IBM

5100

IBM 5100
APL Introduction

IBM 5100
Portable Computer
APL Introduction

SA21-9212-1

Preface

This manual discusses the mechanics of using APL with the IBM 5100. It is intended to provide the users of the 5100 with enough background in the APL language to use the *IBM 5100 APL Reference Manual,* SA21-9213, for answering their questions about how the APL functions work.

Second Edition (December 1975)

This is a major revision of, and obsoletes, the previous edition SA21-9212-0. Changes are continually made to the specifications herein; any such changes will be reported in subsequent revisions or technical newsletters.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for reader's comments is at the back of this publication. If the form is gone, address your comments to IBM Corporation, Publications, Dept 245, Rochester, MN 55901.

Contents

CHAPTER 1. INTRODUCING THE IBM 5100 . 1
About this Manual 1
About the APL Language 1
About the 5100 1
Alphameric Keys 2
Numeric Keys 2
Operating Keys 2
APL System Command Keywords 4
Arithmetic Function Keys 4
Getting Started 4
Entering and Displaying Data 5
Correcting Keying Errors 8
Replacing a Character 8
Deleting a Character 9
Inserting a Character 10

CHAPTER 2. INTRODUCING THE APL LANGUAGE 14
Types of Functions in APL 14
Addition, Subtraction, Multiplication, and Division 14
Another Arithmetic Function—Raising a Number to a Power 16
Finding the Root of a Number 17
Storing Data in the IBM 5100 for Later Use . . . 17
Performing Several Operations in the Same Instruction 18
Specifying the Order of Execution—Using Parentheses 19
Using Strings of Numbers and Tables 19
Using APL with Strings of Numbers (Vectors) . 20
Using APL with Tables of Numbers (Matrices) . 22
Referring Only to Certain Numbers in a String or Table of Numbers (Indexing) 24
You Are Not Limited to Using Only Numbers . . 26

CHAPTER 3. APL FUNCTIONS THAT REQUIRE ONE ARGUMENT 28
How Many Arguments Are Required by an APL Built-In Function? 28
APL Function Symbols That Are a Combination of Two Characters 28
Determining the Whole Numbers Nearest a Fraction 29
Rounding to the Nearest Whole Number . . . 29
Sorting a Vector in Ascending or Descending Sequence 30
Generating a Random Number 31
Generating Consecutive Numbers 31
Generating an Empty Vector 32
Finding the Shape of an Existing Variable . . . 32

CHAPTER 4. APL FUNCTIONS THAT REQUIRE TWO ARGUMENTS 33
Relational Functions 34
Why Two Numbers Identical in Appearance Are Not Always Equal 36
An Example Using a Relational Function . . 36
Logical Functions 37
And 38
Or 39
Finding the Larger of Two Numbers 41
Finding the Smaller of Two Numbers 42
Finding the Index of a Value in a Vector 43
Generating a Random Sequence of Numbers . . . 43
Selecting Certain Elements From (Compressing) a Vector or Matrix 44
Expanding a Vector or Matrix 45
Joining Two Items Together 47
Building a Vector of Results Using Catenation . 49
Finding the Logarithm of a Number 51

CHAPTER 5. APPLYING THE SAME OPERATION TO ALL THE ELEMENTS OF A VECTOR COLLECTIVELY (REDUCTION) . . 53
Plus Reduction 53
Using Plus Reduction to Find the Average . . 53
Using Plus Reduction to Sum the Products of Two Vectors 56
Minus Reduction (Alternating Sum) 56
Maximum Reduction: Finding the Largest Value in a Vector 57
Minimum Reduction: Finding the Smallest Value in a Vector 57
Or Reduction: Checking for a Specific Value in a Vector 58
And Reduction: Checking for All Values in Two Vectors Being Equal 58
CHAPTER 6. USING THE BUILT-IN FUNCTIONS 59
Now Let's Look at the Things You Can Do . . . 59

CHAPTER 7. FUNCTION DEFINITION . . . 71
What Is Function Definition? 71
How Is a Function Defined? 71
Testing Your Function Before Using It 73
How To Use the Trace and Stop Features . . . 75
Trace T∆ 75
Stop S∆ 75
What To Do If You Make a Mistake When Defining Your Function 77
Displaying the Entire Function 78
Adding One or More Statements at the End of the Function 79
Replacing Statements within a Function . . . 80
Inserting One or More Statements in a Function 81
Deleting a Statement from a Function 82
Displaying a Specific Statement or from a Specific Statement to the End of a Function . 83
Modifying a Single Statement 83
Editing the Function Header 84
A Faster Way to Add, Replace, or Insert One Statement in a Function 85
Types of Function Headers 87
Why Have a Result Variable? 88
Local and Global Names 89
Branching, Labels, and Looping 91
Branching and Labels 91
Looping 94
How To Enter Data during Function Execution . 95

CHAPTER 8. WHAT YOU CAN DO WHEN YOUR FUNCTION STOPS 98
When the Attention Key Is Pressed 98
When the Stop Control Feature Is Used 100
When an Error Is Encountered in the Function . . 101
When a OUT Character Is Entered for a ⍞ Input Request 102
Finding Out What Functions Are Suspended . . . 102
Using the Hold Key To Stop Processing 103

CHAPTER 9. USING YOUR TAPE CARTRIDGE (LIBRARY) 104
What Are System Variables? 108

APPENDIX A. OVERSTRUCK CHARACTERS . 112

INDEX 114

Chapter 1. Introducing the IBM 5100

ABOUT THIS MANUAL

This manual will show you how to operate the IBM 5100 using the APL language. If you are not familiar with the APL language, you should do the suggested keying operations or examples on your 5100 while reading the manual from cover to cover. If you are familiar with the APL language, you should read Chapters 1 and 2 to learn how to operate the 5100; however, you may then want to skip to Chapter 7. Not all of the features or functions of the APL language are covered in this manual. For more information about the 5100 or the APL language, see the *IBM 5100 APL Reference Manual*, SA21-9213.

This manual was written with the assumption that the 5100 has been set up and checked out. If the 5100 has not been set up, use the set up procedure in the *IBM 5100 APL Reference Manual* before continuing to read this manual.

ABOUT THE APL LANGUAGE

APL has many built-in functions that allow you to effectively solve your problems. However, if you need a special function to solve a problem, APL also allows you to define your own functions. The functions you define are similar to programs written in other computer languages.

APL is a good language to experiment with; nothing you do from the keyboard can damage the 5100, and the more you experiment, the more you will learn about APL.

ABOUT THE 5100

The IBM 5100 (Figure 1) is a portable computer designed to help you solve problems. The display screen and indicator lights communicate information to you, and the keyboard and the switches allow you to control the operations the 5100 will perform.

Before you begin to use the 5100, you should become familiar with the keys and control panel (Figure 1). The control panel switches will be explained later. A brief description of the keys follows; how you use the keys will be discussed later.

Alphameric Keys

The alpha keys are similar to those on a standard typewriter, except that there are no lowercase characters. The alpha characters are all uppercase, even though they are in the lowercase position on the keys. Thus, you *do not* use the shift key for alpha characters.

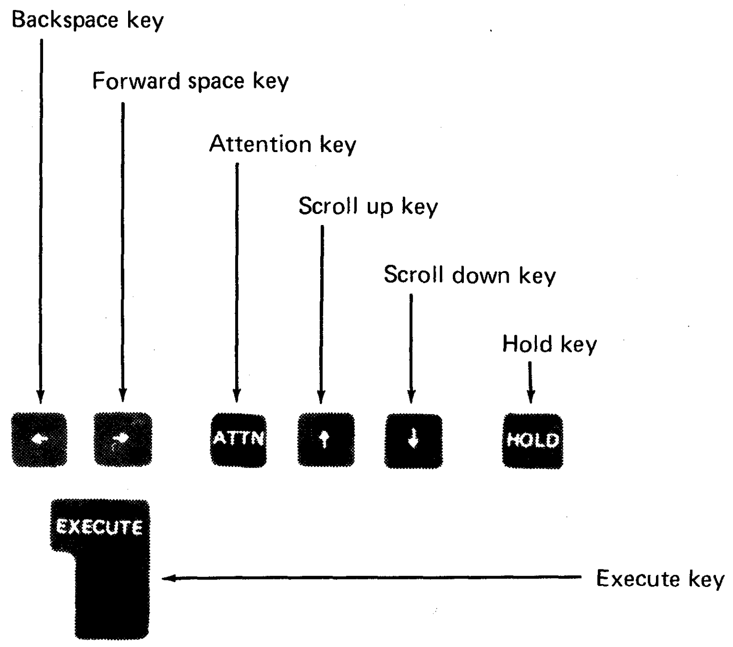
If you want to enter an upper shift character, you must hold down the shift key and then press the key to enter the character, just as you would to type an uppercase character on an ordinary typewriter.

Numeric Keys

Either the top row of alphameric keys or the special calculator arrangement of numeric keys can be used to enter numbers.

Operating Keys

The black key labeled CMD, the gray keys with the legend names EXECUTE, ATTN, and HOLD, and the gray keys with the arrows are all special operating keys. The keys with the arrows and the space bar, which is used to enter blank characters, automatically repeat the operation they perform when held down.



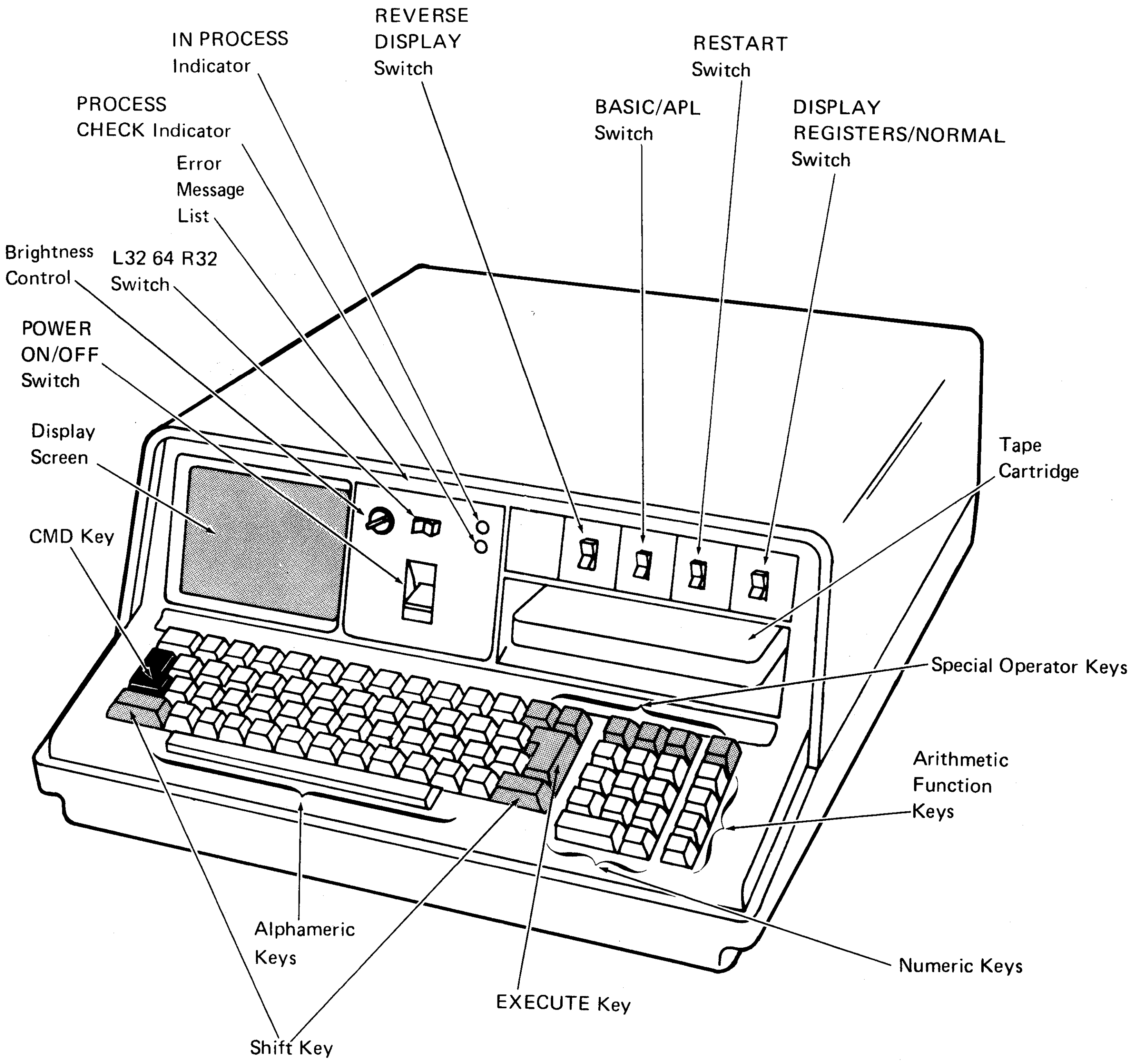


Figure 1. The 5100

APL System Command Keywords

The words that are above the top row of numeric keys are system command keywords, which you can enter by holding down the CMD key and then pressing the key below the desired keyword. For example, to enter)LOAD, hold down CMD and press the 1 key. The system commands and their uses are discussed later, in Chapter 9.

Arithmetic Function Keys

The four keys to the right of the calculator arrangement of numeric keys are the arithmetic function keys. These keys are used to perform division, multiplication, subtraction, and addition. There are also keys on the alphameric keyboard that perform these functions. Notice that the ÷ and x symbols are used for division and multiplication.

GETTING STARTED

Make sure the switches on your IBM 5100 are set as follows:

Switch	Setting
L32 64 R32	64
BASIC/APL (Combined machines only)	APL
DISPLAY REGISTER/NORMAL	NORMAL

If your 5100 has the BASIC/APL switch, it can execute both BASIC and APL language statements. The language to be used is selected by the user before power up or during the restart sequence.

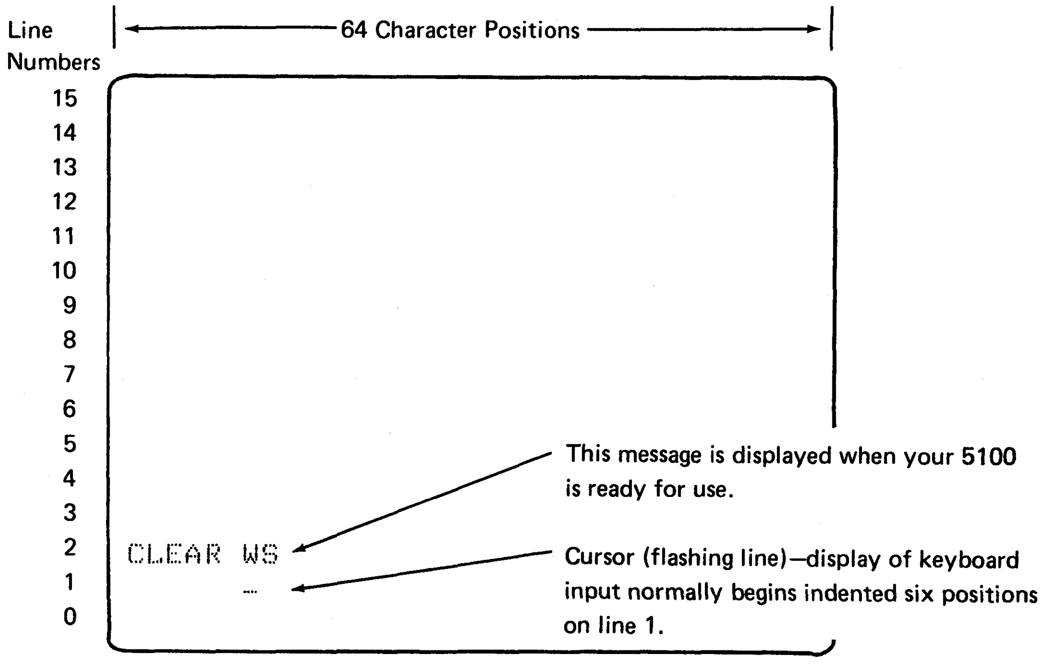
Make sure your 5100 is plugged in and turn power on. If power is already on, press RESTART and wait about 20 seconds. During this time, the 5100 performs internal checks to make sure it is operating correctly.

After 30 seconds, if the message CLEAR WS has not appeared in the lower lefthand corner of the display screen, an error has been detected during the internal checks. In this case, press RESTART. The 5100 will perform the internal checks again. If the CLEAR WS message does not appear after several tries, call your service representative.

ENTERING AND DISPLAYING DATA

First, let's look at the display screen. Normally, information displayed by APL begins at the left edge of the display screen, and the input from the keyboard is indented when it is displayed. The small horizontal flashing line indicates the position on the line where the next input from the keyboard will be displayed. This flashing line is called the *cursor*. The cursor moves as each character is displayed.

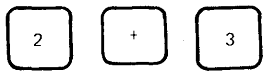
The display screen can contain up to 16 lines of data. Each line has 64 positions across the display screen. The bottom two lines are used to display input, and the remaining 14 lines contain a history of the operations you have performed.



Line Numbers
64 Character Positions
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
CLEAR WS
This message is displayed when your 5100 is ready for use.
Cursor (flashing line)—display of keyboard input normally begins indented six positions on line 1.

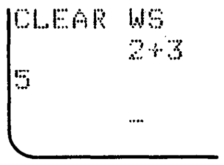
There are 128 positions available for input from the keyboard; that is, there are 64 positions available on line 1 and 64 positions on line 0. When position 64 of line 1 is used as you enter data from the keyboard, the cursor moves to the left margin of line 0. The cursor is then at position 65 of the possible 128 positions available for input.

Now let's enter some data into the 5100 using the numeric keyboard and the arithmetic function keys. Press the following keys:



Notice that the characters are displayed as each key is pressed. To process the data you just keyed, you must press the EXECUTE key. Press the EXECUTE key now.

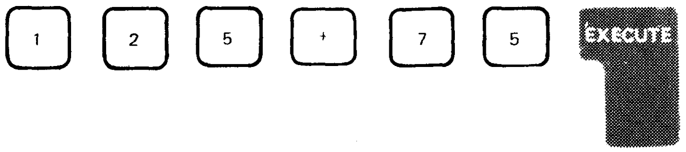
The display screen will look like this:



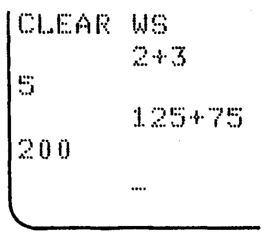
```
CLEAR WS
      2+3
5
      _
```

Notice that the instruction you entered, 2+3, appears indented on the display screen; the answer, 5, appears on the left margin of the next line; and the cursor appears on the next line. The information displayed moves up each time the EXECUTE key is pressed.

Enter and execute the instruction 125+75 by pressing the following keys:



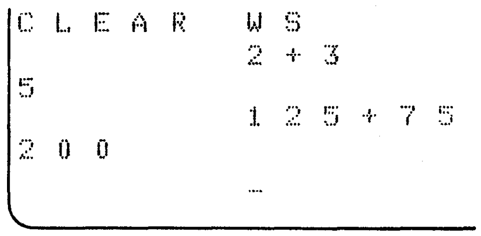
The display screen will look like this:



```
CLEAR WS
      2+3
5
      125+75
200
      _
```

The appearance of your display can be changed by switches on the control panel. The REVERSE DISPLAY switch allows you to change from black characters on a white background to white characters on a black background and vice versa. Change the switch and select the type of display you feel most comfortable with. You may have to adjust the brightness control as you change from one to the other.

Now, watch the display as you set the L32 64 R32 switch to the L32 position. With the switch in this position, the leftmost 32 characters on each line are displayed with an extra space between each character. The rightmost 32 characters on each line will not be displayed. With the switch in the L32 position, your display should look like this:



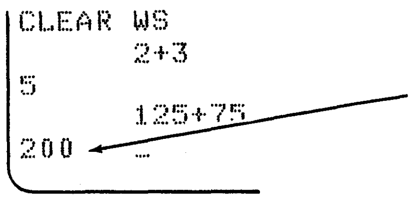
```
C L E A R   W S
            2 + 3
5
            1 2 5 + 7 5
2 0 0
            _
```

In the R32 position, the rightmost 32 characters are displayed with a space between each character. Now, set the switch in the R32 position and notice that the display is blank because there were no characters in the rightmost 32 positions of the display screen.

Return the switch to the 64 position, and notice that all characters are displayed without the space in between. For exercises in the remainder of this book, keep the switch in the 64 position.

There are two keys above the numeric keys that move the display line up or down. The up arrow ↑ (scroll up key) moves the display up one line and the down arrow ↓ (scroll down key) moves the display down one line. As the lines are moved up or down, the displayed information on any line that is moved off of the display screen is lost. Either key continues to move the display lines if it is held down. Now use the down arrow to move the display down one line.

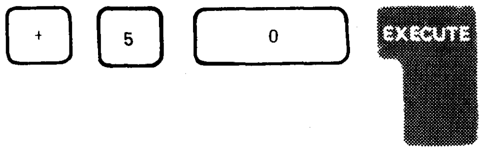
The display will look like this:



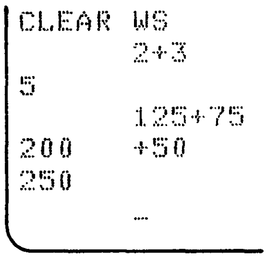
```
CLEAR WS
      2+3
5
      125+75
200   _
```

The value 200 is now on the input line and can be used as input. Notice that input can begin in any position on the line.

Now press the following keys:



The display screen will look like this:



```
CLEAR WS
      2+3
5
      125+75
200   +50
250
      _
```

Now that you are familiar with the display screen, only the line or lines being discussed will be shown.

CORRECTING KEYING ERRORS

The IBM 5100 has a number of very useful features that allow you to correct errors made when data was entered. On a line-by-line basis, at any time, you can:

- Replace a character
- Delete a character
- Insert a character

Replacing a Character

To replace a character, move the cursor with the backspace key or forward space key, until the cursor is positioned at the incorrect character. The cursor moves one character space in the direction of the arrow each time the appropriate key is pressed. These keys continue to move the cursor if they are held down. When the cursor is at the incorrect character, you replace the incorrect character by simply keying the correct character.

For example, you want to do the problem 22+12. But you press the following keys:

2 2 + 1 1

The display screen looks like this:

```
22+11_
```

To correct the error, the cursor must be moved back one position (under the second 1) so that the character can be rekeyed. Now press the backspace ← key one time. Note that the cursor is replaced by a flashing character. The flashing character serves the same function as the cursor; it indicates the position on the line where the next input from the keyboard will be displayed. Now to correct the error and execute the problem, press the following keys:

2 EXECUTE

Deleting a Character

To delete a character, you also use the backspace key ← or forward space key → to move the cursor. Once the cursor is in the position of the character to be deleted (the character is flashing), hold down the CMD key and press the backspace key once. The character is then deleted and any characters to the right are shifted one position to the left to close up the space left by the deletion.

For example, you want to do the problem 13+45. But you press the following keys:

$\boxed{1}\quad\boxed{2}\quad\boxed{3}\quad\boxed{+}\quad\boxed{4}\quad\boxed{5}$

The display screen looks like this:

```
123+45_
```

Press the backspace key and move the cursor (flashing character) back to the 2. Look at the labels that appear above the backspace and forward space keys: DELETE and INSERT. To delete the 2, hold down the CMD key while you press ← once.

The display screen looks like this:

```
13+45
 \
  This character is flashing.
```

Now press the EXECUTE key to execute the problem.

Inserting a Character

To insert a character, position the cursor using the backspace key ← or forward space → key; then hold down the CMD key and press the forward space → key once. This operation moves the flashing character (and all other characters to the right of it) one position to the right, creating the space you need to insert one character. The cursor is not moved. Now, to insert the character, simply press the desired key.

For example, you want to do the problem 123x6. But you press the following keys:

The display screen looks like this:

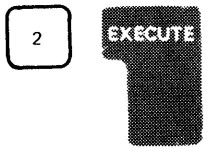
```
13x6_
```

To correct the error, press the backspace key and move the cursor (flashing character) back to the 3. Look at the labels that appear above the backspace and forward space keys: DELETE and INSERT. To perform the insert function, with the cursor positioned at the 3, hold down the CMD key while you press → once.

The display screen looks like this.

```
1_3x6
```

Now to correct the keying error and execute the problem, press the following keys:



There is one more way to correct a keying error. If you make several errors part way through the line, you can backspace the cursor to the character following the last correct character and then press the ATTN (attention) key. Everything from the cursor position to the end of the input line will be cleared from the display.

Since the data from the input line is not processed until the EXECUTE key is pressed, you can visually verify any input before it is processed. However, if you do press the EXECUTE key before you notice a mistake, you can simply enter the input again or you can use the down arrow ↓ (scroll down key) to move the input back down to the input line to correct it. Either way, you must press the EXECUTE key again.

For example, you want to do the problem 135+280, but you enter and execute 134+280. The display screen looks like this:

```
      134+280
414
      ....
```

To correct the input, press the down arrow ↓ three times to clear the result from the screen. The display screen now looks like this:

```
      ....
134+280
```

Then press the up arrow ↑ once to move the original input back up to the first input line so that it can be corrected.

From this point on, we will use examples in the following format to illustrate what we are discussing. You enter the instructions that are indented. The results displayed on your 5100 should be the same as the results shown in this manual.

EXAMPLES:

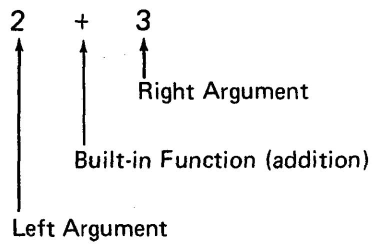
```
      3+4 ◄──────── Instructions to be entered
7 ◄─────────────── Results
```

Remember, the data you key is *not* processed until the EXECUTE key is pressed.

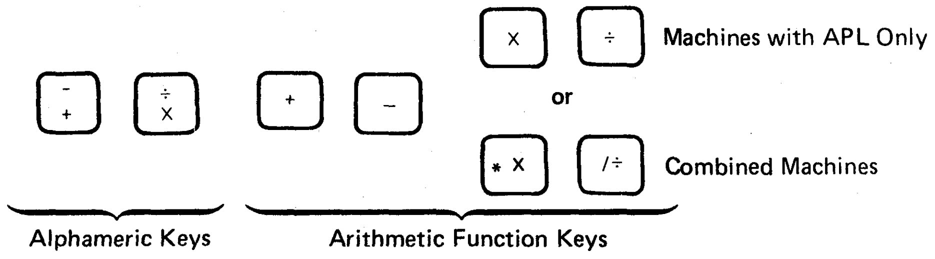
TYPES OF FUNCTIONS IN APL

There are two types of functions in APL: user-defined functions (programs) and those that are built into the APL language. The APL built-in functions are denoted by special symbols. User-defined functions are discussed later, in Chapter 7.

The built-in functions operate on data supplied, called arguments. For example:



ADDITION, SUBTRACTION, MULTIPLICATION, AND DIVISION



Four commonly used built-in functions (+ – x ÷) perform the normal arithmetic operations when they are used. These symbols are located on the top row of the alphameric keys and also to the right of the numeric keys.

EXAMPLES:

```
      3+6
9
      3×6
18
      8-4
4
      4-8
¯4
      8÷4
2
      4÷8
0.5
```

Add 3 and 6.

Multiply 3 times 6.

The right argument is subtracted from the left argument.

The high horizontal bar is the *negative sign*. Compare it with the minus which is the symbol for subtraction; the negative sign appears near the top of the character instead of on the center line.

The left argument is divided by the right argument.

As you have seen in the example, the negative sign is different from the minus. When you are doing arithmetic operations in APL, do not use the minus to represent negative numbers or the negative ¯ sign for a subtract operation.

Problems: Using Addition, Subtraction, Multiplication, and Division

1. Find the total number of cars that a dealer sold during one week if his daily sales were 3, 5, 2, 6, 7, 3 and 4.

2. Find the net number of cars removed from the same dealer's lot if 20 people had trade-ins.

3. Find the dealer's average profit per car if he made a total profit of $2700 for the sales in problem 1.

4. Find the dealer's total earnings if he made $20 on each car sold.

Possible Solutions

Problem 1:

```
        3+5+2+6+7+3+4
30
```

Problem 2:

```
        30-20
10
```

Problem 3:

```
        2700÷30
90
```

Problem 4:

```
        20×30
600
```

ANOTHER ARITHMETIC FUNCTION—RAISING A NUMBER TO A POWER

Another arithmetic operation that you are probably familiar with is raising a number to a power. In APL, you use the ⋆ function to raise the left argument to the power specified by the right argument.

EXAMPLES:

```
        3*2  ◄────── 3 raised to the second power.
9
        2*3  ◄────── 2 raised to the third power.
8
```

Finding the Root of a Number

You can use the power function * to find the root of a number. To do this, you simply raise the number to the power 1÷n, where n is the root you want to find.

EXAMPLES:

```
        4*(1÷2) ←——— The square root of 4.
2
        4*.5 ←—————— Another way to enter the instruction to find a
2                    square root of a number (.5 is the same as 1÷2).
        8*(1÷3)
2 ←————————————————— The cube root of 8.
```

STORING DATA IN THE IBM 5100 FOR LATER USE

You can store data, either direct input that you enter from the keyboard or the result of a calculation. These stored items are called *variables.* Each variable has a name associated with it. Whenever you use the name of a variable, APL supplies the value associated with that name. A variable name can be up to 77 characters long (with no blanks); the first character must be alphabetic; the remaining characters can be any combination of alphabetic and numeric characters. It is good practice to use names that represent the data you are storing. For example, if you want to store a value that is the area of a rectangle you might use the name AREA; or if you want to store some sales data, you might use the name SALES.

You create a variable by assigning the data to a name. To assign a value to a name, you use the assignment arrow ←. The value to the right of the ← is assigned to the name to the left of the ←.

EXAMPLES:

```
      PRICE←99.50
      SALES←PRICE×10
      PRICE
99.5
      SALES
995
```

After you press the EXECUTE key, you have created a variable named PRICE with a value of 99.50.

The result of a calculation can also be assigned to a variable.

If you want to know the current value of a variable, you simply enter the name of the variable.

```
      PRICE←86.75
      PRICE
86.75
      PRICE←PRICE+10
      PRICE
96.75
```

You can change the value of a variable the same way you assigned the original value.

You can also use the variable and change its value in the same instruction.

You cannot use a name as a variable if it does not have a value assigned to it.

```
      COST÷SALES
VALUE ERROR
      COST÷SALES
      ∧
```

The error message indicates why the instruction failed.

The caret (∧) indicates where the instruction failed.

Note: Do not be concerned at this time about the error message that is displayed; all of the 5100 APL error messages and suggested user's responses are described in the *IBM 5100 APL Reference Manual*, SA21-9213.

PERFORMING SEVERAL OPERATIONS IN THE SAME INSTRUCTION

In the preceding examples, only one arithmetic function was used in each example. However, you are not restricted to writing instructions with only one function. Any number of functions can occur in the same instruction. As soon as you use more than one function, however, you must be concerned about the order in which they are used. *In APL, the rightmost function in any instruction is executed first, then the next rightmost, and so on.*

EXAMPLES:

```
                                  Order of execution is right to left.

          3×2+4  ←——  4 is added to 2, and that result is multiplied by 3.
18
          4+3×2  ←——  3 is multiplied by 2, and that result is added to 4.
10
```

Remember that an APL function uses as its right argument the result of the expression to its right.

SPECIFYING THE ORDER OF EXECUTION—USING PARENTHESES

In APL, parentheses are used the same way as they are in conventional arithmetic: the operations inside the parentheses are executed before the operations immediately outside them.

EXAMPLES:

```
          (3×2)+4  ←——  The expression 3x2 is evaluated first and the
10                      result is added to 4.
          (4+3)×2  ←——
14                      The expression 4+3 is evaluated first and the
                        result is multiplied by 2.
```

Remember, the rule of the order of execution is from right to left with the expressions in parentheses resolved first and from right to left as they are encountered.

USING STRINGS OF NUMBERS AND TABLES

A powerful feature of APL is the way it handles strings and tables of data. So far, you have used APL with only single numbers (called scalars): but APL also works with strings of numbers (vectors) and tables (matrices). The operations you have performed using single numbers are simply extended to each number in a string or a table. For example, if you have a string of numbers assigned to a variable named SALES, you can add 2 to each number in the string by simply entering 2+SALES.

Using APL with Strings of Numbers (Vectors)

A string of numbers is called a *vector*. When you enter a string of numbers, there must be at least one blank between each number; each number is called an *element* of the vector.

EXAMPLES:

```
      1 44 2 9 35
1 44 2 9 35
```

You have entered a 5-element vector (a string of five numbers).

```
      STRING←144 16 39 2
      STRING
144 16 39 2
```

A vector can be assigned to a variable name.

```
      SALES←125 220 316 90
      SALES×10
1250 2200 3160 900
```

Each element (number) in the vector can be operated on by a single number.

```
      SALES
125 220 316 90
```

Note that the value of SALES has not changed.

```
      PRICE←.50 1.00 .75 1.10
      TOTAL←SALES×PRICE
      TOTAL
62.5 220 237 99
```

Each element in a vector can be operated on by the corresponding element in another vector with the same number of elements.

```
      1 2 4+4 5 6
5 7 10
      124+456
580
```

There must be at least one blank between each element of the vector, or the result will be different.

```
      1 2 3+4 5
LENGTH ERROR
      1 2 3 + 4 5
           ^
```

You cannot use two vectors that do not have the same number of elements, unless one of the arguments is a single number.

Problems: Using Strings of Numbers

1. Find the squares of the numbers from 1 to 5.

2. Find the squares, cubes, and fourth powers of the numbers 2 and 3.

3. A small mutual fund broker specializes in five funds. He wants to know how much of each fund he had sold at the close of the day. By 4:00 PM, he had sold $1500, $3200, $1200, $2300, and $2400, respectively, of the five funds. In the last hour of the day, he sold $100, $500, $300, $200 and $0 of the respective funds. Write a single APL statement to determine his closing sales figures for each fund.

4. The five funds in problem 3 sold for $7.30, $11.58, $3.45, $2.17 and $5.56 per share. How many shares of each fund were sold?

5. The broker receives the following percentages of commission on the five funds: 3.25, 2.5, 3.0, 3.75 and 3.5. How much did he earn from each fund today? What are his total earnings for the day?

Possible Solutions

Problem 1:

```
      1 2 3 4 5*2
1 4 9 16 25
```

Problem 2:

```
      2 3*2
4 9
      2 3*3
8 27
      2 3*4
16 81
```

or

```
      2*2 3 4
4 8 16
      3*2 3 4
9 27 81
```

Problem 3:

```
      1500 3200 1200 2300 2400+100 500 300 200 0
1600 3700 1500 2500 2400
```

Problem 4:

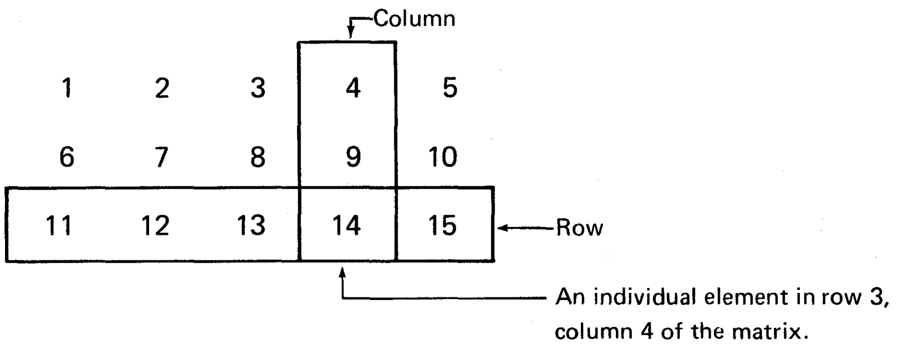
```
      1600 3700 1500 2500 2400÷7.30 11.58 3.45 2.17 5.56
219.18 319.52 434.78 1152.1 431.65
```

Problem 5:

```
      1600 3700 1500 2500 2400×.0325 .0250 .0300 .0375
52 92.5 45 93.75 84
      52.00+92.50+45.00+93.75+84.00
367.25
```

Using APL with Tables of Numbers (Matrices)

A table of numbers is sometimes called a *matrix*. The numbers in the matrix are arranged in rows and columns; each number is called an element of the matrix.



You use the reshape ρ function to create matrices. The left argument specifies the number of rows and columns and the right argument specifies the data or variable name for the data to be placed in the matrix.

EXAMPLES:

```
      2 3ρ1 2 3 4 5 6
1 2 3
4 5 6
```

The first number in the left argument specifies the number of rows; the second number specifies the number of columns.

The right argument specifies the values to be placed element by element into the rows of the matrix.

There must be a blank between the numbers specifying rows and columns.

```
      TABLE←2 3ρ1 2 3 4
      TABLE
1 2 3
4 1 2
```

If there are not enough elements in the right argument to fill the matrix, the elements are repeated.

```
      VECTOR←1 2 3 4 5 6 7 8 9 10 11 12
      MATRIX←3 3ρVECTOR
      MATRIX
1 2 3
4 5 6
7 8 9
```

If there are more elements in the right argument than are required to fill the matrix, only the first (leftmost) elements are used.

The reshape ρ function can also be used when creating a vector.

```
      FOUR←4ρVECTOR
      FOUR
1 2 3 4
```

The number of elements in the vector.

```
      MATRIX+10
11 12 13
14 15 16
17 18 19
```

Each element in the matrix can be operated on by a single number (remember, the value of MATRIX is not changed).

```
      NUMBERS←1 2 3 4 5 6 7 8 9
      EXAMPLES←3 3ρNUMBERS+5
      EXAMPLES
 6  7  8
 9 10 11
12 13 14
```

Remember that APL executes an instruction from right to left—the result of NUMBERS+5 is used as the right argument for the reshape ρ function.

```
      RESULTS←EXAMPLES+MATRIX
      RESULTS
 7  9 11
13 15 17
19 21 23
```

Each element in a matrix can be operated on by the corresponding element in another matrix of the same shape.

EXAMPLES—continued

```
      LESS←2 2⍴NUMBERS
      LESS
1 2
3 4

      MATRIX×NUMBERS←──────── A matrix and a vector
RANK ERROR
      MATRIX×NUMBERS           or
      ^
      MATRIX×LESS←─────────── two matrices that do not have
LENGTH ERROR                  the same shape (number of rows
      MATRIX×LESS             and columns) cannot be used
      ^                       unless one of the arguments is
                              a single number.
```

REFERRING ONLY TO CERTAIN NUMBERS IN A STRING OR TABLE OF NUMBERS (INDEXING)

Indexing is a way to refer to only certain elements in a string or table by specifying the position of the element you want. The numbers you use to specify the positions of the elements are called index numbers. These index numbers are enclosed in brackets [ ] following the vector or matrix to which they apply.

EXAMPLES:

```
      TEMP←68 74 78 65 80 85 72
      TEMP[2]←────────────────── You can refer to a single
74                               element.
      TEMP[3 1 2]
78 68 74 └────────────────────── You can refer to several
                                 elements. Notice that the
                                 elements are displayed in
                                 the order in which you
                                 indexed them.
      TEMP[2]+TEMP[3 1 2]
152 142 148 └─────────────────── You can index and perform
                                 other operations in the
                                 same instruction.

      TEMP[7]←88 ←────────────── You can change a single
      TEMP                       element of a vector.
68 74 78 65 80 85 88
      TEMP[3 6]←70←
      TEMP[3 6]  ─────────────── You can also change several
70 70                            elements.
```

```
      TEMP[5 1]←32 56
      TEMP
56 74 70 65 32 70 88
```

Notice that the new values are assigned in the same order as the index numbers.

For a matrix, you need an index number for the rows and an index number for the columns—these numbers are separated by a semicolon.

```
      TIMES←3 3⍴NUMBERS
      TIMES
 1 2 3
 4 5 6
 7 8 9
```

Remember, we have previously assigned a value to NUMBERS.

Left side of the ; specifies the row(s).

Right side of the ; specifies the column(s).

You can refer to a single element

```
      TIMES[2;2]
5
```

or

you can refer to several elements. In this case, you have referred to the second and third elements in the third row.

```
      TIMES[3;2 3]
8 9
      TIMES[1;1 2]
1 2
      TIMES[2 3;2 3]
 5 6
 8 9
```

Notice that when you refer to more than one row and more than one column, your result is a matrix.

```
      TIMES[2;]
4 5 6
      TIMES[;3]
3 6 9
```

If you do not specify a column, you get the whole row.

If you do not specify a row, you get the whole column.

These values (the third column) are displayed horizontally, because they are a string of numbers (vector).

```
      TIMES[;1 3]
 1 3
 4 6
 7 9
```

Note: Even when selecting entire rows or columns, the semicolon is still required to make it clear whether the index number is for the rows or columns.

```
      TIMES[2;1]×TIMES[3;3]
36
      TIMES[2;2]←0
      TIMES
 1 2 3
 4 0 6
 7 8 9
      TIMES[1;1]←2+3
      TIMES[1;1]
5
```

You can index and perform other operations in the same instruction.

You can change the value of elements in a matrix.

YOU ARE NOT LIMITED TO USING ONLY NUMBERS

Although the examples so far have used only numeric data, APL also works with character data. Character data, for example, can be used for headings on a table or to create a list of names. When you enter character data, you must enclose the data in single quote characters '. These single quote characters indicate that the data is character data and is not a variable name, a number, or a function. When character data is displayed, the single quote marks do not appear.

Character data, like numeric data, can be a single character (scalar), a string of characters (vector), or a table of characters (matrix). Unlike numeric data, when you have a character vector or matrix, each character is a separate element and is not separated from the other elements by a blank. In fact, a blank in the character data is also a character (blank character).

EXAMPLES:

```
      'A'
A
      'ABC'
ABC
      '2+3'
2+3
```

Single character (scalar).

String of three characters (3-element vector).

This instruction does not yield a result of 5, because the values are characters not numbers.

```
      NUMBER←456
      '123'+NUMBER
DOMAIN ERROR
      '123'+NUMBER
           ^
```

You cannot add character data and numeric data.

```
      'DON''T DO THAT'
DON'T DO THAT
      THANKS←'YOU ARE WELCOME'
      THANKS
YOU ARE WELCOME
```

To place a quote within the character string, you must use a pair of quotes.

Character data can be assigned to a variable name.

```
      NAMES←'SAM JOHNJACKTOM '
      MATRIXN←4 4⍴NAMES
      MATRIXN
SAM
JOHN
JACK
TOM
      NAMES[5 6 7 8]
JOHN
```

Blank characters.

Create a character matrix, each row represents a name.

Indexing works with character data also.

So far, you have used APL with some common arithmetic operations. You have also seen how APL works with scalars (single data items), vectors (strings of data), and matrices (tables of data). However, you are not limited to just the functions we have discussed so far. In the following chapters, you will be introduced to more things you can do with APL.

Chapter 3. APL Functions That Require One Argument

In this chapter you will use some APL functions to do the following:

- Determine the whole numbers nearest a fraction.
- Sort a vector into ascending or descending order.
- Generate a random number.
- Find the shape of an existing variable.

There are additional APL functions that require one argument; however, these functions will be discussed later, in Chapter 6.

HOW MANY ARGUMENTS ARE REQUIRED BY AN APL BUILT-IN FUNCTION?

In this chapter, you will use APL functions with one argument. In the next chapter, you will use some of the same APL function symbols with two arguments. As you will see, these symbols perform different APL functions when they are used with one and with two arguments. When you use an APL function with one argument, the argument must be to the right of the function symbol.

APL FUNCTION SYMBOLS THAT ARE A COMBINATION OF TWO CHARACTERS

Some of the APL function symbols you will use are a combination of two characters. You remember that when correcting keying errors, if you positioned the cursor at a certain character and pressed another key, a new character would replace the original character. However, certain APL symbols require two characters, one struck over the other. For these symbols, key the first character, backspace, and key the second character. It does not matter in which order the characters are keyed. The symbols that are a combination of two characters are called *overstruck* characters. Appendix A shows the overstruck characters and the keys required to enter them.

Note: If you key an overstruck character and then want to change it, you can position the cursor at the character and key another character. The new character will replace the overstruck character.

DETERMINING THE WHOLE NUMBERS NEAREST A FRACTION

When you want to disregard the fractional part of a number and just consider the nearest whole number, you can use the floor ⌊ and ceiling ⌈ functions. The floor function will round the number down to the next smaller whole number and the ceiling function will round the number up to the next larger whole number.

EXAMPLES:

```
      B←3.529
      ⌊B
3
      ⌈B
4
      C←3
      ⌊C
3
      ⌈C
3
      B←¯3.529
      ⌊B
¯4
      ⌈B
¯3
```

If the number is already a whole number, the result is the same as the argument.

The result for the floor and ceiling functions is determined according to the number's position on the number line:

```
(smaller)  ┼──┼──┼──┼──┼──┼──┼──┼──┼  (larger)
           ¯4 ¯3 ¯2 ¯1 0  1  2  3  4
```

Rounding to the Nearest Whole Number

It is a common practice to round numbers to the nearest whole number. You can do this by adding .5 to the number and then using the floor function.

EXAMPLES:

```
      X←4.4+.5          Rounds 4.4 to the nearest whole number.
      ⌊X
4
      X←4.6+.5          Rounds 4.6 to the nearest whole number.
      ⌊X
5
      ⌊X←4.4+.5
4
      X←⌊4.6+.5
      X
5
```

These examples could also be entered this way.

SORTING A VECTOR IN ASCENDING OR DESCENDING SEQUENCE

The grade up ⍋ and the grade down ⍒ functions can be used to sort a vector into ascending or descending sequence, because they give you the indices of the argument in ascending or descending order.

EXAMPLES:

```
      A←80 45 62 37 29 74 58 15 96
      ⍋A
8 5 4 2 7 3 6 1 9
```

The largest value is the ninth element.

The smallest value is the eighth element.

```
      B←A[⍋A]
      B
15 29 37 45 58 62 74 80 96
```

Indexing A this way sorts the elements of A in ascending order.

Remember, when indexing elements in a vector, the index numbers or the index expression must be enclosed in [ ].

```
      ⍒A
9 1 6 3 7 2 4 5 8
      C←A[⍒A]
      C
96 80 74 62 58 45 37 29 15
```

The elements of A are sorted in descending order.

GENERATING A RANDOM NUMBER

To generate a random number, you can use the roll function ?, which generates a random number between 1 and the value of the argument.

EXAMPLES:

```
      X←?6 ←────────────── Generates a number between 1 and 6.
      X
5 ←──────────────────────── The result can be any number between 1 and 6.
      X←?6 6 6 6 ←──────── When this function is used with a vector, a
      X                    random number is generated for each element.
1 3 1 3
```

GENERATING CONSECUTIVE NUMBERS

There are times when you will want to generate a vector of consecutive numbers from one value to another value. You can do this by entering an instruction like this:

```
      VECTOR←1 2 3 4 5 6 7 8
      VECTOR
1 2 3 4 5 6 7 8
```

However, you can also use the index generator function ⍳, which generates consecutive numbers from 1 to the value specified by the argument.

EXAMPLES:

```
      ⍳8
1 2 3 4 5 6 7 8 ←────────── Eight consecutive numbers
      VECTOR←5+⍳5 ←──────── Five added to each consecutive number
      VECTOR
6 7 8 9 10
      2*⍳6 ←─────────────── First 6 powers of 2
2 4 8 16 32 64
```

Generating an Empty Vector

An empty vector is just that—a vector with nothing in it (no elements). Why have a vector with nothing in it? As you will see later, when joining two items together or branching in a user-defined function, there are times when you will want to generate an empty vector. One way to generate an empty vector is to use ⍳0.

EXAMPLES:

```
      NAME
VALUE ERROR  ←——— An error occurs if you use a variable name that does
      NAME            not have a value assigned.
      ^

      NAME←⍳0  ←——— Generate an empty vector.
      NAME

                      The result is a blank display line (no value).
```

FINDING THE SHAPE OF AN EXISTING VARIABLE

As you learned in Chapter 2, the left argument of the reshape function determined the number of elements in a vector or the number of rows and columns in a matrix. Thus, the number of elements in a vector or a matrix is referred to as the *shape* of the vector or matrix. For example, the shape of matrix M, which has two rows and three columns, is: 2 3. To find the shape of an existing variable, you can use the shape function ρ.

EXAMPLES:

```
      SCALAR←4
      VECTOR←2 4 6 8
      MATRIX←2 3ρ⍳6      ——— Reshape function (has two arguments).
      ρSCALAR
      ←——————————————————— Blank display line—the shape of a scalar
      ρVECTOR                is an empty vector.
4  ←——————————————————————— Number of elements in the vector.
      ρMATRIX
2 3  ←————————————————————— Number of rows and columns in the matrix.

      EMPTY←⍳0  ←—————————— Generates an empty vector.
      EMPTY
      ←——————————————————— Blank display line.
      ρEMPTY
0  ←——————————————————————— Number of elements in an empty vector.
```

Chapter 4. APL Functions That Require Two Arguments

In this chapter you will use some APL functions that require two arguments. You can use these functions to do the following:

- Compare the arguments to determine if one is equal to, greater than, or less than the other argument.
- Process logical data—true (1's) and false (0's) data.
- Find the larger of two numbers.
- Find the smaller of two numbers.
- Find the index of a value in a vector.
- Generate a random sequence of numbers.
- Compress (select certain elements from) a vector or matrix.
- Expand a vector or matrix by inserting zeros or blanks.
- Join two items together.
- Find the logarithm of a number.

There are additional APL functions that require two arguments; however, these functions will be discussed later, in Chapter 6.

RELATIONAL FUNCTIONS

When solving problems with APL, you might want to test the relationship between two values. For example, you might want to test a counter to see if it has reached a certain value; or you might want to do something different in the solution to your problem, depending on whether a certain condition is true or false. The following APL functions are used to test the relationship between two values:

Function	Symbol	Key
Greater than	>	$\boxed{\substack{> \\ 7}}$
Less than	<	$\boxed{\substack{< \\ 3}}$
Greater than or equal to	≥	$\boxed{\substack{\geq \\ 6}}$
Less than or equal to	≤	$\boxed{\substack{\leq \\ 4}}$
Equal to	=	$\boxed{\substack{= \\ 5}}$
Not equal to	≠	$\boxed{\substack{\neq \\ 8}}$

When these functions are used, the relationship between the two values is evaluated, and a 1 results if the relationship is true, and a 0 if false.

EXAMPLES:

```
      A←10
      B←20
      A=B
0
      A=B-10
1
      A≤B
1
      A≠B
1
      A≥B
0
      A←'ABC'
      B←'DEF'
      A=B
0 0 0
      A≠B
1 1 1
```

The = and ≠ operators also work with character data. Remember, each element is compared with the corresponding element in the other argument.

Why Two Numbers Identical in Appearance Are Not Always Equal

APL stores all numeric values with an internal precision of 15 decimal digits; however, decimal values with more than five significant digits are normally rounded off to five digits before they are displayed. Thus, occasionally, different numbers will look alike when displayed.

EXAMPLES:

```
      A←1÷3
      B←.33333
      A
0.33333                    Only five of the 15 digits are displayed.
      B
0.33333
      A=B                  The values are not equal.
0
      ⎕PP                  ⎕PP is a system variable that determines how
5                          many significant digits will be displayed. This
                           variable is automatically set to 5 when the
                           power is turned on or RESTART is pressed.
                           (The system variables are discussed in
                           Chapter 9.)

      ⎕PP←15               Set the ⎕PP system variable so that 15
      A                    significant digits will be displayed.
0.333333333333333
      B
0.33333                    Notice the difference between the two values.
      ⎕PP←5
                           Set the ⎕PP system variable back to 5.
```

Remember, the value displayed may not be the exact value that the 5100 has stored for the variable.

An Example Using a Relational Function

Suppose the correct answer to a problem has been stored as a variable called RIGHT and the answer supplied by a student has been stored as a variable called ANSWER. To keep track of the student's score, you want to add 1 to his score if his answer is the same as the right answer; otherwise, you want to leave his score unchanged.

If the student got the problem right, it is true that ANSWER=RIGHT. To add 1 to his score only if his answer is equal to the right answer, you could enter this instruction:

```
SCORE←SCORE+ANSWER=RIGHT
```

Then the amount added to SCORE is 1 when the two values are equal and 0 when they are not equal.

Suppose that instead of adding 1 when the student is right, you want to give some problems more weight than others. The weight of the current problem is stored under the variable WEIGHT. If the student gets the problem right, you want to add WEIGHT to his score; otherwise, you want to leave his score unchanged. You could enter this instruction:

```
SCORE←SCORE+WEIGHTxANSWER=RIGHT
```

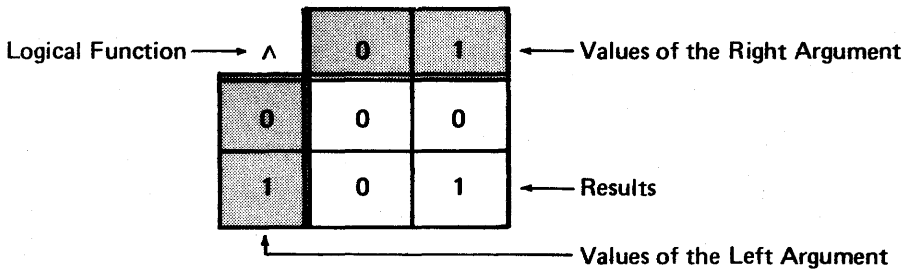
If the student's answer is equal to the right answer, then ANSWER= RIGHT has the value 1, so the amount added is WEIGHT x 1. But if the answers are not equal, then the amount added is WEIGHT x 0, which is 0.

LOGICAL FUNCTIONS

The logical functions take only ones and zeros as arguments and are used to check for certain conditions. (They usually check the results of relational functions.) The fundamental logical functions are:

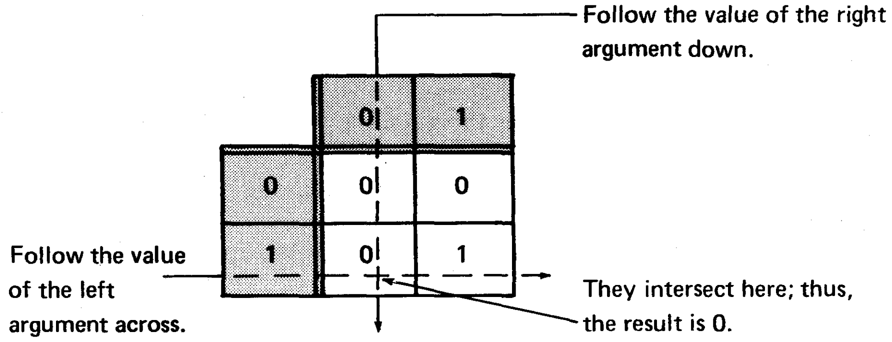
Function	Symbol	Key
And	∧	∧
Or	∨	∨

In our discussion of the logical functions, we will use tables like the following one to show the possible results of the logical functions:



∧	0	1
0	0	0
1	0	1

To use this table, simply find the value of the right argument on top of the table and the value of the left argument on the left side of the table. Then, follow the column represented by the right argument down and the row represented by the left argument across. Where they intersect is the result of the logical function when those values are supplied as arguments. For example, find out what the result of 1 ∧ 0 is as follows:



Follow the value of the right argument down.
0
1
0
0
0
1
0
1
Follow the value of the left argument across.
They intersect here; thus, the result is 0.

And ∧

∧	0	1
0	0	0
1	0	1

Right argument.

The result is 1 only if both arguments are 1.

Left argument.

The And function is used to check for two conditions being true.

For example, suppose you want to know when the items that cost more than $100.00 have a quantity less than 10. You could use the following instruction:

```
(COST>100)∧(QUANTITY<10)
```

The result is 1 when the quantity is less than 10.

The result is 1 when the cost is greater than 100.

Notice how the parentheses in this instruction specify the order of execution.

EXAMPLES:

```
      QUANTITY←8
      COST←120
      (COST>100)∧(QUANTITY<10)
1
      QUANTITY←25
      (COST>100)∧(QUANTITY<10)
0
```

1 — Both conditions are true.

0 — At least one condition is *not* true.

Or $\boxed{\vee}$

∨	0	1
0	0	1
1	1	1

Right argument.

The result is 1 if either argument (or both) is 1.

Left argument.

The Or function is used to check for at least one of two conditions being true.

For example, suppose you want to know when either the inventory for a certain item is less than 10 or the orders for that item exceed the inventory. You could use the following instruction:

```
(INVENTORY<10)∨(ORDERS>INVENTORY)
```

(ORDERS>INVENTORY) — The result is 1 when the orders are greater than the inventory.

(INVENTORY<10) — The result is 1 when the inventory is less than 10.

EXAMPLES:

```
      INVENTORY←15
      ORDERS←5
      (INVENTORY<10)∨(ORDERS>INVENTORY)
0 ←─────────────────────────────────────── Both conditions
      ORDERS←25                            are false.
      (INVENTORY<10)∨(ORDERS>INVENTORY)
1 ←─────────────────────────────────────── At least one of
                                           the conditions
                                           is true.
```

Problems: Using Relational and Logical Functions

1. It is vital to build error checking into all space systems to prevent catastrophy. For example, two indicators checking one condition are commonplace. If either or both of the indicators show danger, action must be taken.

   Assume that the *A* indicator is over its limit at 1.3725 amps and the *B* indicator is over its limit at 1.5365 amps. Enter an expression that will result in a 1 when one or both indicators are outside their limits; the indicators read 1.3732 and 1.5362, respectively.

2. A survey was conducted by the PTA in which the teacher and the parent of the child each evaluated ten of the child's characteristics.

   One child's teacher replied 1, 0, 1, 1, 0, 1, 0, 0, 1, 0 to the questions dealing with his characteristics. His parent answered 1, 0, 0, 1, 0, 1, 1, 0, 0, 0.

   Show which questions the teacher and parent both replied to with a 1.

Possible Solutions

Problem 1:

```
      (1.5365≤1.5362)∨1.3725≤1.3732
1
```

Problem 2:

```
      (⍳10)×1 0 1 1 0 1 0 0 1 0∧1 0 0 1 0 1 1 0 0 0
1 0 0 4 0 6 0 0 0 0
```

FINDING THE LARGER OF TWO NUMBERS

The result of the maximum ⌈ function is the larger of the two arguments.

EXAMPLES:

```
      A←5
      B←6
      A⌈B
6
      (B×B)⌈A×A
36
```

To see how you could use the maximum function, suppose you work for a department store. Each month the store calculates the amount charged and the amount paid by each customer. Your job is to find the difference between the total accumulated charges and the total accumulated payments for each customer. This difference is stored in a variable named BALDUE. The store also charges a service charge of 1.5% of the unpaid balance each month. You could find this charge with the following instruction:

```
CHARGE←BALDUE×.015
```

However, some of the customers have overpaid their bills. For them, BALDUE is a negative number and shows as a credit on their monthly statements. If you calculate the service charge by the instruction just shown, you will be paying them interest at a rate of 1.5%. Instead, the store prefers to calculate the service charge as 1.5% of the balance due or of 0, whichever is greater. To do this, you could use the following instruction:

```
CHARGE←.015×0⌈BALDUE
```

FINDING THE SMALLER OF TWO NUMBERS

The result of the minimum ⌊ function is the smaller of the two arguments.

EXAMPLES:

```
        A←5
        B←6
        A⌊B
 5
        (B×B)⌊A×A
25
        C←1 4 7 9
        D←3 2 8 7
        C⌊D
1 2 7 7
```

Problems: Using the Maximum and Minimum Functions

1. Find the largest dollar expenditure for the following gasoline purchases:
   a. 16.8 gal at 52.9 cents per gal
   b. 13.5 gal at 55.9 cents per gal
   c. 15.6 gal at 57.9 cents per gal

2. For the following purchases, find the smallest quantity of nuts received:
   a. 71 cents for walnuts at 33 cents per lb
   b. 53 cents for cashews at 27 cents per lb
   c. 64 cents for pecans at 29 cents per lb

Suggested Solutions

Problem 1:

```
        (16.8×.529)⌈(13.5×.559)⌈15.6×.579
9.0324
```

Problem 2:

```
        (71÷33)⌊(53÷27)⌊64÷29
1.963
```

FINDING THE INDEX OF A VALUE IN A VECTOR

When you want to find out if a value is an element in a vector, and if it is, which element it is, you use the index of ι function. The index of function gives you the position (index) of the first occurrence in the left argument of the values in the right argument. If a value in the right argument is not in the left argument, the result is 1 plus the length of the left argument.

EXAMPLES:

```
      N←8
      23 33 23 8 16 29⍳N
4
      A←'ABCDEFG'
      A⍳'CAFE'
3 1 6 5
      B←2 4
      C←1 3 2 5 4 4
      C⍳B
3 5
      (⍳7)⍳9
8
```

Index of the first occurrence.

Index generator function.

Index of function.

Value does not occur in the left argument; the result is 1 plus the length of the left argument.

GENERATING A RANDOM SEQUENCE OF NUMBERS

In Chapter 3, you used the roll function (? with one argument) to generate one random number. But by using the deal function (? with two arguments) you can generate a random sequence of numbers without generating the same number twice. That is, the deal function generates the number of random numbers specified by the left argument from 1 through the value specified by the right argument. The random numbers are selected so that no two numbers are the same. Therefore, the left argument cannot be greater than the right argument. If you specify the left argument equal to the right argument, you get all the numbers from 1 through the number specified by the right argument, in random order.

EXAMPLES:

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

May be any two different numbers between 1 and 5.

These numbers can be in any order, as you will see if you enter this instruction several times.

SELECTING CERTAIN ELEMENTS (COMPRESSING) FROM A VECTOR OR MATRIX

You can use the compress function / to select certain elements from a vector or matrix. The left argument must be a vector of all 1's and 0's or an expression that results in such a vector. When selecting elements from a vector the number of elements in each argument must be the same; the corresponding elements of the right argument are retained for each 1 in the left argument.

EXAMPLES:

```
      V←1 2 3 4 5 6 7 8
      1 0 1 1 0 1 1 1/V
1 3 4 6 7 8
      0 0 0 0 0 0 0 0/V

      A←1 0 0 0 0 0 0 1
      A/V
1 8
      1 0 1 0/'ABCD'
AC
```

Result is an empty vector.

When selecting elements from a matrix, you must select and omit entire rows or columns. To do this, you must specify the coordinate (rows or columns) to be acted on by using an index value [I]. The index value is 1 if the first coordinate (rows) will be acted on and 2 if the second coordinate (columns) will be acted on.

EXAMPLES:

```
      B←3 4⍴⍳12
      B
 1  2  3  4
 5  6  7  8
 9 10 11 12           Remember, the left argument must contain a
                      1 for each item to be selected and a 0 for each
                      item to be omitted.
      0 1 0/[1]B
5 6 7 8               The first coordinate (rows) is specified.
      1 0 1 0/[2]B
 1  3                 The second coordinate (columns) is specified.
 5  7
 9 11
      1 0 1 0/B
 1  3
 5  7                 If no index entry is specified, the last
 9 11                 coordinate (columns) is acted on.
```

EXPANDING A VECTOR OR MATRIX

You can use the expand \ function to insert blanks or zeros in a vector or matrix. The left argument must be a vector of all 1's or 0's or an expression that results in such a vector. The number of 1's in the left argument must be equal to the number of elements in the right argument. The 0's in the left argument indicate where the blanks or zeros will be inserted; blanks are inserted in a character vector or matrix and zeros are inserted in a numeric vector or matrix.

EXAMPLES:

```
      1 1 0 1 0 1\1 2 3 4
1 2 0 3 0 4
      1 1 0 1 0 1\'ABCD'
AB C D
```

When expanding a matrix, entire rows or columns of blanks or zeros are inserted. As when using the compress function, you must specify the coordinate (rows or columns) to be acted on by using an index value [I]. The index value is 1 if the first coordinate (rows) is to be acted on and 2 if the second coordinate (columns) is to be acted on.

EXAMPLES:

```
      B←2 2⍴⍳4
      B
1 2
3 4
      1 0 1\[2]B          The left argument must contain a 1 for each
                          row or column being acted on.
1 0 2
3 0 4                     Insert columns.
      1 1 0\[1]B
1 2
3 4                       Insert rows.
0 0

      1 0 1\B
1 0 2
3 0 4
      C←2 2⍴'ABCD'        If no index entry is specified, the last
      C                   coordinate (columns) is acted on.
AB
CD
      1 0 1\C
A B
C D
```

Problems: Using the Compress and Expand Functions

1. Define a vector called ACCTS containing these five accounts: 56 103 100 13 0. Select those with balances of $100 or more.

2. Define the matrix DATA←3 3⍴ ⍳9. Then insert a row in DATA, with the values 20, 21, and 22, after the first row.

Possible Solutions

Problem 1:

```
      ACCTS←56 103 100 13 0
      (ACCTS≥100)/ACCTS
103 100
```

Problem 2:

```
      DATA←3 3⍴⍳9
      DATA
1 2 3
4 5 6
7 8 9
      DATA←1 0 1 1\[1]DATA
      DATA
1 2 3
0 0 0
4 5 6
7 8 9
      DATA[2;]←20 21 22
      DATA
 1  2  3
20 21 22
 4  5  6
 7  8  9
```

JOINING TWO ITEMS TOGETHER

You use the catenate , function to join two vectors together to make a single vector by placing a comma between the left and right arguments. The number of elements in the resulting vector is the sum of the number of elements in the two vectors being joined (catenated).

EXAMPLES:

```
      A←1 2 3
      B←4 5 6
      A,B
1 2 3 4 5 6
      B,A
4 5 6 1 2 3
      C←'CAT'
      D←'EN'
      E←'ATION'
      C,D,E
CATENATION
```

```
      A,C
DOMAIN ERROR
      A,C
      ^
```

A vector must be either all numbers or all characters; therefore, you cannot catenate character data to numeric data.

You also use the catenate function to join two matrices together. To do this, you can use an index value [I] to specify which coordinate is to be extended (that is, whether the number of rows or the number of columns is to increase). The index value is 1 if the first coordinate (number of rows) is to be extended and 2 if the second coordinate (number of columns) is to be extended.

EXAMPLES:

```
      A←2 2⍴⍳4
      B←2 2⍴4+⍳4
      A
1 2
3 4
      B
5 6
7 8
      A,[2]B ←———— You have just joined two columns to two
1 2 5 6            existing columns (increased the number of
3 4 7 8            columns).
      A,B
1 2 5 6
3 4 7 8            When no coordinate is specified, the last
      A,[1]B       coordinate (columns) is acted on.
1 2
3 4
5 6                In this case, you have joined two rows to two
7 8                existing rows (increased the number of rows).
```

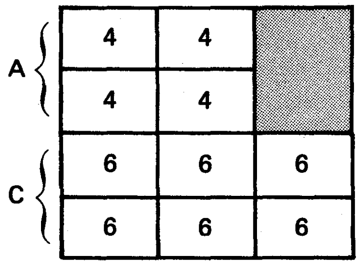
When catenating two matrices, the arguments must conform—that is, the lengths of the columns must be the same if the columns are to be catenated and the length of the rows must be the same if the rows are being catenated.

EXAMPLES:

```
      A←2 2ρ4
      A
4 4
4 4
      C←2 3ρ6
      C
6 6 6
6 6 6
```

```
      A,[1]C
LENGTH ERROR
      A,[1] C
      ^
```

The length error was caused because the row coordinate was specified when A and C have rows of different lengths.



```
      A,[2]C
4 4 6 6 6
4 4 6 6 6
```

Note that the matrices can be joined along the column coordinate, since the lengths of the columns are the same.

Building a Vector of Results Using Catenation

Suppose that as you work through a series of problems you want to accumulate the answers. One way to do this is to catenate each new result to a vector of results previously obtained. If the most recent result is in a variable called LATEST and all the former results are in a vector called RESULT, you could use the following instruction:

```
RESULT←RESULT,LATEST
```

Note: The first time this instruction is executed, there is no value for RESULT. Therefore, before you use this instruction, you should enter the following instruction:

```
RESULT←⍳0
```

This instruction gives RESULT an initial value (makes it an empty vector).

EXAMPLES:

```
      LATEST←10+5

      RESULT←RESULT,LATEST
VALUE ERROR ←──────────────── RESULT does not have a
      RESULT←RESULT,LATEST    value; therefore, it is not a
            ^                 variable and cannot be used
                              in an instruction.

                        ──── Give RESULT an initial
      RESULT←⍳0 ←────        value (empty vector).
      RESULT
←──────────────────────────── Blank display.
      RESULT←RESULT,LATEST
      RESULT ←──────────────── Now RESULT can be used.
15
      LATEST←15+10
      RESULT←RESULT,LATEST
      RESULT
15 25
```

Problem: Using the Catenate Function

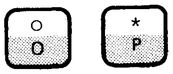
Assign codes to variables as follows: A←'I', B←'T', C←'D', D←'R', E←'GH', F←'YO', G←' ', and H←'U'. Then see what message is displayed if you catenate the variables in the following sequence:

F H G C A C G A B G D A E B

Possible Solution

```
          A←'I'
          B←'T'
          C←'D'
          D←'R'
          E←'GH'
          F←'YO'
          G←' '
          H←'U'
          F,H,G,C,A,C,G,A,B,G,D,A,E,B
    YOU DID IT RIGHT
```

FINDING THE LOGARITHM OF A NUMBER



You use the logarithm ⍟ function to find the log of the right argument to the base specified by the left argument. The log of a number B to a base A is the power needed to raise A to the value B.

EXAMPLES:

```
          A←2
          B←A*3
          B
    8
          A⍟B ←———— The log of B to the base A.
    3
```

Problem: Using the Logarithm Function

1. What is the logarithm of 256 to the base 2?

2. To what power must 10 be raised in order for it to equal 100000?

Possible Solutions

Problem 1:

```
      2⍟256
8
```

Problem 2:

```
      10⍟100000
5
```

Chapter 5. Applying the Same Operations to all the Elements of a Vector Collectively (Reduction)

It is often useful to have the sum (or the product, or the maximum, for example) of all the elements in a vector. APL has a simple procedure for applying the same operation to all the elements of a vector collectively. This operation is called *reduction*, because it reduces a numeric vector down to a single number that represents the sum, the product, or the maximum, for example. The reduction operator is /. The left argument is the function that is applied to all the elements in a vector; the vector is the right argument.

You may have noticed that the reduction operator and the compress function are the same symbol. However, you can tell the difference between the compress function and the reduction operator by the left argument. For the compress function, the left argument is a vector of 1's and 0's and for the reduction operator, the left argument is an APL built-in function.

PLUS REDUCTION

EXAMPLES:

```
      A←1 2 3 4 5
      +/A
15
      1+2+3+4+5 ←———— Adding all the elements of A together is the
15                    same as +/A.
```

Using Plus Reduction To Find the Average

The reduction operator is useful for finding the average of the elements in a vector. Suppose vector X is as follows:

```
X←2 4 3 3 2.5 2
```

The following instruction could be used to find the average of the elements in X:

```
          AVG←(+/X)÷ρX
          AVG
 2.75
```

Now let's analyze the previous instruction.

1. We find the number of elements in X (the length of X):

```
          ρX
 6
```

2. Then we calculate the sum of the elements in X:

```
          +/X
 16.5
```

3. Now we can find the average by dividing 16.5 by 6:

```
          AVG←16.5÷6
          AVG
 2.75
```

Problems: Using Plus Reduction

1. Using reduction, find the average amount that a certain family spends each week on food. The weekly grocery bills for November were $31.05, $29.78, $25.44, and $35.98.

2. Temperatures of a laboratory solution were recorded over a 12-hour period:

6	AM	–	75.8°
7	AM	–	71.9°
8	AM	–	77.0°
9	AM	–	80.3°
10	AM	–	85.1°
11	AM	–	82.2°
12	Noon	–	83.2°
1	PM	–	84.9°
2	PM	–	85.3°
3	PM	–	85.0°
4	PM	–	82.5°
5	PM	–	80.9°
6	PM	–	78.4°

Find the average temperature.

Possible Solutions

Problem 1:

```
      BILLS←31.05 29.78 25.44 35.98

      AVG←(+/BILLS)÷⍴BILLS

      AVG

30.563
```

Problem 2:

```
      TEMP←75.8 71.9 77.0 80.3 85.1 82.2 83.2 84.9 85.3 85.0 82.
      5 80.9 78.4
      AVG←(+/TEMP)÷⍴TEMP

      AVG

80.962
```

Using Plus Reduction to Sum the Products of Two Vectors

Suppose that PRICE is a variable that contains the price list for various items sold by a store, and Q1 and Q2 are two vectors indicating the quantity of these items ordered by two customers. Then the total bill for customer 1 is the sum of the product of PRICE times Q1, and the total bill for customer 2 is the sum of the product of PRICE times Q2.

EXAMPLES:

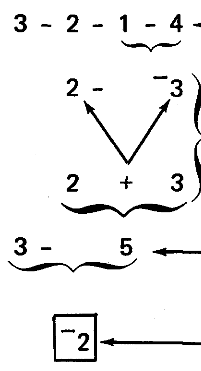
```
      PRICE←.66 1.40 27.10 2.39 14.00 7.60 8.45 2.80
      Q1←0 0 2 1 0 0 0 0
      Q2←12 7 0 5 0 0 0 10
      +/Q1×PRICE
56.59
      +/Q2×PRICE
57.67
```

MINUS REDUCTION (ALTERNATING SUM)

EXAMPLES:

```
      A←3 2 1 4
      -/A
¯2
      3-2-1-4←———— -/A is the same as this instruction.
¯2
```

The following illustration shows why the answer is ¯2.



Direction of processing is from right to left.

First operation (subtract 4 from 1; the result is ¯3).

Second operation (subtract ¯3 from 2; the result is 5).

Third operation (subtract 5 from 3; the final result is ¯2).

Result.

MAXIMUM REDUCTION: FINDING THE LARGEST VALUE IN A VECTOR

To select the largest single element in a vector, you can reduce the vector with the maximum ⌈ function.

EXAMPLES:

```
      BALDUE←62.15 127 4.42 18.65
      ⌈/BALDUE
127
```

Amount owed by all the customers of a store.

Largest amount owed.

MINIMUM REDUCTION: FINDING THE SMALLEST VALUE IN A VECTOR

To select the smallest single element in a vector, you can reduce the vector with the minimum ⌊ function.

EXAMPLES:

```
      NUMBER←1 16 4 7 ¯9
      ⌊/NUMBER
¯9
```

OR REDUCTION: CHECKING FOR A SPECIFIC VALUE IN A VECTOR

Suppose you want to know whether a certain value exists in a long vector. You could use Or ∨ reduction to find the answer.

EXAMPLES:

```
      NUMBERS←50?100
      ∨/NUMBERS=8
0
```

Generate a vector of 50 random numbers.

The result of NUMBERS=8 is a vector consisting of a 0 for each element of NUMBERS that does not equal 8 and a 1 for any element that does equal 8.

When the vector (result of NUMBERS=8) is reduced (the Or function is placed between each element), the result is 1 if at least one of the elements was 1.

A displayed result of 1 indicates that the value 8 was in NUMBERS and a 0 indicates that it was not.

AND REDUCTION: CHECKING FOR ALL VALUES IN TWO VECTORS BEING EQUAL

You can use And ∧ reduction to determine whether corresponding elements of two vectors are equal.

EXAMPLES:

```
      KEY←1.01 1.763 1.888 1.2346 1.2272
      LOCK←1.01 1.763 1.898 1.2346 1.2272
      ∧/KEY=LOCK
0
```

Two vectors that have the same number of elements.

At least one of the elements of KEY does not match the corresponding element of LOCK.

Chapter 6. Using the Built-in Functions

This chapter contains a summary of the things you can do with the APL built-in functions. Some of the functions have already been discussed in the previous chapters and all of the functions are described in the *IBM 5100 APL Reference Manual,* SA21-9213. Also there is an example included for each function; you should enter these examples on your 5100 to see how these functions work.

Note: Many of these functions provide special mathematical capabilities.

NOW LET'S LOOK AT THE THINGS YOU CAN DO

Things You Can Do	Function Name	Keys

APL Functions That Require One Argument (see Chapter 3 for more information)

Things You Can Do	Function Name	Keys
• Determine the next larger whole number	Ceiling	⌈ S

```
      ⌈4.68 6 ←——— If the number is already a whole number,
5 6                the same number is the result.
```

Things You Can Do	Function Name	Keys
• Determine the next smaller whole number	Floor	⌊ D

```
      ⌊4.68 2 ←——— If the number is already a whole number,
4 2                the same number is the result.
```

Things You Can Do	Function Name	Keys
• Sort a string of numbers in ascending order	Grade up	∆ H ∣ M

```
      ⍋A←3 7 2 9 1 ——— Indices of A in ascending order
5 3 1 2 4 ←
      A[⍋A] ←——— Sorts A using the indices
1 2 3 7 9
```

Things You Can Do	Function Name	Keys
• Sort a string of numbers in descending order	Grade down	∇ G, ∣ M

```
        ⍒A←3 7 2 9 1 ──── Indices of A in descending order
4 2 1 3 5 ←
        A[⍒A]← ─────────── Sorts A using the indices
9 7 3 2 1
```

• Generate a random number	Roll	? Q

```
        ?6
3 ←────────────────────── The result can be any number between
                          1 and 6.
```

• Generate a consecutive string of numbers	Index generator	⍳ I

```
        ⍳5
1 2 3 4 5←─────────────── Generates a string of five consecutive
                          numbers.
```

• Determine the length of a string or the number of rows and columns in a table	Shape	⍴ R

```
        ⍴A ────────────── Length of the string named A
5←
        ⍴MATRIX←2 3⍴⍳6← Creates a table and finds its shape in the
2 3                     same instruction (the number of rows and
        MATRIX          columns)
 1 2 3
 4 5 6                  Reshape function (discussed in Chapter 2)

                        Shape function
```

Things You Can Do	Function Name	Keys
APL Functions That Require Two Arguments (see Chapter 4 for more information) The result from the following six functions is 1 if the relationship specified by the APL function is true; otherwise the result is 0.		

• Determine whether two values are equal
`      33=33`
`1`
	Equal to	= 5

• Determine whether the left argument is greater than the right argument
`      16>7`
`1`
	Greater than	> 7

• Determine whether the left argument is less than the right argument
`      3<4`
`1`
	Less than	< 3

• Determine whether the left argument is greater than or equal to the right argument
`      12≥11 12`
`1 1`
	Greater than or equal to	≥ 6

• Determine whether the left argument is less than or equal to the right argument
`      6≤6 9`
`1 1`
	Less than or equal to	≤ 4

• Determine whether two values are not equal
`      7≠77 7`
`1 0`
	Not equal to	≠ 8

Things You Can Do **Function Name** **Keys**

The following two logical functions are usually used to check the results from relational operations. Logical functions can use only 1's and 0's as arguments. The result is 1 when the condition being checked for is met; otherwise, the result is 0.

- Determine whether two conditions are true — And — ∧ 0

```
      1∧1 0
1 0
```

- Determine whether at least one of two conditions is true — Or — ∨ 9

```
      1∨1 0
1 1
```

- Find the larger of two numbers — Maximum — ⌈ S

```
      5⌈4
5
```

- Find the smaller of two numbers — Minimum — ⌊ D

```
      5⌊4
4
```

- Find the index of a given value in a vector — Index of — ⍳ I

```
      9 8 7 6 5⍳7
3
```

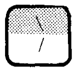
The right argument is found in the third position of the left argument, which is a vector.

- Generate a specific number of different random numbers — Deal — ? Q

```
      3?6
2 3 1
```

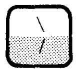
Can be any three different numbers between 1 and 6

Things You Can Do **Function Name** **Keys**

- Compress (select certain elements from) a vector or matrix — Compress 

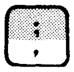
```
      1 0 0 1/2 3 4 5
2 5
```

Selects the elements that correspond to the ones in the left argument

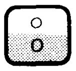
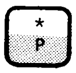
- Expand a vector or matrix — Expand 

```
      1 0 1 0 1 0 1 0\2 3 4 5
2 0 3 0 4 0 5 0
```

Inserts elements according to the zeros in the left argument

- Join two arguments together — Catenate 

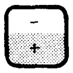
```
      'CAT','EN','ATION'
CATENATION
```

- Find the log of a number — Logarithm  

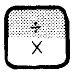
```
      2⍟8
3
```

Log of 8 to the base 2

APL Functions In Addition To The Ones Already Discussed In Previous Chapters (see the IBM 5100 APL Reference Manual, SA21-9213, for more information)

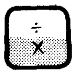
- Change the sign of a number — Negation 

```
      -3 ¯4
¯3 4
```

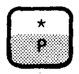
- Find the sign of a number — Signum 

```
      ×¯2 0 2
¯1 0 1
```

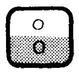
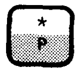
The result is ¯1 for a negative number, 0 for 0, and 1 for a positive number.

- Find the reciprocal of a number — Reciprocal 

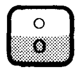
```
      ÷3
0.33333
```

Things You Can Do	Function Name	Keys
• Raise e (2.71828) to a power	Exponential	

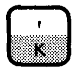
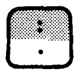
```
      *1 3
2.7183 20.086
```

• Find the log of a number to the base e	Natural log	 

```
      ⍟2.7183 20.086
1 3
```

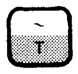
• Multiply a number by pi (3.14159)	Pi times	

```
      ○1 3
3.1416 9.4248
```

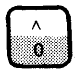
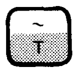
• Find the product of all whole numbers between 1 and a specified number	Factorial	 

```
      !4
24 ←
```

The result is the same as 1x2x3x4.

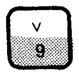
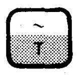
• Change a 1 to a 0 or a 0 to a 1	Logical not	

```
      ~1 0
0 1
```

• Determine whether at least one of two conditions is false	Nand	 

```
      1⍲1 0 ←
0 1
```

The result is 1 when at least one argument is 0; otherwise the result is 0.

• Determine whether two conditions are false	Nor	 

```
      1⍱1 0 ←
0 0
```

The result is 1 when both arguments are 0; otherwise the result is 0.

Things You Can Do | **Function Name** | **Keys**

- Change a scalar or matrix into a vector — Ravel — ; ,

```
      MATRIX←2 3⍴⍳6
      MATRIX
 1 2 3
 4 5 6
      ,MATRIX
1 2 3 4 5 6 ←————The matrix is changed to a vector.
```

- Execute a character string as an APL expression — Execute — ∘ J ⊥ B

```
      ⍎'2×4'
8
```

- Convert numeric data into character data — Format — ∘ J ⊤ N

```
      A←⍕24 ←————How to use this function with two arguments
      A          is discussed in the IBM 5100 APL Reference
24←              Manual, SA21-9213.
      ⍴A
                 This is a character value.
2←
                 A is a 2-element (character) vector.
```

How to use this function with two arguments is discussed in the *IBM 5100 APL Reference Manual,* SA21-9213.

- Find the value of a number without regard to the sign of the number — Absolute value — | M

```
      |53 ¯46
53 46
```

- Invert a square matrix or compute the pseudo-inverse of a rectangular matrix — Matrix inverse — ÷ X ⎕ L

```
      ⌹A←2 2⍴1 3 5 7
 ¯0.875        0.375
  0.625       ¯0.125
```

- Reverse the elements in a vector or matrix — Reverse — | M ○ O

```
      ⌽'LIVE'
EVIL
```

Things You Can Do	Function Name	Keys
• Find the remainder left over from a divide operation	Residue (remainder)	| M

```
      3|8
2
```

2 is the remainder of 8 divided by 3.

• Find the values for the trigonometric functions of an angle	Circular	○ O

```
      3○B←○÷4
1
```

The left argument specifies the trigonometric function (in this case, tangent).

The result is the tangent of 45° (π ÷ 4 radians).

• Find the number of combinations of a number taking so many at a time	Binomial (combination)	' K : .

```
      2!4
6
```

Four items taken two at a time can make six different combinations.

• Find out if a certain value (left argument) exists in a vector or matrix	Membership	∊ E

```
      'ABC'∊'BANANA'
1 1 0
```

The result is 1 if the value in the left argument exists in the right argument; otherwise the result is 0.

• Express a value in another number system	Decode (base value)	⊥ B

```
      24 60 60⊥1 30 15
5415
```

Expresses 1 hour 30 minutes 15 seconds in all seconds

• Represent a value in a specified number system	Encode (representation)	⊤ N

```
      24 60 60⊤5415
1 30 15
```

Represents 5415 seconds in hours, minutes, and seconds

Things You Can Do	Function Name	Keys
• Solve one or more sets of linear equations with coefficient matrices	Matrix divide	÷ X ⎕ L

```
       26 9⌹B←2 2⍴3 5 1 2
7 1
```

• Take a certain number of elements from a vector or matrix	Take	↑ Y

```
       3↑A←1 2 3 4 5
1 2 3 ←———————— These three elements were taken from the
                vector.
```

• Drop a certain number of elements from a vector or matrix	Drop	↓ U

```
       3↓A
4 5 ←———————— The result is the elements remaining after
              the specified number of elements have
              been dropped.
```

• Join two arguments together by forming an array with an additional dimension	Laminate	; ,

```
     1 2 3 4 5,[.5]6 7 8 9 0
1 2 3 4 5 ←———————— Two vectors are joined to form a matrix.
6 7 8 9 0
```

• Rotate the elements in a vector or matrix as specified by the left argument	Rotate	∣ M ○ O

```
       2⌽1 2 3 4 5
3 4 5 1 2 ←———————— Rotates the vector two positions
```

Things You Can Do	Function Name	Keys
• Create data arrangements with at least one dimension (a data arrangement with two dimensions has both rows and columns)	Reshape	ρ R

```
       ARRAY←2 3 3ρ⍳18
       ARRAY
 1  2  3
 4  5  6
 7  8  9

10 11 12
13 14 15
16 17 18
       ARRAY[2;3;1]
16
```

Each number in the left argument is called a coordinate—this N-rank array has three coordinates.

Last coordinate is the columns.

Next to the last coordinate specifies rows.

Leftmost coordinate is the planes.

Planes

You can index elements within N-rank arrays by putting a semicolon between each coordinate.

Things You Can Do	Function Name	Keys
• Interchange coordinates (such as rows and columns of a matrix) of an array	Transpose or generalized transpose	○ O \ /

```
       ⍉ARRAY
 1 10
 4 13
 7 16

 2 11
 5 14
 8 17

 3 12
 6 15
 9 18
```

When used with one argument, this function reverses the coordinates.

Note: This function could also be used with a left argument that specifies how the coordinates are to be interchanged.

Things You Can Do **Operator Name** **Keys**

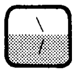
APL Operators

An APL operator applies certain built-in functions to a vector or matrix. The reduction operator has already been discussed in Chapter 5.

- Apply the same operation collectively to all the elements of a vector — Reduction

```
      +/1 2 3 4 5
15 ◄──────────────────────── The sum of the elements
      ⌈/32 6 77 19 2
77 ◄──────────────────────── The largest element
      ⌊/32 6 77 19 2
2 ◄───────────────────────── The smallest element
```

- Apply the same operation cumulatively to each element of a vector (the result of each operation is used in the next operation) — Scan

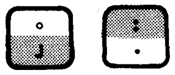


```
      +\1 2 3 4
1 3 6 10

      1
1
      1+2
3
      1+2+3
6
      1+2+3+4
10
```

The scan function works the same as if you entered these instructions.

- Generate operation tables for various APL functions and data — Outer product



```
      A←1 2 3 4
      A∘.×A
 1  2  3  4
 2  4  6  8 ◄──── A multiplication table of numbers
 3  6  9 12       1 through 4
 4  8 12 16
```

Things You Can Do	Operator Name	Keys
• Find the matrix product of two matrices	Inner product	: .

```
          A←2 2⍴1 2 3 4
          B←2 2⍴ 5 6 7 8
          A+.×B
    19 22 ←———————— The matrix product of matrices A and B
    43 50
```

WHAT IS FUNCTION DEFINITION?

Although APL has many built-in functions, there will be times when you want a special function to solve a problem. APL allows you to define your own functions (called *user-defined functions*) and store them for repeated use.

HOW IS A FUNCTION DEFINED?

You use existing APL functions to create a new user-defined function. The new function consists of:

- A function header containing the name of the function and other information (the types of function headers are discussed later in this chapter).
- An instruction or series of instructions, called statements, which define the operation(s) to be performed.

When executing APL instructions, the IBM 5100 is in execution mode; however, before a new function can be defined, the mode must be changed to function definition mode. The ∇ (del) symbol is used to change the 5100 from one mode to another. For example, to change from execution mode to function definition mode, a ∇ is entered as the first character in the function header; then after the function is defined, another ∇ is entered to close the function definition and change the mode back to execution mode. Once the 5100 is back in execution mode, you can execute your user-defined function.

Now, to show how a function is defined, let's create a function to find the hypotenuse of a right triangle. The instruction used for this could be written as ((A*2)+(B*2))*.5, where we square the lengths of the two sides A and B and then take the square root of their sum, which is the length of the hypotenuse. The function must have a name by which it can be identified, so let's name this function HYP. Now enter the opening ∇ (to place the 5100 in function definition mode) and the function header, as follows:

```
      ∇HP←A HYP B ←—— Function header.
[1] ←——————————————— APL responds with the number of the first
                     statement (instruction) to be entered.
```

As each statement is entered, the next statement number is displayed. Now enter the remainder of the function as follows:

```
HP←((A*2)+(B*2))*.5  ←———— Instruction.
∇ ←———————————————————————— Closing ∇ — Changes mode back
                            to execution mode.
```

Notice that the names in the function header (other than the function name itself) are all used in the body of the function. In particular, notice how the result variable name, HP, is assigned the final result by a statement in the function.

The display screen will now look like this:

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

Note: If you make a mistake when entering this function, see *What To Do If You Make a Mistake When Defining Your Function* later in this chapter. The up arrow ↑ (scroll up key) and down arrow ↓ (scroll down key) do not work during editing of user-defined functions.

When you entered the closing ∇, the function HYP was stored in your active workspace, so you can use it just like any other APL function with two arguments.

EXAMPLE:

```
                     ———— Lengths of the two sides.
      3 HYP 4
5 ←————————————————— Length of the hypotenuse.
      X←6
      Y←8
      X HYP Y
10
      R←3 6                Like other APL functions, the arguments can be
      L←4 8                in different forms.
      R HYP L
5 10
```

Whenever you want to use HYP, just enter its name with the arguments you want. The symbol for the calculation of the hypotenuse of a right triangle is HYP, just as the symbol for addition is +.

A function can have only one instruction, like HYP, or it can contain many instructions.

EXAMPLE:

```
      ∇HP←A HYPL B  ←—— The function HYP could also have been
[1] A2←A*2               defined like this.
[2] B2←B*2
[3] S←A2+B2      ←—— Note that the closing ∇ can also be on the
[4] HP←S*.5∇             same line as the last instruction.
      3 HYPL 4
5  ←———————————————— Same result as HYP.
```

Problems: Using Function Definition

1. Define a function that displays the sum of any two numbers. Then use the function.

2. Define a function that displays the area of any rectangle. Then use the function.

Possible Solutions

Problem 1:

```
      ∇S←M SUM N
[1] S←M+N∇
      6 SUM 3
9
```

Problem 2:

```
      ∇A←LENGTH AREA WIDTH
[1] A←LENGTH×WIDTH
[2] ∇
      4 AREA 5
20
```

TESTING YOUR FUNCTION BEFORE USING IT

Once you define your function, you should always try using it with data that will give you a known result. For example, suppose that in the function HYP you used the following expression by mistake:

```
                  ┌——— Should have been *
                  ↓
  ((A*2)+(B*2))×.5
```

You would get an answer, but it would not be the right answer for the hypotenuse of a right triangle.

When you test your function, one of the following will occur:

- The 5100 will display the result you expect.
- The 5100 will display an error message.
- The 5100 will display a result, but not the result you expect.
- Nothing will happen.

If the 5100 Displays the Result You Expect

Great! Your function works.

Note: Even though your function worked one time, you may want to test it some more to make sure it will work for each application you intend to use it for.

If the 5100 Displays an Error Message

You can use the *IBM 5100 APL Reference Manual,* SA21-9213, to find out what the error message means and what you must do to correct it.

Note: An error condition will cause the execution of your function to stop; see Chapter 8 for more information on what to do when your function stops executing.

If the 5100 Displays a Result Other Than the One You Expect, or If Nothing Happens

In either of these cases, you have two alternatives:

- Display the entire function and check it for errors. *Displaying the Entire Function* is discussed later in this chapter.
- Use the trace and stop features (discussed next) to help find the problem.

Note: When a user-defined function is used and nothing happens (that is, neither result nor the cursor appears on the display screen) or a result is repeated continuously, the function is probably *looping.* In this case, press the ATTN key to stop (suspend) function execution. Chapter 8 contains information on what to do when your function stops.

HOW TO USE THE TRACE AND STOP FEATURES

Trace T∆

The trace feature allows you to watch the execution of your function, statement by statement. That is, the final result calculated for each statement traced is displayed. You can either trace all of the statements or just certain statements in a function. To use the trace feature, enter T∆, the function name, ←, and the statement numbers to be traced. For example:

$$\underbrace{\texttt{T∆EXAMPLE}}_{\text{The name of the function to be traced}}\texttt{←}\underbrace{\texttt{1 2 3 4 5 6}}_{\text{The statement numbers to be traced}}$$

The previous statement could also be entered as follows:

$$\texttt{T∆EXAMPLE←}\underbrace{\texttt{⍳6}}_{\text{Generates a vector of numbers from 1 to 6}}$$

Stop S∆

The stop feature allows you to stop the execution of your function just before a specified statement is executed. That is, function execution is temporarily suspended (suspended functions will be discussed in greater detail in Chapter 8). After function execution has stopped, the 5100 displays the number of the next statement to be executed. To use the stop feature, enter S∆, the function name, ←, and the numbers of the statements before which function execution is to stop. For example:

$$\underbrace{\texttt{S∆EXAMPLE}}_{\text{The name of the function}}\texttt{←}\underbrace{\texttt{3 6}}_{\text{The specified statement numbers}}$$

After function execution has stopped, you can start it again by entering →⎕LC. ⎕LC is a *system variable* that contains the next statement number to be executed; see Chapter 9 for more information about system variables, and the *IBM 5100 APL Reference Manual,* SA21-9213, for a complete description of the ⎕LC system variable.

Now let's use trace and stop to find a problem in a function.

EXAMPLES:

```
      ∇HP←A HYPX B
[1]  'THE HYPOTENUSE IS'
[2]  A2←A*2
[3]  B2←B*2
[4]  S←A2+B2
[5]  HP←S×.5∇
```

Defines a function that calculates the hypotenuse of a right triangle.

This function has an error in it.

```
      3 HYPX 4
THE HYPOTENUSE IS
12.5
```

Tests the function using data for which the correct result is known. The result should be 5.

Using the trace feature to find the problem

```
      T∆HYPX←2 3 4 5
      3 HYPX 4
THE HYPOTENUSE IS
HYPX[2] 9
HYPX[3] 16
HYPX[4] 25
HYPX[5] 12.5
12.5
```

The 5100 responds with the function name, statement number, and the result of the statement being traced.

The correct result was obtained in each statement except statement 5; therefore, statement 5 probably contains the error.

```
      T∆HYPX←⍳0
```

To turn off the trace feature, use ⍳0 as the statement to be traced.

```
      S∆HYPX←4 5
```

Using the stop feature to find the problem

```
      3 HYPX 4
THE HYPOTENUSE IS

HYPX[4]
      A2
9
      B2
16
```

The 5100 responds with the function name and the next statement number to be executed.

When the function is stopped, you can enter the variables to see if they contain the expected values.

```
      →⎕LC
```

Continue execution by entering →⎕LC.

```
HYPX[5]
      S
25
      →⎕LC
12.5
```

Execution stops at the next statement specified for the stop feature.

All the variables contained the correct values; therefore, statement 5 must be in error.

```
      S∆HYPX←⍳0
```

To turn off the stop feature, use ⍳0 as the statement to be stopped at.

Note: How to correct an error in a function is discussed next.

WHAT TO DO IF YOU MAKE A MISTAKE WHEN DEFINING YOUR FUNCTION

If you make a mistake when defining your function, you can correct it by editing the function. When editing a function, you can do the following:

- Display the entire function.
- Add one or more statements at the end of the function.
- Replace statements.
- Insert one or more statements.
- Delete a statement from the function.
- Display a specific statement or from a specific statement to the end of the function.
- Modify a single statement.

If you notice your mistake as you are defining your function, you can correct it without reopening the function definition (the 5100 is already in function definition mode). However, if the function definition is closed, you must first reopen it. To do this, you must enter the ∇ followed only by the function name. If you enter the complete function header, you will get an error message.

Now, let's define a function to use in doing some function editing. Enter the following:

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

This function calculates the average, smallest, and largest number in a vector of numbers. Notice that this function does not have a result variable in the function header; however, it will still display the results. The reason for having a result variable in your function will be discussed later.

Displaying the Entire Function ([⎕ ⌊)]

To display a function, you enter [⎕] immediately after any statement number or as shown in the following example.

EXAMPLE:

```
          ∇STAT[⎕]∇
      ∇ STAT X
[1]       N←⍴X
[2]       (+/X)÷N
[3]       ⌊/X
[4]       ⌈/X
      ∇

          STAT 2 9 1
4
1
9
```

This instruction opens, displays, and closes the function definition.

Displayed function.

Try the function.

Adding One or More Statements at the End of the Function

To add statements to a function, you open the function definition and the number of the first available line is displayed. Then you can enter the statements you want to add.

EXAMPLE:

```
                              The 5100 displays the number of the first
                              available line.
      ∇STAT                 ← Open the function.
[5]  (⌈/X)-⌊/X∇             ← Add this statement to find the range of the
                              numbers in the vector. The ∇ closes the function
                              (you are only adding one line).

      ∇STAT [⎕]∇            ← Display the function.
    ∇ STAT X
[1]   N←⍴X
[2]   (+/X)÷N
[3]   ⌊/X                   ← Displayed function.
[4]   ⌈/X
[5]   (⌈/X)-⌊/X
    ∇

      STAT 9 2 1            ← Try the function.
4
1
9
8
```

Replacing Statements within a Function

To replace statements, the statement number to be replaced must be enclosed in brackets [ ] followed by the new statement.

EXAMPLE:

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

∇STAT [⎕] — This instruction opens and displays the function.

Lines [1] through [5] — Displayed function.

```
[6]  [2]  (+/X)÷⍴X∇
```

[6] — The 5100 displays the number of the first available line.

[2] — Notice that you can specify another statement number by enclosing it in brackets. Now, replace statement 2 with this statement for finding the average. The ∇ closes the function.

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

∇STAT[⎕]∇ — Display the modified function.

Lines [1] through [5] — Displayed function.

```
        STAT 9 1 2
4
1
9
8
```

Inserting One or More Statements in a Function

To insert statements in a function, you must use a decimal statement number that is between the numbers of the statements where you want to insert the new statement. For example, to insert a statement between statements 1 and 2, you could use the statement number 1.5 or any decimal number between 1 and 2.

EXAMPLE:

```
        ∇STAT
[6] [1.5] X
[1.6] ∇
```

Open the function.

The 5100 displays the number of the first blank line.

Insert a statement between statements 1 and 2; the inserted statement displays the vector of numbers.

If you do not enter ∇, the 5100 responds with another decimal statement number.

Enter the closing ∇.

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

         STAT 9 2 1
9 2 1
4
1
9
8
```

Display the function.

Notice that the 5100 has renumbered the statement numbers.

Deleting a Statement from a Function

To delete a statement from a function, you enter [Δn], where n is the number of the statement you want to delete.

EXAMPLE:

```
           ∇STAT[⎕]
        ∇  STAT X
[1]        N←⍴X
[2]        X
[3]        (+/X)÷⍴X
[4]        ⌊/X
[5]        ⌈/X
[6]        (⌈/X)−⌊/X
        ∇
[7]     [Δ4]
[5]     ∇
```

Open and display the function.

Displayed function.

The 5100 displays the next available statement number.

Remove statement 4; you no longer need to know the smallest number.

Note: The closing ∇ must *not* be entered on the same line as [Δ n]; you must enter it on another line or an error will occur.

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

           STAT 2 9 1
2 9 1
4
9
8
```

Display the modified function.

Displayed function—the original line 4 was deleted and the statements were renumbered.

Displaying a Specific Statement or from a Specific Statement to the End of a Function

You have already seen how to display the entire function; you can also display only one statement or each statement from a certain statement to the end of the function. To display one statement, you enter [n⎕], where n is the statement number you want to display. To display each statement from a certain statement to the end of the function, you enter [⎕n], where each statement from statement n to the end of the function is to be displayed.

EXAMPLE:

```
        ∇STAT[3⎕]∇←——— Display statement 3.
[3]     (+/X)÷⍴X

        ∇STAT[⎕4]∇←——— Display each statement from statement 4 to
[4]     ⌈/X            the end of the function.
[5]     (⌈/X)-⌊/X
```

Modifying a Single Statement

You can correct keying errors in a statement of a function the same way you correct keying errors made during entering of instructions in execution mode. That is, the same procedures for inserting, deleting, or replacing characters are used. To correct keying errors in function definition mode, you must currently be entering the statement in error or you must display the statement you want to correct.

Note: You cannot use the up or down arrows (scroll up or scroll down keys) when the 5100 is in function definition mode.

EXAMPLE:

```
        ∇STAT[2⎕]       ← Open the function and display statement 2.
[2]     X
[2]     N               ← Enter an N to replace the X in the displayed line.
[3]  ∇                    (You now want to know the number of elements
                          in the vector.)

                        The 5100 responds with [3] ; now enter the closing ∇.

        ∇STAT[⎕]∇       ← Display the function.
      ∇ STAT X
[1]     N←⍴X
[2]     N
[3]     (+/X)÷⍴X        ← N has replaced the X.
[4]     ⌈/X
[5]     (⌈/X)-⌊/X
      ∇

        STAT 2 9 1
3
4
9
8
```

Editing the Function Header

You can edit the function header the same way you would edit any other statement in the function. To do this, you specify statement 0 as the statement to be edited.

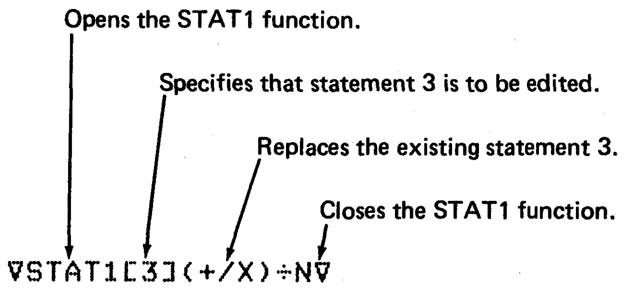
EXAMPLE:

```
        ∇STAT[0]STAT1 X ∇   ← The original function header is
        ∇STAT1[⎕]∇            replaced with this function header.
      ∇ STAT1 X
[1]     N←⍴X                  Display the function.
[2]     N
[3]     (+/X)÷⍴X              Note: Do not be concerned at
[4]     ⌈/X                   this time if the error message
[5]     (⌈/X)-⌊/X             SI DAMAGE is displayed; this
      ∇                       error message and a suggested
                              user response is described in the
                              IBM 5100 APL Reference Manual,
                              SA21-9213.

        ∇STAT[⎕]∇           ← You cannot display the function
DEFN ERROR                    STAT because the function no
               ∇STAT          longer has that name.
                    ^
```

A Faster Way to Add, Replace or Insert One Statement in a Function

If your function is closed and you have only one statement to add, replace, or insert, it can be done using only one instruction. For example, the following instruction opens, changes, and closes the function definition:


Opens the STAT1 function.
Specifies that statement 3 is to be edited.
Replaces the existing statement 3.
Closes the STAT1 function.

```
∇STAT1[3](+/X)÷N∇
```

EXAMPLE:

```
        ∇STAT1[⎕]∇ ←———————— Display the STAT1 function.
      ∇ STAT1 X
[1]     N←ρX
[2]     N
[3]     (+/X)÷ρX
[4]     ⌈/X
[5]     (⌈/X)-⌊/X
      ∇

        ∇STAT1[6]'THIS STATEMENT WAS ADDED '∇
                ←———————— Add a statement to the function.

        STAT1 2 9 1 ←———————— Now try the function.
3
4
9
8
THIS STATEMENT WAS ADDED

        ∇STAT1[3](+/X)÷N∇←———— Replace a statement.
        ∇STAT1[3.5]⌊/X∇←
        ∇STAT1[⎕]∇ ←             Insert a statement.
      ∇ STAT1 X
[1]     N←ρX                   Display the modified function.
[2]     N
[3]     (+/X)÷N
[4]     ⌊/X
[5]     ⌈/X
[6]     (⌈/X)-⌊/X
[7]     'THIS STATEMENT WAS ADDED '
      ∇

        STAT1 2 9 1
3
4
1
9
8
THIS STATEMENT WAS ADDED
```

TYPES OF FUNCTION HEADERS

Like the APL built-in functions, you can have user-defined functions with one or two arguments. You can also have user-defined functions without any arguments. The number of arguments required by a function is defined in the function header. For example:

```
∇RESULT←ARGUMENT1 FUNCTIONNAME ARGUMENT2
```

This function requires two arguments.

```
∇RESULT←FUNCTIONNAME ARGUMENT
```

This function requires one argument.

```
∇RESULT←FUNCTIONNAME
```

This function requires no argument.

When a function is executed, the value used for an argument is assigned to the variable name that appears as the argument in the function header. This variable is then used in the function. For example, you might have the following function:

```
      ∇R←A DIVIDE B
[1]  R←A÷B∇
```

If you enter 10 DIVIDE 2, the value 10 is assigned to A and the value 2 is assigned to B. Now when the statement A÷B is executed, the result is 5.

Note: For some user-defined functions (as with some built-in functions), it is important that you enter the arguments in the proper order. For example, if you enter 2 DIVIDE 10, the answer would be 0.2 instead of 5.

When defining a function with one argument, the argument must be to the right of the function name; otherwise, the argument will be treated as the function name, and vice versa.

EXAMPLES:

```
      ∇R←A AREA1 B
[1]  R←A×B∇
      12 AREA1 12
144
```

Two arguments—this function finds the area of a rectangle.

```
      ∇R←SQRT X
[1]  R←X*.5∇
      A←1 4 9 16 25 36
      SQRT A
1 2 3 4 5 6
```

One argument—this function finds the square root of a number.

The argument can be a vector.

```
      ∇R←DICE
[1]  R←?6 6
[2]  ∇
      DICE
1 5
      DICE
3 4
```

No argument—this function simulates the roll of two dice.

The results can be any pair of numbers between 1 and 6.

WHY HAVE A RESULT VARIABLE?

So far in our discussion of user-defined functions, we have usually defined functions with a result variable. A result variable is a variable name with which the result of a function is temporarily stored for use in an APL instruction. When your function has a result variable, it is said to have an *explicit result.* Without an explicit result, your function cannot be used in an APL expression.

The following function has a result variable; therefore, it has an explicit result.

Result Variable

```
      ∇RESULT←QTY ITEMX COST
[1]  RESULT←COST÷QTY ∇
```

Result Variable

The result variable must appear in both the *function header* and the *body* of the function (it must be included in the statement where the final result is determined).

EXAMPLES:

```
      ∇QTY ITEM COST
[1] COST÷QTY∇
      10 ITEM .60
0.06

      STORE←10 ITEM .60
0.06
VALUE ERROR
      STORE←10 ITEM 0.6
      ^
      10+10 ITEM .60
0.06
VALUE ERROR
      10+10 ITEM 0.6
        ^

      ∇RESULT←Q ITEMY C
[1] RESULT←C÷Q∇

      10 ITEMY .60
0.06

      STORE←10 ITEMY .60
      STORE
0.06
      10+10 ITEMY .60
10.06
```

Define a function without an explicit result.

The result of the function cannot be used in APL expressions.

Define a function with an explicit result.

The result of the function can now be used in an APL expression.

Remember, if you plan to use the function you are defining in calculations, you must provide a result variable.

LOCAL AND GLOBAL NAMES

A name appearing in a user-defined function can be either *local* or *global.* A global name has the same value during the execution of a function as it has outside of the function. A local name has a value only while a function is active. Any name appearing in the function header (except the function name) is a local name. So far we have seen that a function header can contain a result variable and arguments. Since these variable names are contained in the function header, they are local to the function. But other names can also be made local to the function by placing them in the function header following the right argument (if any) with a semicolon preceding each name. For example, the function header ∇ LOOP R;I;J makes the right argument R and the variables I and J local to the function. Now to see how local and global names work, let's use some.

EXAMPLES:

```
      ∇GLOBAL
[1]  GA←3
[2]  GB←4
[3]  GC←5
[4]  GA+GB+GC∇
      GLOBAL
12
      GA
3
      GB
4
      GC
5
```

Define a function without any local names.

Since these names are global variables, they also exist outside the function.

```
      ∇LOCAL  ;LA;LB;LC
[1]  LA←3
[2]  LB←4
[3]  LC←5
[4]  LA+LB+LC∇
      LOCAL
12
      LA
VALUE ERROR
      LA
      ^
      LB
VALUE ERROR
      LB
      ^
      LC
VALUE ERROR
      LC
      ^
```

Define a function with all local names.

Notice how the names are made local to the function.

Execute LOCAL, then enter the variable names to see what values they represent.

Since these variable names are local to the function, they only represent a value during the execution of the function.

```
      ∇COMBINATION;GA;GB
[1]  GA←6
[2]  GB←7
[3]  GC←8
[4]  GA+GB+GC∇
      COMBINATION
21
      GA
3
      GB
4
      GC
8
```

Define a function using both local (GA and GB) and global (GC) names.

Local names that are the same as existing global names.

Global name.

Notice that outside the function, the existing global values (previously established by the function GLOBAL) are used. The new values (6 and 7) existed only during the execution of the function.

Since this variable name is not local to the function, the global value was changed.

Now, you are probably wondering why you should make variable names local to a function. Following are some reasons for using local variables:

- Let's assume you have defined a function named COUNT that uses a variable named X. At some later time, you assign the result of an important calculation to a global variable named X. Now if you execute COUNT, the following conditions can occur:

  1. If X was made local to COUNT, the global value of X is not changed.

  2. If X was not local to COUNT, the global value of X (the results of your important calculation) is changed.

- You can conserve space in your active workspace by not storing the values for variables you do not use outside of a function.

BRANCHING, LABELS, AND LOOPING

Branching and Labels

Statements in a user-defined function are normally executed in the order indicated by the statement numbers, and execution terminates at the end of the last statement in the sequence. However, this normal order of execution can be modified by *branching* (transferring to another point in the sequence). Branching is indicated by a right arrow → followed by a label that specifies the statement to be branched to.

For example, the expression →START means branch to a statement labeled START. When a label is assigned to a statement, the label is followed by a colon and must precede the statement. The colon separates the label from the statement:

```
[2] START:N←N+1
     .
     .
     .
[5] →START
```

In the previous illustration, the label START is assigned to the second statement in the function. In this case, START has a value of 2; however, if the function is edited and the statement is no longer the second statement in the function, START will automatically be given the value of the new statement number. Now as the function executes, when statement 5 is executed, a branch is taken to the statement labeled START.

Labels are local to a function; that is, they can only be used within that function. Following are some rules that apply exclusively to the use of labels:

- They must *not* appear in the function header.

- You cannot assign values to them.

There are two types of branch statements you can use—unconditional branches and conditional branches:

- *Unconditional branches* are branches that are taken each time the branch statement is executed. You have already seen an example of an unconditional branch, [5] →START, where the branch to the statement labeled START is taken each time statement 5 is executed. Another common use of an unconditional branch is →0, which causes the execution of the function to be terminated.

- *Conditional branches* are branches that are taken depending upon some condition that exists at the time the branch statement is executed. Conditional branches are used, for example, to branch to a statement if a condition is true and to otherwise continue with the next statement (fall through). This type of branch can be entered like this:

  ```
  →(CONDITION)/N
  ```

  The branch to statement N is taken if the condition is true; otherwise the next statement is executed. For example, APL executes the branch statement →(I≥N)/START as follows:

  1. First, the condition (I≥N) is evaluated; the result is 1 if the condition is true and 0 if the condition is false.

  2. The result of step 1 is then used as the left argument for the compress (/) function:
     a. If the result of step 1 was 1, START is selected from the right argument and a branch to the statement labeled START is taken.
     b. If the result of step 1 was 0, nothing is selected from the right argument (an empty vector is the result). A branch to an empty vector means execute the next statement in sequence (fall through).

In the following example, you will use two variations of a function to determine the sum of each number from 1 to the value of the argument (each function will use a different method of branching).

EXAMPLES:

```
      ∇S←SUM2 N
[1]  S←0
[2]  I←1
[3]  CHECK:→(I>N)/0 ←—— Branch to 0 (terminate the function) or fall
[4]  S←S+I               through to the next statement.
[5]  I←I+1
[6]  →CHECK∇ ←————————— Unconditional branch to CHECK.
      SUM2 5
15

      ∇S←SUM3 N
[1]  S←0
[2]  I←0
[3]  CHECK:S←S+I
[4]  I←I+1
[5]  →(I≤N)/CHECK∇ ←——— Branch to CHECK or fall through.
      SUM3 5
15
```

Looping

A repeated segment of a function is called a loop; when you have a loop in your program, you must provide a way to get out of the loop.

EXAMPLE:

```
        ∇LOOP ←─────────────────────────── This function executes a
[1]  I←0                                   continuous loop.
[2]  LABEL:'THIS PROGRAM CONTAINS A LOOP'
[3]  I←I+1
[4]  →LABEL∇
        LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
THIS PROGRAM CONTAINS A LOOP
```

Note: To stop execution of LOOP, press the ATTN key.

```
LOOP[3] ←───────────────────────────────── The name of the function
                                           and the statement number
                                           where execution stopped is
                                           displayed.

        ∇LOOP[4]→(I≠3)/LABEL∇ ←─────────── Provide a way to get out of
        ∇LOOP[⎕]∇ ←                        the loop.
      ∇ LOOP
[1]     I←0                                Display the function.
[2]   LABEL:'THIS PROGRAM CONTAINS A LOOP'
[3]     I←I+1
[4]     →(I≠3)/LABEL
      ∇
        LOOP
THIS PROGRAM CONTAINS A LOOP⎫
THIS PROGRAM CONTAINS A LOOP⎬←──── The loop is executed three
THIS PROGRAM CONTAINS A LOOP⎭      times.
```

HOW TO ENTER DATA DURING FUNCTION EXECUTION ⎕ L ⍞ K

So far you have defined functions for which you have supplied the data for the function as arguments. This method of supplying data limits you to two input arguments, and you must be familiar with the function so that you can enter the required arguments in the correct order. However, you can also define user-defined functions that display requests for input data as the function executes. This type of function allows you to input any amount of data; and you can also define your function so that it specifies what type of data is to be entered. To do this, you use the ⎕ (quad) or ⍞ (quad quote) symbols in your function to request input from the keyboard. When a ⎕ is encountered in a function, execution stops and ⎕: is displayed to indicate that the system is waiting for numeric or character input (character data must be enclosed in single quotes) for the keyboard. When a ⍞ is encountered in a function, execution stops, the cursor appears, and the system waits for input from the keyboard; but in this case, everything on the input line from position 1 to the cursor or the last character entered (whichever is the farthest on the input line) is treated as character input, *even though* you do not use enclosing single quotes when you enter the data.

EXAMPLE:

Enter the following user-defined function to determine the final score of a baseball game:

```
      ∇BASEBALL
[1]  'ENTER THE NAME OF THE VISITING TEAM'
[2]  VISIT←⍞
[3]  'ENTER THEIR SCORE BY INNING'
[4]  VSCORE←⎕
[5]  'ENTER THE NAME OF THE HOME TEAM'
[6]  HOME←⍞
[7]  'ENTER THEIR SCORE BY INNING'
[8]  HSCORE←⎕
[9]  'THE FINAL SCORE WAS:'
[10] VISIT
[11] +/VSCORE
[12] HOME
[13] +/HSCORE∇
```

The input from the keyboard will replace the ⎕ or ⍞ and be assigned to the variables.

The score by inning was:	REDS	– 0 1 0 2 0 3 2 5 0
	BLUES	– 0 0 0 2 3 1 3 0 0

EXAMPLE (continued)

Now execute the function:

```
      BASEBALL
ENTER THE NAME OF THE VISITING TEAM
REDS


ENTER THEIR SCORE BY INNING
⎕:
      0 1 0 2 0 3 2 5 0
ENTER THE NAME OF THE HOME TEAM
BLUES
ENTER THEIR SCORE BY INNING
⎕:
      0 0 0 2 3 1 3 0 0
THE FINAL SCORE WAS:
REDS
13
BLUES
9
```

Notice how the messages identify the type of keyboard input required.

This character data is not enclosed in single quotes, since it was requested by a ⍞ in the function.

This is *not* character data, since it was requested by a ⎕ and is not enclosed in single quotes.

Note: A ⎕: indicates that the keyboard input is requested by ⎕ in the function; no ⎕: (blank line) indicates that the keyboard input is requested by ⍞ in the function.

When you are using interactive functions, there may be times when you will need to escape from a request for input. Normally pressing the ATTN key will cause the execution of your function to stop; however, pressing the ATTN key during a request for input does not stop the function (the function will continue to wait for input to be entered). Therefore, APL provides a way to escape from input requests. To escape from a ⎕ input request, you enter →, which will cause execution of your function to be terminated.

To escape from a ⍞ input request, you must enter the Ⓤ character. This character is entered by holding the CMD key and pressing the $\boxed{\substack{-\\+}}$ key once followed by the EXECUTE key. This will cause the execution of your function to stop. What you can do when your function stops is discussed next, in Chapter 8.

EXAMPLE:

Let's use the BASEBALL function to show how to escape from input requests.

→

Entering → in response to a ⎕ input request causes the execution of the function to be terminated.

Try escaping from a ⍞ by entering →. Your entry was treated as a character, and used as the visiting team's name.

Enter some numbers so that the next ⍞ input request will be displayed.

Entering the Ⓤ character (holding CMD and pressing the $\boxed{\begin{smallmatrix}-\\+\end{smallmatrix}}$ key once) causes the execution of the function to stop.

Chapter 8. What You Can Do When Your Functions Stops

The execution of your user-defined function will stop when:

- The ATTN key is pressed.
- The stop feature is used.
- An error is encountered in the function.
- A ⍝ character (the CMD key held and the $\boxed{\overset{-}{+}}$ key pressed once) is entered for a ⍞ input request.

A function that has stopped executing for one of the preceding reasons is called a *suspended* function. A suspended function is still active, since its execution can be resumed later.

Now let's look at what you can do when your function stops executing.

WHEN THE ATTENTION KEY IS PRESSED

When you press the ATTN key during the execution of your user-defined function, the function stops executing at the end of the statement currently being executed. In this case, the 5100 displays the function name and the next statement number to be executed.

After your function stops executing, you can do one of the following:

- Edit the function.
- Execute the function again.
- Execute another user-defined function.
- Execute system commands except for)SAVE,)COPY, and)PCOPY. The system commands are described in the *IBM 5100 APL Reference Manual*, SA21-9213.
- Terminate the function by entering →.

Generally, after you have stopped your function by pressing the ATTN key, you will want to resume execution of the function at a later time. To do this, you enter →⎕ LC. ⎕ LC is a system variable that contains the statement number of the next statement to be executed (see the *IBM 5100 APL Reference Manual*, SA21-9213, for a complete description of the ⎕LC system variable).

Note: If you wanted to resume execution at a statement other than the one immediately following the last statement executed, enter →n (where n is the statement number at which you want to resume execution).

EXAMPLES:

```
      ∇SFUNCTION;COUNT
[1]  COUNT←0
[2]  LOOP:'THIS FUNCTION CONTAINS A LOOP'
[3]  COUNT←COUNT+1
[4]  →LOOP
[5]  'THIS FUNCTION LOOPED'
[6]  COUNT
[7]  'TIMES'∇
```

Define a function with a continuous loop.

```
      SFUNCTION
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
```

Press the ATTN key to stop execution of the function.

```
SFUNCTION[3]
```

The function is suspended at the statement number shown in the [ ] on your display screen.

```
      ∇SFUNCTION[4]→(COUNT<3)/LOOP∇
```

Edit the function so that it does not contain a continuous loop.

```
      →⎕LC
```

Resume execution of the function.

EXAMPLES—continued

```
THIS FUNCTION LOOPED
7 ←——————————————————————————————————————————— The value shown
TIMES                                           here on your display
                                                screen is the number
                                                of times the function
                                                looped.

      SFUNCTION ←————————————————————————————— Now execute the
THIS FUNCTION CONTAINS A LOOP                   function again.
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION CONTAINS A LOOP
THIS FUNCTION LOOPED
3
TIMES
```

Note: When the ATTN key is pressed twice during the execution of an APL statement or expression (either within or outside of a user-defined function), the execution of the statement or expression stops immediately. The message INTERRUPT, the statement being processed, and the caret (∧) that indicates where the statement was interrupted is displayed. You can use this method to interrupt statements that take a long time to execute. However, any results generated by the statement or expression before it was interrupted might not exist after the interrupt.

WHEN THE STOP FEATURE IS USED

You are already familiar with the stop feature, which was discussed in Chapter 7. When using the stop feature (as when using the ATTN key), you can do the following:

- Edit the function.
- Execute the function again.
- Execute another user-defined function.
- Execute system commands except for)SAVE,)COPY, and)PCOPY.
- Resume function execution by entering →⎕LC.
- Terminate the function by entering →.

WHEN AN ERROR IS ENCOUNTERED IN THE FUNCTION

The reason the execution of your function stopped in this case, unlike the reasons in the other two cases, cannot be controlled by you. That is, the 5100 automatically stops the execution of your function and displays an error message when an error occurs in the function. The error messages and a suggested user's response for each error are described in the *IBM 5100 APL Reference Manual*, SA21-9213.

Errors in a user-defined function are sometimes difficult to find and correct. The error message displayed indicates where the execution of the statement stopped, and why; but the reason the failure occurred at that point might have been because a mistake (either a keying error or an error in the solution to the problem) was made earlier in the statement or because a mistake was made in an even earlier statement in the function. Following are some hints to help you find errors in a statement or expression that is failing or giving the wrong results.

- Check the expression (statement) you entered for any keying errors.
- Analyze the execution of the expression from right to left. Remember, APL executes an expression from right to left with the expressions in parentheses resolved (right to left) as they are encountered.
- Use the shape ρ function to make sure the shapes of the arguments are what you expect. For example, suppose you have a function named CAT that catenates two vectors together to form one vector; however, one of the arguments you supplied was a matrix.
- Enter the names to check the values of the arguments to make sure they are what you expect (local names in a suspended function can be displayed, since the function is still active).
- Break the expression down and execute it in smaller segments. The up ↑ and down ↓ arrows (scroll up and scroll down keys) make it easy for you to break the expression down; that is, you can execute the expression like APL does (from right to left with expressions in parentheses resolved as they are encountered). To do this, you enter the first operation performed by APL, for which the result will be displayed. Then press the down arrow three times and the up arrow once to remove the previous result from the display screen (so that it is not on the input line when the EXECUTE key is pressed again) and to place the instruction you just entered in a position for you to add more operations. Now you can add the next operation to the instruction, and the next, until the error in the instruction is found.

It is important that you maintain a history (either a printout on the IBM 5103 Printer or a handwritten copy) of what you did when you were trying to find the cause of an error. Then if you cannot find the error and you think the problem is caused by the 5100, this history will help your service representative determine where the problem is.

When a function has stopped because an error occurred, as when pressing the ATTN key or using the stop feature, you can do the following:

- Edit the function.
- Execute the function again.
- Execute another user-defined function.
- Execute system commands except for)SAVE,)COPY, and)PCOPY.
- Resume execution of the function by entering →⎕LC.
- Terminate the function by entering →.

WHEN A Ⓞ CHARACTER IS ENTERED FOR A ⍞ INPUT REQUEST

In Chapter 7, you used the Ⓞ character to escape from a ⍞ input request and to stop function execution. In that case, the 5100 displayed the message INTERRUPT, the function name, and the statement that requested the input. After your function stops, you can do the same operations that you did when the function stopped for any other reason. However, in most cases, you will want to terminate the function by entering →.

FINDING OUT WHAT FUNCTIONS ARE SUSPENDED

The state indicator contains the function name and the number of the statement to be executed next for each suspended function. To display the state indicator, you enter)SI or)SIV. See the *IBM 5100 APL Reference Manual*, SA21-9213, for more information on the state indicator.

USING THE HOLD KEY TO STOP PROCESSING HOLD

We have already discussed the ways a user-defined function can be suspended. You can also stop the execution of a function by pressing the HOLD key once. In fact, this stops the entire system from processing any data. To resume processing after pressing the HOLD key, you must press the HOLD key again. The HOLD key is useful when the information on the display screen is changing rapidly; that is, you can stop processing, read the displayed information, and then resume processing.

EXAMPLES:

```
      ∇HOLDF ←————————— Define a function.
[1]  H←0
[2]  'PRESS THE HOLD KEY TO STOP PROCESSING'
[3]  LOOP:H←H+1
[4]  H
[5]  →(H≠25)/LOOP∇
```

```
      HOLDF
PRESS THE HOLD KEY TO STOP PROCESSING
1 ⎫
2 ⎪
3 ⎬ ←————————— The value displayed here on your
. ⎪             display screen indicates how many
. ⎭             times the function looped before
                processing stopped.
```

Now press the HOLD key again to resume processing.

Note: If your 5100 is not processing any data or user-defined functions and the cursor is not flashing on the display screen, the HOLD key might have been pressed once, stopping all processing.

Chapter 9. Using Your Tape Cartridge (Library)

So far you have used only the IBM 5100 active workspace. The active workspace is the part of the 5100's internal storage where the calculations are performed; it is also the place where the variables and user-defined functions are stored. When you set the 5100 POWER ON/OFF switch to off or press RESTART, the data in the active workspace is lost. However, before turning the power off or pressing RESTART, you can save the data in your active workspace by writing the contents of the active workspace on a tape cartridge. This tape cartridge is like a library; that is, you can write the contents of your active workspace on the tape (like placing a book on the library shelf) and, at a later time, put the information stored on the tape back into the active workspace (like taking the book off the library shelf to use it again).

The tape library consists of one or more files (each file is like a book), and just as each book in a library has a name, each file that contains information on the tape also can have a name (file identification).

The IBM 5100 system commands are your means of controlling the active workspace and tape (library). Look at the labels above the alphameric keyboard; these system command keywords can be entered by simply pressing the CMD key with the appropriate key below the label. The system command keywords can also be entered character by character. Notice that each system command begins with a) symbol. There are some system commands that do not appear on the labels above the keyboard. All of the 5100 system commands are discussed in detail in the *IBM 5100 APL Reference Manual,* SA21-9213.

In the following example, you will see how some of the system commands work. First, a tape cartridge must be inserted into your 5100. Be sure the tape contains no data required for any further use, and that the SAFE switch (Figure 2) does *not* point to SAFE. Now insert the tape cartridge (Figure 3).

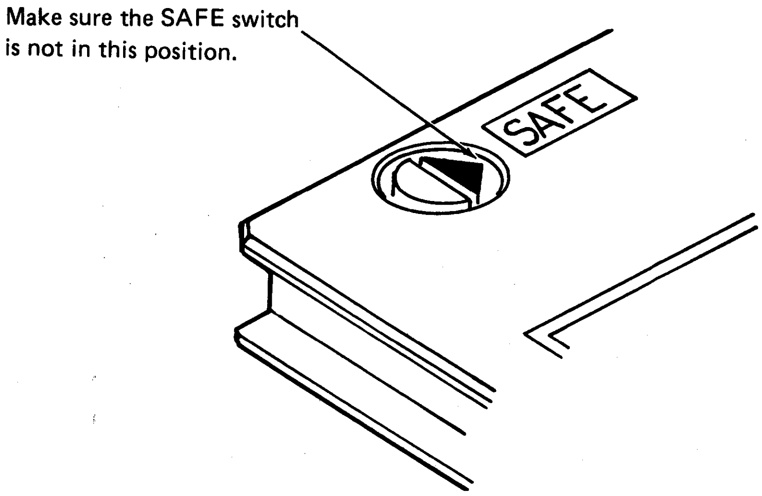


Figure 2. The SAFE Switch

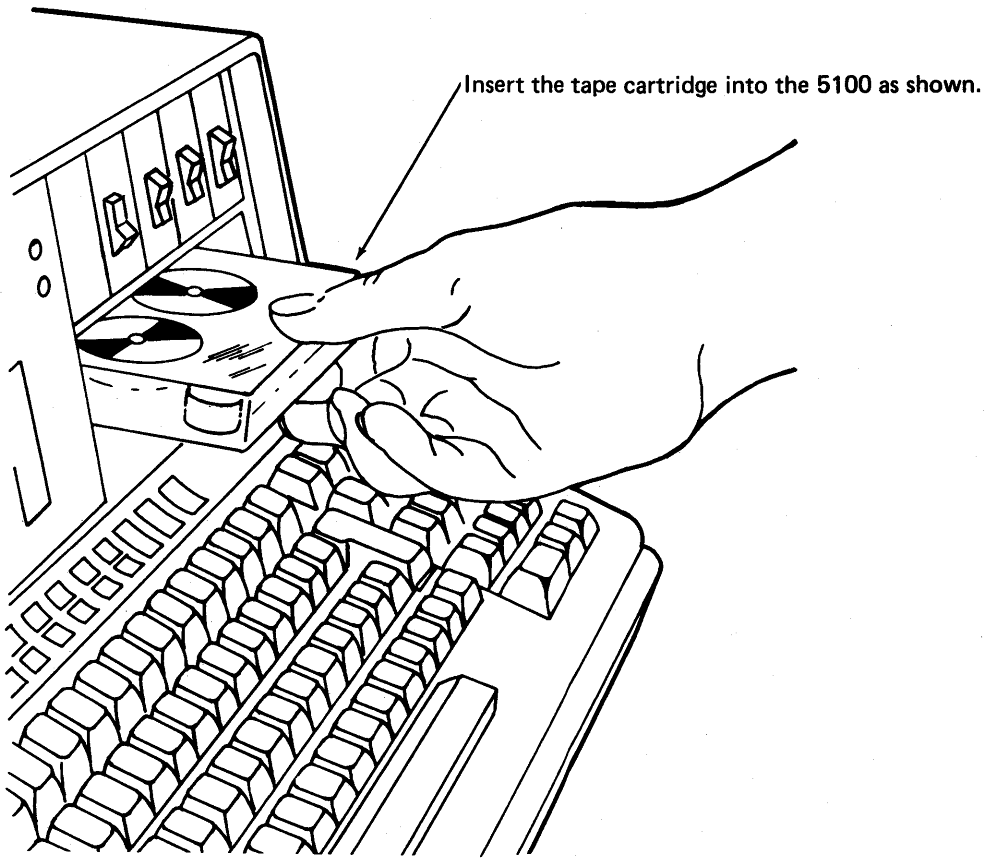


Figure 3. Inserting a Tape Cartridge into Your IBM 5100

EXAMPLES:

Press RESTART on your 5100; all the data that was in the active workspace is now lost.

```
CLEAR WS ←——————————————— This message will be displayed
                          when the 5100 is again ready
                          for you to enter data.
```

Enter the following function and variable so that you can store them on tape for later use:

```
      ∇EXAMPLE;R;NAME
[1]  'THIS FUNCTION COUNTS THE CHARACTERS IN YOUR NAME'
[2]  'NOW ENTER YOUR NAME'
[3]  NAME←⍞
[4]  'THERE ARE'
[5]  ⍴NAME
[6]  'CHARACTERS IN YOUR NAME'∇

      VARIABLE←'LET''S SAVE THIS DATA'
```

Now try the function EXAMPLE to see if it works.

```
      )FNS ←——————————————— The )FNS system command
EXAMPLE                     displays user-defined function
                            names in the active workspace.

      )VARS←——————————————— The )VARS system command
VARIABLE                    displays the global variable
                            names in the active workspace.
```

Before a tape can be used, the files you want to use must be formatted.

The)MARK command formats files on the tape. This command specifies:
- Size of the files to be formatted
- Number of files to format
- Starting file number

```
      )MARK 16 3 1
MARKED 0003 0016
```

APL will respond with MARKED, number of the last file marked, and the size of the files. If the file you want to use has been marked before, you will get a message ALREADY MARKED. In this case, enter GO and press the EXECUTE key to reformat the tape files.

The files are formatted in blocks of 1024 bytes. For example, the size of the files just formatted is sixteen 1024 byte blocks (or 16384 total bytes). See the *IBM 5100 APL Reference Manual,* SA21-9213, for information on what size to format files.

Now let's write the contents of the active workspace on the tape.

```
      )CONTINUE 1001 INFO
CONTINUED 1001 INFO
```

This becomes the name of the file on tape.

This specifies the device/file number (device 1, file 001) where the contents of the active workspace are written.

```
      )CLEAR
CLEAR WS
```

You do not have to turn the power off or press RESTART to clear all of the existing data out of the active workspace; you can use this system command.

The data in a stored workspace can be placed back into the active workspace.

```
      )LOAD 1001 INFO
LOADED 1001 INFO
```

The stored workspace name (workspace ID).

The device/file number where the stored workspace will be loaded from.

```
      )FNS
EXAMPLE
      )VARS
VARIABLE
```

Now the data that was stored on tape is in the active workspace once again.

The remaining system commands are described in the *IBM 5100 APL Reference Manual*, SA21-9213. Try using these system commands to see how they work.

So far, you have learned how to write the entire contents of the active workspace on tape. However, you can also write one variable at a time to a file on tape. This data can then be read from tape at a later time in the same order as it was written to tape. For more information on how to do this, see Chapter 8, *Tape and Printer Input and Output* in the *IBM 5100 APL Reference Manual*, SA21-9213.

WHAT ARE SYSTEM VARIABLES?

System variables are variables within the active workspace that control the system. All system variables begin with the ⎕ symbol and are set to an initial value by the 5100 in a clear workspace. See the *IBM 5100 APL Reference Manual*, SA21-9213, for a complete description of each system variable. In the following example, you will see how the value of some system variables can be changed and how this affects certain APL functions.

EXAMPLES:

The index origin ⎕IO system variable determines the index origin. The value of the ⎕IO system variable can be either 0 or 1, which means that the first element of a vector or array is indexed with a 0 or 1 depending upon what the ⎕IO system variable is set to. The APL functions ⍳ ? ⍋ ⍒ are affected by the ⎕IO system variable.

```
      ⎕IO
1
      ⍳5
1 2 3 4 5
```

You can display the value of a system variable the same way you display the value of any variable.

The ⎕IO system variable is initially set to 1 by the system.

Results when the ⎕IO system variable is set to 1.

```
      3?3
3 1 2
      ⎕IO←0
      ⍳5
0 1 2 3 4
      3?3
1 2 0
```

These numbers can be in any order.

You can change the value of some system variables.

Notice how the results of these APL functions change when the ⎕IO system variable is changed.

These numbers can be in any order. Notice that the values start from 0.

The printing precision ⎕PP system variable determines the number of significant digits displayed.

```
      ⎕PP
5
      1÷3
0.33333
      ⎕PP←2
      1÷3
0.33
```

The ⎕PP system variable is initially set to 5 by the system.

Five significant digits are displayed.

Now only two significant digits are displayed.

The comparison tolerance ⎕CT system variable determines how close two numbers must be when you are using the relational, floor, or ceiling functions.

```
      ⎕CT
1E¯13
      .5555556=.5555557
0
      ⎕CT←1E¯5
      .5555556=.5555557
1
```

The ⎕CT system variable is initially set to 1E¯13 by the system.

These two values are not considered equal.

Now these two values are considered equal.

```
      )CLEAR
CLEAR WS
```

The workspace is clear and the system variables are once again set to their original values.

REMEMBER, APL IS A GOOD LANGUAGE TO EXPERIMENT WITH. THE MORE YOU EXPERIMENT, THE MORE YOU LEARN.

Appendix A. Overstruck Characters

Name	Character		Keys	
Comment	⍝		∩ C	∘ J
Compress	⌿	(See note)	\ /	¯ +
Execute	⍎		⊥ B	∘ J
Expand	⍀	(See note)	\ /	¯ +
Factorial, Combination	!		' K	: .
Format	⍕		⊤ N	∘ J
Grade Down	⍒		∇ G	| M
Grade Up	⍋		∆ H	| M
Logarithm	⍟		* P	○ O
Matrix Division	⌹		⎕ L	÷ X
Nand	⍲		∧ 0	~ T
Nor	⍱		∨ 9	~ T
Protected Function	⍫		∇ G	~ T

Name	Character	Keys	
Quad Quote	⍞	⎕ L	' K
Rotate, Reverse	⌽	| M	○ O
Rotate, Reverse	⊖ (See note)	○ O	− +
Transpose	⍉	\ /	○ O

Note: These are variations of the symbols for these functions; they are used when the function is to act on the first coordinate of an array.

Index

)CLEAR command 107, 109
)CONTINUE command 107
)FNS command 106
)LOAD command 107
)MARK command 107
)VARS command 106
[I] 45, 46, 48
[⎕] 78
[⎕n] 83
[n⎕] 83
[∆n] 82
⎕ 95
⎕: 95
⎕CT 109
⎕IO 108
⎕PP 36, 109
⍞ 95
→⎕LC 76, 99
⍝ character 96, 102
∇ symbol 71
⌈ function 29, 41
⌊ function 29, 42
⍋ function 30
⍒ function 30
? function 31, 43
⍳ function 22, 32
ρ function 32
< function 34
≤ function 34
> function 34
≥ function 34
= function 34
≠ function 34
∧ function 37, 38
∨ function 37, 39
/ function 44, 45
, function 47
⍟ function 51

active workspace 104
adding statements 79
addition 14
alpha keys 2
alphameric keys 2
ALREADY MARKED 107
alternating sum 56
and function ∧ 37, 38
and reduction 48
APL language 1
APL system command keywords 4
arguments 14, 28, 87
arithmetic function keys 4
arithmetic operations 14
assign a value 17
assignment arrow 17
attention key (see ATTN key)
ATTN key 4, 12, 74, 98
average 53

backspace key 4, 8, 9, 10
BASIC/APL switch 4
black background 7
black characters 7
blank character 26
branching 91
brightness control 7
built-in functions 1, 14, 59

calculator arrangement 4
catenate function , 47
ceiling function ⌈ 29
character data 26
CLEAR WS 4
closing ∇ 72
CMD key 4, 9
column 22
command keywords 4
comparison tolerance ⎕CT 109
compress function / 44
conditional branch 92
conform 48
consecutive numbers 31
control panel 1

coordinate, acted on 45, 46
correct a keying error 11
correcting keying errors 8
cursor 5

deal function ? 43
del symbol 71
delete a character 9
deleting statements 82
display appearance 7
display global variable names 106
display registers/normal switch 4
display screen 1, 5
displayed information 5, 6
displaying a statement 83, 84
displaying data 5
displaying from a specific statement 83
displaying the entire function 78
displays user-defined function names 106
division 14
down arrow 7, 12, 72

editing a function 77
editing the function header 84
empty vector 32, 50
entering data 5
entering data during function execution 95
equal to function = 34
error in the function 101
error message 18, 74, 101
escape from input requests 96
EXECUTE key 4, 6, 12
execution mode 71
expand a vector or matrix 45
expand function \ 45
explicit result 88

features 1
files 104
flashing line 5
floor function ⌊ 29
format files 107

forward space key 4, 8, 9, 10
function body 72
function definition 71
function definition mode 71
function editing 77
function execution stops 98
function header 71, 84, 88, 89
functions 1

global names 89
global variable names 106
grade down function ⍒ 30
grade up function ⍋ 30
greater than function > 34
greater than or equal to function ≥ 34

history 5
HOLD key 4, 103
HYP function 71

index generator function ⍳ 31
index numbers 24
index of a value 43
index of function ⍳ 43
index origin ⎕IO 108
index value [I] 45, 46, 48
indexed 30
indexing 24
indicator lights 1
input 5
input line 7
insert a character 10
insert blanks or zeros 45
insert tape cartridge 104
inserting statements 81
instructions 71
interactive functions 96
internal checks 4
internal precision 36

join two items 47

keyboard 1
keying errors 8
keys 1
keywords 4, 104

labels 91
larger of two numbers 41
less than function < 34
less than or equal to function ≤ 34
library 104
local names 89
local variables 91
logarithm function ⍟ 51
logarithm of a number 51
logical functions 37
looping 74, 94
lost lines 7
L32 64 R32 switch 4, 7

matrices 19
matrix 22, 26
maximum function ⌈ 41
maximum reduction 57
minimum function ⌊ 42
minimum reduction 57
minus 15
minus reduction 56
modifying a statement 83
move display lines 7
multiplication 14

negative numbers 15
negative sign 15
not equal to function ≠ 34
numeric keys 4

operating keys 4
or function ∨ 37, 39
or reduction 58
order of execution 18, 19
overstruck character 28

parentheses 19
place a stored workspace into the active workspace 107
plus reduction 53
portable computer 1
power function ⋆ 16
power up sequence 4
printing precision ⎕PP 109

quad input 95
quad quote input 95

random number 31, 43
reduction 53
relational functions 34
reopen function definition 77
replace a character 8
replacing statements 80
request for input data 95
reshape function 22
restart sequence 4
restart switch 4
result variable 72, 77, 88
reverse display switch 7
roll function ? 31
root of a number 17
rounding off numbers 29
row 22

SAFE switch 104
scalar 26
scroll down 72, 83
scroll down key 4, 7
scroll up 72, 83

scroll up key 4, 7
selecting certain elements 44
semicolon 25
set up procedure 1
shape 32
shape function ρ 32
shift key 2
single quote chatacters 26
smaller of two numbers 42
sorting a vector 30
standard typewriter 2
state indicator 102
statement number 72
statements 71
stop control feature 100
stop execution 75
stop feature 75
stop processing 103
storing data 17
strings of numbers 19
subtraction 14
suspended functions 75, 98
switches 1, 4
system command keywords 4, 104
system commands 104
system variables 108

tables 19
tape cartridge 104
testing your function 73
trace feature 75
turn off stop 76
turn off trace 76
typewriter 2

unconditional branch 92
up arrow 7, 12, 72
user-defined functions 14, 71

variable name 17
variables 17
vectors 19, 26

white background 7
white characters 7
write the active workspace to tape 107

READER'S COMMENT FORM

IBM 5100
APL Introduction

SA21-9212-1

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page *Comment*

I would like a reply. ☐

Name ____

Address ____

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SA21-9212-1

Cut Along Line

Fold

Fold

FIRST CLASS
PERMIT NO. 387
ROCHESTER, MINN.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold

Fold

IBM 5100 APL Introduction Printed in U.S.A. SA21-9212-1

IBM

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

IBM 5100
APL Introduction

SA21-9212-1

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page	*Comment*

I would like a reply. ☐

Name ______________________________

Address ______________________________

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SA21-9212-1

Cut Along Line

Fold

Fold

FIRST CLASS
PERMIT NO. 387
ROCHESTER, MINN.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold

Fold

IBM 5100 APL Introduction Printed in U.S.A. SA21-9212-1

IBM®

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

IBM 5100 **SA21-9212-1**
APL Introduction

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page ***Comment***

I would like a reply. ☐

Name ____

Address ____

● Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SA21-9212-1

Cut Along Line

Fold Fold

FIRST CLASS
PERMIT NO. 387
ROCHESTER, MINN.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold Fold

IBM 5100 APL Introduction Printed in U.S.A. SA21-9212-1

IBM

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

IBM 5100 **SA21-9212-1**
APL Introduction

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page *Comment*

I would like a reply. ☐

Name ______________________________

Address ______________________________

● Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SA21-9212-1

Cut Along Line

Fold Fold

FIRST CLASS
PERMIT NO. 387
ROCHESTER, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold Fold

IBM 5100 APL Introduction Printed in U.S.A. SA21-9212-1

IBM®

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

SA21-9212-1

IBM 5100 APL Introduction Printed in U.S.A. SA21-9212-1

IBM

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
Atlanta, Georgia 30301
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)