, 5.5
Tilde, ~, 6.4
Transpose, ⍉
  monadic, 18.6
  dyadic, 18.8
Trigonometric functions, 5.5

Unconditional branch, 15.2
Unquote (execute), ⍎, 19.5
User defined function, 8.1

Value error, 20.6
Variables, 3.1
  assigning, 3.1
  global, 10.1
  listing, ⎕*VA*, 3.2
  local, 10.1
Vectors, 3.8
  of length 0, 11.11, 14.7

Workspace
  clear, ⎕*WC*, 21.3
  available, ⎕*WA*, 21.13
Workspace full error, 20.7

MICRO COMPUTER
MACHINES INC.
4 LANSING SQUARE
WILLOWDALE
ONTARIO CANADA
M2J 1T1
TEL. 416/492 1693

PRINTED IN CANADA