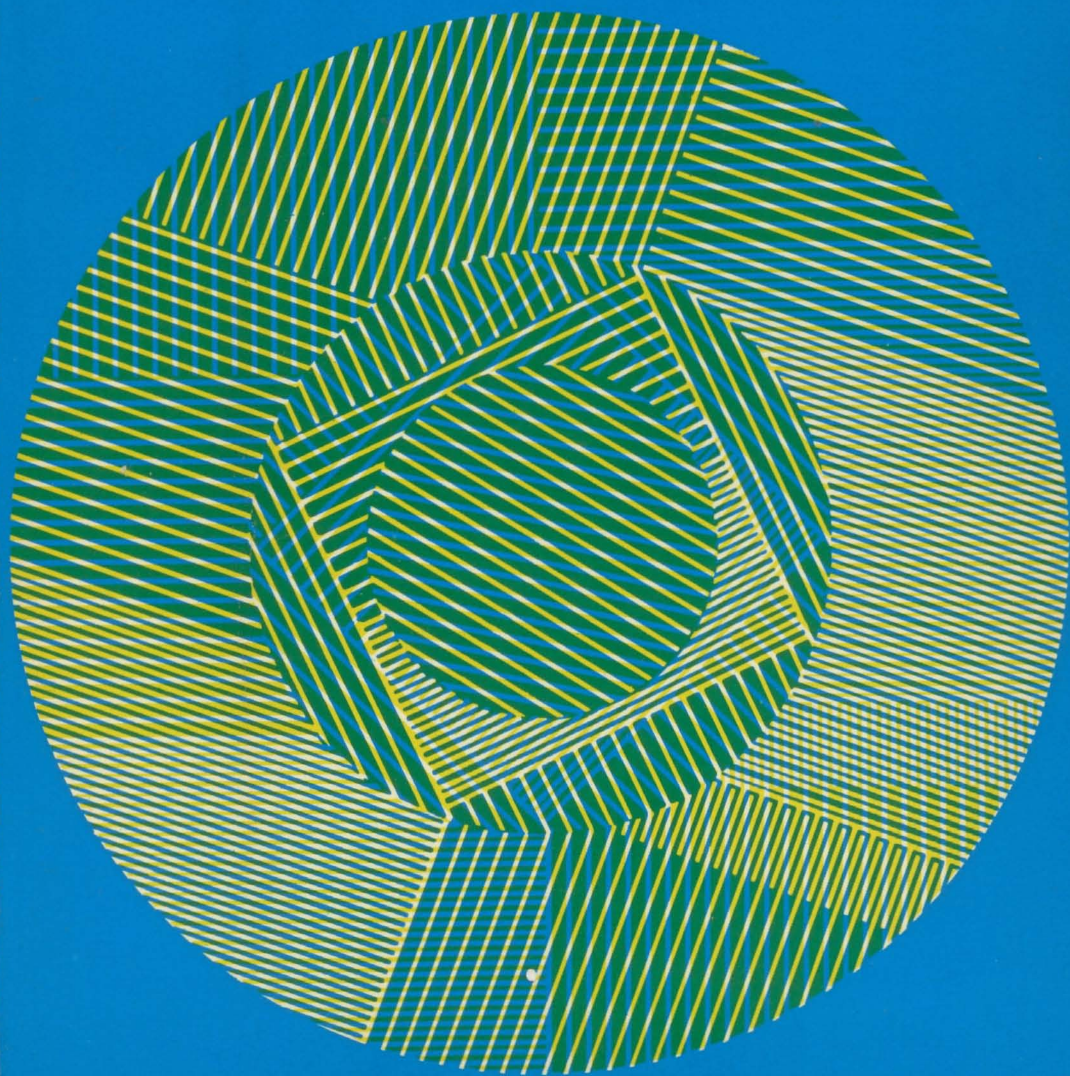
# AN INTRODUCTION TO APL

WILLIAM PRAGER



# An Introduction to APL

This book is part of
the Allyn and Bacon series

PROGRAMMING LANGUAGES OF THE 70's

*Consulting Editor*

PETER WEGNER
Brown University

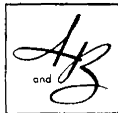
# An Introduction to APL

WILLIAM PRAGER

*University Professor of Engineering and Applied Mathematics*

*Brown University*

Allyn and Bacon, Inc. Boston



© Copyright 1971 by Allyn and Bacon, Inc.
470 Atlantic Avenue, Boston.

All rights reserved. Printed in the United States of America. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any informational storage and retrieval system, without written permission from the copyright owner.

Library of Congress Catalog Card Number: 76-150848

# Contents

Preface vii

List of APL Symbols ix

1 Getting Acquainted With the System 1
- 1.1 Keyboard 1
- 1.2 Signing On 2
- 1.3 Signing Off 4
- 1.4 Correction of Typing Errors 5
- 1.5 Evaluation of Arithmetic Expressions 5

2 Arrays (part 1) 13
- 2.1 Multiple Evaluations of an Expression 13
- 2.2 Forming and Indexing of Arrays 16
  Operator ⍴. Operator ⍳. Indexing.
- 2.3 Arithmetic Operations on Arrays 23
  Monadic Operators. Dyadic Operators. Special Operators.
- 2.4 Boolean Expressions 26

3 Defined Functions (part 1) 29
- 3.1 Defining Simple Functions 30
- 3.2 Branching 33
  Growthtable. Quadratic. Zero.
- 3.3 Local and Global Variables 40
- 3.4 Checking Function Definitions 43

4 System Commands (part 1) 49
- 4.1 Starting a Library 49
- 4.2 Adding to a Library 52

| | | |
|---|---|---|
| **5** | **Additional Operators** | **55** |
| 5.1 | Dyadic Use of ⍟, !, \|, and ○ | 55 |
| 5.2 | Operators ⌊, ⌈, and ? | 56 |
| 5.3 | Operators ~, ∧, ∨, ⍲, and ⍱ | 60 |
| 5.4 | Basic Operators | 61 |
| **6** | **Arrays (part 2)** | **63** |
| 6.1 | Arrays of Higher Dimensionality | 63 |
| 6.2 | Special Operators<br>Reduction. Inner Product. Outer Product.<br>Grade Up and Grade Down. Take and Drop.<br>Decode and Encode. Transpose.<br>Compression and Expansion. | 67 |
| **7** | **Character Manipulation** | **85** |
| 7.1 | Character Data | 85 |
| 7.2 | Operations on Character Data | 89 |
| 7.3 | Sorting, Coding, Decoding, and Translating | 94 |
| **8** | **Defined Functions (part 2)** | **101** |
| 8.1 | Headline Types | 101 |
| 8.2 | Branching | 102 |
| 8.3 | Labels | 104 |
| 8.4 | Checking Function Definitions: Stop Control | 105 |
| 8.5 | Editing Function Definitions<br>Insertion of a Command. Deletion of a Command.<br>Displays. | 108 |
| 8.6 | Error Reports | 111 |
| 8.7 | Recursive Functions | 115 |
| **9** | **System Commands (part 2)** | **117** |
| 9.1 | Digits, Width, and Origin | 117 |
| 9.2 | Inquiry Commands<br>Functions. Variables, Groups, and Group.<br>Library. Workspace Identification.<br>Status Indicator. Ports. System Information. | 119 |
| 9.3 | Library Control Commands<br>Reactivation of Stored Workspace. | 125 |
| 9.4 | Hold Commands | 127 |
| 9.5 | Trouble Reports | 128 |
| | References | 129 |
| | Index | 131 |

# Preface

This book is an introductory text for classroom use or independent study, not a reference manual. It consists of two parts comprising Chapters 1 through 4 and Chapters 5 through 9. After reworking the examples of the first part at the terminal, the reader should be reasonably proficient in applying APL to the solution of a wide variety of *numerical* problems. Study of the second part will extend his knowledge of the language and enable him to handle nonnumerical problems as well. At this stage, the reader should try to improve some of the function definitions in the text. This will be possible because these definitions were written with an emphasis on transparency rather than elegance. Finally, to test his ability to follow complex function definitions, the reader may call for the display of the definition of functions such as PLOT in Workspace PLOTFORMAT of Public Library 1. Further examples of this kind may be found in the literature listed in the References.

There remains the pleasant task of thanking all those who, in various ways, contributed to the publication of this book: Dr. K. E. Iverson of the Thomas J. Watson Research Center of IBM for defining APL, and IBM for implementing the language; Professor Peter Wegner of Brown University for the invitation to write the book for the series; Professors C. Mylonas and R. A. Vitale of Brown University and Dipl. Ing. W. Münzner of the IBM Research Laboratory at Zurich for their numerous helpful comments; Mrs. D. Archambault for the most careful preparation of the manuscript; and the editorial staff of the publisher for their splendid cooperation.

*William Prager*
*Providence, Rhode Island*

# List of APL Symbols

The order of symbols in this list corresponds to their arrangement on the keyboard, starting with the top row of keys and going from left to right. Overstruck and composite symbols are listed last. The letters *d* and *D* stand for basic dyadic operators.

| | |
|---|---|
| − | minus (as part of the specification of a negative number) |
| < | less than |
| ≤ | less than or equal to |
| = | equal to |
| ≥ | greater than or equal to |
| > | greater than |
| ≠ | not equal to |
| ∨ | or |
| ∧ | and |
| - | minus (as arithmetic operator) |
| + | plus |
| ÷ | divided by (dyadic); reciprocal of (monadic) |
| × | times (dyadic); sign of (monadic) |
| ? | random choice (dyadic or monadic) |
| ∊ | membership of |
| ⍴ | structure from (dyadic); size of (monadic) |
| ~ | not |
| ↑ | take |
| ↓ | drop |

| | |
|---|---|
| ⍳ | index of (dyadic); integers up to (monadic) |
| ○ | trigonometric or hyperbolic functions (dyadic); pi times (monadic) |
| ⋆ | to the power (dyadic); *e* to the power (monadic) |
| → | branch to command |
| ← | specified as |
| ⌈ | the greater one of (dyadic); integer just above (monadic) |
| ⌊ | the lesser one of (dyadic); integer just below (monadic) |
| ∇ | del (opening or closing of function definition) |
| ∆ | delta (in trace and stop control) |
| ∘ | small circle (see *comment* and *outer product*) |
| ' | quote (to enclose character data) |
| ⎕ | quad calling for output ( ⎕← ) or input ( ←⎕ ) |
| ( ) | parentheses (grouping of terms in expressions) |
| [ ] | brackets (enclosing indices of arrays) |
| ⊥ | decode |
| ⊤ | encode |
| ∣ | remainder of (dyadic); absolute value of (monadic) |
| ; | semicolon (to separate indices of a matrix or an array of higher dimensionality) |
| , | catenate with (dyadic); unravel (monadic) |
| : | colon (to separate label from command) |
| . | decimal point |
| \ | expand |
| / | compress |
| ⍱ | nor |
| ⍲ | nand |
| ⌽ | rotate (dyadic); reverse (monadic) |
| ⍟ | logarithm for any base (dyadic); base *e* (monadic) |
| ⍉ | transpose (dyadic or monadic) |
| ⍒ | grade down |
| ⍋ | grade up |
| ! | number of combinations (dyadic); factorial (monadic) |
| ⍞ | quote-quad (calling for character input ←⍞ ) |
| ⍝ | comment |
| ⌶ | system information |
| *d*/ | reduction with respect to last index |
| *d*⌿ | reduction with respect to first index |
| *d.D* | inner product |
| ∘.*d* | outer product |

# 1 Getting Acquainted With the System

APL (*A Programming Language*) is best learned at the terminal. After studying a section of this book, the reader should repeat the examples at the terminal before proceeding to the next section. To enable the reader to follow this advice, the present chapter begins with brief sections on the APL keyboard, the procedures for signing on and off, and the correction of typing errors. This is followed by a longer section on the evaluation of elementary APL expressions involving the most frequently used APL operators. Additional operators are discussed in Ch. 5.

## 1.1 KEYBOARD

The APL characters appear on the front rather than the top face of each key. Each key carries two characters. To type the lower one, it is sufficient to hit the key, but if the upper character is to be typed, the shift key must be depressed while the key is hit.

Figure 1 shows the distribution of the APL characters over the keys. Numerals, letters, comma, and period are in the customary positions; however, there are only upper-case letters, which are typed without the use of the shift key. The shift key must be used for almost all special characters; the only exceptions occur at the end of each row

of keys and involve the characters + × (top row), ← (second row), [ ] (third row), and / (bottom row).

```
¨   ¯   <   ≤   =   ≥   >   ≠   ∨   ∧   -   ÷
1   2   3   4   5   6   7   8   9   0   +   ×

?   ⍵   ∊   ⍴   ~   ↑   ↓   ⍳   ○   ⋆   →
Q   W   E   R   T   Y   U   I   O   P   ←

⍺   ⌈   ⌊   _   ∇   ∆   ∘   '   ⎕   (   )
A   S   D   F   G   H   J   K   L   [   ]

⊂   ⊃   ∩   ∪   ⊥   ⊤   |   ;   :   \
Z   X   C   V   B   N   M   ,   .   /
```

*FIGURE 1.1*
*APL Keyboard*

Note that there are two forms of the minus sign: the sign ¯ (upper character on the second key of the top row) is used as part of the specification of a negative number, e.g., ¯4 ; the sign - (upper character on the second to last key of the top row) is used as an arithmetic operator as in 5-2 . The symbol for division is ÷ (upper character on the last key of the top row). Note that the solidus / (lower character on the last key of the bottom row) *cannot* be used in the place of ÷ . The zero (lower character on the third to last key of the top row) and the letter *O* cannot be used interchangeably.

## 1.2 SIGNING ON

Since several types of *terminals* and several types of *connections* between terminal and central computer are available for APL systems, local instructions must be consulted for precise information on sign-on and sign-off procedures. For the purpose of the following qualitative description, it will be assumed that an IBM 2741 Communications Terminal and a DATA SET acoustic coupler are used.

Every authorized APL user has an account number. In the discussion below, this number is assumed to be 9999. Sign-on instructions for the assumed equipment are as follows:

(a) Make sure that the terminal switch marked LCL/COM at the left side of the terminal is in position COM, and the switch ON/OFF of the keyboard is in position ON.

(b) Lift receiver of the phone next to the terminal and dial one of the numbers listed in the local instructions. If you get the busy signal or no answer after a few rings, hang up and try again using another one of these numbers. If you get a high-pitched continuing tone, cradle the receiver on DATA SET with the receiver flex at the far end.

(c) Before trying to sign on, check whether the keyboard is unlocked, i.e., whether depressing the shift key produces a rotation of the typing element.

(d) If the keyboard is unlocked and you are signing on for the first time, type a closing parenthesis followed without blanks by your account number:

```
)9999
```

and depress the RETURN key. The response should be a line giving the number of your port (terminal), the time and date of your sign-on, and your user code, which is distinct from your account number. This is followed by a blank line and by the headline *APL*\360. You are now ready to start using the computer.

(e) If, at the time you are trying to sign on, somebody is inadvertently or fraudulently signed on under your account number, you will receive the message

```
NUMBER IN USE
```

and will not be able to sign on until the other user has signed off. To protect yourself against being locked out in this

fashion, you should use a *password* consisting of up to eight letters or digits, without blanks. Your complete identification for signing on will then be your account number followed without blanks by a colon, followed without blanks by your password. Thus, if you had chosen the password *LOCK* in signing off last (see Sec. 1.3), you would sign on by typing

```
)9999:LOCK
```

It is good practice to change your password at regular intervals. If, however, you forget your password (or your account number), the response to your attempt to sign on will be

```
NUMBER NOT IN SYSTEM
```

Before discussing the evaluation of arithmetic expressions, we will describe the termination of a work session.

## 1.3 SIGNING OFF

It is most important that each work session be terminated by an appropriate sign-off procedure. The instructions for this are as follows:

(a) If you have completed a computation and no longer need the material from the current work session, sign off by typing

```
)OFF
```

and depress the RETURN key. When you next sign on, you must then use the same password as for the current work session. If, however, you wish to change the password, say to *NEWLOCK*, sign off by typing

```
)OFF:NEWLOCK
```

and depress the RETURN key.

(b) In either case, a three-line message will be typed that gives the time and date of the sign-off, your user code, and accounting information. After this is completed, turn off the terminal by putting the ON/OFF switch of the keyboard in the OFF position, and return the receiver to the phone.

(c) If you have to interrupt a computation before it is completed, sign off with *)CONTINUE* or *)CONTINUE:NEWLOCK* depending on whether you wish to retain the old password or change it to *NEWLOCK*. In either case, the material from the current work session will be available when you next sign on, and this availability will be indicated by a message that starts with *SAVED* and gives the time and date of the last sign-off.

## 1.4 CORRECTION OF TYPING ERRORS

While you are typing a command or an expression that you want to have evaluated, you may discover typing errors. If you have not yet *entered* the command or expression by depressing the RETURN key, these errors can be corrected as follows. Backspace to the first erroneous character, and depress the ATTN* key. The response to this will be the symbol ∨ typed under the first erroneous letter, indicating that this and everything to its right have been erased. Now complete the typing of the command or expression and depress the RETURN key.

## 1.5 EVALUATION OF ARITHMETIC EXPRESSIONS

The APL symbols for addition, subtraction, multiplication, division, and exponentiation, respectively, are + - × ÷ ⋆ . The following examples illustrate their use as operation symbols appearing *between two numbers* (*dyadic* use). In each example, the first line, which is indented by six spaces, states the arithmetic problem as typed by

---

*On some terminals, this key is labelled INT.

the user, while the second, not indented, line is the response given by the computer after the RETURN key has been depressed. In the second example, note the raised minus sign used as part of the specification of a negative number. The customary minus sign is reserved for use as an arithmetic operator.

```
      13.5+2487.2
2500.7
      13.5-2487.2
¯2473.7
      13.5×200
2700
      2700÷13.5
200
      12*2
144
      2*0.5
1.414213562
```

The last example is concerned with the evaluation of $2^{0.5}$ or $\sqrt{2}$. Note that up to ten digits of the result will be typed out unless you specify another number, say 7, by typing the command )*DIGITS* 7, and depress the RETURN key. The response to this command, indicating the number of significant digits previously used, is shown below together with a new evaluation of $\sqrt{2}$. Note that the seventh and following digits of the earlier, more accurate value were 3562. The seventh (and last) digit of the new value is 4 as required by proper rounding.

```
      )DIGITS 7
WAS 10
      2*0.5
1.414214
```

The maximum number of digits that you may call for is sixteen. Note that the *DIGITS* command does not limit the number of digits you may use in input, nor does it affect the number of digits carried in the computation; it simply specifies the number of significant digits to which the result of a computation is typed out. It may,

however, affect the form in which this result is presented. The following example illustrates the evaluation of the same product under )*DIGITS* 5 and )*DIGITS* 4.

```
        )DIGITS 5
WAS 10
        100.2×200
20040
        )DIGITS 4
WAS 5
        100.2×200
2.004E4
```

The last result, stated in *exponent notation,* is to be read as 2.004 x $10^4$. Similarly, 2.004E¯3 would be read as 2.004 x $10^{-3}$. Note that the exponent notation for a number consists of a positive or negative number, with magnitude less than 10 but not less than 1, that is followed without blanks by the letter *E* , which is in turn followed without blanks by a positive or negative integer. A number may be *entered* in either ordinary or exponent notation. The examples below indicate how numbers are *returned* by the computer. If *D* is the number of digits currently displayed (10 unless changed by a )*DIGITS* ... command), a number will be returned in exponent notation if its magnitude is equal to or greater than $10^D$ or smaller than $10^{-4}$, unless the number was entered in ordinary notation as an integer of magnitude less than ¯1+2*31 , in which case it will be returned in the form in which it was entered.

```
        )DIGITS 5
WAS 10
        1.1E4
11000
        1.1E5
1.1E5
        0.0008
0.0008
        0.00008
8E¯5
        100056
100056
```

```
        )DIGITS 3
WAS 5
        1.1E2
110
        1.1E3
1.1E3
        0.0008
0.0008
        0.00008
8E¯5
        100056
100056
```

In APL, the value of a composite expression, such as `4-1-3×8÷2` , in which numbers alternate with dyadically used operators, is obtained as follows: imagine the expression rewritten with an opening parenthesis inserted after each operator and the corresponding closing parenthesis added at the end of the expression. Thus, the expression above becomes `4-(1-(3×(8÷2)))` . To evaluate this, we must begin with the innermost parenthesis.* This means that the original, parentheses-free expression is evaluated from right to left. First 8 is divided by 2; next the result 4 of this operation is multiplied by 3; the result of 12 of this operation is then subtracted from 1 furnishing −11, which is finally subtracted from 4 yielding 15 as the value of the expression. It will be useful to familiarize yourself with this evaluation of composite expressions by doing some examples and hand checking your results at the terminal.

To display the manner in which a composite expression is evaluated, insert the symbol pair `⎕←` to the left of each number except the last. Each pair calls for the typing out of the result obtained as the expression is evaluated from right to left up to the considered pair. The following lines show this kind of display for the example considered above.

```
        ⎕←4-⎕←1-⎕←3×⎕←8÷2
4
12
¯11
15
```

*For brevity, an expression within parentheses is referred to as *the parenthesis.*

Parentheses may be used to indicate modes of evaluation that deviate from the rule formulated above. For example, the APL equivalent of

$$\frac{2}{3} - \frac{3}{4} + \frac{4}{5} - \frac{5}{6}$$

is (2÷3)-(3÷4)-(4÷5)-(5÷6). Note that the alternating signs in the ordinary notation become minus signs in APL notation.

In a longer computation, the same intermediate results may have to be used repeatedly. To avoid recomputing them every time they are needed, one may use identifiers as illustrated by the following example. The arrow indicates that the expression on the right should be given the identification on the left of the arrow. Note that the parentheses in line 3 are needed if $A^2 + B^2$ is intended, because without the parentheses, 2+B⋆2 would be taken as the exponent of A .

```
         A←4-12÷6
         B←3×4-2
         (A⋆2)+B⋆2
40
         (A⋆3)-B⋆3
¯208
```

In addition to the dyadic use of the operators + - × ÷ ⋆ , APL also allows their *monadic* use, in which case they are *immediately followed but not preceded by a number or variable.* The monadic use of + and - has the customary algebraic meaning; for example, +15 and -15 are respectively equivalent to 15 and ¯15 . For × ÷ ⋆, the following conventions are adopted. The monadic use of × furnishes the *signum function*; that is, ×C equals 1,0 ⋆C or ¯1 depending on whether $C$ is positive, zero, or negative. The monadic use of ÷ furnishes the *reciprocal*; that is, ÷C equals 1÷C . Finally, the monadic use of ⋆ furnishes the *exponential function*; ⋆C equals $e^C$. Note that as a consequence of the monadic use of operators, two or more operators may immediately follow each other as in ÷⋆C and ⋆-C , which both equal $e^{-C}$, or in -⋆-C , which equals $-e^{-C}$.

Other frequently used monadic operators are ⍟ ! | and ○ . The first two of these are formed by overstriking. For example, to obtain

⍟ , type ○ (or ⋆ ), backspace once, and type ⋆ (or ○ ). The operator ! is formed in a similar manner by overstriking an apostrophe and a period. The meaning of these monadic operators is as follows: ⍟$C$ is the *natural logarithm* of $C$, where $C$ must be positive; !$C$ is the *factorial* of $C$, where $C$ must be a positive integer* or zero, the factorial of zero being defined as 1; |$C$ is the *absolute value* of $C$; and ○$C$ is $C$ times $\pi$. Further APL operators are discussed in Ch. 5.

The rule for the evaluation of a composite expression that was stated earlier in this section must now be extended to include monadic as well as dyadic operators. When an expression is ready for numerical evaluation, the values of all quantities it contains must be known regardless of whether these values appear in explicit numerical form in the expression or are represented by the identifiers of variables. To describe the manner in which a composite expression is evaluated, consider, for instance, the expression of the symbolic form

$(QoQooQ)ooQoQ$

where each $Q$ indicates the numerical value of some *q*uantity and each *o* indicates some *o*perator. Note that the symbolic form of a valid expression may contain the sequence *oo* but not the sequence $QQ$.

The parenthesis in the symbolic expression above contains an expression that must be evaluated by the same rule as the considered expression. For the purposes of the present discussion, the parenthesis may therefore be replaced by the symbol $Q$. The considered symbolic expression is thereby reduced to

$QooQoQ$

This is scanned from right to left. The first operator encountered in this scan cannot be interpreted monadically, because $oQ$ would then be a numerical value that could be represented by $Q$, so that the expression would end with the invalid pair $QQ$. It follows that *a*

---

*When $C$ is not an integer, !$C$ is the value of the gamma function for $C+1$.

*single operator between two quantities must always be interpreted dyadically.* The symbols *QoQ* at the right of the considered expression thus represent a numerical value, and may therefore be replaced by the symbol *Q*. The expression is thereby reduced to the form

*QooQ*

Here *the right member of the sequence oo can only be interpreted monadically,* and *oQ* may be replaced by *Q*. It follows that *the left member of the sequence oo must be interpreted dyadically.*

Note that in a sequence such as *QooooQ* only the leftmost operator is dyadic; all other operators are monadic. For example, 3+÷!8 is the sum of 3 and the reciprocal of the factorial of 8. The operators ÷ and ! are used monadically, but the operator + is used dyadically.

Note also that in the expression of the symbolic form

*Qoo*(*QoQooQ*)

the parentheses are unnecessary.

# 2 | Arrays (part 1)

Much of the remarkable power of APL stems from the provisions made for the manipulation of arrays of numbers or characters. The first section of this chapter illustrates this remark by examples concerning the evaluation of an algebraic expression for several values of the independent variable. The subsequent sections treat the forming and indexing of numerical vectors and matrices, and arithmetic and Boolean operations on arrays of this kind. Additional material on arrays is found in Ch. 6 (arrays of numbers) and 7 (arrays of characters).

## 2.1 MULTIPLE EVALUATIONS OF AN EXPRESSION

As an introduction to the discussion of arrays, we shall consider an important application of the array concept. It often becomes necessary to evaluate an expression containing a parameter for a set of values of this parameter. In APL, this is facilitated by the possibility of regarding these values of the parameter as the elements of a *vector.* For example, for a given interest rate, say 5%, we may wish to compute the factor by which the value of a bank deposit increases in one year if interest is compounded annually, semiannually, quarterly, monthly, or daily, i.e., $N = 1, 2, 4, 12$, or 365 times a year. The APL formulation of the problem and the result of the computation are shown in Example 1. The first line defines $N$ as the vector with

*EXAMPLE 1*

```
      N←1 2 4 12 365
      (1+0.05÷N)*N
1.05  1.050625  1.050945337  1.051161898  1.051267496
```

*EXAMPLE 2*

```
      X←0.05
      N←0 1 2 3 4
      T←(X*N)÷!N
      T
1  0.05  0.00125  2.083333333E¯5  2.604166667E¯7
```

the elements 1, 2, 4, 12, 365. Note that in APL notation the numbers are separated by blanks rather than commas. The next line defines the desired factor in terms of $N$, and the last line shows the resulting values that this factor assumes for the five values of $N$.

As this example illustrates, an expression containing a *single* vector as parameter represents a vector whose elements are the values that the expression assumes for the successive elements of the parameter vector.

As another example, consider the infinite series $e^X = 1 + X + X^2/2! + \ldots$ . The evaluation of its first five terms for $X = 0.05$ is shown in Example 2.

Note that in the third line the expression `(X*N)÷!N` has been assigned the identifier `T` for easy reference in the next example. As a consequence of this assignment, which implies later use of the numerical values of the five terms of the series, these values will not be automatically displayed but must be called for by the command `T` .

The following APL features facilitate the evaluation of partial sums of series. If `T` is a vector of `K` elements and `J` is an integer not exceeding `K` , the vector consisting of the first `J` elements of `T` is the value of the expression `J↑T` . Furthermore, the sum of the elements of `T` is the value of the expression `+/T` . While the material from the preceding example is still available, the third, fourth, and fifth partial sums of the series for $e^{0.05}$ can be obtained as shown below.

```
      +/3↑T
1.05125
      +/4↑T
1.051270833
      +/5↑T
1.051271094
```

The fact that the last two partial sums agree in the first six digits suggests that we have already obtained a good approximation to $e^{0.05}$. Note that $e^{0.05}$ is the factor by which a bank deposit would

grow in a year if 5% annual interest were compounded continuously rather than once a month or quarter.

APL also provides a simple means of alternately adding and subtracting the elements of a vector. If *T* is a vector, the expression -/*T* is evaluated by changing the signs of all even-numbered elements and then summing the elements of the vector obtained in this manner. Accordingly, the lines below show the evaluation of the analogous partial sums of the series for $e^{-0.05}$.

```
      -/3↑T
0.95125
      -/4↑T
0.9512291667
      -/5↑T
0.9512294271
```

A final example shows the approximate evaluation of e from the fifth partial sum of the series

$$e^{-1} = 2\left(\frac{1}{3!} + \frac{2}{5!} + \cdots + \frac{n}{(1+2n)!} + \cdots\right).$$

```
      ÷2×+/(⍳5)÷!1+2×⍳5
2.718281843
      *1
2.718281828
```

We have chosen a few operations on vectors to illustrate the power of the array concept. These and other operations on vectors and matrices will be discussed more comprehensively in the remaining sections of this chapter.

## 2.2 FORMING AND INDEXING OF ARRAYS

The position of an element in an array is specified by an index or several indices. For example, in a plane rectangular array, the position of an element may be specified by stating that it is located in

the third row from the top and the fifth column from the left. If the considered array is denoted by $M$ , the element in question would be denoted by $M$[3;5]. Note that in APL the indices of an element are separated by semicolons rather than commas and are enclosed in brackets rather than parentheses.

The number of indices needed to specify the position of an element in an array is called the *dimensionality* of this array.* Arrays of the dimensionalities 1 and 2 are known as *vectors* and *matrices.* Although arrays of higher dimensionality may be used in APL, the following discussion is restricted to vectors and matrices.

When dealing with arrays, we may occasionally find it convenient to regard a single number (scalar) as an array of the dimensionality 0. This extension of the dimensionality concept is consistent because no indices are required to specify the position of the single element in this kind of array.

### Operator ρ

A vector $V$ of, say, six elements will be said to have the *size* 6, and a matrix $M$ of, say, 3 rows and 8 columns will be said to have the *size* specified by the vector 3 8 .† In APL, *monadic* use of the operator ρ (rho), called the *structuring operator,* provides the size of an array.

Thus, for the vector $V$ and the matrix $M$ above, ρ$V$ and ρ$M$ respectively have the values 6 and 3 8 . Accordingly, ρρ$V$ and ρρ$M$ have the values 1 and 2, which give the dimensionalities of $V$ and $M$.

---

*In APL literature, the term *rank* is frequently used instead of dimensionality, but this will not be used here to avoid confusion with the customary use of the term rank in matrix algebra.

†In APL literature, the size of a vector is frequently called its *length*, but this term will not be used here to avoid confusion with the customary meaning of the length of a vector. Similarly, the term *dimension*, which is often used instead of size, will not be used here to avoid confusion with the concept of dimensionality introduced above.

The *structuring operator* ρ is also used dyadically. For example, to specify a matrix $M$ of the size 2 3 , we start with the vector consisting of the elements of the first row of $M$ , followed by the elements of the second row and those of the third row—say, the vector 2 3 5 7 11 13. We then structure this vector into the desired matrix $M$ by the command ⎕←M←2 3ρ2 3 5 7 11 13, where the symbol pair ⎕← preceding $M$ calls for the typing of the matrix identified by $M$ . Note that ρM and ρρM furnish size and dimensionality of $M$ . Note also that the matrix $M$ may be reconverted to the vector $V$ from which it was formed by the command V←,M. In APL manuals, the term "ravel" is used for the operation expressed by the monadically used comma; the term "unravel" might be more appropriate.

```
          ⎕←M←2 3ρ2 3 5 7 11 13

      2   3   5
      7  11  13

          ρM
  2  3
          ρρM
  2
          ⎕←V←,M
  2  3  5  7  11  13
```

If this vector $V$ is used to structure a matrix $N$ that requires fewer elements than are in $V$ , only the leading elements are used. Similarly, if $V$ has fewer elements than are needed for $N$ , the elements of $V$ are used repeatedly. The following examples illustrate these structuring operations, and similar structuring of other vectors from $V$.

```
        2 2ρV

  2  3
  5  7
```

```
          3 5⍴V
  2   3   5   7  11
 13   2   3   5   7
 11  13   2   3   5

          4⍴V
2  3  5  7
          9⍴V
2  3  5  7  11  13  2  3  5
```

**Operator ⍳**

Vectors such as `1 1.25 1.50 1.75 2` or `8 6 4 2`, in which the difference between successive elements has a constant value, frequently occur in applications. In APL, vectors of this kind are readily specified by the use of the *index generator* ⍳ (iota). If *N* is a non-negative number, ⍳*N* denotes the vector consisting of the integers from 1 through *N*. Some uses of the index generator are shown below.

```
          ⍳5
1  2  3  4  5
          .75+.25×⍳5
1  1.25  1.5  1.75  2
          .25×3+⍳5
1  1.25  1.5  1.75  2
          -⍳4
¯1  ¯2  ¯3  ¯4
          10-2×⍳4
8  6  4  2
          ⍳0

          V←⍳1
          ⍴V
1
```

Note that ⍳0 is the *empty vector*, which is represented by a blank. An important (but by no means the only) use of this seemingly useless vector is discussed under *Indexing*. Note also that ⍳1 , which consists of the single element 1, is a vector of size 1.

The index generator ⍳ is also used dyadically. If *V* is a row vector, and *E* is the value of *one* of its elements, the expression *V*⍳*E* gives the position of this element in *V* . If the vector *V* contains several elements of value *E* , the position of the first of these will be returned. If, on the other hand, *V* contains no element of value *E* , the "position" returned will be 1+⍴*V* . If *A* is an array, *V*⍳*A* is the array obtained by performing the operation *V*⍳ on each element of *A* . The following examples illustrate these rules.

```
      V←6 3 5 5
      V⍳3
2
      V⍳5
3
      V⍳2
5
      A←1 5 6 2 3 6
      V⍳A
5  3  1  5  2  1
      M←2 3⍴A
      V⍳M

 5  3  1
 5  2  1
```

## Indexing

When a vector *V* or a matrix *M* has been defined, individual elements or groups of elements may be selected by indexing. *V*[5] denotes the fifth element of the vector *V* , provided that this vector has at least five elements. Similarly *V*[2 4 5 6] denotes the vector consisting of the second, fourth, fifth, and sixth elements of *V* . Note that the command *V*[12] in the example below produces the message *INDEX ERROR*, beneath which the unacceptable command is repeated with a caret suggesting what is wrong (in this case, that the vector *V* has less than twelve elements). *M*[2;4] is the element at the intersection of row 2 and column 4 of *M* . Similarly, *M*[1 4;2 5] is a matrix consisting of the elements at the intersections of rows 1 and 4 with columns 2 and 5. The examples opposite illustrate these and other indexing operations.

```
      ⎕←V←1+2×⍳9
3  5  7  9  11  13  15  17  19

      ⎕←M←4 5⍴V

  3  5  7  9 11
 13 15 17 19  3
  5  7  9 11 13
 15 17 19  3  5

      V[6]
13
      V[2 4 5 6]
5  9  11  13
      V[12]
INDEX ERROR
      V[12]
      ^
      M[2;4]
19
      M[1 4;2 5]

  5 11
 17  5
      M[;2 5]

  5 11
 15  3
  7 13
 17  5
      M[1 4;]

  3  5  7  9 11
 15 17 19  3  5
```

Note that `M[;2 5]` is the matrix consisting of the second and fifth columns, and `M[1 4;]` is the matrix consisting of the first and fourth rows of `M` .

Indexing may also be used to change individual elements or groups of elements. For the vector `V` defined in the first line of the example below, the third line redefines the fourth element as 0. The fourth line calls for the typing of the new vector. Note that the substitutions

called for in the sixth line are not made in the original vector *V* but in the vector as changed by the preceding substitution. Similar changes may be made in a matrix.

```
      V←1+⍳9
      M←3 4⍴V
      V[4]←0
      V
2 3 4 0 6 7 8 9 10
      V[3 7]←1
      V
2 3 1 0 6 7 1 9 10
      M[1;2]←5
      M
  2  5  4  5
  6  7  8  9
 10  2  3  4

      M[2 3;1 4]←20
      M
  2  5  4  5
 20  7  8 20
 20  2  3 20
```

It is often desirable to record successive intermediate results of a computation as they are obtained and before they are transformed by further computational steps. A convenient way of doing this is to gather the intermediate results into a vector *R* . In discussing this task, we shall assume that each intermediate result is obtained, by the same computational steps, from the preceding one, and is therefore assigned the same identifier, say *A* , as it is obtained. Having initially defined `I←0` , we should like to increase *I* by 1 every time a new intermediate result *A* has been found, and then execute the command `R[I]←A` . This substitution of an element into the vector *R* is only possible, however, if we have defined an initial vector *R* . If it is known, for example, that ten intermediate results will be computed, this can be done by the initial command `R←10⍴0` , which forms a vector consisting of ten zeros.

However, the number of intermediate results $A$ may not be known beforehand. These results may, for instance, be successive partial sums of a series, and we may wish to continue the computation until two successive partial sums agree to within a specified number of digits. In this case, the *catenation operator*, which is the comma used dyadically, proves valuable. If $V$ and $W$ are vectors, $V,W$ denotes the vector consisting of the components of $V$ followed by those of $W$. Using catenation, we may build up the vector $R$ by the repeated execution of the command R←R,A, provided we have initially specified $R$ as the empty vector by R←⍳0 . Note that the comma used as a *monadic* operator converts a scalar into a one-element vector as shown below.

```
        A←5
        ⍴A

        A←,5
        ⍴A
1
```

## 2.3 ARITHMETIC OPERATIONS ON ARRAYS

### Monadic Operators

If $m$ denotes a monadically used operator and $A$ is an array, $mA$ is the array obtained by performing the operation $m$ on each element of $A$:

```
        A←¯2 0 3 4
        -A
2  0  ¯3  ¯4
        ×A
¯1  0  1  1
        |A
2  0  3  4
        A←2 2⍴⍳4
        ÷A

      1               0.5
      0.3333333333    0.25
```

## Dyadic Operators

If *d* denotes a dyadically used operator and *A* and *B* are *arrays of the same size,* *A d B* denotes the array obtained by performing the operation *d* on each pair of corresponding elements of *A* and *B*. When *A* is a scalar, *A d B* is the array obtained by performing the operation *d* on *A* and each element of *B*. A similar statement applies when *A* is an array and *B* a scalar. The following examples illustrate these conventions.

```
      A←1 3 5 7
      B←4 3 2 1
      A-B
¯3  0  3  6
      A×B
4  9  10  7
      10×A
10  30  50  70
      A←2 2ρA
      B←2 2ρB
      A÷B

   0.25          1
   2.5           7

      B÷5

   0.8           0.6
   0.4           0.2
```

Note that the command *S*+*A*-*A* , where *S* is a scalar and *A* an array, does not yield the scalar *S* but an array of the size ρ*A* , each element of which has the value *S* .

## Special Operators

If *d* denotes a dyadically used operator and *A* is a vector, *d*/*A* is the scalar obtained by inserting the operator *d* between each pair of successive elements of *A* and performing the operations specified in this manner. Thus, *d*/*A*←*A*[1] *dA*[2]*d*...*dA*[ρ*A*] with evaluation

from right to left. It follows that +/*A* is the sum of the elements of *A*, and ×/*A* is their product. Note that −/*A* is the alternating sum (see the end of Sec. 2.1), and ÷/*A* is the quotient of the product of all odd-numbered elements of *A* by the product of the even-numbered elements. If *A* is a matrix, *d*/*A* is a vector with elements obtained by performing the operation *d*/ on each of the rows of *A*. Similarly, *d*⌿*A*, where ⌿ is typed by overstriking / and −, is the vector consisting of the elements obtained by performing the operation *d*/ on each of the columns of *A*.

```
      A←⍳10
      +/A
55
      -/A
¯5
      ×/A
3628800
      !10
3.6288E6
      ÷/A
0.24609
      (1×3×5×7×9)÷(2×4×6×8×10)
0.24609
      A←3 3⍴⍳9
      +/A
6  15  24
      +⌿A
12  15  18
```

Note that the polynominal $c_1 + c_2x + c_3x^2 + \cdots + c_{n+1}x^n$ may be evaluated as +/*C*×*X*⋆¯1+⍳⍴*C*, where *C* is the vector with the elements $c_1, c_2, \ldots, c_{n+1}$ and $n > 0$. If the degenerate case $n = 0$ is to be included, the expression ⍳⍴*C* in the command above must be replaced by ⍳⍴,*C* because ⍴*C* is empty if *C* is a scalar but has the value 1 if *C* is a one-element vector.

If *A* and *B* are *vectors of the same size*, their *scalar product S* is defined as the sum of the products of corresponding elements of *A* and *B*. Thus, *S* is obtained by the command *S*←+/*A*×*B*. An alternative command is *S*←*A*+.×*B*. If *M* and *N* are matrices, and *M has*

*as many columns as N has rows*, their product P is defined as the matrix with the typical element `P[I;J]←+/M[I;]×N[;J]` . Note that for given values of I and J , the expressions `M[I;]` and `N[;J]` are vectors of the same size, and `P[I;J]` is obtained by multiplying corresponding elements of the two vectors and adding these products. In APL, the product P of the matrices M and N is obtained by the command `P←M+.×N`.

Since it is sometimes necessary to take the *transpose* of a matrix (i.e., interchange its rows and columns) before it can be used as a factor in a matrix product, APL provides the monadic operator ⍉ (typed by overstriking ∘ and \ ) to transpose a matrix; `⍉M` is the transpose of M . The following examples illustrate the operations discussed above.

```
          A←1 3 5
          B←¯2 1 3
          +/A×B
    16
          A+.×B
    16

          M←2 3ρι6
          N←3 2ρ¯3+ι6
          M+.×N

       4  10
       4  19

          (⍉N)+.×⍉M

       4   4
      10  19
```

## 2.4 BOOLEAN EXPRESSIONS

If *r* is a *relational symbol* such as ≤ and A and B are scalars, the assertion ArB is attributed the value 1 or 0 according to whether it is true or false. Typical relational symbols are < ≤ = ≥ > ≠ and ∊ .

The last of these asserts membership in an array. For example, `5∊1 4 8` asserts that 5 is an element of the vector `1 4 8` ; since this assertion is false, the expression `5∊1 4 8` has the value 0. For other relational symbols see Sec. 5.3.

If $A$ and $B$ are vectors of the same size, and $r$ is a relational symbol other than ∊ , then $A\,r\,B$ is a vector of the same size as $A$ and $B$ ; its $I$th element is 1 or 0 according to whether the assertion $A[I]\,r\,B[I]$ is true or false. $A∊B$ , where the vectors $A$ and $B$ need not have the same size, is a vector of the size of $A$ ; its $I$th element is 1 or 0 according to whether the assertion $A[I]∊B$ is true or false.

Note that the comparison demanded by a relational symbol is relative in the following sense: if two numbers agree to within a critical amount called *fuzz* (approximately $10^{-13}$), they will be regarded as equal for the purpose of this comparison.

The following examples illustrate the usefulness of expressions of this kind, which are called *Boolean expressions.* The expression that is evaluated first has the value 1 for $-1 \leqslant X \leqslant 1$ and the value 0 elsewhere; the second expression has the value 1 for $-1 \leqslant X \leqslant 2$ and vanishes elsewhere; and the last expression equals $-1$ for $X < -1$, $X$ for $-1 \leqslant X \leqslant 1$, and 1 for $X > 1$.

```
        ⎕←X←0.5×¯4+⍳9
¯1.5   ¯1   ¯0.5   0   0.5   1   1.5   2   2.5
        1-(|X)>1
0  1  1  1  1  1  0  0  0
        1-(X<¯1)+X>2
0  1  1  1  1  1  1  1  0
        X+((¯1-X)×(¯1>X))+(1-X)×(X>1)
¯1  ¯1  ¯0.5  0  0.5  1  1  1  1
```

The last expression could also be obtained as `¯1⌈1⌊X` (see Sec. 5.2 for the operators ⌊ and ⌈ ).

If $B$ is a Boolean vector, i.e., a vector consisting exclusively of elements with values 0 or 1, and if $C$ is a vector of the same size as $B$ , then $B/C$ is the vector consisting of those elements of $C$ that correspond to elements of value 1 in $B$ . If all elements of $B$ are 0, then

*B*/*C* is an empty vector. If *S* is a scalar, 1/*S* is a vector whose only element has the value *S* , and 0/*S* is empty.

```
        B←1 0 0 1 1
        C←⍳5
        B/C
1  4  5
        B←5⍴0
        B/C

        ⎕←V←1/S←4
4
        ⍴V
1
        ⎕←V←0/S

        ⍴V
0
        X←2
        Y←1
        ((X<Y),(X=Y),X>Y)/3 7 8
8
```

# 3 Defined Functions (part 1)

In addition to $e^X$ and ln $X$, which have already been discussed in Sec. 1.5, the trigonometric and hyperbolic functions and also the inverse trigonometric and hyperbolic functions are directly available in APL. The APL expressions for sin $X$, cos $X$, and tan $X$, respectively, are `1○X` , `2○X` , `3○X` , and cot $X$ is obtained as `÷3○X` . The argument $X$ of the trigonometric functions must be given in radians. If $D$ denotes an angle measured in degrees, sin $D$, for example, is found as `1○○D÷180`. For the inverse trigonometric functions arc sin $X$, arc cos $X$, and arc tan $X$, the notations `¯1○X` , `¯2○X`, `¯3○X` are used in APL. For the hyperbolic functions sinh $X$, cosh $X$, tanh $X$, and their inverse functions, see Sec. 5.1.

If, in the course of a computation, some other function has to be evaluated repeatedly for different values of the argument, it would be inconvenient to write the commands by which it is computed every time the function is needed. APL provides the possibility of defining a function once and for all, so that it can be used as readily in a work session as the functions discussed above.

As has been seen in preceding sections, rather involved computations may be specified by a single APL command. The first section of this chapter deals with function definitions that consist of a single command. A more involved type of function definition contains several commands that have to be executed in an order that depends on the

values of data or intermediate results. A few examples of this type are discussed in Sec. 3.2. The important distinction between local and global variables is the subject of Sec. 3.3, and some ways of checking function definitions are presented in Sec. 3.4. Additional material on defined functions is contained in Ch. 8.

## 3.1 DEFINING SIMPLE FUNCTIONS

Let us denote the growth factor in the second line of the first example of Sec. 2.1 by *G* . The commands

```
D←1354.20

N←12

G←(1+0.05÷N)*N

D×G
```

will compute the final value of a deposit of $1354.20 at the end of a year during which it has been compounded monthly at 5%. If we then want to compute the final value of some other deposit—for example, $2485.00 at 5% compounded semiannually for a year, we may enter the commands

```
D←2485.00

N←2

G←(1+0.05÷N)*N

D×G
```

While *D* and *N* must obviously be redefined, it is irksome that the command for *G* must be retyped although it has not changed. To avoid this, we may define a function called *GROWTH* of the variable *N* once and for all as described below.

Whereas each of the commands above is executed as soon as it is entered by pressing the RETURN key, the commands in the definition of a function cannot be so executed because the value of the

argument is not known at the time the function is defined. To indicate that we wish to change from the *execution mode* to the *definition mode,* we begin the *headline* of a function by typing the character ∇ (called "del"). This is followed by the identifier of the value of the function (here *G* ), a leftward arrow, the name of the function (here *GROWTH* ), which must differ from the identifier of its value, a blank, and the identifier of the independent variable (here *N* ). Thus, the headline of our function definition is

```
∇G←GROWTH N
```

When this is entered, the response is [1] typed at the beginning of the next line to indicate that this is to be the first line of the function definition. We now continue this line by typing G←(1+0.05÷N)*N∇ , where the final del indicates the desire to close the definition mode. As long as the material from the current work session is available, we may call for this definition to be displayed by entering the command ∇GROWTH[⎕]∇ as shown below together with the statements and solutions of the problems considered earlier. Note that the final ∇ appears below the last line rather than at its end.

```
          ∇GROWTH[⎕]∇

       ∇ G←GROWTH N
[1]      G←(1+0.05÷N)*N
       ∇
          1354.20×GROWTH 12
1423.48
          2485.00×GROWTH 2
2610.8
```

In the definition of the function *GROWTH* above, the interest rate was set at the fixed value of 0.05. If we want to make the interest rate *R* another independent variable of the function *GROWTH* , we may use *R* as a *left argument* and *N* as a *right argument* as shown below, where the function definition is followed by an example in which a deposit of $3000 is left for twelve years at 6.5% compounded quarterly. Note that the first attempt to define the new

function *GROWTH* was not successful; the old function *GROWTH* had to be erased before a new function with the same name could be defined. If it is desirable to retain the old function, the new one must be given a different name, for instance *GROWTH*1.

```
        ∇G←R GROWTH N
DEFN ERROR
        ∇G←R GROWTH N
                   ∧
        )ERASE GROWTH
        ∇G←R GROWTH N
[1]     G←(1+R÷N)*N ∇
        3000×(0.065 GROWTH 4)*12
6503.51
```

An alternative way of defining a function that furnishes the growth factor *G* as a function of *R* and *N* is to use the vector *V*←*R*,*N* as right argument:

```
        )ERASE GROWTH
        ∇G←GROWTH V
[1]     G←(1+V[1]÷V[2])*V[2] ∇
        3000×(GROWTH 0.065 4)*12
6503.51
```

Note that if there is only one argument, this must be the right argument.

Suppose that in command [1] of the last function definition, we forget to depress the SHIFT key while typing the operator ÷ between *V*[1] and *V*[2] and hence obtain a × sign instead of the ÷ sign. If the error is detected before the command is entered, it may be corrected in the manner described in Sec. 1.4. If, on the other hand, we have entered the erroneous line, then we have left the definition mode on account of the terminal "del" of this line. To correct the line, we must re-enter the definition mode, correct the line, and leave the definition mode. This can be done by typing

```
    ∇GROWTH[1]G←(1+V[1]÷V[2])*V[2]∇
```

While there are many other ways of correcting typing errors, it is strongly recommended that, in the beginning, you restrict yourself to those discussed above to avoid introducing further errors.

## 3.2 BRANCHING

The function definitions of the preceding section are particularly simple, because each of them requires only a single command. A definition consisting of a set of commands that are always executed in the same order is not much more complex. Frequently, however, a function definition involves a set of commands that have to be executed once or repeatedly in an order that depends on the values of data or intermediate results. We shall now present some examples to illustrate ways of specifying a more or less complex path through a set of commands.

### Growthtable

This function has as its left argument a vector *P* consisting of several annual interest rates (stated in percent), and as its right argument a vector *N* consisting of several annual frequencies of interest compounding. The function furnishes a table of growth factors in which each row starts with the relevant value of the interest rate and then gives the factors by which a deposit left at this rate will increase in a year when interest is compounded with the various frequencies that make up the vector *N* .

The desired table is typed out as a matrix *T* of *J* rows and *K* columns, where *J* is the number of elements in *P* (command [1]) and *K* is the number of elements in *N* increased by 1 to accommodate the leading column of values of the interest rate (command [2]). The *I*th row of this matrix is defined by command [6], which states that this row should consist of the value `P[I]` catenated by the growth factors `(1+0.01×P[I]÷N)*N` corresponding to `P[I]` and the elements of *N* . This command, however, will only be acceptable if the size of the matrix *T* has been specified in advance. This is

```
      ∇GROWTHTABLE[⎕]∇

    ∇ T←P GROWTHTABLE N
[1]   J←ρP
[2]   K←1+ρN
[3]   T←(J,K)ρ0
[4]   I←0
[5]   →(J<I←I+1)/0
[6]   T[I;]←P[I],(1+0.01×P[I]÷N)*N
[7]   →5
    ∇

      (2.5+0.5×⍳7) GROWTHTABLE (1 2 4 12)

     3          1.03        1.0302      1.0303      1.0304
     3.5        1.035       1.0353      1.0355      1.0356
     4          1.04        1.0404      1.0406      1.0407
     4.5        1.045       1.0455      1.0458      1.0459
     5          1.05        1.0506      1.0509      1.0512
     5.5        1.055       1.0558      1.0561      1.0564
     6          1.06        1.0609      1.0614      1.0617
```

done by command [3], which provisionally defines $T$ as a matrix of $J$ rows and $K$ columns, all of whose elements are 0. (Any other number or array of numbers would serve as well, because these elements are later replaced by those specified by command [6]). The commands [4], [5], and [7] see to it that the variable $I$ in command [6] successively assumes the values $1, 2, \ldots, J$. Initially, $I$ is set to 0 by command [4]. Command [5] is a *switch* that either calls for the computation of another row of the matrix $T$ or terminates the computation. Whereas commands normally are executed in the order indicated by their numbers, the computation will follow one of two or more *branches* every time a switch is reached. Particular care must be taken in writing the switch command to make sure that it accomplishes the desired branching.

In an APL function definition, a switch begins with a rightward arrow. If this arrow is not followed by an expression, the switch calls for termination of the computation. If the arrow is followed by an expression, the value $V$ of this expression, or if the expression represents a vector, the value $V$ of its first element, indicates the manner in which the computation is to proceed. $V$ must be a positive integer, zero, or empty. If $V$ is the number of one of the commands of the function definition, this command is executed next. If $V$ is zero or greater than the number of the last command of the function definition, the computation is terminated. Finally, if $V$ is empty, e.g., if $V \leftarrow 0/S$, where $S$ is a scalar, the computation proceeds to the next command or terminates if the switch is the last command of the function definition.

After the $I$th row of the matrix $T$ has been computed according to command [6], command [7] thus effects an unconditional switch to command [5]. As the parenthesis in the latter command is evaluated from right to left, the current value of $I$ is increased by 1 and the new value of $I$, which is the number of the next row to be computed, is compared to the fixed value of $J$. If the $J$th, i.e., last, row of the matrix $T$ has just been computed, this new value of $I$ is $J+1$, and since the assertion $J<I$ in command [6] is true, the parenthesis has the value 1. Since 1/0 has the value 0, the computation is terminated. If, on the other hand, the current value of $I$ is

less than *J* , i.e., if at least one more row of *T* has to be computed, the increase of *I* by 1 in command [6] yields a new value satisfying *J*≥*I* . The assertion *J*<*I* is therefore false and has the value 0. Now, 0/0 is empty; accordingly the computation proceeds to the next command (command [6] ).

### Quadratic

This function furnishes the roots of the quadratic equation $x^2 + c_1 x + c_2 = 0$. It does not use a left argument; its right argument is the vector *C* , the two elements of which are equal to the coefficients $c_1, c_2$ of the quadratic equation. The desired roots are found in terms of a four-element vector *R*, with *R*[1], *R*[3] being the real and *R*[2], *R*[4] the imaginary parts of the roots.

```
      ∇ R←QUADRATIC C
[1]     R←4ρ0
[2]     D←(0.25×C[1]*2)-C[2]
[3]     →(D<0)/7
[4]     R[1]←-(0.5×C[1])-D*÷2
[5]     R[3]←-(0.5×C[1])+D*÷2
[6]     →0,R[2]←R[4]←0
[7]     R[1]←R[3]←-0.5×C[1]
[8]     R[4]←-R[2]←(|D)*÷2
      ∇

        QUADRATIC 1 ¯6
2  0  ¯3  0

        QUADRATIC ¯4 13
2  3  2  ¯3

        QUADRATIC ((¯10 8)÷3)
2  0  1.333333333  0
```

Command [1] of *QUADRATIC* provisionally specifies a vector *R* with four elements of value 0. Command [2] defines the discriminant *D* of the quadratic equation, which is used in command [3] to effect appropriate branching. If *D*≥0, the expression that follows the

rightward arrow in command [3] is equivalent to 0/7, which is empty. In this case, the next command to be executed is [4]. On the other hand, if $D<0$, command [3] causes branching to [7]. Each of the commands [4] and [7] begins a series of commands that defines the elements of *R* for nonnegative discriminant (real roots) or negative discriminant (conjugate complex roots).

Note that the expression `0,R[2]` in command [6] represents a two-element vector because the comma indicates catenation. Since the convention has been adopted that a rightward arrow in front of a vector disregards all elements of this vector but the first, the computation is terminated after the part of command [6] that is to the right of the comma has been executed.

The first two uses of *QUADRATIC* shown above concern the equations

$$x^2 + x - 6 = 0 \quad \text{and} \quad x^2 - 4x + 13 = 0,$$

the roots of which are found to be

$$x_1 = 2, x_2 = -3 \quad \text{and} \quad x_1 = 2 + 3i, x_2 = 2 - 3i.$$

The third example concerns the equation $3x^2 - 10x + 8 = 0$, which must first be brought into the standard form for *QUADRATIC* by dividing the coefficients `¯10` and `8` by the coefficient `3` of the quadratic term. Note that the argument of *QUADRATIC* in this example could also be typed as `(¯10 8)÷3` or even `¯10 8÷3`. Note also that, in many cases, a quadratic equation of the form $a_1x^2 + a_2x + a_3 = 0$ has to be solved when the coefficients $a_1, a_2, a_3$, instead of being given outright, are intermediate results of the computation in the course of which the equation must be solved. It is then not known in advance that $a_1 \neq 0$, and *QUADRATIC* must be replaced by a more involved function that takes account of all special cases.

### Zero

This function yields an approximation to a real zero of the separately defined function *FUNCTN* when two values of the independent

variable are given for which this function has values of opposite signs. (A zero of a function is a value of the independent variable for which the function has the value 0.)

```
      ∇ F←TOL ZERO X
[1]     →(0≤(FUNCTN X[1])×(FUNCTN X[2]))/6
[2]     →(TOL≥|G←FUNCTN F←0.5×+/X)/0
[3]     →(0<G×FUNCTN X[1])/5
[4]     →2,X[2]←F
[5]     →2,X[1]←F
[6]     'ERROR'
      ∇
```

The method used to find the approximate value of the zero is called *binary search.* It is an iterative method, in each step of which the length of the interval known to contain a zero is halved. The iteration is broken off when a value of the independent variable has been found for which the absolute value of *FUNCTN* is equal to or less than the tolerance specified by the left argument *TOL* of *ZERO* . The right argument of *ZERO* is a vector *X* of size 2, *X*[1] and *X*[2] being the given values of the independent variable, for which *FUNCTN* is supposed to have values of opposite signs.

Command [1] of *ZERO* checks whether *X*[1] and *X*[2] satisfy or violate this condition, and respectively causes branching to either [2] or [6]. In the latter case, the message *ERROR* is typed out and the computation is terminated. Interpreted from right to left, the parenthesis in command [2] calls for the computation of *F*←0.5×(*X*[1] +*X*[2]) , and evaluation of *G*←*FUNCTN F* and the Boolean expression *TOL*≥|*G* , where |*G* stands for the absolute value of *G* . The value 1 of this Boolean expression causes branching to 0, i.e., terminates the computation. If, however, the absolute value of *G* exceeds *TOL* , the computation proceeds to command [3], which effects branching to [5] or [4] depending on whether the values of *FUNCTN* at *X*[1] and *F* have the same or opposite signs. In the first case, the zero lies between *F* and *X*[2] . The substitution of *F* for *X*[1] and the return to [2] demanded by [5] thus initiate the next step of the iteration. If, on the other hand, *FUNCTN* has

values of opposite signs at *X*[1] and *F* , the computation proceeds from [3] to [4] and *F* is substituted for *X*[2] before the branch to [2] starts the next step of the iteration.

The function *FUNCTN* defined in the example below is $x^2 - 3x - 2$; its right argument and value have deliberately been named *X* and *F* to show that no confusion arises from the fact that both the function *ZERO* and the function *FUNCTN* , which appears in *ZERO* , use the same symbols for right argument and value.

```
      ∇ F←FUNCTN X
[1]     F←¯2+X×X-3 ∇

      1E¯6 ZERO 0 4
3.561553001
      0.5×(3+17*÷2)
3.561552813

      G
7.775970516E¯7

      X
VALUE ERROR
      X
      ^
```

Since *FUNCTN* has the values ¯2 and 2 , respectively, for the argument values 0 and 4, there is a zero in the interval (0, 4). The command 1*E*¯6 *ZERO* 0 4 yields 3.561553001 as the approximate value *F* of this zero. A check can be made of this result by solving the quadratic equation directly via the command 0.5×(3+17*0.5) which yields the "exact" value 3.561552813. As stipulated, the variable *G*, computed in command [2] of *ZERO* and equal to *FUNCTN F*, has an absolute value less than the tolerance 1*E*¯6 .

A more concise version of *ZERO* is shown on the next page together with the evaluations of the same zero of $x^2 - 3x - 2$ for two values of the tolerance.

```
      ∇ F←TOL ZERO X
[1]     (0<(FUNCTN X[1])×FUNCTN X[2])/4
[2]     →(TOL≥|G←FUNCTN F←0.5×+/X)/0
[3]     →2,X[1+(0≥G×FUNCTN X[1])]←F
[4]     'ERROR'
      ∇

        1E¯6 ZERO 3 4

3.561553001

        1E¯8 ZERO 3 4

3.561552815
```

## 3.3 LOCAL AND GLOBAL VARIABLES

For the purpose of *defining* a function, its arguments and value must be given identifiers. In the function *ZERO* , for instance, these are *TOL* , *X* , and *F* . There is no need, however, to employ the same identifiers for these variables when the function is *used.* For example, after specifying *A*←1*E*¯6 and *B*←0 4 , we may call for the evaluation of, say, *Z*←*A ZERO B* . Thus, the identifiers *TOL* , *X* , and *F* appearing in the definition of *ZERO* are mere *dummies* that may be replaced by other identifiers when the function is used. After a function has been evaluated, the identifiers used for the independent variables are associated with numerical values only if such values were assigned to them by a command that preceded the start of the function evaluation. Accordingly, the command *X* at the bottom of the example on p. 39 brings the response *VALUE ERROR*, followed by a repetition of the unacceptable command below which the variable without value is indicated by a caret. If, on the other hand, we had called for the evaluation of a zero of *FUNCTN* by the commands

```
X←0 4
1E¯6 ZERO X
```

the command $X$ after the evaluation of the zero would have yielded the initial value `0 4` , even though this value has been changed during the execution of the function $ZERO$ .

As the example on p. 39 shows, the final numerical value that the variable $G$ assumed just before the evaluation of $ZERO$ was terminated by the switch in command [2] remains available after this evaluation. A variable whose value is only available during the evaluation of a particular function is said to be *local* to this function; a variable whose value is not restricted in this manner is called *global.* All variables in APL functions are global unless they appear as dummies in function headers or are specifically declared as local.

To avoid the inadvertent use of the same identifier for distinct variables, it is *good practice* to declare as local to a function all variables whose numerical values are not likely to be required after the function has been evaluated or that can be readily recomputed should they be so required. In $ZERO$ , for instance, $G$ could be declared as local because its last value, if desired, can be obtained as shown below.

```
        ⎕←Z←1E¯6 ZERO 0 4

3.561553001

        FUNCTN Z
7.775970516E¯7
```

To declare variables as *local* to a function, they must be listed at the end of its header, each being preceded by a semicolon. For example, the header

```
∇T←P GROWTHTABLE N;I;J;K
```

indicates that the variables $I$, $J$, $K$ are local to the function $GROWTHTABLE$ (see the beginning of Sec. 3.2), in which $J$ and $K$ determine the size of the table and $I$ is a counter used to terminate the computation after $J$ rows of the table have been obtained. The

numerical values of these variables are obviously no longer needed when the function *GROWTHTABLE* has been evaluated.

Note that the value of a global variable is not available during the execution of a function that has a local variable with the same identifier. For instance, command [4] could be omitted from the original definition of *GROWTHTABLE* provided that the assignment *I*← 0 was made before this function was invoked, but this change could not be made in the definition in which *I* has been made local to the function.

Since the numerical values of the local variables of a function are available throughout the evaluation of this function, they are also available during the evaluation of a second function that appears in the definition of the first function, provided that the second function does not have local variables with the same identifiers. On the other hand, local variables of the second function, while available during its evaluation, are not available in the first function once the evaluation of the second function has been completed. These facts are illustrated by the example below, in which the function *CONVERGENCE* uses the function *AREA* , which in turn uses the function *Y* .

The function *N AREA B* approximately evaluates the area bounded by the *X* axis, the graph of the function *Y X* , and the lines with the equations *X*=*B*[1] and *X*=*B*[2] as follows: the strip bounded by these two lines is divided into *N* strips of equal width; in each of these strips, the graph of *Y X* is replaced by its secant, and the areas of the trapezoids formed in this way are added.

The function *P CONVERGENCE C* displays the approximate areas for *N*=2**P*[1], 2**P*[1]+1, ..., 2**P*[2], the numerical values of the elements of *B* being the same as those of *C* .

Although *I* is declared local to *CONVERGENCE* , it is available to *AREA* , which is used by *CONVERGENCE* . On the other hand, if *CONVERGENCE* , instead of listing each value of *I* followed by the corresponding value of *AREA* , were to list each value of *H* (see command [1] of *AREA* ) followed by the corresponding value of *AREA*, the variable *H* could not be made local to *AREA* . Note that

no confusion arises from the fact that *C* is used as a dummy in *CONVERGENCE* and as a local variable in *AREA* .

```
        ∇CONVERGENCE[⎕]∇

      ∇ P CONVERGENCE C;I
[1]     I←2*P[1]-1
[2]     →((2*P[2])<I←2×I)/0
[3]     →2,⎕←I,(I AREA C)
      ∇

        ∇AREA[⎕]∇

      ∇ A←N AREA B;H;C
[1]     H←(B[2]-B[1])÷N
[2]     C←B[1]+H×⍳(N-1)
[3]     A←H×(+/Y C)+0.5×(Y B[1])+Y B[2]
      ∇

        ∇Y[⎕]∇

      ∇ F←Y X
[1]     F←X*2
      ∇

        5 8 CONVERGENCE 1 3
32   8.66797
64   8.66699
128   8.66675
256   8.66669
```

(Note that the exact value of the desired area is 26÷3 .)

## 3.4 CHECKING FUNCTION DEFINITIONS

To illustrate various means of checking function definitions that are to be used in a computational task, suppose that the functions *CONVERGENCE* and *AREA* of the last section were erroneously defined as follows, while the function *Y* was defined as before.

```
        ∇CONVERGENCE[⎕]∇

      ∇ P CONVERGENCE C;I
[1]     I←2*P[1]-1
[2]     →(2*P[2]<I←2×I)/0
[3]     →2,⎕←H,(I AREA C)
      ∇

        ∇AREA[⎕]∇

      ∇ A←N AREA B;H;C
[1]     H←(B[2]-B[1])÷N
[2]     C←B[1]+H×⍳(N-1)
[3]     A←H×(+/Y C)+0.5(Y B[1])+Y B[2]
      ∇

        1 2 CONVERGENCE 1 3
```

The attempt to execute `1 2 CONVERGENCE 1 3` produces a blank line, as the example shows. This suggests that command [3] of *CONVERGENCE*, which would produce the first value of *H* followed by the corresponding value of *AREA* , has not been reached, because command [2] has effected branching to [0] instead of [3]. To check what has gone wrong, we may request that the output from commands [1] and [2] of *CONVERGENCE* be *traced*. This request and the response to it are shown below.

```
        T∆CONVERGENCE←1 2
        1 2 CONVERGENCE 1 3
CONVERGENCE[1] 1
CONVERGENCE[2] 0

        T∆CONVERGENCE←0
```

It is seen that command [1] yields the correct result 1 (initial value of *I* ), while command [2] furnishes the incorrect result 0 (instead of the empty vector, which indicates branching to the next command). Indeed, evaluating command [2] from right to left, we first obtain 2 as the new value of *I* , then 0 as the value of `P[2]<2`, then 1 as the value of `2*0` , and finally 0 as the value of 1/0, thus

producing branching to [0] instead of [3]. For the latter branching to take place, the final result from [0] should be the empty vector 0/0. To achieve this, we must type `(2*P[2])<I` instead of `2*P[2]<I`. Before we make this correction, we cancel the request for a trace of *CONVERGENCE* as shown in the last line of the example above.

The necessary correction can be made as described at the end of Sec. 3.1. In making corrections, we must guard against introducing new errors. Suppose, for instance, that we attempt to correct command [3] as shown below, inadvertently omitting the rightward arrow. The attempt to execute the "corrected" function produces the report of a *SYNTAX ERROR* in *AREA*[3] .

```
      ∇CONVERGENCE[2] ((2*P[2])<I←2×I)/0 ∇

      1 2 CONVERGENCE 1 3

SYNTAX ERROR
AREA[3] A←H×(+/Y C)+0.5(Y B[1])+Y B[2]
                             ∧
      )SI
AREA[3] *
CONVERGENCE[3]
      →
      ∇AREA[3⎕1]
[3]   A←H×(+/Y C)+0.5(Y B[1])+Y B[2]
                     1
[3]   A←H×(+/Y C)+0.5×(Y B[1])+Y B[2]
[4]   ∇
```

This shows the computation has now reached *CONVERGENCE*[3], which requires the evaluation of *I AREA C* . The caret located under the reproduction of the incorrect command indicates the source of trouble: the multiplication symbol between the factor 0.5 and the opening parenthesis has been omitted. To correct command [3] of *AREA* , we would have to enter the definition mode. This, however, cannot be done directly because the error report interrupted the execution of *CONVERGENCE* . To find out exactly where we stand, we call for a status report by entering the system command

)*SI* , which stands for *status indicator.* The response to this lists all currently active functions (i.e., functions whose evaluation has not yet been completed), beginning with the most recent one, and indicates for each the command whose execution is to be completed next. An asterisk indicates that the function is in *suspended execution.* Before the erroneous function definition can be corrected, the status indicator must be cleared by entering a rightward arrow for each asterisk—in the example above, this means one rightward arrow.

Since the command *AREA*[3]is rather long, it would be bothersome to retype the entire command just to insert a single multiplication symbol. A more convenient way is shown above. In the command ∇*AREA*[3⎕1], the 3 is the number of the command that is to be corrected, and the 1 (or any other digit) indicates that we wish to correct without complete retyping. In response, the current form of the erroneous command is reproduced. Beneath the opening parenthesis following the factor 0.5, we type a 1 to indicate that the correction requires *one* empty space to the left of the parenthesis.* Depression of the RETURN key yields another copy of the incorrect command with the demanded blank space, and positions the type head at this space. We now type the multiplication symbol that had been omitted and depress the RETURN key. The response to this is the number (here [4]) of the next command (if there was such a command). If we do not wish to change this command or add a command with this number, we enter a "del" to close the definition mode.

The attempt to execute *CONVERGENCE* produces the report of a *VALUE ERROR*, indicating that in *CONVERGENCE*[3] the value of *H* is not defined. Inspection of *AREA* shows that *H* has been made local to this function; its value thus is not available to *CONVERGENCE* once *AREA* has been evaluated. We first ask for the status indicator and then clear it. The next few lines show how the header of *AREA* (referred to as *AREA*[0] ) is corrected by using

---

*The digits from 1 to 9 may be used in this manner to demand the insertion of up to nine empty spaces, and the letters *A*, *B*, *C* . . . may be used to indicate the need for 5, 10, 15, . . . empty spaces.

```
      1 2 CONVERGENCE 1 3

VALUE ERROR
CONVERGENCE[3] →2,⎕←H,(I AREA C)
                        ∧
      )SI
CONVERGENCE[3] *
      →
      ∇AREA[0⎕1]
[0]   A←N AREA B;H;C
               //
[0]   A←N AREA B;C
[1]   ∇
      1 2 CONVERGENCE 1 3

1  9

0.5  8.75
0
0.25  8.6875
0
0.125  8.67187
0
0.0625  8.66

      )SI
AREA[3] *
CONVERGENCE[3]
      →
      ∇CONVERGENCE[2⎕1]
[2]   ((2*P[2])<I←2×I)/0
      1
[2]   →((2*P[2])<I←2×I)/0
[3]   ∇
      1 2 CONVERGENCE 1 3
1  9
0.5  8.75
```

slashes to indicate deletion of characters. The next attempt at evaluating *CONVERGENCE* yields two correct lines of the result; however, these are separated by a blank line that should not be there and are followed by an apparently unending sequence of further lines. To

interrupt this, we depress the ATTN key. The response to this indicates the command whose execution is to be completed next. We again request a status report and clear the status indicator. The fact that the computation is not terminated shows that the desired branching to [0] in *CONVERGENCE*[2] is not performed. Inspection of this command reveals that the rightward arrow has been omitted. After this mistake has been corrected, 1 2 *CONVERGENCE* 1 3 finally yields the correct result.

# 4 System Commands (part 1)

System commands, and only system commands, have a closing parenthesis as the first character. Accordingly, we have already encountered some system commands, namely the commands for signing on or off, the )*DIGITS* command, the )*ERASE* command, and the )*SI* command. In this chapter, only a limited number of additional system commands will be discussed, which are used in the organization and maintenance of a library of function definitions.

The ease with which a user may build his own function library or copy functions from a public library or another user's library is a major asset of the APL system. The two brief sections of this chapter show how a user may start his function library and make additions to it. A more complete discussion of system commands is found in Ch. 9.

## 4.1 STARTING A LIBRARY

A *workspace* is a block of space in the computer's memory. Each user is entitled to at least three workspaces: the *active workspace* in which he is working, a workspace called *CONTINUE* in which the active workspace will be stored if sign-off is with )*CONTINUE*, and a workspace in which a library of function definitions may be organized.

To illustrate the establishment of such a library, let us assume that the system command *)OFF* was last used at the end of the work session preceding the one in which the functions *ZERO* and *FUNCTN* were defined, and that since then only the command *)CONTINUE* was used to terminate the sessions in which these functions and the functions *CONVERGENCE*, *AREA*, and *Y* were defined and tested. The example below shows how a library containing the relevant function definitions from these work sessions may be established.

Because the systems command *)CONTINUE* has been used at the end of the last work session, the content of the *CONTINUE* workspace is copied into the active workspace when we sign on. This is indicated by a line containing the word *SAVED* and the time and date of the last sign-off. Note that the active workspace, which is a copy of the *CONTINUE* workspace, now takes on the workspace *id*entification *CONTINUE*. This would be displayed in response to the system command *)WSID* , but this is not shown in the example. Since we may not remember the precise contents of the *CONTINUE* workspace, we call for a listing of *fun*ction*s* in the active workspace by the system command *)FNS* . The response to this reminds us that, in addition to the useful functions *AREA*, *CONVERGENCE*, and *ZERO* , the workspace also contains the functions *FUNCTN* and *Y* , which were only used to test the other functions. We therefore erase *FUNCTN* and *Y* by the next system command, to which there is no typed response. A new command *)FNS* shows, however, that the desired erasure has been performed.

Next, we use the command *)VARS* to call for a listing of global *var*iable*s* in the active workspace that possess specific numerical values. These are listed as *G* , *H* , and *Z* . The command *G* yields the last value of this variable (see p. 39). Since this value as well as the values of *H* and *Z* are of no interest in further uses of library functions, we call for their erasure.

Because the function *CONVERGENCE* uses the function *AREA* , it will be convenient to form a *group* composed of these functions. This is done by the systems command consisting of the characters *)GROUP* followed by the identifier (*ARCON*) that we wish to give to

```
   A P L \ 3 6 0

SAVED  11.03.50 07/01/70
      )FNS
AREA    CONVERGENCE     FUNCTN   Y       ZERO
      )ERASE FUNCTN Y
      )FNS
AREA    CONVERGENCE     ZERO
      )VARS
G       H       Z
      G
7.775970516E¯7
      )ERASE G H Z
      )GROUP ARCON AREA CONVERGENCE
      )WSID LIBR
WAS CONTINUE
      )SAVE
  11.06.19 07/01/70 LIBR
      )CLEAR
CLEAR WS
      )COPY LIBR ARCON
SAVED   11.06.19 07/01/70
      )FNS
AREA    CONVERGENCE
      )GRPS
ARCON
      )VARS

      )CLEAR
CLEAR WS
      )COPY LIBR
SAVED   11.06.19 07/01/70
      )FNS
AREA    CONVERGENCE     ZERO
      )VARS

      )GRPS
ARCON
```

the group and by the identifiers of the members of the group (*AREA CONVERGENCE*). There is no typed response to this command. We now choose an identifier for our function library, say

*LIBR* , by the systems command )*WSID LIBR*. The response to this gives the previous identifier of the active workspace. Finally, the command )*SAVE* is used to store the contents of the active workspace as a function library named *LIBR* .

In preparation for an illustration of uses of this library, we clear the active workspace by the systems command )*CLEAR* , the response to which indicates that we now have a clear active workspace.

If we need only the functions of the group *ARCON* , we may use the system command consisting of the characters )*COPY* followed by the identifier of the workspace (*LIBR*) containing the object (variable, function, or group) to be copied, followed by the identifier (*ARCON*) of this object. Note that *only one* object can be copied at a time. If we had not formed the group *ARCON* , we would have had to use two copy commands to get both *AREA* and *CONVERGENCE* copied. Note also that the entire named workspace (*LIBR*) will be copied if its identifier is not followed by that of a specific object.

The response to our copy command indicates when *LIBR* was saved, and the responses to requests for listings of *fn*ctions, *gr*oups, and *var*iables in the active workspace show that we have the functions *AREA* and *CONVERGENCE* and the group *ARCON* , but no global variables with numerical values. (To exhibit this absence more clearly in the example, the paper has been manually advanced by one line after the lack of a typed response was noted.)

As is shown in the rest of the example, a copy command that does not name a specific object (variable, function, or group) causes the entire named workspace (*LIBR*) to be copied.

## 4.2 ADDING TO A LIBRARY

To show how additions may be made to an already established library, let us assume that at the end of the work session in which *ZERO* was defined and tested, this function was stored as the first member of the library *LIBR* , that the command )*OFF* was used to terminate this session, that the functions *AREA* and *CONVERGENCE*

were defined and tested in subsequent sessions, which were terminated by the command )*CONTINUE* after the last numerical value of the variable *H* had been erased.

```
     A P L \ 3 6 0

SAVED  16.42.51 07/01/70
      )FNS
AREA    CONVERGENCE
      )VARS
      )GRPS
      )GROUP ARCON AREA CONVERGENCE
      )COPY LIBR
SAVED   16.40.47 07/01/70
      )WSID LIBR
WAS CONTINUE
      )SAVE
  16.44.56 07/01/70 LIBR
      )CLEAR
CLEAR WS
      )COPY LIBR
SAVED   16.44.56 07/01/70
      )FNS
AREA    CONVERGENCE     ZERO
      )VARS
      )GRPS
ARCON
```

After signing on again, we examine the contents of the active workspace and form the group *ARCON* as before. We then ask the library *LIBR*, which at this time contains only *ZERO* , to be copied into the active workspace, change the identifier of this space to *LIBR*, and save the active workspace. The rest of the example shows that *LIBR* now contains the functions *AREA* , *CONVERGENCE*, and *ZERO* , no global variables with numerical values, and the group *ARCON*.

Note that the command )*SAVE* is destructive in the sense that the original function library is now replaced by the new library.

# 5 | Additional Operators

In Sec. 1.5, only the most frequently used operators were treated, and for some of them only their monadic use was discussed. Section 5.1 is concerned with the dyadic use of the latter operators, while Secs. 5.2 and 5.3 introduce new operators in both monadic and dyadic use. Basic operators are defined in Sec. 5.4.

## 5.1 DYADIC USE OF ⍟, ! , / , AND ○

The monadic use of these operators was explained at the end of Sec. 1.5, and some of the dyadic uses of ○ were mentioned at the beginning of Ch. 3. In the following discussion of the dyadic use of these operators, *M* and *N* will denote nonnegative integers, while *A* and *B* may or may not be integers.

If *A*>0 , *B*>0 , and *A*≠1 , the value of *A*⍟*B* is the logarithm of *B* for the basis *A* .

The binomial coefficient (!*N*)÷(!*M*)×!*N*-*M* is given by the expression *M*!*N*. Note that *M*!*N* has the values 0 and 1 for *M*<*N* and *M*=0, respectively. For 1≤*M*≤*N*, the value of *M*!*N* may be interpreted as the number of combinations of *N* items taken *M* at a time.

If *A*≠0, the value of *R*←*A*|*B*, the residue of *B* modulo *A* , is the

```
      2 5⍴10⍟⍳10
 0              0.30103       0.477121      0.60206       0.69897
 0.778151       0.845098      0.90309       0.954243      1

      0 1 2 3 4!4
1   4   6   4   1

      A←12.3 234.5 456.7
      A-1|A←A+0.5
12  235  457

      ∇ S←C SINES X
[1]   S←+/C×1○X×⍳⍴,C ∇

      1 ¯0.5 0.25 SINES ○÷6
0.316987
```

smallest nonnegative number *R* such that *B* can be expressed as *R*+*N*×*A* . If *A*=0 , then *B* must be nonnegative, and *A*/*B* has the value *B* .

The meaning of *M*○*A* and (-*M*)○*A* for 0≤*M*≤7 is given in Table 5.1.

*TABLE 5.1*

| *M* | *M*○*A* | (-*M*)○*A* |
|---|---|---|
| 0 | (1-*A**2)*0.5 | (1-*A**2)*0.5 |
| 1 | sin *A* | arcsin *A* |
| 2 | cos *A* | arccos *A* |
| 3 | tan *A* | arctan *A* |
| 4 | (1+*A**2)*0.5 | (¯1+*A**2)*0.5 |
| 5 | sinh *A* | arsinh *A* |
| 6 | cosh *A* | arcosh *A* |
| 7 | tanh *A* | artanh *A* |

The examples opposite illustrate the dyadic use of these operators. Note the rounding of the positive elements of the vector *A* that is achieved by means of the operator | . The function *SINES* evaluates the series $c_1 \sin x + c_2 \sin 2x + \cdots + c_n \sin nx$. The left argument *C* is the vector of coefficients, and the right argument is the value of *X* .

## 5.2 OPERATORS ⌊, ⌈, AND ?

Since these operators have not yet been discussed, both their monadic and dyadic uses will be explained below, where *M* and *N* denote positive integers.

On the real number axis that is directed toward the right, the points with the abscissas ⌊*A* and ⌈*A* are the points with integer abscissas that are immediately to the left and right of the point with the

abscissa *A* . For example, ⌊¯3.2 and ⌈¯3.2 have the values ¯4 and ¯3 , respectively. If *A* is an integer, both ⌊*A* and ⌈*A* have the value *A* .

The value of *A*⌊*B* is the smaller of the values *A* and *B* . Similarly, the value of *A*⌈*B* is the greater of the values *A* and *B* . Consequently, ⌊/*V* and ⌈/*V* yield respectively the smallest and greatest elements of the vector *V* . The examples below indicate various uses of the operators ⌊ and ⌈ . Note the last command, which provides a convenient rounding of the elements of *V* to the nearest integers.

```
      V←¯3.2 ¯3 4 4.6
      ⌊V
¯4  ¯3  4  4
      ⌈V
¯3  ¯3  4  5
      ⌊/V
¯3.2
      ⌈/V
4.6
      ⌊V+0.5
¯3  ¯3  4  5
```

The function *SMALL* shown next uses the operations just discussed to locate the indices *I* and *J* , and the value *A*[*I*;*J*] , of the smallest element of a matrix *A* (or one of several mutually equal elements that are smaller than any other element).

```
    ∇ S←SMALL A;V;I;J
[1]   I←V⍳⌊/V←⌊/A
[2]   J←V⍳⌊/V←⌊⌿A
[3]   S←I,J,A[I;J]
    ∇

      A←3 3⍴4 2 6 8 7 1 3 5 1

      SMALL A
2  3  1
```

The operator *?* is used in connection with the selection of "random" elements from a vector. In particular, the expression *?N* represents

an integer randomly chosen from the elements of the vector ⍳*N* , each element having the same chance of being chosen. For a given value of *N* , the expression ?*N* thus does not have a fixed value, as is shown below by the response to the command ?(10⍴6), which corresponds to a tenfold repetition of the command ?6 . Note that this response simulates the result of ten consecutive throws of a die. If the command ?(10⍴6) is given once more, the response differs from the preceding one—just as the next ten throws of the die do not duplicate the first ten throws. On the other hand, after a clear active workspace has been provided by the command )*CLEAR* , two successive commands ?(10⍴6) yield the same results as before. This shows that there is some system to the "randomness," so that the term "pseudo-random" would be preferable to the term "random." For brevity, however, the latter term will be used here.

```
      ?10⍴6
1  5  3  4  2  1  5  5  6  3
      ?10⍴6
4  5  1  1  4  5  1  3  1  3
      )CLEAR
CLEAR WS
      ?10⍴6
1  5  3  4  2  1  5  5  6  3
      ?10⍴6
4  5  1  1  4  5  1  3  1  3

      )WIDTH 55
WAS 120
      (?36⍴6)+?36⍴6
11  10  7  12  8  5  6  9  8  8  6  8  2  9
      5  6  11  10  6  8  7  10  10  9  2
      5  11  8  7  10  8  8  8  5  3  6
```

Note the systems command )*WIDTH* used to control the length of the typed line. Note also that terminating a work session with )*OFF* has the same effect on subsequent random operations as the command )*CLEAR* . The last command in the example above simulates the sums obtained in thirty-six consecutive throws of a pair of dice.

If *M*≤*N* , the expression *M*?*N* represents a vector of *M* elements randomly chosen from the elements of the vector ⍳*N* , with the

provision that any element may be chosen only once. Thus, *N?N* is a random arrangement of the elements of ⍳*N*, and 2|(2×*N*)?2×*N* is a random Boolean vector with exactly *N* zeros and *N* ones. The last command in the example below shows how a random arrangement of the elements of an arbitrary vector may be obtained.

```
      5?9
5  4  2  8  1
      10?10
8  6  4  10  5  9  7  2  1  3
      2|10?10
1  0  1  0  1  0  1  1  0  0
      V←1 2 3 5 8 13 21 34 55
      V[(ρV)?ρV]
3  21  34  8  13  55  5  1  2
```

## 5.3 OPERATORS ~, ∧, ∨, ⍲, AND ⍱

The operator ~ ("not") is only used monadically, and the operators ∧ ("and"), ∨ ("or"), ⍲ ("nand"), and ⍱ ("nor") are only used dyadically. In the following discussion of these operators, *B* and *C* will be used as identifiers of Boolean scalars or vectors.

The value of the expression ~*B* is obtained from that of *B* by replacing each 0 by 1 and each 1 by 0. For example, if the vectors *B* and *V* have the same size, *B*/*V* is the set of elements in *V* that correspond to elements of value 1 in *B* (see Sec. 2.4), and (~*B*)/*V* is the complementary set of elements.

The expression *B*∧*C*, where *B* and *C* are Boolean scalars, has the value 0 unless both *B*=1 *and* *C*=1, in which case *B*∧*C* has the value 1. This meaning of the operator ∧ is displayed below as the response to the command 0 0 1 1 ∧ 0 1 0 1 , and similar displays are given for the operators ∨, ⍲, and ⍱.

In a function definition, the command →8×((*X*>5)∧(*Y*≤3)∨*Z*=6) effects a branch to command 8 only if the assertion *X*>5 *and* at least one of the assertions *Y*≤3 *or* *Z*=6 are true; in any other case, the

computation is terminated because the expression following the rightward arrow has the value 0.

```
      B←0 0 1 1
      V←1 2 3 4
      B/V
3 4
      (~B)/V
1 2
      C←0 1 0 1
      B∧C
0 0 0 1
      B∨C
0 1 1 1
      B⍲C
1 1 1 0
      B⍱C
1 0 0 0

      ∇BRANCH[⎕]∇

    ∇ B←BRANCH V
[1]   B←8×((V[1]>5)∧(V[2]≤3)∨V[3]=6)
    ∇
      BRANCH 7 2 4
8
      BRANCH 7 2 6
8
      BRANCH 5 2 4
0
```

## 5.4 BASIC OPERATORS

In the following, the monadically or dyadically used operators + - × ÷ * ⍟ ! | ⌊ ⌈ ○, the monadically used operators ? ~, and the dyadically used operators < ≤ = ≥ > ≠ will be called *basic*. At the beginning of Sec. 2.3, rules were given regarding the use of operators with vectors and matrices, but no attempt was made to define the class of operators to which these rules apply, because only

a few operators had been defined at that time. It is now appropriate to state that the symbols *m* and *d* used there respectively stand for any basic monadic or dyadic operator. Some illustrations are given below.

```
        V←⍳5
        ⎕←W←V[5?5]
4  5  3  1  2
        ⎕←R←V>W
0  0  0  1  1
        ⎕←S←V≥W
0  0  1  1  1
        R∧S
0  0  0  1  1
        R∨S
0  0  1  1  1
```

# 6 | Arrays (part 2)

The discussion of arrays in Ch. 2 was restricted to vectors and matrices and to the most important special operations on these arrays. Section 6.1 is concerned with arrays of any dimensionality, while Sec. 6.2 treats additional special operations on arrays.

## 6.1 ARRAYS OF HIGHER DIMENSIONALITY

An array of the dimensionality 3 may be visualized as a deck of file cards, in which each card carries a matrix of the same size. For example, an array *A* of the size `3 2 4` corresponds to a deck of three cards, each of which carries a matrix of 2 rows and 4 columns. The element `A[I;J;K]` is found on the *I*th card at the intersection of the *J*th row and *K*th column. The display of such an array is card by card, as shown in the example below, which also indicates how the elements of a vector *V* are structured into an array of dimensionality 3.

Note that the command `,A` yields the vector *V* from which *A* was formed. Note also that the commands `A[3;2;4]` and `A[3;1 2;4]`, respectively, yield a single element and a vector of size 2, while the commands `A[2 3;1 2;3]` and `A[2 3;1 2;2 3 4]` yield a matrix

```
      ⎕←A←3 2 4ρV←⍳24
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
      ,A
1 2 3 4 5 6 7 8 9 10 11 12 13
      14 15 16 17 18 19 20 21 22 23
      24

      A[3;2;4]
24
      A[3;1 2;4]
20 24
      A[2 3;1 2;3]

11 15
19 23
      A[2 3;;3]

11 15
19 23
      A[2 3;1 2;2 3 4]

10 11 12
14 15 16

18 19 20
22 23 24
```

of size 2 2 and an array of size 2 2 3. Finally, note that the two central indices in A[2 3;1 2;3] exhaust the range of the second index of A and may therefore be omitted.

An array A of the dimensionality 4 may be visualized as a set of decks of cards, each deck containing the same number of cards, and each card carrying a matrix of the same size. The element A[I;J;K;L]

is found at the intersection of row *K* and column *L* on card *J* of deck *I* . The display of such an array is deck by deck and, within each deck, card by card. The example below shows how a vector of size 24 or a matrix of size 3 8 are restructured into an array of size 2 2 2 3 , and how various groups of elements may be selected from such an array.

```
        ⎕←A←2 2 2 3ρM←3 8ρV←ι24

  1   2   3
  4   5   6

  7   8   9
 10  11  12


 13  14  15
 16  17  18

 19  20  21
 22  23  24
        A[1;2;1;3]
9
        A[1; ; ;3]

  3   6
  9  12
        A[;;;3]

  3   6
  9  12

 15  18
 21  24
```

Arrays may be indexed by arrays. For example, in the expression *M*[2 3;4 2], the matrix *M* is indexed by the vectors 2 3 and 4 2 . If *B* , *C*, and *D* are matrices, *A*←*B*[*C*;*D*] defines an array *A*

of size (ρ*C*),ρ*D* with the typical element *A*[*I*;*J*;*K*;*L*]←*B*[*C*[*I*;*J*]; *D*[*K*;*L*]]. For this definition to make sense, the elements of *C* and *D* must be positive integers not exceeding ρ*B*[1] and ρ*B*[2], respectively. If this condition is not fulfilled, an *INDEX ERROR* is reported. Similarly, *A*←*B*[*C*;] defines an array of the size (ρ*C*),ρ*B*[2] with the typical element *A*[*I*;*J*;*K*]←*B*[*C*[*I*;*J*];*K*]. The extension to arrays of higher dimensionality is immediate. In the example below, this kind of indexing is used to substitute a ¯1 for each 0 in a Boolean matrix *B*. Note the alternative ways of obtaining the same result.

```
      B←2 3ρ1 0 1 1 0 0
      ¯1 1[B+1]

1 ¯1  1
1 ¯1 ¯1

      B-~B

1 ¯1  1
1 ¯1 ¯1

      ¯1+2×B

1 ¯1  1
1 ¯1 ¯1
```

The rules for using basic operators with vectors and matrices that were given at the beginning of Sec. 2.3 and further illustrated in Sec. 5.4 apply also to arrays of higher dimensionality, as is shown below.

```
      ⎕←A←2 2 3ρι12

  1   2   3
  4   5   6

  7   8   9
 10  11  12
```

```
      ÷A

 1              0.5            0.33333
 0.25           0.2            0.16667

 0.14286        0.125          0.11111
 0.1            0.090909       0.083333
      3×A

  3    6    9
 12   15   18

 21   24   27
 30   33   36
      ⎕←B←2 2 3ρ13-ι12

 12   11   10
  9    8    7

  6    5    4
  3    2    1
      A×B

 12   22   30
 36   40   42

 42   40   36
 30   22   12
```

## 6.2 SPECIAL OPERATORS

In Sec. 2.3 only a few of the more important special operations on arrays were discussed. A more comprehensive treatment is presented below, where *d* and *D* denote dyadically used basic operators.

### Reduction

When *A* is a vector or a matrix, the value of the expression *d*/*A* is the scalar *A*[1] *dA*[2]*d*...*dA*[ρ*A*] or the matrix *A*[;1]*dA*[;2]*d* ... *dA*[;(ρ*A*)[2]]. Similarly, when *A* is an array of dimensionality

3, the value of *d*/*A* is the matrix `A[;;1]dA[;;2]d...d A[;;(ρA)[3]]`. Because the operation *d*/ reduces the dimensionality of an array by 1, it is called *reduction.* In the examples above, the size of the reduced array is obtained from that of the original array by deleting the last element. This kind of reduction will be called *reduction with respect to the last index*.

APL also provides reduction with respect to other indices. For an array *A* of dimensionality 4, for instance, the expression `d/[2]A` indicates reduction with respect to the second index yielding `A[;1;;]d A[;2;;]d...dA[;(ρA)[2];;]`. Note that *d*⌿ and *d*/ are used as convenient abbreviations of `d[1]/A` and `d/[ρρA]A`.

The following examples illustrate reduction of arrays.

```
      */10 2 3
1E8
      ⍟/10 10 100
0.30103
      ∧/1 0 1
0
      ∨/1 0 1
1
      ⎕←A←2 3 4ρ1 1 1 1 0

 1 1 1 1
 0 1 1 1
 1 0 1 1

 1 1 0 1
 1 1 1 0
 1 1 1 1
      +⌿A

  2  2  1  2
  1  2  2  1
  2  1  2  2
      =/[2]A

 0 0 1 1
 1 1 0 0
```

```
      A←2 2 4⍴(⍳4),(¯1+⍳3),⍳5
      ⌊/⌊/⌊/A
0
      ⌊/,A
0
      ⌈/,A
5
```

Note that repeated reduction by the operator ⌊ may be used to obtain the smallest element of an array, but that there is an equivalent command using only a single reduction by ⌊ .

### Inner Product

The scalar product and the matrix product discussed in Sec. 2.3 are special forms of an operation that is called *inner product.* It involves two arrays $A$ and $B$ that must be *conformable* in the sense that the last element of the size of $A$ must be the same as the first element of the size of $B$. The result of the operation is an array $C$, whose size is obtained by deleting the last element from the size of $A$ and the first element from the size of $B$ and catenating the results. For example, if $A$ and $B$ have the sizes 2 3 4 and 4 5 6, respectively, then $C$ will be of the size 2 3 5 6. For conformable arrays $A$ and $B$ of, say, the dimensionality 3, the inner product $A\,d.D\,B$ is defined by the statement that `C[I;J;K;L]= d/A[I;J;]DB[;K;L]`, where $d$ and $D$ are any dyadically used basic operators. Note that for fixed values of $I$, $J$, $K$, or $L$, the expressions `A[I;J;]` and `B[;K;L]` are vectors of the same size because $A$ and $B$ are supposed to be conformable. The operation $D$ is performed for each pair of corresponding elements of these vectors, and the resulting vector is reduced by $d/$.

The following examples show uses of the inner product. The function *POL* evaluates the polynomial $c_1 + c_2x + c_3x^2 + \cdots + c_{n+1}x^n$; the vector $C$ has the elements $c_1, c_2 \ldots, c_{n+1}$, and the degenerate case n = 0 is included. The function *ERROR* furnishes the error committed when the function $e^x$ is replaced by the $(N + 1)$th partial sum of its

power series. The function *DECIMAL* yields the decimal form of the number whose binary digits constitute the vector *B* . The function *COMB* evaluates the number of combinations of *N* items when up to *M*≤*N* are taken at a time. The final example gives an approximate evaluation of the maximum of the function defined by the statement that, for any value of $x$, it equals the smaller one of the values $2 - x^2$ and $1 + x^3$.

```
        ∇POL[⎕]∇

      ∇ P←C POL X
[1]     P←C+.×X*¯1+ιρ,C
      ∇
        1 2 3 POL 5
86

        ∇ERROR[⎕]∇

      ∇ E←N ERROR X;M
[1]     E←(*X)-1+(X*M)+.÷!M←ιN
      ∇
        5 ERROR 0.3
1.0576E¯6

        ∇DECIMAL[⎕]∇

      ∇ D←DECIMAL B
[1]     D←B+.×2*(ρ,B)-ιρ,B
      ∇
        DECIMAL 1
1
        DECIMAL 1 0 1 1
11

        ∇COMB[⎕]∇

      ∇ C←M COMB N
[1]     C←(ιM)+.!MρN
      ∇
        3 COMB 5
25

        (2-X*2)⌈.⌊1+(X←0.1×(¯1+ι11))*3
1.36
```

```
      X←0.2×Y←¯1+⍳5
      (Y∘.+X)*2

 0           0.04        0.16        0.36        0.64
 1           1.44        1.96        2.56        3.24
 4           4.84        5.76        6.76        7.84
 9          10.24       11.56       12.96       14.44
16          17.64       19.36       21.16       23.04

      P←2.5+0.5×⍳7
      N←1 2 4 12
      (1+0.01×P∘.÷N)*7 4⍴N

     1.03        1.0302      1.0303      1.0304
     1.035       1.0353      1.0355      1.0356
     1.04        1.0404      1.0406      1.0407
     1.045       1.0455      1.0458      1.0459
     1.05        1.0506      1.0509      1.0512
     1.055       1.0558      1.0561      1.0564
     1.06        1.0609      1.0614      1.0617
```

## Outer Product

If *d* is a dyadically used basic operator and *V* and *W* are vectors, the *outer product* `P←V∘.dW` is a matrix of the size `(ρV),ρW` with the typical element `P[I;J]←V[I]dW[J]`. Accordingly, with `V←⍳9` , the outer product `V∘.×V` yields a multiplication table in which each element is the product of its row and column numbers. The first example on the last page shows how the outer product may be used to produce the body of a table of squares, in which the line of an entry corresponds to the integer part (0, 1, 2, 3, 4) and the column to the fractional part (0.0, 0.2, 0.4, 0.6, 0.8) of the argument. For example, the element in the second row and third column of the table is the square of 1.4.

The second example shows how the body of *GROWTHTABLE* in Sec. 3.2 may be obtained in a similar way. Note that the parenthesis in the last command represents a matrix *M* of the size `7 4`. To raise all elements in a column of *M* to the power corresponding to the *N* value for this column, we must form a matrix of the size `7 4` consisting of 7 identical lines with the elements of *N* , and use this matrix as the "exponent" of *M* . Indeed, `⋆` used as dyadic operator between two matrices requires that these matrices have the same size.

The following examples show how the outer product lends itself to the construction of useful matrices. For example, the matrix *S* in the first example may be used in the command `(,S)+.×,M`, which furnishes the sum of the elements in the principal diagonal of a matrix *M* of size `3 3`.

```
      ⎕←S←V∘.=V←⍳3
1 0 0
0 1 0
0 0 1
      V∘.<V
0 1 1
0 0 1
0 0 0
```

```
        V∘.≥V
1 0 0
1 1 0
1 1 1
        V∘.+3ρ0
 1   1   1
 2   2   2
 3   3   3
```

The function *POLYN* evaluates the polynomial with the coefficient vector *C* for each element of the argument vector *X*. (Note that *C*[1] is the constant term.)

```
        ∇POLYN[⎕]∇
      ∇ P←C POLYN X
[1]     (X∘.*¯1+ιρ,C)+.×C
      ∇
        1 2 3 POLYN ¯2 ¯1 1 3 5
9  2  6  34  86
```

The outer product *P*←*A*∘.*dB* of a matrix *A* with an array *B* of the dimensionality 3 is an array *P* of the size (ρ*A*),ρ*B* with the typical element *P*[*I*;*J*;*K*;*L*;*M*]←*A*[*I*;*J*]*dB*[*K*;*L*;*M*]. The extension of this definition to arrays of any size is immediate.

The function *TABLE* furnishes a more elegant version of the table of squares considered above. The table is presented in the form of an array *T* of dimensionality 3 consisting of two cards, each of which contains a matrix of the size 6 6 . The elements in the first column of this matrix serve as row labels and give the integer part of the argument. The elements in the first row are column labels and give the fractional part of the argument. Outer products are used in commands [3] (outer product of two vectors) and [6] (outer product of vector and matrix, which is itself the outer product of two vectors).

```
      ∇TABLE[⎕]∇

    ∇ T←TABLE C;X;Y;Z
[1]   Z←5×(¯1+⍳C)
[2]   T←(C, 6 6)⍴0
[3]   T[;1+⍳5;1]←Z∘.+Y←¯1+⍳5
[4]   T[;1;]←(C,6)⍴0,X←0.2×Y
[5]   T[;1+⍳5;1+⍳5]←(Z∘.+(Y∘.+X))*2
    ∇
      TABLE 2

 0         0       0.2      0.4      0.6      0.8
 0         0       0.04     0.16     0.36     0.64
 1         1       1.44     1.96     2.56     3.24
 2         4       4.84     5.76     6.76     7.84
 3         9      10.24    11.56    12.96    14.44
 4        16      17.64    19.36    21.16    23.04

 0         0       0.2      0.4      0.6      0.8
 5        25      27.04    29.16    31.36    33.64
 6        36      38.44    40.96    43.56    46.24
 7        49      51.84    54.76    57.76    60.84
 8        64      67.24    70.56    73.96    77.44
 9        81      84.64    88.36    92.16    96.04
```

## Grade Up and Grade Down

The operators ⍋ ("grade up," typed by overstriking ∆ and | ) and ⍒ ("grade down") are only used monadically and only on vectors. If the elements of the vector *V* are *distinct*, the value of ⍋*V* is the permutation of the elements of ⍳⍴*V* that produces an *ascending* order of the elements of *V* when ⍋*V* is used as index of *V* . In other words, each element of *V*[⍋*V*] is greater than the preceding one. Similarly, each element of *V*[⍒*V*] is smaller than the preceding one. If an element value occurs repeatedly in *V* , the first occurrence is listed first in ⍋*V* and ⍒*V* , the second occurrence is listed next, and so on. The following examples illustrate uses of these operators. The function *ROWORDER* has as its argument a matrix *M* and yields the matrix obtained from *M* by separately putting the elements of each row in ascending order.

Note that (1+⌈/,*M*)-⌊/,*M* is the difference between the greatest and smallest elements of *M* , augmented by 1. If this value is denoted by *C* , the outer product in command [1] is a matrix *N* of the same size as *M* . All elements of the *I*th row of *N* have the value *I*×*C* for 1≤*I*≤(⍴*M*)[1] . This means that no element will change its row when ,*M*+*N* is ordered and then structured into a matrix of the size ⍴*M* .

## Take and Drop

The operators ↑ ("take") and ↓ ("drop") are only used dyadically. If the right argument is a vector, the left argument must be an integer *I* . The expression *W*←*I*↑*V* represents a vector *W* of the size |*I* . If *I* is positive, *W* consists of the first *I* elements of *V* or, if *I*>⍴*V* , of the elements of *V* followed by *I*-⍴*V* zeroes. If *I* is negative, *W* consists of the last *I* elements of *V* or, if (|*I*)>⍴*V*, of (|*I*)-⍴*V* zeroes followed by the elements of *V* . If *I*=0 , *W* is empty.

Similarly, if (|*I*)<⍴*V* , the vector *W*←*I*↓*V* is obtained by dropping the first or last *I* elements from *V* depending on whether *I* is positive or negative. The vector *W* is empty if (|*I*)>⍴*V* , and identical with *V* if *I*=0 .

```
      V←3 ¯3 4 1 2
      ⍋V
2  4  5  1  3
      V[⍋V]
¯3  1  2  3  4
      ⍒V
3  1  5  4  2
      V[⍒V]
4  3  2  1  ¯3

      ∇ROWORDER[⎕]∇

    ∇ R←ROWORDER M
[1]   R←(⍴M)⍴(,M)[⍋,M+(((1+⌈/,M)-⌊/,M)×⍳(⍴M)[1])∘.+(⍴M)[2]⍴0]
    ∇
      M←3 4⍴-⍳12
      ROWORDER M

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

If the right argument is an array *A* of at least the dimensionality 2, the left argument *A* must be a vector of size ⍴⍴*A* that consists of integers. The *J* th element of *I* operates on the *J* th index of *A* in accordance with the rules given above for a vector as a right argument. For example, for a matrix *M* of the size 3 4 , the matrix 2 ¯3↑*M* consists of the elements at the intersections of the first two rows and the last three columns of *M* . Similarly, for an array *A* of the size 3 3 4 , the array that consists of only the first three elements of the first rows of cards 1 and 2 is obtained by the command 2 1 3↑*A* .

The examples below illustrate these rules. Note that the command ¯1↑*V* , which takes only the last element of the vector *V* , furnishes a vector of size 1 rather than a scalar. Similarly, the command ¯1 ¯1↑*M*, which takes only the element at the lower right corner of the matrix *M* , yields a matrix of the size 1 1 rather than a scalar. Note also that the command ¯3 ¯4↑*M* adds a first row and a first column of zeros to the matrix *M* of size 2 3 . This kind of command may be used to shorten the definition of the function *TABLE* (p. 74) as follows: command [2] is replaced by the command

```
[2] T←((⍴,Z),¯6 ¯6)↑(Z∘.+(Y∘.+X))*2
```

and command [5] is deleted.

```
      V←⍳6
      (2↑V),¯3↑V
1  2  4  5  6
      2↓¯1↓V
3  4  5
      ⎕←W←¯1↑V
6
      ⍴W
1
      (8↑V),4↓V
1  2  3  4  5  6  0  0  5  6
      (¯8↑V),¯4↓V
0  0  1  2  3  4  5  6  1  2
```

```
          M←2 3⍴⍳6
          ⍴⎕←¯1 ¯1↑M
    6
1 1
          ¯3 ¯4↑M
    0 0 0 0
    0 1 2 3
    0 4 5 6
          0 1↓M
    2 3
    5 6
```

An extension of the catenation concept, announced by IBM but not yet implemented at the time of this writing, will provide a more efficient way of adding rows or columns to a matrix. For example, if *A* and *B* have the sizes 3 5 and 3 2 , then *C*←*A*,[2]*B* has the size 3 7 , the first five and the last two columns of *C* consisting of the matrices *A* and *B* , respectively.

**Decode and Encode**

The operators ⊥ ("decode") and ⊤ ("encode") are only used dyadically. If *A* and *B* are vectors *of the same size,* the value of *A*⊥*B* is

```
B[⍴B]+A[⍴B]×B[¯1+⍴B]+A[¯1+⍴B]×+∘.∘+A[2]×B[1]
```

(Note that the value of *A*[1] does not affect the result and may therefore be chosen arbitrarily.) For example, 1 24 60 60⊥ 10 13 9 30 is the number of seconds in 10 days, 13 hours, 9 minutes, 30 seconds. Accordingly, 2 2 2⊥1 0 1 1 is the decimal representation of the binary number with digits 1, 0, 1, 1. Note that the same result is furnished by the command 2⊥1 0 1 1 . Similarly, 3.1⊥2.2 ¯1.5 ¯2.6 7 is the value of the polynomial $2.2x^3 - 1.5x^2 - 2.6x + 7$ for $x = 3.1$. The fifth partial sum of the power series for $e^{0.3}$ may therefore be obtained by the command

`0.3⊥÷!5-⍳5`. Note, however, that it is not possible to obtain the first five partial sums of this series by the command `0.3⊥÷!N-⍳N←⍳5` because `⊥` is not a basic operator.

```
        1 24 60 60 ⊥ 10 13 9 30
911370
        2 ⊥ 1 0 1 1
11
        3.1 ⊥ 2.2 ¯1.5 ¯2.6 7
50.065
        0.3 ⊥ ÷!5-⍳5
1.3498
```

The operator `⊤` , which must have a scalar as its right argument, is inverse to `⊥` in the following sense: if the vectors $A$ and $B$ are of *the same size,* then `(N↓A)⊤A⊥B` has the value `N↓B` . It therefore follows from the first example given for the operator `⊥` that `0 24 60 60⊤911370` has the value `10 13 9 30` , while `60 60⊤911370` has the value `9 30` , which indicates that 911,370 seconds amount to an integer number of days plus 9 minutes and 30 seconds. The number of hours may be obtained by the command `0 60 60⊤911370` , which furnishes the result `253 9 30` . A few examples are given below. Note that for a vector $A$ and a scalar $S$ , the expression `A⊤S` is a vector of size `⍴A` .

```
        0 24 60 60 ⊤ 911370
1   13  9  30
        60 60 ⊤ 911370
9  30
        0 60 60 ⊤ 911370
253  9  30

        (5⍴2)⊤28
1  1  1  0  0
        (7⍴2)⊤28
0  0  1  1  1  0  0
        (3⍴2)⊤28
1  0  0
        (0,(3⍴2))⊤28
3  1  0  0
```

**Reverse and Rotate**

The operator ⌽ (typed by overstriking ○ and | ) is used both monadically ("reverse") and dyadically ("rotate").

If $V$ is a vector, `W←⌽V` is the vector that consists of the elements of $V$ in reverse order, i.e., `W→V[1+(⍴V)-⍳⍴V]` . If $M$ is a matrix, `P←⌽[1]M` is the matrix obtained from $M$ by putting the rows in reverse order, and `Q←⌽[2]M` is the matrix obtained from $M$ by putting the columns in reverse order. Thus, `P←M[⌽⍳(⍴M)[1];]` and `Q←M[;⌽⍳(⍴M)[2]]` . Note that `⌽[2]M` may be abbreviated as `⌽M` . Similarly, for an array $A$ of dimensionality 3, the expressions `⌽[1]A` or `⌽[2]A` , or `⌽[3]A` , are respectively obtained from $A$ by putting the cards, or the rows on each card, or the columns on each card, into reverse order. Thus, `⌽[2]A` has the value `A[;⌽⍳(⍴A)[2];]` . Note that `⌽[3]A` may be abbreviated as `⌽A` . The extension to arrays of higher dimensionality is immediate.

```
      0.3⊥÷!⌽0,⍳4
1.3498

      A←2 2 3⍴⍳12
      ⌽[2]A

  4   5   6
  1   2   3

 10  11  12
  7   8   9
```

Note that the first example above gives another command yielding the fifth partial sum of the power series for $e^{0.3}$.

Used dyadically, the operator ⌽ takes a positive or negative integer as its left argument and an array as its right argument. If $V$ is a vector and $I$ is a positive integer, `I⌽V` is obtained from $V$ by moving `(⍴V)|I` elements from the head of $V$ to the tail; if $I$ is negative, `(⍴V)|I` elements are moved from the tail of $V$ to the head. If $M$ is a matrix, `I⌽[1]M` is obtained from $M$ by moving `(⍴M)[1]|I` rows from the top of $M$ to the bottom, or from the bottom to the top, depending on whether $I$ is positive or negative.

The expression I⌽[2]M , which may be abbreviated as I⌽M , indicates analogous operations on the columns of M . The extension of these conventions to arrays of higher dimensionality is immediate.

```
      2⌽⍳5
3  4  5  1  2
      ¯2⌽⍳5
4  5  1  2  3

      A←3 2 2⍴⍳12
      2⌽[1]A

   9  10
  11  12

   1   2
   3   4

   5   6
   7   8
```

**Transpose**

The operator ⍉ ("transpose"), which is typed by overstriking ○ and \ , is used both monadically and dyadically. We have already encountered the monadic use of ⍉ with a matrix M as an argument: ⍉M is the transpose of M (see Sec. 2.3). Applied to an array of higher dimensionality, the monadic use of ⍉ effects the exchange of the last two indices. For example, if A has the dimensionality 3 and B←⍉A , then B[I;J;K] has the value A[I;K;J], that is, each card of B carries the transpose of the matrix on the corresponding card of A .

In dyadic use, the operator takes as right argument an array, say A , of dimensionality D≥2 , and as left argument a vector, say V , of size D . The elements of V must be taken from ⍳D , and if G is the greatest element of V , all elements of ⍳G must occur in V .

If V consists of a permutation of the elements of ⍳D , the array B←V⍉A is obtained from A by letting the first index of A become the V[1]th index of B , the second index of A become the V[2]th

index of $B$ , and so on. Thus, the transpose of a matrix $M$ may be obtained as `2 1⍉M` . If $A$ has the dimensionality 4, `B←2 1 4 3⍉A` has the typical element `B[I;J;K;L]←A[J;I;L;K]`. The same rules apply when $V$ does not contain all elements of `⍳D` . Thus, `B←2 1 1⍉A` is an array of dimensionality 2 with the typical element `B[I;J]←A[J;I;I]` , and the command `1 1⍉M` furnishes the principal diagonal of the matrix $M$ .

The following examples illustrate uses of the operator ⍉ .

```
      A←2 2 3⍴⍳12
      ⍉A

1   4
2   5
3   6

7  10
8  11
9  12

      3 1 2⍉A

1   7
2   8
3   9

4  10
5  11
6  12

      1 2 1⍉A

1   4
8  11
```

## Compression and Expansion

We have already encountered the first of these operations in Sec. 2.4. If $B$ is a Boolean vector, that is, a vector consisting exclusively of elements of value 0 or 1, and $A$ is an arbitrary vector of the same

size as `B` , then `B/A` , the *compression* of `A` by `B` , is the vector consisting of those elements of `A` that correspond to elements of value 1 in `B` . Similarly, if `A` is an array with `(⍴A)[I]` equal to `⍴B` , then `B/[I]A` is the array obtained from `A` by deleting all elements for which the `I`th index corresponds to an element of value 0 in `B` . Thus, if `A←3 3⍴⍳9` , then `1 1 0/[1]A` consists of the first two rows of `A` , while `1 0 1/[2]A` , which may be abbreviated as `1 0 1/A` , consists of the first and last columns of `A` . Similarly, for an array `A` with `⍴A←2 3 3` , the expressions `1 0/[1]A` and `0 1 1/[3]A` , respectively, consist of the first card of `A` and the last two columns on each card of `A` .

Whereas compression deletes certain groups of elements (rows, columns, cards, etc.) from an array, *expansion* inserts groups that consist exclusively of elements of value 0. If `B` is a Boolean vector of more than `(⍴A)[I]` elements such that `+/B` equals `(⍴A)[I]` , then `C←B\[I]A` is an array such that `⍴C` is obtained from `⍴A` by replacing `(⍴A)[I]` with `⍴B` . Furthermore, `B/[I]C` equals `A` , while all elements not copied from `A` have the value 0. For example, if `A←3 3⍴⍳9` and `B←1 0 1 1` , then `B\[1]A` is of size `4 3` , with the second row consisting exclusively of zeros while rows 1, 3, and 4 are respectively identical with rows 1, 2, and 3 of `A` . Similarly, `B\[2]A`, which may be abbreviated as `B\A` , is obtained by inserting a column of zeros between columns 1 and 2 of `A` .

The following examples illustrate compression and expansion.

```
      A←3 3⍴⍳9
      1 1 0/[1]A

1  2  3
4  5  6

      1 0 1/A

1  3
4  6
7  9
```

```
      1 0 1 1\[1]A

1  2  3
0  0  0
4  5  6
7  8  9

      1 0 1 1\A

1  0  2  3
4  0  5  6
7  0  8  9
```

# 7 Character Manipulation

Whereas the preceding chapters were exclusively concerned with manipulation of numbers, this chapter treats manipulation of characters. Section 7.1 deals with input and output of character data. Operations on character data are discussed in Sec. 7.2, and are further illustrated by functions in Sec. 7.3 that concern sorting, coding, decoding, and translating.

## 7.1 CHARACTER DATA

All keyboard characters, valid overstruck combinations of these characters (e.g. $\underline{A}$ or ⌽ but not $\not{F}$ or ⌷ ), the space, and the carriage return may be used as *character data.* To distinguish them from characters used as identifiers of variables, character data must be enclosed in quotes in input, but the quotes do not appear in output. A quote that forms part of the character data must be typed as a double quote ('').

A single character between quotes is treated as a scalar, but a string of characters between quotes is treated as a vector, the elements of which are the individual characters. The example below shows how a page of text can be organized as a matrix $P$ . The command $P$ yields the typed page, while the command $P[4;]$ yields the fourth

line. Note the alternative ways of handling *short lines,* i.e., lines of less than thirteen characters (lines 3 and 6).

```
        P←6 13ρ' '
        P[1;]←'THIS IS A DI-'
        P[2;]←'MINUTIVE PAGE'
        P[3;]←'ILLUSTRATING '
        P[4;]←'THE ORGANIZA-'
        P[5;]←'TION OF A PA-'
        L←'GE.'
        P[6;ιρL]←L

        P

THIS IS A DI-
MINUTIVE PAGE
ILLUSTRATING
THE ORGANIZA-
TION OF A PA-
GE.

        P[4;]
THE ORGANIZA-
```

Another way of organizing a page of text as a matrix is shown in the next example. The first input line contains the identifier of the page, the leftward arrow, and the opening quote. The subsequent lines contain the text followed by a line with only the closing quote. Short lines, such as the last line of the example, must be extended to the standard length by the addition of spaces. Since the carriage return at the end of each line is counted as a character, each line contains fourteen characters. The carriage return after the opening quote is an additional character, so that the response to the command `ρPAGE` would be 99. In organizing the text as a matrix of size `7 14`, we must drop the initial carriage return—hence the command `M←7 14ρ(1↓PAGE)`. The commands `M[3;]` or `M[3;5]` then yield the third line or the first five characters of this line, the space being counted as a character.

```
      PAGE←'
THIS IS A DI-
MINUTIVE PAGE
TO ILLUSTRATE
THE ORGANIZA-
TION OF A PA-
GE IN FORM OF
A MATRIX.
'
      M←7 14ρ1↓PAGE
      M[3;]
TO ILLUSTRATE

      M[3;⍳5]
TO IL
```

Note that the command *M* now would furnish the text typed double-spaced because the carriage returns included in the text are added to the normal carriage returns used in the output of a matrix.

The quote-quad ⍞ (typed by overstriking ⎕ and ' ) provides another way of defining a vector of characters. In response to the command consisting of an identifier for the vector, the leftward arrow, and the quote-quad, the carriage moves to the left in readiness for character input *without enclosing quotes.* Note that this state of readiness (*quote-quad state*) cannot be terminated by entering a command such as )*CLEAR* , because this will be interpreted as a vector of characters rather than a system command. To leave the quote-quad state without providing character input, enter *O* , backspace, *U* , backspace, *T* . Note also that only a single line of characters can be directly entered in this manner, but catenation of lines may be used for more extensive input, as shown below, where the final *Z* in the second line has only been typed to show that four spaces have been entered after the word *LINES*. Since this *Z* is the thirty-ninth character of the input, and only thirty-eight characters are needed for the matrix *C* , the *Z* does not appear in the output.

It is often desirable to combine the results of numerical or non-numerical operations with segments of explanatory text. This can

```
      C←⍞
AN EXAMPLE OF CATE-
      C←C,⍞
NATION OF LINES     Z
      C←2 19⍴C
      C

AN EXAMPLE OF CATE-
NATION OF LINES

      C[2;]
NATION OF LINES
```

be done by enclosing the text segments in quotes and separating them by semicolons from the commands for output, as shown in the example below.

```
      M←2 2⍴5-⍳4
      'THE TRANSPOSE OF THE MATRIX';M;'
IS THE MATRIX';⍉M
THE TRANSPOSE OF THE MATRIX
  4  3
  2  1
IS THE MATRIX
  4  2
  3  1
```

Note the quote at the end of the first line, which indicates that the character input is not yet complete. If this quote were omitted, an error report would be received upon entering the first line on account of the terminal semicolon. The symbol ⍝ (typed by overstriking ∩ and ∘ ) at the head of a line indicates that this line contains a

```
      ⍝ EVALUATION OF POLYNOMIAL P
      ⍝ C=VECTOR OF COEFFICIENTS
      ⍝ BY DESCENDING POWERS OF X

      C←2 3 ¯4 5
      X←2
      ⎕←P←X⊥C
25
```

*comment* rather than a command. Segments of a longer computation may be labelled in this manner to facilitate review at a later time.

## 7.2 OPERATIONS ON CHARACTER DATA

The only basic operators that have meaning when used with character data are = and ≠ . For example the expression '*PEARS* '= '*PEACHES*' has the value 1 1 1 0 0 0 0 because only the first three letters of the two words are identical.

Inner products with = or ≠ as the second operator, or outer products with one of these operators, are also meaningful. For example, if *WORD*1 and *WORD*2 are the identifiers of two character vectors of equal size, the expression *WORD*1∧.=*WORD*2 will have the value 1 only if the vectors are identical. Similarly, the expression *V*←+/*WORD*1∘.=*WORD*2 , where the character vectors *WORD*1 and *WORD*2 need not have the same length, has as its value a numerical vector *V* such that *V*[*I*] indicates how often the *I*th character of *WORD*1 occurs in *WORD*2 .

```
          'CAT'∧.='HAT'
0
          +/'ORANGE'∘.='GRAPEFRUIT'
0  2  1  0  1  1
```

The monadically used operators , ρ and the dyadically used operators , ρ ι ∊ as well as all operators such as ⌽ ⍉ ↑ that rearrange, choose, or discard elements of an array have the usual meaning when they are applied to a character array. Some examples are given below. Note that expansion of a character array inserts spaces rather than zeros. Note also that the response to 2 ¯1↓*M*, where *M* is a two-row matrix, is the empty vector.

```
          M←2 5ρ'APPLESAUCE'
          M

APPLE
SAUCE
```

```
        ρ,M
10
        ρM
2  5
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ'⍳M

   1  16  16  12   5
  19   1  21   3   5

        M∊'A'

 1  0  0  0  0
 0  1  0  0  0

        ⌽M

ELPPA
ECUAS
        1⌽[1]M

SAUCE
APPLE

        ⍉M

AS
PA
PU
LC
EE
        2 ¯1 ↓M

        1 ¯1 ↓M

SAUC
        1 0 1 0 1 0 1 0 1\M

A P P L E
S A U C E
```

The function *WORDS* on page 92, which extracts the words from a phrase, uses some of these operations. Its right argument *PHRASE* is the character vector corresponding to the phrase. Command [2] deletes any character (such as a digit or punctuation mark) that is not a letter or a space, appends two spaces to the shortened phrase, and gives the identifier *PHRASE* to the result. Command [4] yields the position *I* of the first space. If *I* has the value 1, then ⍳1<*I* is empty, and we proceed to commands [5] and [6], in which the space in position 1 is dropped, the result is given the identifier *PHRASE* , and an unconditional branch to [4] is executed. If the shortened *PHRASE* still has a space in position 1, this too is deleted by another execution of commands [5] and [6], and so on. If the first character of *PHRASE* is not a space, there is a branch to command [7], which causes the first *I* characters of *PHRASE* , the last of which is a space, to be typed. There follows an unconditional branch to command [3], which drops the processed part of *PHRASE* (i.e., the first *I* characters) and tests whether the remaining part consists of a single character (i.e., the second appended space), in which case the operation is terminated. If there remain at least two characters in *PHRASE* , command [4] is executed next.

Note that the last two commands of *WORDS* could not be combined into →3,⎕←*I*↑*PHRASE* because this would mean catenation of a number and characters, and APL does not provide for mixed vectors of this kind.

The output of the function *WORDS* is a series of typed lines, each of which contains a word of the input phrase. Because these lines have not been assigned identifiers, they cannot be further manipulated—for instance, alphabetically sorted.

Note that the function *WORDS* will not yield the desired result if there is a space between the last character of the input phrase and the final quote. This possibility is taken care of in the function *WORDMATRIX* shown on page 93, which organizes the words of a phrase as a matrix and thus makes them available for further manipulation.

```
      ∇WORDS[⎕]∇

     ∇ WORDS PHRASE;I;J
[1]    I←0
[2]    PHRASE←((PHRASE∊' ABCDEFGHIJKLMNOPQRSTUVWXYZ')/PHRASE),'  '
[3]    →4×1<ρPHRASE←I↓PHRASE
[4]    →7×⍳1<I←PHRASE⍳' '
[5]    PHRASE←1↓PHRASE
[6]    →4,I←I-1
[7]    ⎕←I↑PHRASE
[8]    →3
     ∇
       PHRASE←'A TEST OF THE FUNCTION  WORDS:'
       WORDS PHRASE
A
TEST
OF
THE
FUNCTION
WORDS
```

```
      ∇WORDMATRIX[⎕]∇

    ∇ W←N WORDMATRIX PHRASE;I;J;K;WORD
[1]   K←1+I←0
[2]   PHRASE←((PHRASE∊' ABCDEFGHIJKLMNOPQRSTUVWXYZω')/PHRASE),' ω '
[3]   W←(1,N)⍴' '
[4]   →7×⍳1<I←(PHRASE←I↓PHRASE)⍳' '
[5]   PHRASE←1↓PHRASE
[6]   →4,I←I-1
[7]   →8×0=+/(WORD←I↑PHRASE)∊'ω'
[8]   W[K;⍳⍴WORD]←WORD
[9]   W←((K←K+1),N)↑W
[10]  →4
    ∇
      PHRASE←'THIS IS  A  TEST OF THE FUNCTION WORDMATRIX  '
      M←15 WORDMATRIX PHRASE
      M

THIS
IS
A
TEST
OF
THE
FUNCTION
WORDMATRIX

      M[7;]
FUNCTION
```

The left argument $N$ of this function indicates the length of the rows of the output matrix, and must at least be equal to the number of letters in the longest word of the input phrase. Command [2] appends to this phrase a space followed by an ⍵ followed by another space. Command [3] sets up an empty matrix of the size `1,N`, into which the first word is entered by command [8]. Command [9] adds an empty row to this matrix, into which the second word is entered when command [8] is executed again.

The other commands of *WORDMATRIX* have functions similar to those of the corresponding commands in *WORDS*.

## 7.3 SORTING, CODING, DECODING, AND TRANSLATING

Further examples of character manipulation are found in the function definitions of this section.

The function *SORT* has as its right argument $M$ , a matrix of words such as may be furnished by the function *WORDMATRIX* of the preceding section. The left argument $N$ of *SORT* indicates the number of initial characters of each word that will be considered in the alphabetic sorting process performed by *SORT* . The first factor of the inner product in the single command of *SORT* converts the matrix consisting of the first $N$ columns of $M$ into a numerical matrix of the same size by replacing each character with its position number in the vector consisting of space, comma, period, and the letters of the alphabet. The inner product regards the $N$ elements in each row of this matrix as the digits of a number expressed in the system with base 30 and computes the equivalent decimal numbers, which are then ordered by ascending magnitude. Finally, the rows of the character matrix are arranged in the corresponding order.

Note that sorting on the first three characters does not produce the desired order, but sorting on the first seven characters does produce it.

A slightly more compact form of `SORT[1]` will be possible when the operator ⊥ applies to matrices, an extension announced by IBM but

```
      ∇SORT[⎕]∇

    ∇ S←N SORT M
[1]   S←M[⍋(' ,.ABCDEFGHIJKLMNOPQRSTUVWXYZ'⍳((⍴M)[1],N)↑M)+.×(30*(¯1+⌽⍳N));]
    ∇
      M←4 10⍴'DOES,N.C. DOE,J.B.  DOE,J.    ABEL,W.N. '

      3 SORT M

ABEL,W.N.
DOES,N.C.
DOE,J.B.
DOE,J.

      A←7 SORT M
      A[2 4;]

DOE,J.
DOES,N.C.
      A[1;3 4]←'LE'
      A[1;]
ABLE,W.N.
```

not yet implemented at the time of this writing. The same extension will be provided for the operator ⊤ .

There are numerous systems of transforming a "clear" message, e.g., an English sentence, into a cipher that, hopefully, can only be understood by the person for whom it is meant. A fairly simple system breaks the clear message into five-character groups and treats each group as follows: each character is replaced by its position number in the character vector consisting of the letters of the alphabet, followed by space, comma, and period. For each group, these five numbers are interpreted as the digits of a five-digit number expressed in a number system with a base greater than 30. The decimal equivalent of this number is the coded form of the five-character group, and these

```
      ∇CODE[⎕]∇

    ∇ C←BASE CODE CLEAR;ALPH
[1]   C←⍳0
[2]   ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.'
[3]   CLEAR←CLEAR,(5-5|⍴CLEAR)⍴' '
[4]   →5×((⍴CLEAR)≥5)
[5]   C←C,BASE⊥ALPH⍳(5↑CLEAR)
[6]   CLEAR←5↓CLEAR
[7]   →4
    ∇
      ∇DECODE[⎕]∇

    ∇ C←BASE DECODE CIPHER;ALPH;I;V
[1]   ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.'
[2]   C←⍳0
[3]   V←5⍴BASE
[4]   I←0
[5]   →6×(⍴CIPHER)≥I←I+1
[6]   C←C,ALPH[V⊤CIPHER[I]]
[7]   →5
    ∇
      CLEAR←'RETURN IMMEDIATELY'
      ⎕←CIPHER←31 CODE CLEAR
16792222  13742716  4745469  4999986

      31 DECODE CIPHER
RETURN IMMEDIATELY
```

decimal numbers are catenated (with at least one space between consecutive numbers).

The functions *CODE* and *DECODE* respectively code a clear message (*CLEAR*)or decode a coded message (*CIPHER*). The left argument *BASE* of each function is the base of the number system used. The right arguments are *CLEAR* and *CIPHER*, respectively. Command [3] of *CODE* appends enough spaces to *CLEAR* to extend this to a size that is divisible by 5. The reader should experience no difficulties in understanding the purpose of any other command.

Automatic language translation is a field of computer science that has attracted much attention, to some extent because it is difficult for the layman to conceive how it can be accomplished. The function *TRANSLATE* is a very modest example. It yields the German numeral for an integer right argument *X* specified by at most three decimal digits. The first thirteen commands built up three character matrices *A*, *B*, and *C* containing the German numerals for 0, 1, . . ., 9; 10, 11, . . ., 19; and 20, 30, . . ., 100, respectively. Command [14] sets up an empty vector that will later receive the various parts of the desired German numeral. Command [15] breaks the given number *X* into its three digits (e.g., 123 is replaced by the vector 1 2 3, and 25 is replaced by the vector 0 2 5), and gives the identifier *X* to this vector.

The remaining commands take care of all special cases that may occur. For example, commands [16] to [18] furnish the output *NULL* if the sum of the three elements of *X* is zero, or switch to command [19] if this condition is not fulfilled. Command [19] switches to [24] or [20] depending on whether the first element of *X* is or is not 0. Command [20] switches to [21] or [23] depending on whether the first element of *X* is or is not 1. If *X*[1] has another positive value, say 3, command [23] will yield the character vector *DREI HUNDERT*, because this will be the beginning of the desired German numeral. (Note that that command [38] will, at the end, delete all spaces from the numeral.) The reader should not experience any difficulty in following the remaining commands of *TRANSLATE*. However, he should note the comparatively large number of special cases that must be considered even for this extremely simple translation problem.

```
      ∇TRANSLATE[⎕]∇

     ∇ Z←TRANSLATE X;A;B;C
[1]    A←'NULL   EINS   ZWEI   DREI   '
[2]    A←A,'VIER   FUENF  SECHS  SIEBEN '
[3]    A←A,'ACHT   NEUN   '
[4]    A← 10 7 ρA
[5]    B←'ZEHN      ELF       ZWOELF    '
[6]    B←B,'DREIZEHN  VIERZEHN  FUENFZEHN '
[7]    B←B,'SECHZEHN  SIEBZEHN  '
[8]    B←B,'ACHTZEHN  NEUNZEHN  '
[9]    B← 10 10 ρB
[10]   C←'ZWANZIG  DREISSIG VIERZIG  '
[11]   C←C,'FUENFZIG SECHZIG  SIEBZIG  '
[12]   C←C,'ACHTZIG  NEUNZIG  HUNDERT  '
[13]   C← 9 9 ρC
[14]   Z←ι0
[15]   X←(3ρ10)⊤X
[16]   →(0≠+/X)/19
[17]   Z←Z,A[1;]
[18]   →0
[19]   →(0=X[1])/24
[20]   →(1≠X[1])/23
[21]   Z←Z,C[9;]
[22]   →24
[23]   Z←Z,A[X[1]+1;],C[9;]
[24]   →(0≠X[2]+X[3])/26
[25]   →38
[26]   →(0≠X[2])/29
[27]   Z←Z,A[X[3]+1;]
[28]   →38
[29]   →(1≠X[2])/32
[30]   Z←Z,B[X[3]+1;]
[31]   →38
[32]   →(0=X[3])/37
[33]   →(1≠X[3])/35
[34]   A[2;]←'EIN    '
[35]   Z←Z,A[X[3]+1;],'UND',C[X[2]-1;]
[36]   →38
```

```
[37]  Z←Z,C[X[2]-1;]
[38]  Z←(Z≠' ')/Z
    ∇

      TRANSLATE 10
ZEHN
      TRANSLATE 200
ZWEIHUNDERT
      TRANSLATE 308
DREIHUNDERTACHT
      TRANSLATE 560
FUENFHUNDERTSECHZIG
      TRANSLATE 999
NEUNHUNDERTNEUNUNDNEUNZIG
```

# 8 Defined Functions (part 2)

The great variety of features that APL provides for the definition of functions is an important asset, but it is apt to confuse the beginner. For this reason, the discussion in Ch. 3 was deliberately restricted to a few essential features. The present chapter completes this limited information. It contains sections on headline types, branching, the use of labels, the checking and editing of function definitions, error reports, and recursive functions.

## 8.1 HEADLINE TYPES

All functions discussed in Ch. 3 had right arguments, and some also had left arguments. Moreover, all these functions had *explicit results*—that is, the function headline started with ∇ followed by a dummy identifier for the output, followed by a leftward arrow. There are, however, other types of function headlines. For example, in the headline of the function *WORDS* of Sec. 7.2, there is no identifier for the output, because the output is caused by the symbol pair ⎕← in command [7], which furnishes one word every time it is executed.

A function headline may also consist of the function name alone. For example, to make the pattern of zeros in a Boolean matrix with the identifier *BOOLEAN* more readily recognizable, one may wish to

replace each 0 by a circle (○) and each 1 by an asterisk (⋆). The function *PICTURE* below accomplishes this; it has neither arguments nor explicit result.

```
          ∇PICTURE[⎕]∇

        ∇ PICTURE
[1]       ⎕←'○⋆'[BOOLEAN+1]
        ∇
          ⎕←BOOLEAN←5 5ρ0 1 1

  0 1 1 0 1
  1 0 1 1 0
  1 1 0 1 1
  0 1 1 0 1
  1 0 1 1 0

          PICTURE

○⋆⋆○⋆
⋆○⋆⋆○
⋆⋆○⋆⋆
○⋆⋆○⋆
⋆○⋆⋆○
```

Since a function may have no argument, only a right argument, or both left and right arguments, and may or may not have an explicit result, there obviously are six headline types. It does not seem necessary, however, to give further examples illustrating types that have not been encountered in the preceding sections.

## 8.2 BRANCHING

To avoid confusion, only a single type of switch was mentioned in the discussion of branching in Sec. 3.2. This type is illustrated by command [5] of *GROWTHTABLE*.

There are, however, many other types of switch, and three important groups of these are discussed below, where the symbols $a_1, a_2, \ldots$ and $n_1, n_2, \ldots$ respectively denote assertions and command numbers.

The switches of the first group effect branching to the command $n_1$ or the command immediately following the switch according to whether the assertion $a_1$ is true or false. The switches used in Ch. 3 belong to this group; it is represented by

$$\rightarrow a_1/n_1.$$

Other members of this group are

$$\rightarrow a_1\rho n_1 \quad \text{and} \quad \rightarrow n_1\times\rho\,\iota a_1.$$

Note, however, that the last type of switch will only operate properly with index origin 1 (see Sec. 9.1), because with index origin 0, the expression $\iota 1$ has the value 0, while $\iota 0$ is empty. The switches of the second group effect branching to commands $n_1$ or $n_2$ (neither one of which needs to immediately follow the switch) according to whether the assertion $a_1$ is true or false. The types

$$\rightarrow(a_1,\sim a_1)/n_1,n_2 \quad \text{and} \quad \rightarrow n_2,n_1[1+a_1]$$

belong to this group. Note the order $n_2,n_1$ in the second switch. Note also that this switch requires index origin 1. For index origin 0, it takes the form $\rightarrow n_2,n_1[a_1]$.

Finally, the switches of the third group effect branching to one of several commands. For example,

$$\rightarrow(a_1,a_2,\ldots,a_m)/n_1,n_2,\ldots,n_m$$

effects branching to command $n_i$ if $a_i$ is the first assertion in the sequence $a_1,a_2,\ldots,a_m$ that is true. If all assertions are false, the command immediately following the switch is executed next. Another member of this group is

$$\rightarrow C\phi n_1,n_2,\ldots,n_m$$

where $C$ is a *counter* that is set at an earlier stage of the computation. If $C$ has the value 3, for example, the operation $3\phi$ applied to

the vector of command numbers brings the fourth element to the head and a branch to command $n_4$ takes place. This type of switch is useful in a computation in which corresponding branches must be taken at various stages. After the first branching has occurred, a separate value of the counter variable $C$ is specified in each branch before the branches join again for a common part of the computation. The value of the counter then assures that, at the next branch point, the course of the computation corresponds to the branch taken at the first branch point. Note that it may not be possible to achieve the same effect by repeating the original switch at the second branch point, because the values of the variables in the switch may have been changed in the intervening computation.

A command consisting of only a rightward arrow terminates execution of the function in which it occurs as well as the execution of any function that directly or indirectly calls for the evaluation of this function. For example, if the function $F$ calls for the evaluation of the function $G$ , which in turn calls for the evaluation of the function $H$ , the command → in $H$ terminates not only the execution of $H$ but also that of $G$ as well as $F$ .

## 8.3 LABELS

When developing a function definition, we may wish to insert additional commands between those of an earlier version of the definition. (See Sec. 8.5 for the manner in which this is done.) Insertions of this kind cause an automatic renumbering of commands by successive integers upon exit from the definition mode. For example, if a command is inserted between the commands originally numbered [3] and [4], the new command takes the number [4] and the old command with this number becomes command [5]. If command [3] was a switch to [4] or termination of the computation, it will now cause branching to the new command [4] or terminate the computation, and the computation will not proceed as originally planned. Insertion of new commands thus would necessitate updating of command numbers in switches. This trouble can, however, be avoided by the use of *labels* for all commands to which branching may occur.

Any variable name that is not otherwise used in a function definition may be used as label. To label a command, we insert the label followed by a colon between the number of the command and its first character. No matter how the commands of a tentative function definition are reshuffled to obtain the final form of this definition, the ultimate value of a label is the final number of the command to which the label has been attached. Accordingly, if *L*1 is used as the label of a command in the tentative version, a switch such as (10≥*I*←*I*+1)/*L*1 will cause the desired branching to the labelled command, no matter what its number may be in the final version. Labels are particularly useful in the development of a function such as *TRANSLATE* that involves a complex pattern of switching.

Note that a label is *local* to the function in which it is used, but its value is available to another function that is invoked by the first function.

## 8.4 CHECKING FUNCTION DEFINITIONS: STOP CONTROL

As was discussed in Sec. 3.4, the trace of a command in a function definition furnishes the value of this command every time it is executed. Particularly for a lengthy command containing several leftward arrows, this value may not indicate an error with sufficient clarity. Consider, for instance, command [2] of the function *ZERO* in Sec. 3.2. To check whether this command effects the appropriate branching, we may wish to know not only the value of this command, but also the values of the variables *G* , *F* , and *X* . There is no way of obtaining these by a trace.

*Stop control,* which operates in a similar manner as a trace, is useful in this respect. It is initiated by a command consisting of the characters *S*∆ followed by the name of the considered function, a leftward arrow, and the vector (*stop control vector*) consisting of the numbers of the commands just before whose execution the computation is to be temporarily halted. For example, to obtain the desired values of *G* , *F* , and *X* , in command [2] of *ZERO*, we give the command *S*∆*ZERO*←3 . There is no typed response to this. After

```
      ∇TRANSLATE[⎕]∇

     ∇ Z←TRANSLATE X;A;B;C
[1]    A←'NULL   EINS   ZWEI   DREI   '
[2]    A←A,'VIER   FUENF  SECHS  SIEBEN '
[3]    A←A,'ACHT   NEUN   '
[4]    A← 10 7 ρA
[5]    B←'ZEHN       ELF        ZWOELF    '
[6]    B←B,'DREIZEHN  VIERZEHN  FUENFZEHN '
[7]    B←B,'SECHZEHN   SIEBZEHN  '
[8]    B←B,'ACHTZEHN  NEUNZEHN  '
[9]    B← 10 10 ρB
[10]   C←'ZWANZIG  DREISSIG VIERZIG  '
[11]   C←C,'FUENFZIG SECHZIG  SIEBZIG  '
[12]   C←C,'ACHTZIG  NEUNZIG  HUNDERT  '
[13]   C← 9 9 ρC
[14]   Z←⍳0
[15]   X←(3ρ10)⊤X
[16]   →(0≠+/X)/L1
[17]   Z←Z,A[1;]
[18]   →0
[19]  L1:→(0=X[1])/L3
[20]   →(1≠X[1])/L2
[21]   Z←Z,C[9;]
[22]   →L3
[23]  L2:Z←Z,A[X[1]+1;],C[9;]
[24]  L3:→(0≠X[2]+X[3])/L4
[25]   →L9
[26]  L4:→(0≠X[2])/L5
[27]   Z←Z,A[X[3]+1;]
[28]   →L9
[29]  L5:→(1≠X[2])/L6
[30]   Z←Z,B[X[3]+1;]
[31]   →L9
[32]  L6:→(0=X[3])/L8
[33]   →(1≠X[3])/L7
[34]   A[2;]←'EIN    '
[35]  L7:Z←Z,A[X[3]+1;],'UND',C[X[2]-1;]
[36]   →L9
```

```
[37] L8:Z←Z,C[X[2]-1;]
[38] L9:Z←(Z≠' ')/Z
    ∇

      TRANSLATE 0
NULL
      TRANSLATE 16
SECHZEHN
      TRANSLATE 231
ZWEIHUNDERTEINUNDDREISSIG
      TRANSLATE 460
VIERHUNDERTSECHZIG
      TRANSLATE 700
SIEBENHUNDERT
```

binary search for a zero of the function *FUNCTN* in Sec. 3.2 has been initiated by a command such as `1E¯6 ZERO 0 4` , the computation is halted after the first execution of command [2] , and `ZERO[3]` is typed out to indicate the next command to be executed. We are now free to call for the current values of the considered variables by giving the command `G, F, X`. After these values have been furnished, the command `→3` restarts the computation, which comes to another halt after [2] has been executed once more. The command `G, F, X` now gives the second set of values of these variables, and the two sets may be sufficient to indicate that command [2] operates correctly or that it has been incorrectly formulated. The command `→` will then terminate the execution of *ZERO* , and the command `S∆ZERO←0` will remove the stop control.

Note that erasing a function on which a stop control has been set also erases the stop control vector. Editing of a line for which a stop control has been set removes the stop control for this line. Similar statements also apply to a trace.

## 8.5 EDITING FUNCTION DEFINITIONS

Several ways of modifying function definitions were already used in Secs. 3.1 and 3.4; others are discussed below, where a function with the name *FNCT* is considered.

### Insertion of a Command

Suppose that the need for insertion of the command `→2×(ρV)<I` between commands [3] and [4] is discovered when the computer has asked for command [7] . To achieve this insertion, complete the line as shown below:

```
[7] [3.5]→2×(ρV)<I
```

When the line is entered, the command is inserted with the number [3.5] and the system asks for command [3.6] . If no further com-

mand is to be inserted, we may return to command [7] by typing [7] followed by that command.

Note that any number between 3 and 4 could have been used instead of 3.5. If, for instance, 3.28 had been used, the system would have asked next for command [3.29], and this number could have been overridden by [7] in the same way as above.

When the definition of *FNCT* is closed, the commands are renumbered by integers. Accordingly, the insertion of a command between [3] and [4] causes the previous commands [4], [5], . . . to be renumbered [5], [6], . . . . This may necessitate changes of command numbers in switches unless labels were used consistently.

If the need for the considered insertion is discovered after the definition of *FNCT* has been closed, the insertion is effected by the command

```
∇FNCT[3.5]→2×(ρV)<I∇
```

which also leads to a renumbering of the previous commands [4], [5], . . . .

Note that a command that should precede the original command [1] can be inserted with a number between 0 and 1. To add, for example, command [8] when the definition has been closed by a del at the end of the line containing command [7], enter the command ∇*FNCT*, which will make the system ask for command [8].

### Deletion of a Command

To delete command [4] (i.e., command [3.5] inserted above) of *FNCT*, when the system is asking, say, for command [7] of this function, complete the line by typing [4] and depress the ATTN and RETURN keys (in this order). The system will now ask for command [5]. This number can be overridden by [7] as above.

If the definition of *FNCT* has already been closed when the need for

deleting command [4] is discovered, the command

```
∇FNCT[4⎕]
```

will yield the response

```
[4]   →2×(ρV)<I
[4]
```

Depress the ATTN and RETURN keys, in this order, to delete command [4].

Note that, just as insertion does, deletion changes command numbers and may necessitate changes in switches unless labels have been used consistently.

**Displays**

The editing of a function definition is greatly facilitated by the various ways of displaying the current form of commands of this function. In discussing commands for display, we shall assume that the current form of *FNCT* consists of *seven* commands.

As was already indicated in Sec. 3.1, the command ∇*FNCT*[⎕]∇ causes the entire definition of *FNCT* to be typed out. After this has been done, the system returns to the execution mode. If, however, the final del in the display command is omitted, the system stays in definition mode and asks for command [8] after the definition of *FNCT* has been typed. If a command [8] is to be added to the definition, this may now be entered. On the other hand, if command [4] is to be changed, type [4] followed by the new form of this command.

When editing a function definition, we may be satisfied that the first part of this definition is all right, and be interested in seeing only the commands starting with, for instance, command [5]. To accomplish this, enter one of the commands ∇*FNCT*[⎕5]∇ or ∇*FNCT*[⎕5]. The

first of these returns the system to the execution mode after the requested part of the definition of *FNCT* has been typed; the second command leaves the system in definition mode and in expectation of a change in the last command.

Finally, to display a single command, say command [4], enter ∇*FNCT*[4⎕]∇ or ∇*FNCT*[4⎕] . In particular, to display the headline, enter ∇*FNCT*[0⎕]∇ or ∇*FNCT*[0⎕] .

## 8.6 ERROR REPORTS

A command may be faulty because it does not specify the intended computation or because it cannot be interpreted or executed by the computer. For $y = 3x^2 + 4$, the commands *Y*←3×*X*⋆2+4 and *Y*←3(*X*⋆2)+4, in which *X* is supposed to have been specified by an earlier command, illustrate these two kinds of error. Only the second kind can be detected by the computer.

Since a command is executed from right to left, a faulty command may be partially executed when an error of the second kind is found. Further execution of the command is then abandoned, and the result obtained so far is not retained except for such specifications as may already have been made during the partial execution of a command with multiple specifications. A diagnosis of the error is typed out, followed by a copy of the faulty command with a caret approximately indicating the place at which the error was noted. The examples below illustrate these remarks.

```
      ((1536/5)-/÷(¯1+2×⍳10)⋆5)⋆÷5
SYNTAX ERROR
      ((1536/5)-/÷(¯1+2×⍳10)⋆5)⋆÷5
               ∧
      ((1536/5)×-/÷(¯1+2×⍳10)⋆5)⋆÷5
DOMAIN ERROR
      ((1536/5)×-/÷(¯1+2×⍳10)⋆5)⋆÷5
            ∧
      ((1536÷5)×-/÷(¯1+2×⍳10)⋆5)⋆÷5
3.14159
```

```
      ((1536/5)×F←-/÷(¯1+2×⍳10)*5)*÷5
DOMAIN ERROR
      ((1536/5)×F←-/÷(¯1+2×⍳10)*5)*÷5
             ∧
      F
0.996158
```

Note that for a command containing several errors, these are reported one at a time. The *syntax error* reported first is the omission of an operator (here × ) between two expressions. Other syntax errors are unmatched parentheses and the use of a function without all the arguments required by its header.

The *domain error* reported next consists in the use of an operation (here compression) with operands outside the domain for which this operation has been defined. While 0/5 or 1/5 could be interpreted by the system, 1536/5 cannot be interpreted. A frequent cause of domain error reports is the attempted division by zero.

Note that in the last example, the value of the partial result *F* is retained even though the complete expression cannot be evaluated due to the domain error occurring to the left of the specification of *F* .

The following examples illustrate other error reports concerning faulty formation of an expression.

```
      V←⍳4
      M←3 3⍴⍳9

      +7M
CHARACTER ERROR
      +
       ∧

      V[6]
INDEX ERROR
      V[6]
      ∧
```

```
        V+M[1;]
LENGTH ERROR
        V+M[1;]
        ∧

        M[3]
RANK ERROR
        M[3]
        ∧

        W[2]
VALUE ERROR
        W[2]
        ∧
```

When two characters are overstruck to form a symbol without defined meaning, a *character error* is reported. The use of a nonexistent element of an array results in the report of an *index error.* In the third example, $V$ is a vector of size 4 and $M$[1;] is a vector of the size 3. The attempt to add these vectors of different sizes causes a *length error** to be reported. Similarly, the attempt to form the inner product $M$+.×$N$ of matrices $M$ and $N$ with the respective sizes 3 4 and 5 2 would result in the report of a length error.

The next example illustrates a *rank error.*† The variable $M$ has been defined as a matrix, but the command $M$[3] implies that it is a vector. The final command calls for an undefined value and hence causes the report of a *value error*. The attempt to use a function that is not in the active workspace also produces this kind of error report.

The report *DEFN ERROR* (*definition error*) was already encountered in Sec. 3.1, where it was caused by the attempt to define a function *GROWTH* when a function of this name was in the active workspace.

---

*The term *length* is here used synonymously with *size.* See the second footnote on p. 17.

†The term *rank* is here used synonymously with *dimensionality.* See first the footnote on p. 17.

The first example below again illustrates this situation. As the second example shows, the same type of error report is made when the name of a function that is to be defined is the same as the identifier of a variable already in the active workspace. The third and fourth examples show that a report of definition error results from incorrect requests for displays of functions. Note, however, that the choice of a function name as an identifier of a variable leads to a report of *syntax error*.

```
      )FNS
FUNCTN  SORT     WORDMATRIX
      )VARS
BEGIN   END

      ∇F←FUNCTN X
DEFN ERROR
      ∇F←FUNCTN X
                 ∧

      ∇B←BEGIN
DEFN ERROR
      ∇B←BEGIN
              ∧

      ∇∇SORT[0⎕]∇
DEFN ERROR
      ∇
       ∧

      ∇SORT[0⎕]∇
[0]   S←N SORT M
      ∇N SORT M [⎕] ∇
DEFN ERROR
      ∇N SORT M [⎕] ∇
                     ∧

      SORT←0
SYNTAX ERROR
      SORT←0
      ∧
```

While one function may invoke a second function, which may in turn invoke a third function, and so on, excessive length of a chain of this kind will lead to the report of a *depth error.*

The report *WS FULL* indicates that the capacity of the workspace has been exhausted. After the status indicator has been cleared, the contents of the workspace should be examined and objects no longer needed should be erased. Similarly, the report *SYMBOL TABLE FULL* indicates that too many names are being used. Some functions or variables should be erased, and the commands

```
)SAVE CONTINUE

)CLEAR

)COPY CONTINUE
```

should be given in this order.

The report *RESEND* indicates that an error has occurred in the transmission from terminal to computer. The last command should be repeated. If resend requests occur frequently this fact should be reported to the operator. The report *SYSTEM ERROR* indicates a malfunction of the machine, which results in the loss of the contents of the active workspace.

## 8.7 RECURSIVE FUNCTIONS

In the course of its evaluation, a function may invoke itself. Functions of this kind are called *recursive.*

To give an example of a recursive function definition, consider the difference table of a function, $f = x^3$, that is tabulated for $x = 1$, 2, . . . . In the body of Table 8.1, the first row contains these function values, which will be denoted by $f_1, f_2, \ldots$ . The second row contains the *first differences,* $d_1^1, d_2^1, \ldots$, where $d_n^1 = f_{n+1} - f_n$. The second differences $d_1^2, d_2^2, \ldots$ are obtained from the first differences

in exactly the same way in which these are obtained from the function values, and so on.

The function *DIFF* shown below takes, as left and right arguments *M* and *N* , the order $m$ and the position $n$ of the difference $d_n^m$ , and evaluates this difference in a recursive way.

*TABLE 8.1*
*Difference Table for $f = x^3$*

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $fn$ | 1 | 8 | 27 | 64 | 125 | 216 | 343 |
| $d_n^1$ | 7 | 19 | 37 | 61 | 91 | 127 | |
| $d_n^2$ | 12 | 18 | 24 | 30 | 36 | | |
| $d_n^3$ | 6 | 6 | 6 | 6 | | | |
| $d_n^4$ | 0 | 0 | 0 | | | | |

```
          ∇DIFF[⎕]∇

        ∇ D←M DIFF N
[1]       →(M=0)/4
[2]       D←((M-1) DIFF(N+1))-(M-1) DIFF N
[3]       →0
[4]       D←FUNC N
        ∇

          ∇FUNC[⎕]∇

        ∇ F←FUNC X
[1]       F←X*3
        ∇

          2 DIFF 5
36
```

# 9 System Commands (part 2)

The discussion of system commands in Ch. 4 was restricted to a few commands that are essential to the organization of a user's library. The present chapter surveys a wider array of system commands. It contains sections on the digits, width, and origin commands, inquiry commands, library control commands, hold commands, and on trouble reports.

System commands are readily recognized by their first character, a closing parenthesis. No other properly formed command can begin with this character. System commands cannot be used in APL expressions or function definitions. Similarly, APL expressions cannot be used in system commands, as is illustrated by the following example.

```
      I←5
      )DIGITS I
INCORRECT COMMAND
```

## 9.1 DIGITS, WIDTH, AND ORIGIN

We have already encountered the *DIGITS* and *WIDTH* commands in Secs. 1.5 and 5.2. In a clear workspace, at most 10 significant

digits of a result will be displayed, and at most 120 characters per line. This can, however, be changed by commands such as

```
)DIGITS 4

)WIDTH 65
```

which respectively call for a maximum of 4 significant digits of results and 65 characters per line. From 1 to 16 digits and 30 to 130 characters may be requested by these commands. For the manner in which the *DIGITS* command affects the use of the exponent notation, see Sec. 1.5.

In a clear workspace, the *index origin* is 1. Thus, the first element of a vector *V* or the first column of a matrix *M* are denoted by *V*[1] and *M*[;1]. It may occasionally be convenient to use the index origin 0. For example, in a paper describing an algorithm for determining the zeros of a polynomial, this polynomial may be written as $c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$. In defining a function for this algorithm, it would then be convenient to use the index origin 0 and denote the elements of the vector of coefficients by *C*[0], *C*[1],..., *C*[*N*] rather than first transcribe the formulas of the paper for the index origin 1. This change of index origin is accomplished by entering the command

```
)ORIGIN 0
```

which will yield the response

```
WAS 1
```

This change of index origin, however, may cause trouble if the function that is being defined invokes other functions that were defined with index origin 1. For example, with index origin 1, the command

```
→(0 3)[1+I≠J]
```

will terminate the computation if *I*=*J* but effect a branch to command [3] if *I*≠*J* . When a function containing this switch is copied

into a workspace with index origin 0, an index error will be reported if *I≠J* , while a branch to command [3] will occur if *I=J* .

This kind of trouble can only be avoided if either index origin 0 or index origin 1 is used consistently. Since a clear workspace has index origin 1, it seems best to use this origin in all work except, possibly, special jobs that are independent of all other work. If, for a job of this kind, the index origin has been changed to 0, it can be restored to 1 by the command

```
)ORIGIN 1
```

which will yield the response

```
WAS 0
```

## 9.2 INQUIRY COMMANDS

These commands enable the user to obtain information concerning the current content of the active workspace. The commands *)FNS* , *)GRPS* , and *)VARS* that were discussed in Sec. 4.1 as well as the command *)SI* discussed in Sec. 3.4 are members of this class of system commands; others are described below.

### Functions

In response to the system command *)FNS* , an alphabetically ordered list of the names of all functions available in the active workspace is displayed. Note that in this ordering *A̲, B̲,...* follow *Z* . If the function list is lengthy, and if we only need to know whether a function with, for example, the name *TRANSFER* is available, we may give the command

```
)FNS T
```

which yields an alphabetically ordered list of function names beginning with *T* or a later letter of the alphabet. As soon as *TRANSFER*

has been listed, we may press the ATTN key to discontinue the listing.

### Variables, Groups, and Group

If the commands )*VARS* or )*GRPS* are followed by a letter, only the variables or groups with names beginning with this or a later letter of the alphabet will be listed.

The command )*GRP* followed by the name of a group yields a list of the members of this group.

### Library

The command )*LIB* yields a list of saved workspaces (including *CONTINUE* ) in the user's private library.

If followed by a number (from 1 to 999) that refers to a *public library,* the command )*LIB* yields a list of workspaces in this library. A public library may have been distributed with the system or organized locally. Public Library 1, which is distributed with the system, contains many useful functions. The list of its workspaces is shown below together with the function list of the workspace *PLOTFORMAT* .

```
      )LIB 1
COMPPROB
ADVANCEDEX
APLCOURSE
CLASS
NEWS
PLOTFORMAT
TYPEDRILL
WSFNS

      )COPY 1 PLOTFORMAT
SAVED    9.26.53 07/01/70
      )FNS
AND     DESCRIBE        DFT       EFT       PLOT      VS
```

A description of the function *PLOT* will be given in response to the command

*HOWPLOT*

*PLOT* yields a rough graph of a function of one independent variable. The left argument of *PLOT* is a two-element vector that controls the size of the graph; the right argument is a two-column matrix, each row of which contains a value of the independent variable followed by the corresponding value of the function. In the example on the next page, the function plotted is the cubic polynomial $x^3 - 16x^2 + 76x - 96$. The right argument for this application of *PLOT* is furnished by the function *POLYNOMIAL*, which takes as its left argument the vector `1 ¯16 76 ¯96` of coefficients and as its right argument the vector of the chosen values of the independent variable. Note that the function values for $x = 3$ and $x = 4$ are 15 and 16, but the ordinates of the plotted points for these abscissas are equal because each line of the typed graph corresponds to an ordinate interval of length 2.

### Workspace Identification

The command *)WSID* yields the name of the active workspace.

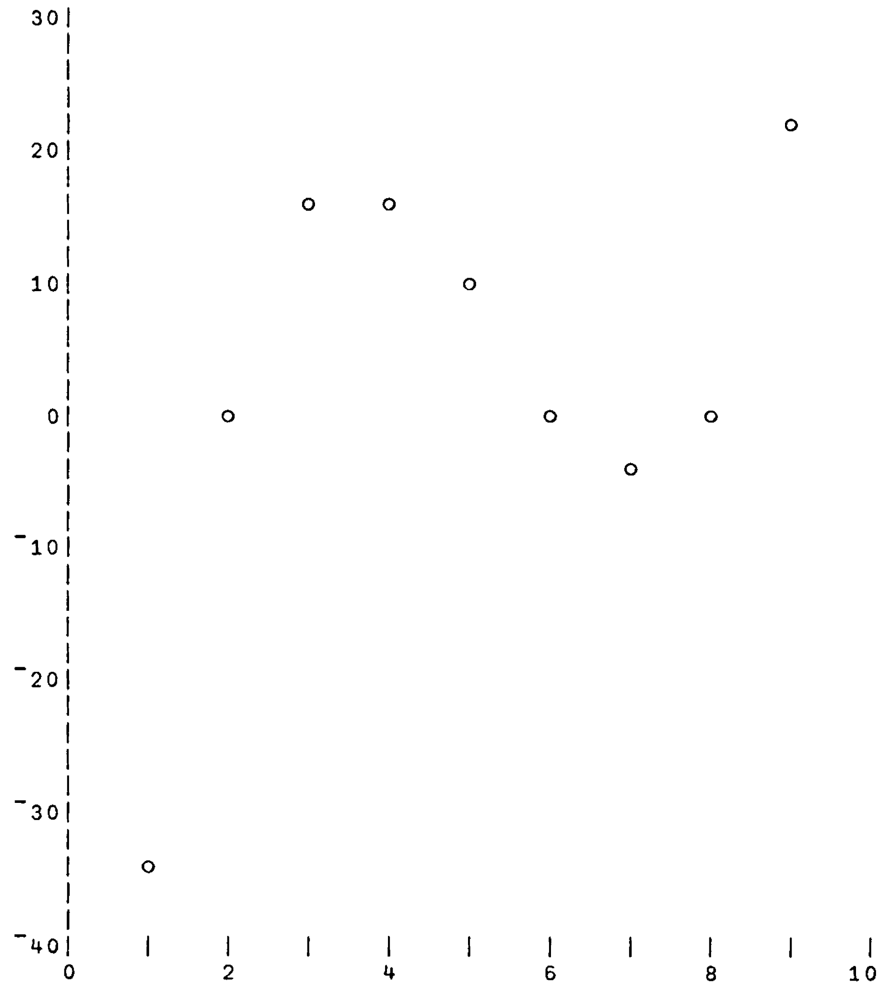
### Status Indicator

The execution of a function may be suspended because the ATTN key has been pressed, an error has been detected, or the APL operator has sent a *PA* (*P*ublic *A*ddress) message, or because a stop control was set for a command, say command [5], of the function. When a suspension occurs, the name of the suspended function is typed out followed by the number of the "next" command to be executed. In the first three cases above, this will be the command during whose execution the suspension occurred. In the stipulated case, this would be [5]. However, a suspension that is due to a stop control occurs after a command has been completely executed, and in this case the number of the next command is typed out.

```
      ∇ P←C POLYNOMIAL X
[1]     P←(2,⍴X)⍴0
[2]     P[1;]←X
[3]     P[2;]←(X∘.*(⍴C)-⍳⍴C)+.×C
      ∇
        C←1 ¯16 76 ¯96
        C POLYNOMIAL ⍳9

   1    2    3    4    5    6    7    8    9
 ¯35    0   15   16    9    0   ¯5    0   21

        40 40 PLOT (⍉C POLYNOMIAL ⍳9)
```



The system commands )*SI* and )*SIV* yield the name of each active function, followed by the appropriate command number, followed by an asterisk if the function is suspended, and, in the case of )*SIV*, by a list of the identifiers of all local variables of the function. A function in this list that has no asterisk is called *pendant.* Note that the most recently active function is listed first, and so on.

The values of all local variables of the function at the top of the list furnished by )*SI* or )*SIV* may be requested by the appropriate commands. The value of a local variable of a function farther down in the list is accessible only if the identifier of this variable is distinct from the identifiers of all local variables listed in preceding lines of the list.

Note that a suspended function may be edited, but a pendant function cannot be edited. The following example will illustrate these rules. The function *PRIMES* with positive integer *N* as right argument furnishes a list of all prime numbers that do not exceed *N* . The function *TWINS* with positive integer right argument *N* furnishes the smaller member of each pair of primes up to *N*+2 whose difference does not exceed 2. (Note the commands *PRIMES* 30 and *TWINS* 30 .) After a stop control has been set for command [5] of *PRIMES*, the execution of *TWINS* 30 is stopped just before command [5] of *PRIMES* is executed for the first time, and this is signalled by the report *PRIMES*[5] . The response to the system command )*SIV* shows that *PRIMES* is suspended and *TWINS* is pendant. The current values of all local variables of *PRIMES* and the current value of the local variable *T* of *TWINS* could be requested by the commands *PR*,*N*,*P*,*Q* and *T*, but the values of the local variables *N* and *P* of *TWINS* are not accessible, because the command *N*,*P* would yield the values of the local variables *N* and *P* of *PRIMES*.

```
        ∇TWINS[⎕]∇

      ∇ T←TWINS N;P
[1]     N←N+2
[2]     P←PRIMES N
[3]     T←(((1↓P)-(¯1↓P))≤2)/¯1↓P
      ∇
```

```
        ∇PRIMES[⎕]∇

      ∇ PR←PRIMES N;P;Q
[1]     PR←1↓⍳N
[2]     Q←N*÷P←2
[3]     →(Q<P)/0
[4]     PR←((0≠P|PR)∨P=PR)/PR
[5]     P←PR[1+PR⍳P]
[6]     →3
      ∇

        PRIMES 30
2  3  5  7  11  13  17  19  23  29
        TWINS 30
2  3  5  11  17  29

        SΔPRIMES←5
        TWINS 30

PRIMES[5]
        )SIV
PRIMES[5] *     PR      N       P       Q
TWINS[2]        T       N       P
        →
```

### Ports

The command )*PORTS* yields a list of the numbers and user codes of all connected ports (terminals).

The command )*PORT* followed by a user code yields only the port (or ports) for this user code.

### System Information

There are some inquiries concerning the system to which the answers are obtained not by system commands but by special functions. The name of each of these functions consists of the symbol ⌶ (overstruck ⊥ and ⊤ ) followed by a two-digit number. The more important of these functions are described below.

The value of ⌶19 is the total time (in 60ths of a second) during which the keyboard has been unlocked (i.e., ready to receive entries)

during the current work session. When the RETURN key is depressed, the keyboard is locked and cannot receive new entries until the present command has been executed (and results have been typed out if the command called for this).

The value of ⌶22 is the as yet unused space (in bytes) in the active workspace. (A byte is equivalent to eight binary digits.) By giving the command ⌶22 after the active workspace has been cleared, the size of this space may be obtained.

The values of ⌶20 and ⌶24 are the time of the day (since midnight) and the time of the sign-on for the current work session (in 60ths of a second).

The value of ⌶25 is a six-digit number, the two-digit groups of which give month, day, and year of the current date. The conventional form of this date may be obtained by the function *DATE* below.

```
      ∇DATE[⎕]∇

    ∇ DATE;D
[1]   D←(3⍴100)⊤⌶25
[2]   D[1];'/';D[2];'/';D[3]
    ∇

      DATE
10/23/70
```

## 9.3 LIBRARY CONTROL COMMANDS

These commands are concerned with reactivating or deleting a stored workspace, and storing the active workspace or deleting some of its contents.

### Reactivation of Stored Workspace

The system command consisting of the characters )*LOAD* followed by a space and the name of a stored workspace or the number of a public library, another space, and the name of a workspace in this

library, *replaces* the content of the active workspace by that of the named workspace, including digits, width, origin, trace, and stop controls that were in force when this workspace was stored. In response to the command, a message is typed out that starts with the word *SAVED* and gives the time and date of the last storing of the workspace.

Note that the *LOAD* command is destructive in the sense that the current content of the active workspace is lost. The system command consisting of the characters )*COPY* followed by a space and the name of a stored workspace or the number of a public library, a space, and the name of a workspace in this library, *adds* all groups, functions, and global variables of the named workspace to the current contents of the active workspace while retaining the digits, width, origin, trace, and stop controls of the latter. The response to the *COPY* command is the same as that to the *LOAD* command.

If the *COPY* command just described is extended by a space and the name of *one* object (group, function, or global variable) in the stored workspace, only this object is added to the current content of the active workspace. Note that only one object of the stored workspace can be copied at a time in this manner.

If an object in the active workspace has the same name as an object in the stored workspace, the latter will *replace* the former when the *COPY* command is executed. If this effect is not desired, a command beginning with the characters )*PCOPY* (where the *P* stands for "protected") should be used instead of the command beginning with )*COPY* . The *PCOPY* command is particularly useful if a function in the stored workspace is to be replaced by an improved version with the same name that has been developed in the active workspace. To this end, the stored workspace is *P*-copied into the active workspace, which is then given the name of the stored workspace and saved.

Note that when a group is *P*-copied, only those members will be copied whose names do not duplicate names of objects already in the active workspace. If a group in the active workspace has the same name as the group that is to be *P*-copied, the members of the latter group will be copied as far as their names are distinct from the names

of the objects in the active workspace, but they will no longer be recognized as members of a group.

A group in the active workspace may be *dispersed* by the system command consisting of the characters )*GROUP* followed by a space and the name of the group. The members of the dispersed group remain in the active workspace but are no longer recognized as forming a group. On the other hand, if a group formed of functions, variables, and other groups is erased by the command consisting of the characters )*ERASE* followed by a space and the name of the first group, the functions and variables of this group will be deleted, but the other groups will only be dispersed.

Note that a function that is being edited and a pendant function cannot be erased. The name of a stored workspace and its contents may be dropped from the library by the system command consisting of the characters )*DROP* followed by a space and the name of the stored workspace. The response to the drop command is a line with the time and date.

Provided the active workspace has a name (identification), the system command )*SAVE* will store it under this name unless the user's quota of workspaces is exhausted. The response to the command )*SAVE* is a line beginning with *SAVED* and giving the time and date.

Note that the active workspace initially has either no name or the name *CONTINUE*, depending on whether the previous work session was terminated by )*OFF* or by )*CONTINUE*. In the first case, the command )*SAVE* will yield the response *NOT SAVED, THIS WS IS CLEAR*. To save this workspace under the name *LIBR*, the command )*SAVE LIBR* should be given. Alternatively, the workspace may first be given the name *LIBR* by the command )*WSID LIBR* and then be saved by )*SAVE*.

## 9.4 HOLD COMMANDS

A user may have more than one account and may wish to work successively on several of them. To avoid redialing a telephone

connection to the central computer after finishing the work on one account, he may sign off by using the commands *)OFF HOLD* or *)CONTINUE HOLD* followed by a colon and a password if this is desired. The telephone connection will then be maintained for sixty seconds after the time and cost information for the account has been typed out, and during this time the user may sign on again by entering a closing parenthesis and the next account number (and a colon and password if the account is locked).

## 9.5 TROUBLE REPORTS

When a system command is not executed, a trouble report is typed out. Some of these have already been mentioned—for example, *NUMBER IN USE* (Sec. 1.2). This particular trouble report indicates that somebody is already signed on under the given account number. Other examples of trouble reports are *NUMBER NOT IN SYSTEM* or *WS NOT FOUND* , indicating that an account with the given number (and lock) does not exist or that there is no workspace with the given name. As these examples show, trouble reports are sufficiently specific to make their detailed discussion unnecessary.

# References

*APL Programming and Computer Techniques,* by H. Katzan, Jr., New York: D. Van Nostrand Company, Inc., 1970.

*APL\360 User's Manual,* 2nd ed., White Plains, N. Y.: IBM Technical Publications Department, 1970.

*APL\360 Primer,* 2nd ed., White Plains, N. Y.: IBM Technical Publications Department, 1970.

*APL\360 Reference Manual,* by Sandra Pakin, Chicago: Science Research Associates, Inc., 1968.

*A Programming Language,* by K. E. Iverson, New York: John Wiley & Sons, Inc., 1962.

# Index


Account number, 3
Active workspace, 49
AREA, function, 43
Arguments, 31
Array, 13, 16, 63
  arithmetic operations on, 23, 66
ATTN (Attention) key, 5

Basic operators, 61
Binary search, 38
Binomial coefficient, 55
Boolean:
  expression, 26
  vector, 27
Branching, 33, 35, 102
Byte, 125

Catenation:
  of lines, 87
  operator, 23
Character:
  data, 85, 89
  manipulation, 85
CHARACTER ERROR, 113
Checking function definition, 43, 105
CLEAR command, 52
CODE, function, 96
Code, user's, 3, 5
COMB, function, 70
Comments, 88
Compression, 82
Conformability, 69
CONTINUE command, 5
CONTINUE HOLD command, 128
Continue workspace, 49
Convergence, function, 43
COPY command, 52, 126
Counter, 103

DATE, function, 125
DECIMAL, function, 70
Decode, 78
DECODE, function, 96
Defined functions, 30, 101
Definition:
  error, 32, 113
  mode, 31
Deletion of command, 109
DEPTH ERROR, 115
DIFF function, 116
Difference table, 115
DIGITS command, 6, 49, 117
Dimension (*see* Size)
Dimensionality of array, 17, 63

Discriminant, 36
Dispersion of group, 127
Displays, 110
DOMAIN ERROR, 112
Drop, 75
DROP command, 127
Dummies, 40
Dyadic operators, 5, 11, 24, 55

Editing of function definition, 32, 108
Encode, 78
ERASE command, 32, 49
ERROR, function, 70
Error reports, 111
Execution mode, 31
Expansion, 82
Explanatory text, 87
Experimental function, 9, 15
Exponent notation, 7
Expression:
 Boolean, 27
 composite, 8, 10
 multiple evaluation of, 13

Factorial, 10
FNS command, 50, 119
Function:
 pendant, 123
 suspended, 46, 121
Function definition, editing of, 32, 108
Fuzz, 27

Gamma function, 10
Global variables, 41
Grade down, 75
Grade up, 75
GROUP command, 50
GROWTH, function, 31, 32, 113
GROWTHTABLE, function, 34, 41, 72, 102
GRP command, 120
GRPS command, 51, 119, 120

Headline, 31
 types, 101

Hyperbolic functions, 56

Identifier, 9, 40
INDEX ERROR, 20, 66, 113
Index generator, 19, 20
Indexing, 17, 63
Inner product, 69
Inquiry commands, 119
Insertion of command, 108
INT (Interrupt) key, 5
Interest, compound, 31
Intermediate results, catenation of, 23
Inverse hyperbolic functions, 56
Inverse trigonometric functions, 29, 56

Keyboard, 2
Key:
 ATTN (Attention), 5
 INT (Interrupt), 5
 RETURN, 3
 Shift, 1

Label, 104
Language translation, automatic, 97
Length (*see* Size)
LENGTH ERROR, 113
LIB command, 120
Library:
 adding to a, 52
 starting a, 49
Library control commands, 125
LOAD command, 125
Local variables, 41, 123
Logarithm, 10, 55

Matrix, 17
Matrix product, 26, 69
Membership in array, 27
Monadic operators, 9, 11, 23

OFF command, 4
OFF HOLD command, 128
Operations on character data, 89
Operator:
 catenation, 23

Operator *(continued)*:
  structuring, 17
Operators:
  basic, 61
  dyadic, 5, 11, 24, 55
  monadic, 9, 11, 23
  special, 24, 67
ORIGIN command, 118
Outer product, 72

Parentheses, use of, 9
Password, 4
PCOPY command, 126
Pendant function, 123
PICTURE, function, 102
PLOT, function, 122
POL, function, 70
POLYN, function, 73
POLYNOMIAL, function, 122
Port, 3, 124
PORT command, 124
PORTS command, 124
Public address, 121
PRIMES, function, 124

QUADRATIC, function, 36
Quotes, use of, 85
Quote-quad, 87

Random elements, 59
Rank (*see* Dimensionality)
RANK ERROR, 113
Ravel, 18
Reciprocal, 9
Reduction, 68
RESEND, 115
Residue, 55
RETURN key, 5
Reverse, 80
Rotate, 80
ROWORDER, function, 75

SAVE command, 52
Scalar, 17
  product, 25, 69
SI command, 46, 49, 119, 121, 123
Sign-off instructions, 4
Sign-on instructions, 3
Signum function, 9
SIV command, 123
Size of array, 17, 18, 63
SMALL, function, 58
SORT, function, 95
Status indicator, 46, 123
Stop control, 105
Suspended:
  execution, 46
  function, 46, 121
Switch, 35
SYMBOL TABLE FULL, 115
SYNTAX ERROR, 45, 112, 114
System:
  commands, 49
  information, 124
SYSTEM ERROR, 115

TABLE, function, 74, 77
Take, 75
Terminal, 2
Trace, 44, 105
TRANSLATE, function, 98, 100, 106, 107
Translation, automatic language, 97
Transpose, 26, 81
Trigonometric functions, 29, 56
Trouble reports, 128
TWINS, function, 124

Value, absolute, 10
VALUE ERROR, 39, 40, 46, 113
Variable:
  global, 41
  local, 41, 123
VARS command, 50, 119
Vector, 17
  empty, 19

WIDTH command, 59, 118
WORDMATRIX, function, 93
WORDS, function, 92
Workspace, 49
WS FULL, 115
WSID command, 50, 121

ZERO, function, 38

# The Allyn and Bacon Series

## PROGRAMMING LANGUAGES OF THE 70's

Consulting Editor
Peter Wegner
Brown University

C. Joseph Sass
BASIC Programming For Business

**RELATED TITLES**

An Introduction to Computer Science and Algorithmic Processes
Terry M. Walker and William W. Cotterman

Elementary Numerical Analysis With Programming
Gerald B. Haggerty

Program Design in FORTRAN IV
Raymond A. Kliphardt

Introduction to PL/1 Programming
Frank J. Clark

ALLYN AND BACON, INC.
470 Atlantic Avenue, Boston, Massachusetts 02210

203222

AN INTRODUCTION TO APL PRAGER ALLYN AND BACON