APL2 Programming:

Guide

Version 2 Release 1

SH21-1072-00

APL2 Programming:

Guide

Version 2 Release 1

SH21-1072-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page v.

First Edition (March 1992)

This edition applies to Release 1 of APL2 Version 2, Program Number 5688-228, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest edition of the applicable IBM system bibliography for current information on this product.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Department J58, P. O. Box 49023, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interface Information

This book is intended to help programmers code APL2 applications. This book documents General-Use Programming Interface and Associated Guidance Information provided by APL2 Version 2.

General-Use programming interfaces allow the customer to write programs that obtain the services of APL2 Version 2.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

APL2

DATABASE 2

DB2

IBM

SQL/DS

About This Book

This book is for people who are somewhat familiar with APL2*, who perhaps write some APL2 code for their own use, and who now want to produce a “full-blown” application. If you're new to APL or to APL2, you should read *An Introduction to APL2* before you use this book.

If you're an experienced APL user, you might skip Part 1 entirely, or focus only on those areas in the “Fundamentals” chapter that cover features that are new or different with APL2. Two new features are **defined operators**, introduced in the “Operators” section, and **nested arrays**, introduced in the “Arrays” section.

How This Book Is Organized

In Part 1, this book *reviews the fundamentals of the APL language and the features of APL2*. (The language that APL2 gives you is an extension of the APL that you could use with previous IBM products.)

In Part 2, this book *shows you how you can use some of APL2's features to program an inventory control application*. It describes the nature of the application, discusses how you might structure the data, and then illustrates the coding details. The foldout section of Part 2 (inside the back cover of the book) shows all of the functions, variables, and operators used in the sample implementation.

Other APL2 Documentation

This book does not cover all the features of APL2. Things that are important in using APL2, but aren't really “language,” aren't covered here. Instead, topics like APL2 invocation and termination, the APL2 session manager, APL2 auxiliary processors, and APL2 associated processors are covered in *APL2 Programming: System Services Reference*. You'll need to understand how to invoke and terminate APL2. You'll need to know about the session manager and auxiliary processors if you try to do some advanced tasks with APL.

As you become proficient in writing APL2 applications, you'll probably need *APL2 Programming: Language Reference*. That book describes the APL2 language in depth and gives a number of examples.

Figure 1 on page vii shows which APL2 publications should be consulted for help with various tasks.

Figure 1. APL2 Licensed Program Library

Task	APL2 Publication	Publication Number
Evaluating APL2	*APL2 General Information*	GH21-1051
	APL2 Application Environment Licensed Program Specifications	GH21-1063
	APL2 Licensed Program Specifications	GH21-1070
Installing APL2	*APL2 Installation and Customization under CMS*	SH21-1062
	APL2 Installation and Customization under TSO	SH21-1055
Migrating to Version 2 Release 1	*APL2 Migration Guide*	SH21-1069
Finding Reference Information	*APL2 Programming: Language Reference*	SH21-1061
	APL2 Programming: Processor Interface Reference	SH21-1058
	APL2 Programming: System Services Reference	SH21-1054
	APL2 Reference Card	SH21-1071
	APL2 Reference Summary	SX26-3999
Programming	*An Introduction to APL2*	SH21-1073
	APL2 GRAPHPAK: User's Guide and Reference	SH21-1074
	APL2 Programming: Guide	SH21-1072
	APL2 Programming: Using Structured Query Language (SQL)	SH21-1057
	APL2 Programming: Using the Supplied Routines	SH21-1056
Diagnosing Problems	*APL2 Diagnosis*	LY27-9601
	APL2 Messages and Codes	SH21-1059

Related Publications

IBM DATABASE 2 Application Programming Guide for TSO Users, SC26-4081
IBM DATABASE 2 Data Base Planning and Administration Guide, SC26-4077
IBM DATABASE 2 Introduction to SQL, SC26-4082
SQL/Data System Application Programming, SH24-5018
SQL/Data System Planning and Administration, SH24-5043

Part 1. Introduction

Part 1 is a review of the fundamentals of the APL language and the features of APL2. If you're new to APL or to APL2, you should read *An Introduction to APL2* before you use this book.

Chapter 1. APL2 — What Is It?

APL2 is an IBM licensed program that processes APL language requests. APL is a general-purpose language that is used in applications as different as business data processing, system design, mathematical and scientific computing, and the teaching of mathematics. APL has been particularly useful in data base applications, where its computing power and communication features have helped make application programmers and end users more productive.

The language that APL2 accepts is an extension of the language that previous APL products accepted. For instance, an array (the basic unit of data in APL) can mix numbers and characters. Previously an array could have either numbers or characters, but not both. This feature simplifies commonly performed tasks such as displaying tables of data with text headings. Furthermore, APL2 allows the pieces of data in an array, the **array items**, to be arrays themselves. This allows programmers to request operations on complicated collections of data (for instance, part inventory records) in the same way that they request operations on more uniform collections of data. In fact, APL2 is designed to help users concentrate on the result they want an operation to accomplish, and to free them from concern for the structure of the data.

These features and others make APL2 well suited not only to the professional business programmer and the advanced scientific and technical user, but also to the occasional user with little or no previous experience with computers.

Chapter 2. Fundamentals

APL2 Is Interactive

APL2 takes one APL statement at a time, executes it, and then proceeds to the next line. Contrast this sequence with traditional program compilers that convert complete programs to machine language before executing any statements. This allows you a high degree of interaction with the computer. If something that you enter is invalid, you will get quick feedback on the problem before you proceed further.

APL2 and You Take Turns at the Terminal

During an APL2 terminal session, you and APL2 take turns using the terminal. While you type in information, APL2 waits for some signal from you that it is its turn to use the terminal to display results. This signal is the pressing of the ENTER key, the execute key, or the carriage-return key. The name of the key differs between different types of terminals, but the action is the same: It's merely a means of telling APL2 that you've finished typing a line, and that you're ready for it to evaluate that line.

When APL2 displays information for you, it starts each new line at the left margin. After it finishes displaying any such output, it signals that it is ready for you to type in another keyboard input by spacing in six spaces from the left margin and halting. This indented position indicates that “it's your turn.” For example:

```
      2+2                    You typed this statement in
4                            APL2 executed the request
      ←                       Now you can enter something else
```

Characteristics of Terminals Used with APL2

Figure 2 on page 4 shows the keyboard of a typical terminal used with APL2. Terminals that are used with APL systems include a variety of typewriter-like and display-tube devices. Their characteristics vary, but the essential common characteristics are:

- A way of entering and displaying APL characters
- A way of signaling the completion of an entry (and release to the system)
- A way of revising an entry before it's released
- A way to suspend execution at the end of an entry and within an entry
- A cursor (some form of pointer) to show where on the line the next character entered will appear

All examples in this manual are presented as they would appear on an IBM* 3270-series display terminal, *with one exception*: Alphabetic characters are shown in this book in italic font. This is to help you differentiate between explanatory text and characters you might see on your terminal screen, even though your terminal might display upright block characters instead of italics. Here's an example:

```
      A←'ONE1' 'TWO2' 'THREE3'
      A
ONE1 TWO2 THREE3
```

Figure 2. An IBM 3278/3279 APL Keyboard

What Can be Entered and What Gets Displayed

Here's a typical entry in APL2:

```
AREA←3×4
```

The effect of this entry is to assign to the name *AREA* the value of the expression 3×4.

In APL2, a combination of characters, numbers, and symbols that represents something to be evaluated is called an **expression**. Thus, *AREA*←3×4 is an expression (yes, even though it has another expression — 3×4 — within it). Expressions can be, optionally, preceded by a **label** and be, optionally, followed by a **comment**, like this:

```
TEST:AREA←3×4      ⍝ A SIMPLE TEST
```

This line is termed a **statement**. Because the label and comment are optional, calling *AREA*←3×4 a statement is correct.

In any case, the statement may be read informally as "*AREA* gets three times four."

A statement may or may not display a result. For example, the following statement displays a result:

```
      3×4
12
```

But an expression assigned to a name won't display a result:

```
      AREA←3×4
←──────────────No displayed result
```

The APL2 Character Set

The characters that can be used in APL2 are shown in Figure 3.

Figure 3. The APL2 Character Set

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

¨	dieresis	α	alpha[1]	⍭	down caret tilde	∨	~
¯	overbar	⌈	up stile	⍲	up caret tilde	∧	~
<	less	⌊	down stile	⍛	del stile	∇	|
≤	not greater	_	underbar	⍙	delta stile	∆	|
=	equal	∇	del	⍗	delta underbar	∆	_
≥	not less	∆	delta	⍜	circle stile	○	|
>	greater	∘	jot	⍉	circle slope	○	\
≠	not equal	'	quote	⊖	circle bar	○	-
∨	down caret	⎕	quad	⊛	circle star	○	*
∧	up caret	(	left paren	⍪	down tack up tack[1]	⊥	⊤
-	bar	)	right paren	⍫	del tilde	∇	~
÷	divide	[	left bracket	⍦	down tack jot	⊥	∘
+	plus	]	right bracket	⍤	up tack jot	⊤	∘
×	times	⊂	left shoe	⍀	slope bar	\	-
?	query	⊃	right shoe	⌿	slash bar	/	-
ω	omega[1]	∩	up shoe	⍝	up shoe jot	∩	∘
∈	epsilon	∪	down shoe	⍞	quad quote	⎕	'
⍴	rho	⊥	down tack	!	quote dot	.	'
~	tilde	⊤	up tack	⍁	quad divide	⎕	÷
↑	up arrow	|	stile	⍂	quad slope[1]	⎕	\
↓	down arrow	;	semicolon	⍜	quad jot[1]	⎕	∘
⍳	iota	:	colon	[]	left bracket right bracket[1]	[	]
○	circle	,	comma	≡	equal underbar	=	_
*	star	.	dot	≤	epsilon underbar	∈	_
→	right arrow	\	slope	⍳	iota underbar[1]	⍳	_
←	left arrow	/	slash	⍨	dieresis dot[1]	¨	.
			blank (space)	%	percent[1] [2]	/	÷
				&	ampersand[1] [2]	|	∈
				¢	cent[1] [2]	|	⊂
				$	dollar[1] [2]	S	/
				#	pound[1] [2]	N	=
				@	at[1] [2]	Q	∘
				!	exclamation[1] [2]	'	∘
				|	vertical bar[1] [2]	|	⊥
				~	tilde[1] [2]	|	~
				¬	not[1] [2]	/	~
				¦	split bar[1] [2]	;	'
				"	double quote[1] [2]	¨	'
				{	left brace[1] [2]	-	(
				}	right brace[1] [2]	-	)
				\	backslash[1] [2]	\	|
				`	accent[1] [2]	\	¯

Note: The lowercase alphabetics ("*a*" through "*z*") may be typed as "*A*" overstruck with "¯" through "*Z*" overstruck with "¯," respectively.

All overstrike combinations may be entered in either order.

[1]These characters have no assigned purpose, other than use as decorators.

[2]National-use characters may have alternate graphics in different countries, although they do ***not*** have alternate overstrikes.

The characters fall into four main classes: alphabetic, numeric, special, and blank.

The alphabetic characters include the Roman alphabet in uppercase italic font, the same alphabet underscored, plus ∆, and ⍙.

The numeric characters are 0 through 9.

In general, most of the special characters (such as +, -, ×, and ÷) are used to denote operations that have fixed meanings. The alphabetic characters, by comparison, are used to construct names, which may be assigned and reassigned meanings. The blank separates adjacent names.

Lowercase characters are special characters, and are thus not used in APL2 names.

Most of the APL2 characters in Figure 3 on page 5 correspond to single keys on an APL keyboard; some do not. For instance, ⍙ is entered at an IBM 2741 communication terminal by pressing the Δ key, the backspace key, and the | key.

Figure 3 also lists the **overstrike characters**. The overstrike combinations may be entered in either order, with a backspace in between. On an IBM 3270 terminal, you can't overstrike a character in the sense of superimposing one character above the other. If you try, you'll merely replace one character with the other. Fortunately, you can get most of the overstrike characters without actually overstriking; for instance, A̲ and ⍤ correspond to single keys on the keyboard.

On some display terminals, there may be no way to enter the overstrike characters. Therefore, APL2 allows you to designate the underscore (_) as a backspace character in the context of these characters. For example, you can enter the **equal underbar** character (≡) by pressing the = key, the _ key (for a backspace), and the _ key again (this time, for an underbar).

When you start a session with APL2, the backspace character is on. You can make sure of this by entering the following system command:

```
      )PBS
IS _
```

If you see the response is *IS OFF*, you can turn the backspace character back on by entering:

```
      )PBS ON
```

Now, if you want to check whether array A is the same as array B, you can enter:

```
      A=__B
1
```

You use)*PBS* to enter the 10 overstrike characters even if you are on a terminal that uses program symbol sets. However, if you use the symbol set provided by APL2, the session manager will display the correct character. Thus, if you enter the expression above, it will change to the following when you press the ENTER key:

```
      A≡B
1
```

Review of Fundamentals

In the following sections, we will review some of the more important concepts in APL2. You should already know most of the pieces you will need to assemble the application in Part 2: forming names and arrays, applying functions, evaluating expressions, defining and executing user functions, as well as the definitions of the primitive operations.

Numbers

All numbers entered or displayed are in decimal. They can be in **conventional form** (including a decimal point if appropriate):

```
      257
257
      0.346
0.346
```

or in **scaled form**:

```
      2.57E2
257
      346E¯3
0.346
```

In similar fashion, APL2 accepts and displays complex numbers, with a J separating the real and imaginary parts. For example, the square root of negative one can be entered or displayed as:

```
0J1          Standard form
```

Optionally, a polar form is available for entry, with the angle expressed in either radians or degrees. In polar form, the square root of negative one can be entered as:

```
      1R1.570796327    Polar radian form
0J1
      1D90             Polar degree form
0J1
```

The MATHFNS workspace, distributed with APL2, contains functions that provide displays for radians and degrees.

Furthermore, either the real or imaginary part of a complex number can be entered in scaled form:

```
1.2E5J¯4E4
8E3D1E2
```

Notice that APL2 does not display complex numbers in polar form.

Negative numbers are represented by an overbar immediately preceding the number:

```
      ¯257
¯257
      ¯346E¯3
¯0.346
```

The overbar can be used as part of a numeric constant and is distinguished from the bar that denotes negation, as in `-X`. The overbar may *not* be used to denote negation of a value assigned to a name; that is, `¯X` is invalid.

Functions

The word “function” comes from a word that means to execute or to perform. A **function** applies to some data (its **arguments**) and transforms it into new data (its **result**).

A function that applies to one argument (one array) is called a **monadic function**. For example, the function **interval** (represented by the **iota symbol**) applied to the number 7 produces a list of the first seven integers.

```
      ⍳7
1 2 3 4 5 6 7
```

A function that applies to two arguments (two arrays) is called a **dyadic function**. For example, **multiply** applied to two lists produces a new list.

```
      2 3 4 × 10 20 30
20 60 120
```

Functions that are denoted with symbols are called **primitive functions**. You may also define your own functions to do computations not provided by the primitive functions.

Operators

You can change the normal action of a function by applying an **operator** to it. For example, `+` and `×` are primitive functions; applying the **reduce operator** (`/`) to produce `+/` and `×/` modifies their normal operation in a precise, defined manner, and produces a new, **derived function**:

```
      2+3 4 5        Add two to each of the items 3 4 5
5 6 7

      2++/3 4 5      Add two to the sum of 3 plus 4 plus 5
14
```

The value of an operator is that you may use it to produce a whole set of related derived functions, each of which is applied in precisely the same way. For instance, in the above example, `+/` produced the function **summation**, which, when applied to the vector `3 4 5`, produced the sum `12`. This was the equivalent of `3+4+5`.

When the reduce operator is applied to other functions, the same definition holds. For example:

```
      ×/3 4 5   Product
60
      ⌈/3 4 5   Largest
5
```

The derived function produced by an operator may be used as the operand of another operator. For example, the **each operator** (¨) may be applied to the **summation function** (+/) to give another derived function that sums each vector in a collection of vectors.

```
      +/¨ (3 4 5) (9 8 7 6)
12 30
```

Another useful operator is **outer product**. It applies a function between all combinations of items, one from the left argument and one from the right argument. For example:

```
      (⍳3) ∘.×⍳4
1 2 3  4
2 4 6  8
3 6 9 12
```

Operators that are denoted with symbols are called **primitive operators**. You may also define your own operators. This is new in APL2 and is an extremely powerful facility. Following is a **defined operator** that simulates the **reduce function**:

```
[0]  Z←(F RED) R
[1]  Z←F/R

    + RED 2 3 4
9
```

Remember:

- A function applies to one or more data objects and returns a data result.
- An operator applies to one or more data objects or functions and returns a function.

Defined operators are covered in more detail in Part 2.

Arrays

An **array** is an ordered collection of data. Each item of data can be a number, a character, or another array. For example, all the following are arrays:

```
2 3 5 7 9

ADAMS 5  8 1 3 2

NAME    DAY1 DAY2  DAY3
JONES   Y    N     N
SMITH   N    Y     Y
TAYLOR  Y    Y     Y
```

The Structure of Arrays

Arrays have structure; the data in an array is ordered along zero or more directions, called **axes**. The number of axes that an array has is its **rank**. For example, a simple list of numbers has only one axis and therefore is of rank one:

```
        V
2  3  5  7  11  13  17  19
```

Data in a list form like this is referred to as a **vector**.

An example of a rank-two array would be a table of numbers:

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

Two-dimensional data like this is referred to as a **matrix**.

Either of these examples could as easily have used character data, or a mixture of numeric and character data. For instance:

```
        MIX
A  B   C   D
1  2   3   4
9  E  10  11
```

A **simple scalar** is a single number or character; it has no axes and is of rank 0:

```
        S←5
        S
5

        N←'R'
        N
R
```

Arrays range from these dimensionless scalars to multidimensional arrays — arrays of rank two or more.

The **shape** of an array is the number of data items in each axis of the array. You can measure an array's shape by using the **shape function**, denoted by the ρ (rho) symbol:

```
        V
2  3  5  7  11  13  17  19

        ρV
8

        A
ABCDEFGH

        ρA
8
```

In a similar way, you can the measure the **rank** of an array by counting the number of numbers that are returned in the shape. In other words, you can measure the shape of the shape:

```
      ⍴V       Shape of V
8

      ⍴⍴V      Rank of V
1
```

Because a vector has one axis, its shape is one number — actually, a one-item vector. The shape of a matrix, because it has two axes, is a two-item vector:

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

      ⍴N
3 4

      ⍴⍴N
2
```

Notice that the last item (that is, the rightmost item) of the shape vector is the number of columns in the array, and the next-to-last item is the number of rows.

In all the previous examples, the data items in an array have been simple scalars, that is, single numbers or characters, but they don't have to be:

```
      NAMES
 ADAMS 5  8 1 3 2

      ⍴NAMES
3

      ⍴⍴NAMES
1
```

In this example, the first item of *NAMES* is the character vector *ADAMS*, the second item is the scalar `5`, and the third item is the numeric vector `8 1 3 2`. Such an array is called a nested array. A **nested array** is an array in which one or more data items is not a simple scalar, If all the items are single numbers or characters, (as they are in array *N* above), the array is called a **simple array**.

In general, it is difficult to tell what the exact structure of a nested array is from its formatted output. APL2 provides a defined function (the **display function**) that gives a picture of an array. To get the display function, issue a command of the form:

```
)PCOPY n DISPLAY DISPLAY
```

(The number *n* is usually `1`, but your system administrator might have designated another number.) For more information about the display function, see *APL2 Programming: Language Reference*.

Let's display the structure of *NAMES*:

```
      DISPLAY NAMES
 .→--------------------.
 | .→----.   .→------. |
 | |ADAMS| 5 |8 1 3 2| |
 | '-----'   '~------' |
 'ϵ--------------------'
```

The display shows each vector item enclosed within a box. The outer box represents the entire vector *NAMES*. The → in the display indicates that what is in the box is a vector, the ϵ indicates that the array contains at least one item that isn't a simple scalar, and the ~ indicates numeric data.

Here's another example of an array whose data items aren't simple scalars:

```
      ATTENDANCE
 NAME   DAY1 DAY2 DAY3
 JONES  Y    N    N
 SMITH  N    Y    Y
 TAYLOR Y    Y    Y

      ρATTENDANCE
4 4

      ρρATTENDANCE
2
      DISPLAY ATTENDANCE
 .→-------------------------------.
 ↓ .→---.   .→---. .→---. .→---. |
 | |NAME|   |DAY1| |DAY2| |DAY3| |
 | '----'   '----' '----' '----' |
 | .→----.                       |
 | |JONES|   Y      N      N     |
 | '-----'   -      -      -     |
 | .→----.                       |
 | |SMITH|   N      Y      N     |
 | '-----'   -      -      -     |
 | .→-----.                      |
 | |TAYLOR|  Y      Y      Y     |
 | '------'  -      -      -     |
 'ϵ-------------------------------'
```

The ↓ in the display indicates that *ATTENDANCE* is a matrix. Notice that each item in the first row of *ATTENDANCE* is a character vector and each item in the first column is a character vector. The _ under the *Y*s and *N*s indicates that these are scalar characters at the same depth as the vectors *JONES*, *SMITH*, and *TAYLOR*.

The degree of nesting of an array is called its **depth**, and you can measure it with the **depth function**, denoted by ≡. A simple scalar has a depth of zero:

```
      ≡5
0
```

A simple array (that isn't a scalar) has a depth of one:

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

      ≡N
1
```

If the deepest item in an array has a depth of 1, the array has a depth of 2:

```
      ATTENDANCE
 NAME   DAY1  DAY2  DAY3
 JONES  N     Y     N
 SMITH  Y     N     Y
 TAYLOR Y     Y     Y

         ≡N
   2
```

A depth of 3 indicates that the item of greatest depth in the array has a depth of 2. For example:

```
      STOCK
 ITEM   DETAIL1           DETAIL2
 BOLT    118  4.7 5.2      119  4.95 5.25
 FRAME   729  22.8 24       730  23.5 24
 TUBE     23  1.25 1.25     25  1.3 1.35

      ≡STOCK
3
```

Because you can't determine depth from what is printed, let's use the display function to show the structure of *STOCK*:

Notice that the second, third, and fourth rows of *STOCK* include items that are depth-2 vectors. For instance, the second item in the *BOLT* row is a nested vector. The first item in the nested item is `118`, and the second item is the vector `4.7 5.2`.

Note: An easy way to tell the depth of an item is to count the minimum number of lines an arrow would need to cross to reach the item from the outside of the display. For example, an arrow would cross two lines to reach *BOLT*; therefore, it is depth 2. An arrow would cross three lines to reach `1.25 1.25`; therefore, it is depth 3.

Forming Arrays

A vector of length two or more may be formed by listing the arrays that are its items. The items must be separated by a blank or a parenthesis.

```
2 3 5 7              Four numbers
2 'B' 5 'D'          Numbers and letters
'A' 'B' 'C' 'D'      Four letters
```

If every item of a vector is a single character, the vector may be written with a single pair of enclosing quotation marks. Thus the last example above may be written:

```
'ABCD'
```

Any item of a vector may be the result of a computation as well as a constant. For example, the following two vectors are equivalent, and one may be used wherever the other is used:

```
      2 3 (2+3) 7
2 3 5 7
```

Any item of a vector may be another vector or any other array. Here is a vector containing vectors:

```
      (2 3 5) (7 11 13)
2 3 5  7 11 13
```

To form a vector of length one or zero, or an array of rank greater than one, you must apply some functions. The most fundamental function is **reshape** (⍴), which produces an array of requested shape containing the given items. Here's a simple matrix of numbers:

```
      N←2 3 ⍴⍳6
      N
1 2 3
4 5 6
```

Here's a nested matrix containing a simple matrix and a vector:

```
      M←2 2 ⍴1 'X' N (3 4)
      M
      1       X

1 2 3         3 4
4 5 6
```

Given a matrix you can turn it into a vector. The function **ravel** (**,**) produces a vector having the same number of items as its argument:

```
      ,M
1 X    1 2 3  3 4
       4 5 6
      ⍴,M
4
```

The function **enlist** (∈) produces a vector containing all the simple scalars from its argument:

```
      ∈M
1 X 1 2 3 4 5 6 3 4
      ρ∈M
10
```

A nested vector may be turned into a matrix by using the **disclose** (⊃) function:

```
      ⊃(1 2 3) (4 5 6)
1 2 3
4 5 6
      ρ⊃(1 2 3) (4 5 6)
2 3
```

Sometimes the display of a nested vector is too wide to fit properly on the screen. The function **ravel with axis** may be used to turn the vector into a one-column matrix. Example:

```
      V←'A' 'VERY' 'LONG' 'NESTED' 'VECTOR'
      V
A VERY LONG NESTED VECTOR

      ,[⍳0]V
A
VERY
LONG
NESTED
VECTOR
```

In the application in Part 2, you will find several one-column nested matrices. They are represented this way because of the way they display. In a real application, you might leave them as vectors.

Building a Data Structure

You've just seen a few ways to build arrays that assume you already have the data somewhere; the data was merely reshaped, disclosed, or rearranged in some way. Here are a couple of ways to build an array from input supplied from the terminal.

The code you'll see is designed for your personal use — because you never make mistakes! Functions designed for people who *do* make mistakes would need more error checking and trapping like those discussed in Chapter 5, “Writing Input and Error-Checking Routines” on page 31. We'll discuss here some techniques for building a data structure. Each begins from a structure having no data, and then progressively adds data based on entries from the keyboard.

In this book, and in many applications, the data to be represented is organized in tables where each column (or set of columns) represents similar data about related line items. Each row of the table contains all data associated with one line item.

For example, if the table contained banking data, each row could represent one account, and the columns could represent current balance, customer name, and so forth. Such a table is sometimes called a **relation**.

Although the techniques shown below may be used for structures more general than these, we will concentrate on **relational data**.

Let's suppose we want to build the *STOCKS* matrix used in Part 2 of this book. It is a five-column array of numbers, except for column 2, which is made up of character vectors.

Building a Data Structure Using Catenation: We begin by specifying an empty array that defines the column structure of the *STOCKS* array. An **empty array** is an array that has a shape (for instance, it's a vector or a matrix), but has no length in at least one of its dimensions. This happens when you specify at least one of its axes as length zero.

Here's an example:

```
      STOCKS←0 5⍴0 ' ' 0 0 0
```

STOCKS now exists in the workspace.

```
      STOCKS
(blank)
      ⍴STOCKS
0 5
      ⍴⍴STOCKS
2
```

It's empty

Has five columns, but no rows

And it's a matrix

You might wonder how this empty matrix differs from this one:

```
      STOCKS←0 5 ⍴0
```

The answer is — no difference at all. The reason we recommend the first form is that it documents the intended use of the columns. Also, you can write functions that use the column types and they will work even when the table is empty.

Now that *STOCKS* exists, you can add rows to it, but because the empty *STOCKS* has no rows, the first row you add will be the first row in the array:

```
      STOCKS←STOCKS,[1]1234 'A THING' 5.25 100 20
      STOCKS
1234 A THING 5.25 100 20
      ⍴STOCKS
1 5
```

You could enter more statements like this to add more rows to the matrix, but it would be a time-consuming process involving a lot of typing.

Instead, you could write a small function to read input and catenate it to the matrix. In addition to needing less typing, the function could do some error checking.

Building a Data Structure Using a Defined Function: Here's a simple function that you can use to read in lines from the keyboard:

```
    ∇ Z←EVALIN;X
[1]   ⍝ ACCEPT INPUT LINES UNTIL AN EMPTY LINE IS INPUT
[2]    Z←''
[3]   L1:Z←Z,⊂X←⍞          ⍝ ACCEPT INPUT ; APPEND TO RESULT
[4]    →(0≠⍴X)/L1          ⍝ REPEAT IF INPUT NOT EMPTY
[5]    →(0=⍴Z←¯1↓Z)/0      ⍝ DROP EMPTY ENTRY ; EXIT IF NO INPUT
[6]    Z←⊃⍎¨Z              ⍝ EVALUATE AND RETURN A MATRIX
```

The last line of *EVALIN* evaluates each character input line with ⍎ and creates a matrix with ⊃.

Here's how *EVALIN* would look in use:

```
      STOCKS←0 5⍴0 ' ' 0 0 0              ⍝ FIRST TIME ONLY
      STOCKS←STOCKS,[1]EVALIN             ⍝ ALL SUBSEQUENT ADDITIONS
1135 'FIRST GREAT ITEM' 9.95 118 55
9993 'HIGH FLYER WIDGET' 88.73 240 35
3569 'SECOND MONEYMAKER' 24.75 0 30
      (Empty input)
```

Now we have a matrix that has the required data. It is 5 columns wide:

```
      ⍴STOCKS
 3 5
      DISPLAY STOCKS
┌→──────────────────────────────────────┐
↓      ┌→──────────────┐                │
│ 1135 │FIRST GREAT ITEM│  9.95  118 55 │
│      └───────────────┘                │
│      ┌→─────────────┐                 │
│ 9993 │HIGH FLYER WIDGET│ 88.73 240 35 │
│      └─────────────┘                  │
│      ┌→─────────────┐                 │
│ 3569 │SECOND MONEYMAKER│ 24.75 0  30  │
│      └─────────────┘                  │
└∊──────────────────────────────────────┘
```

Building a Data Structure Using EDITOR 2: APL2 EDITOR 2 can be used as a full screen entry program. EDITOR 2 can edit only character arrays, so let's start (as usual) by creating an empty character matrix:

```
TAB←0 0 ⍴''
```

Notice that in this case we don't worry about the number of columns. EDITOR 2 will expand the number of columns as necessary.

You invoke the editor by entering:

```
      ∇TAB
```

Now you can use all the power of the editor to add, delete, change, or rearrange lines. Your screen would look like this:

```
[⍝]∇ TAB.2  ⍴: 0 0
[0]    TAB
```

After you enter 3 lines, it would look like this:

```
[⍝]∇ TAB.2  ⍴: 3 37
[0]    TAB
[1]     1135 'FIRST GREAT ITEM' 9.95 118 55
[2]     9993 'HIGH FLYER WIDGET' 88.73 240 35
[3]     3569 'SECOND MONEYMAKER' 24.75 0 30
```

When you close definition, the variable *TAB* has a character matrix representing what you last saw on the screen.

We can now use a function to turn the character matrix into the data we want:

```
∇ Z←EVAL M
[1] ⍝ EVALUATE EACH ROW OF A MATRIX
[2] Z←⊃⍎¨⊂[1+⎕IO]M

STOCKS←EVAL TAB
```

Or, because *EVAL* is so short, we can say:

```
STOCKS←⊃⍎¨⊂[1+IO]TAB
```

Selecting the Items of an Array

If an array is an ordered collection of data, how do you select individual items in an array? In APL2, you can select items in an array by specifying the position of the items in brackets. This is called **bracket indexing**. For example:

```
      V←2 3 5 7 11 13 17 19
      V[3 1 5]                Selecting items from a simple array
 5 2 11

      X←3 1 5
      V[X]
 5 2 11

      ((2 3 5)(7 11) 13 17 19)[3 1 5]
 13  2 3 5  19                Selecting items from a nested array
      A←'LINE' 'A' 'XZ' 'NEW'
      A[2 4 1]
 A NEW LINE
```

Notice that the position indicators are simple arrays. For arrays of rank two or higher, APL2 needs more than one position indicator to identify the position of individual items. In these cases, the **index** (that is, the arrays in the brackets) is a

composite of the item's position along each axis. The last array in the index identifies the position of the item in the column axis, the next-to-last array in the index identifies the position in the row axis, and so on. The arrays in the index are separated by semicolons. For example:

```
      V←2 3 5 7 11 13 17 19

      M←2 4⍴V

      M
 2  3  5  7
11 13 17 19

      M[2;3]
17

      M[2 1;2 3 4]
13 17 19
 3  5  7

      ⍴M[2 1;2 3 4]
2 3
```

and:

```
      T←2 2 3⍴'ABCDEFGHIJKL'

            T
      ABC
      DEF

      GHI
      JKL

          T[2;1;3]
      I
          T[2;1 2;1 2 3]
      GHI
      JKL

          ⍴T[2;1 2;1 2 3]
2 3
```

Notice that a composite index selects all the items along the pertinent axes. Thus, *M*[2 1;2 3 4] selects the items in row 2, columns 2, 3, and 4, and in row 1, columns 2, 3, and 4.

Also notice that the shape of the result in these examples is the same as the shape of the indexes. This is true of bracket indexing in general. Consequently, bracket indexing gives you a way of making results conform to the shape you want:

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

      '∘□'[1+N>6]
∘∘∘∘
∘∘□□
□□□□
```

The previous example points out another thing about bracket indexing: The index values can be repeated. In the previous example, APL2 evaluates the expression for each item of *N*. For items less than or equal to 6, the expression becomes '∘□'[1], so that the ∘ is selected. For items greater than 6, the expression becomes '∘□'[2], so that □ is selected.

One last word on this subject: Items can't be selected from a scalar through bracket indexing, because a scalar has no axes from which to select its data.

Selecting from a Nested Array

The indexing you've just seen selects some subset of the items of an array. If you want to get inside the items, you can use the function **pick** (⊃). Example:

Suppose you wanted to select the 7. That 7 is item 3 of a vector; that vector is at row 2 column 3 of a matrix; and that matrix is at row 2 column 1 of *P*. This information is packaged into the left argument of pick as follows:

```
      (2 1) (2 3) (3)⊃P
7
```

Notice that 7 is at depth 3, the arrow pointing at the 7 crosses 3 lines, and the left argument of pick is a 3-item vector. In general, a length *n* left argument will pick an item at depth *n*. The vector is sometimes called a **path** to the item. Also notice that, at each level, the rank of the item determines the number of integers needed to pick from it.

Let's look at some other examples. Suppose you wanted to select the entire vector 9 8 7.

Two lines are crossed, so a length-2 index will do the job.

```
      (2 1) (2 3)⊃P
9 8 7
```

If you wanted to select the entire matrix containing the vector, a length-1 index will do. Because there is no notation for writing a length 1 or 0 vector, we must use a function to get one item. Here we use enclose to produce a scalar:

```
      (⊂ 2 1)⊃P
1 2 3
4 5   9 8 7
```

In general, if you want to pick an item from a matrix, you must enclose the row and column index.

The Limiting Case: As long as we've gone this far, let's look at the limiting case. What happens if the left argument of pick is of length zero (an empty vector)? The only line you could draw that crosses no lines must point at the whole array. Thus the following is always true:

```
      A←→(⍳0)⊃A
```

By the way, you can write these expressions on the left of a left arrow and replace an item instead of selecting it. The following example:

```
      ((2 1)(2 3)⊃P)←'ABC'
```

will replace the vector `9 8 7` with a character vector, called **selective specification**.

As you have seen, pick can be used to select a single item at some specified depth. Suppose you wanted to select two items from *P*, as created on the previous page?

```
      I←(2 1) (2 3) (3)
      I⊃P
7
      J←(2 2) (1)
      J⊃P
3
```

You can do that in one operation as follows:

```
      I J ⊃⍤⊂ P
7 3
```

This is an extremely common idiom (it's sometimes called the “chipmunk idiom”). Enclose of *P* (⊂*P*) is a scalar. Pick each is a scalar function, so a scalar argument (the right argument of pick each in this case) extends. Thus this one expression is equivalent to both the previous expressions.

Note: ⊃⍤⊂ cannot be used on the left of a left arrow.

Beyond Fundamentals

In the next part of this book you'll see how you can apply APL2 to meet a data processing need. In particular, you'll learn how you can use APL2 in a business application.

If you want to read more about APL2 fundamentals, read “Fundamentals” in *APL2 Programming: Language Reference*, before you go on to Part 2 of this Programming Guide. That part of the Language Reference gives a comprehensive description of such topics as arrays and evaluating expressions. You'll find that this in-depth coverage of APL2's fundamentals will make it easier for you to follow the application as it develops in the next part of the Programming Guide.

Also consult the Language Reference for detailed information on the following:

- Structure, definition, and execution of defined functions and operators
- The APL2 editors
- Complex arithmetic
- Messages

- System commands
- System functions and variables
- Use of auxiliary processors

Part 2. Creating an APL2 Application

In Part 1 we reviewed some fundamentals of APL2. In this part we'll show how you can apply APL2 to meet a data processing need.

The following chapters present the structure and content of a set of APL2 variables, functions, and operators that you can use to implement a model inventory control application. The introduction of techniques is the sole purpose of the application presentation — it is not intended to present an application that will fulfill the needs of a “real world” inventory control application. For your convenience in following the discussion, the foldout pages inside the back cover of the book show all the functions, variables, and operators used in the sample implementation. We will refer to them as we discuss each part in detail.

You'll get the most out of this application if you try things out at your own terminal as we go along.

As APL2 primitive functions and operators are introduced, they will be shown in **boldface type**. For further information about them, refer to *APL2 Programming: Language Reference*.

Chapter 3. An Inventory Control Application

Imagine that you are an application programmer for a wholesale parts distributor (Jed's Wholesale Parts). Your manager walks into your office and says, “The boss just told me that she wants our customer order and inventory control system automated.” You say, “No problem,” and he quickly leaves before you have a chance to realize what you've just said. Now what do you do?!

After getting over your initial anxiety, you begin to think, “What would an automated order and inventory system do? How might customers and order clerks use it?”

Figure 4. Jed's Wholesale Parts

Using the Application

Before you decide in detail what a customer will do during the ordering of merchandise, you must first define the problem in general terms; then identify the steps required for the application, and discuss the problem with the people who will use it.

Figure 4 shows the layout of the store: The desks that deal with placing orders, filling them, and entering restocking information into the computer system.

A customer walks into the store, looks in the catalog, and fills out an order form. The form includes the customer's name and address as well as information about the items she wants to buy. Figure 5 shows a customer order.

```
                              Jed's Wholesale Parts
                               Customer Order Form
                              ______________________

Name: ______________________________________________________________
Address: ___________________________________________________________
City, State, Zip: __________________________________________________

                                                        Number
        Item                                              of
        Number       Description       Price            Items            Total
        ______       ___________       _____            _____            _____

  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________
  __________________________________________________________________________

                                                         Total Cost: __________
```

Figure 5. The Customer Order

She next takes the form to the order clerk.

When a clerk first enters the inventory control program, he will be prompted for the name of the task he wants to perform:

```
CHOOSE PLACE FILL OR STOCK :
```

PLACE indicates he wants to place an order; *FILL*, that he wants to fill one; and *STOCK* that he wants to restock merchandise.

Placing an Order

This clerk enters *PLACE*. The "place" portion of the program then prompts him for the customer number:

```
CUSTOMER NUMBER:
```

He types in the number:

```
CUSTOMER NUMBER: 55
```

Next, the program prompts him for the item number and quantity, which he types in.

```
ITEM :  5613
AMT ORDERED :  2
ITEM :                ←—— End input; could enter more
```

Figure 6 shows what an invoice would look like if a customer ordered the above item.

CUSTOMER INFORMATION **INVOICE NO**
MAIL HOUSE LTD. 145
711 *RAMBLERS LANE*
ISLAND CITY, S. DAK. 54321

DATE
MAY 25, 1984

ITEM	DESCRIPTION	PRICE	QUANTITY	TOTAL
5613	*MAIL ORDER SPECIAL*	14.99	2	29.98

TOTAL COST 29.98

Figure 6. The Customer Invoice

If the customer is new, the program prompts the clerk for customer information:

```
 CHOOSE PLACE FILL OR STOCK :  PLACE
 CUSTOMER NUMBER :  22                    <------- A new customer

NEW CUSTOMER
  CUSTOMER NAME :  TWENTY TWO
  ADDRESS :  22 TWENTY LANE
  CITY, STATE, ZIP :  TWOTOWN, TENN. 22222
CUSTOMER RECORD UPDATED
  ITEM :

 CHOOSE PLACE FILL OR STOCK :         <------- End input
```

Filling an Order

After getting her invoice from the order clerk, the customer takes it to the clerk who fills orders. That clerk has access to the same Inventory Control Program, and when he gets the invoice, he gets the same initial prompt:

```
CHOOSE PLACE FILL OR STOCK :
```

He enters *FILL*, and the program prompts him for the invoice number:

```
CHOOSE PLACE FILL OR STOCK :  FILL
INVOICE NUMBER: 131
```

Restocking Merchandise

Periodically, a clerk checks the inventory, and orders items the store is in danger of running out of. He enters *STOCK* in response to the initial prompt, and the program prompts him for the stock number:

```
 CHOOSE PLACE FILL OR STOCK :  STOCK
STOCK ITEMS TO CHECK: 7777                    <----Items to check
 ITEM  DESCRIPTION  PRICE  INVENTORY  REORDER LEVEL
 7777  THINGY        1.89           2             1
 LOW ITEMS:   3569                            <----List of low items
STOCK NUMBER: 7777                            <----User selects 7777
 NOW READS:    7777  THINGY 1.89 2 1
INCREMENT: 10                                 <----Add 10 to stock
STOCK NUMBER:                                 <----End input
NEW STOCK ITEMS:
 ITEM :  2222                                 <----Add item 2222
  DESCRIPTION :  TUTU
  PRICE :  22.22
  INVENTORY :  222
  REORDER LEVEL :  22
 ITEM :
```

So there it is — an application that checks the status of a given item, records orders, prints invoices, updates inventory status, and adds new items to the inventory. You're left with the question, “How can I code the application?” Better yet (and not only because this is an APL2 manual) you wonder, “How can I code the application in APL2?”

Overview of the Application

Before you begin to code an application, it is a good idea to have an overview of what it is you are trying to accomplish. One way to do this is by making a list of tasks to be performed. You have already identified the three main tasks:

1. Placing the order
2. Filling the order
3. Restocking merchandise

The three people who will be using your program are the clerk who places the order, the clerk who fills it, and the stock person who maintains the inventory. By talking to those three people, you identify the following expanded application steps:

1. Placing the order

   a. Input

      - Enter customer number.
      - If customer is new, enter customer information.
      - Enter information relating to the order.

   b. Process

      - Verify that stock is available.
      - Put the order into a table.
      - Update the available stock.

   c. Print

      Format the invoice.

      The invoice will be taken to the merchandise area, where the order will be filled.

2. Filling the order

   - Ask for the order number.
   - Delete the order table entry.

You could choose to have the available stock updated at this point rather than when the order was placed, but that would require keeping separate stocks table entries for (or doing calculations for) consigned or unconsigned stock, so that the entry clerk does not consign the same stock twice. The way it will be done, then, is for the stock table to have unconsigned inventory, and the orders table to have the consignments.

3. Restocking merchandise

   - Check for low stock.
   - Restock low items.
   - Enter information about new items.

That should do it for now. You'll begin by writing the application's building blocks on which everything else will be based. Those building blocks (or “tools”) are:

- Tables that define your application world, and that reflect its current state
- Input and error checking routines
- Prompting control operators
- Table updating routines

The next four chapters discuss the building block routines.

Chapter 4. Designing Tables

It appears that there are three types of data that are meaningful in this application:
Data about stock. The data in this category includes —

STOCK NUMBER	DESCRIPTION	PRICE	QUANTITY	REORDER LEVEL

Data about orders. The data in this category includes —

INVOICE NUMBER	CUSTOMER NUMBER	STOCK NUMBER	AMOUNT ORDERED

Data about customers. The data in this category includes —

CUSTOMER NUMBER	NAME	ADDRESS	CITY, STATE, ZIP

It's easy to visualize the input data as three arrays — let's call them *STOCKS*, *ORDERS*, and *CUSTOMERS*. And because it's easy to think of the data as tabular, let's assume that *STOCKS*, *ORDERS*, and *CUSTOMERS* are matrixes.

The STOCKS Array

Now let's examine the content of each array, starting with *STOCKS*. Suppose the matrix looked like this:

```
      STOCKS
1135 FIRST GREAT ITEM    9.95  118  55
2583 A REAL WINNER      49.99   89  10
3569 SECOND MONEYMAKER  24.75    0  30
5613 MAIL ORDER SPECIAL 14.99  225  95
9993 HIGH FLYER WIDGET  88.73  240  35
9998 NONESUCH FRAMMIS    2.69  440  50
```

If *STOCKS* were a simple array, each character in the array would occupy one column; if *STOCKS* were made up exclusively of character data, each row would have 39 columns:

```
      ⍴STOCKS
9 39
```

One problem with processing this structure is that you have to monitor the character positions. If you needed to process the description of an item in *STOCKS* (as you probably would), you'd have to direct APL2 to drop the first 5 columns and then take the next 18 columns:

```
      18↑5↓STOCKS[1;]
FIRST GREAT ITEM
```

But, if *STOCKS* were a nested array, you could avoid all length considerations. If *STOCKS* had a structure like this:

```
       DISPLAY STOCKS
┌→──────────────────────────────────────────┐
↓      ┌→──────────────┐                    │
│ 1135 │FIRST GREAT ITEM│   9.95  118  55   │
│      └────────────────┘                    │
│      ┌→─────────────┐                      │
│ 2583 │A REAL WINNER│      49.99 89   10   │
│      └──────────────┘                      │
│      ┌→─────────────────┐                  │
│ 3569 │SECOND MONEYMAKER│  24.75 0    30   │
│      └──────────────────┘                  │
│      ┌→────────┐                           │
│ 5555 │WHIZBANG│           22.22 43   2    │
│      └─────────┘                           │
│      ┌→───────────────────┐                 │
│ 5613 │MAIL ORDER SPECIAL│ 14.99 225  95   │
│      └────────────────────┘                │
│      ┌→───────┐                            │
│ 7777 │ THINGY│            1.89  2    1    │
│      └────────┘                            │
│      ┌→─────┐                              │
│ 8888 │WHEEE│              4.33  3011 100  │
│      └──────┘                              │
│      ┌→─────────────────┐                  │
│ 9993 │HIGH FLYER WIDGET│  88.73 240  35   │
│      └──────────────────┘                  │
│      ┌→────────────────┐                   │
│ 9998 │NONESUCH FRAMMIS│   2.69  416  50   │
│      └─────────────────┘                   │
└←──────────────────────────────────────────┘
```

you could then pick things out in natural groupings like this:

```
      STOCKS[1;2]
 FIRST GREAT ITEM
```

Notice that if the description of item `1135` were changed to *THE BEST ITEM THERE IS*, you wouldn't have to change the expression you'd use to reference it:

```
      STOCKS[1;2]
 THE BEST ITEM THERE IS
```

Furthermore, nesting *STOCKS* gives it a more natural structure. There are nine stock items, and five columns (not 39) in each row.

The ORDERS Array

ORDERS, on the other hand, can and should be a simple array. Its structure should probably look something like this:

```
        DISPLAY ORDERS
 .→---------------.
 ↓131  55 3569 10|
 |135 312 9998 12|
 |136   7 1135  2|
 |165  55 3569  0|
 '~---------------'
```

The CUSTOMERS Array

So we're left with *CUSTOMERS*. Clearly, this should be nested. Otherwise, your processing would have to account for positioning, just as it would have for a simple *STOCKS* array. It might be structured something like this:

```
      DISPLAY CUSTOMERS
.→-------------------------------------------------------------------------------.
↓     .→-----------------.  .→--------------.      .→------------------.         |
| 7   |CITY TRADERS INC.|  |41 POSTAGE ROAD|      |RIMELA, N.Y. 12345|         |
|     '-----------------'  '---------------'      '------------------'         |
|     .→---------------.    .→-----------------.   .→--------------------------. |
| 55  |MAIL HOUSE LTD.|    |711 RAMBLERS LANE |   |ISLAND CITY, S. DAK. 54321| |
|     '---------------'    '-------------------'   '--------------------------' |
|     .→-----------------. .→--------------------. .→--------------------.       |
| 312 |MANTUP SALES CORP.| |RURALIA FARMS, RFD 2| |SUBURBIA, WIS. 00000|       |
|     '-----------------' '--------------------' '--------------------'       |
'←-------------------------------------------------------------------------------'
```

Chapter 5. Writing Input and Error-Checking Routines

Now that you've decided how the data will be structured, you're going to need a way of getting the initial data into the system and check for input errors.

Naturally, you want the routines to be as “user friendly” as possible. How can you make them that way? One way is to make the prompting for input as natural to the users as possible. Furthermore, you want to make your system forgiving, so you're going to try to anticipate user errors, notify users when they've made an error, and give them a chance to correct it.

Let's organize your thinking into a series of actions that the input handling functions will have to take:

- Prompt for input.
- Check for entry errors (as necessary, issue message and allow users to recover).
- Return the input in a form suitable for processing by table update functions.

Seems fairly straightforward. Maybe that's all the initial planning you have to do for this part of the application. And because you're anxious to begin, why not start coding?

Creating an Input-Handling Function

Let's start with a function that builds the *STOCKS* array:

```
∇INPUT
```

Documenting What the Function Does

When you begin to write a function, the first thing you should do is document what the function does. This information should be in a comment line that appears at the beginning of the function. A comment, which is signified by the **up shoe jot** symbol, (⍝), is used to help describe functions (it's not executed by APL2).[1] In fact, comments at the beginning of a function or to the right of a statement won't slow down execution.

Here's what an opening comment line might look like:

```
[0]    INPUT
[1] ⍝ PROMPT FOR INPUT
```

In essence, this comment is the “abstract” for the function (and, you hope, prevents the remainder of the function from looking abstract!). The abstract line shouldn't exceed 60 to 80 characters. You may find that you can't state the function's purpose in just one line. *Stop, don't merely write a longer one*! That's probably a good indication that you're trying to do too much with that function; narrow its scope a bit and get it down to one operation before you start to write any code.

The header line can also contain a comment:

[1] The up shoe jot symbol is also known as the **lamp** symbol, because it “illuminates.”

```
[0]    Z←AGAIN    ⍝ RETURN 1 TO RETRY
```

As you get into the “nuts and bolts” of the code, you should also document how the function operates. APL2 accepts comments on lines by themselves or to the right of a statement or statement labels, so that you can liberally sprinkle the function with a running commentary. This added effort will pay off when the function is maintained, particularly if the maintainer isn't you.

Prompting for Input

There are two ways you can have a user specify input to an APL2 operation (function or operator). You can have the user supply the input as one or more arguments to an operation:

```
STOCK 1135
```

or have an operation prompt the user for input:

```
      STOCK
ENTER STOCK NUMBER:
```

For users of *STOCK*, prompting is much better. (Remember, you want to shield them as much as possible from the programming system.)

Let's look at some ways to prompt for input. For character input, you'll use **quad quote** (⍞).

```
[0]    Z←INPUT MSG
[1]    ⍝ PROMPT FOR INPUT
[2]    MSG
[3]    Z←⍞
```

If you execute *INPUT* `'ENTER STOCK NUMBER:'`, the user sees:

```
ENTER STOCK NUMBER:
```

and can then respond:

```
ENTER STOCK NUMBER:
1135
```

At *INPUT*`[3]`, *Z* is assigned the character vector 1135. In fact, it's always assigned a character vector; if the user's response to the prompt were `7`, then *Z* would be assigned a one-character vector.

Note that ⍞ input is always accepted as text. If you need to have numbers entered, you'll either have to use the **execute function** (⍎) on the data or use the evaluated input prompt mechanism. The execute function evaluates the expression that a character vector represents. For example:

```
      ⍎'3+4'
7
```

Prompting and Keeping the Response on the Same Line

This is a slight variation of the character input technique:

```
[0]     Z←INPUT MSG
[1]   ⍝ PROMPT FOR INPUT
[2]     ⍞←MSG,': '    ⍝ DISPLAY THE PROMPT
[3]     Z←⍞            ⍝ ACCEPT INPUT
```

When *INPUT 'ENTER STOCK NUMBER'* is executed, the prompt response looks like this:

```
      INPUT'ENTER STOCK NUMBER'
ENTER STOCK NUMBER: 1135
```

For the order clerks, this is probably the most natural-looking dialog.

You might think that the character vector `1135` gets assigned to *Z*, but that's not all. *Z* is also assigned the prompt, or characters that replace the prompt. This depends on the value of the system variable **prompt replacement** ⎕*PR*. If ⎕*PR* is an empty vector, the prompt is assigned as is:

```
      ⎕PR←''
      A←INPUT'ENTER STOCK NUMBER'
ENTER STOCK NUMBER: 1135
      A
ENTER STOCK NUMBER: 1135
      ⍴A
24
```

Otherwise, ⎕*PR* contains a character (the default in a clear workspace is the blank character), and that character replaces the prompt:

```
      ⎕PR←' '
      A←INPUT'ENTER STOCK NUMBER'
ENTER STOCK NUMBER: 1135
      A
                  1135
      ⍴A
24
```

It's worth noting that *INPUT* in its current form depends on the global variable of ⎕*PR* to work properly. In actual practice, it may be set to some other value in whatever workspace this function gets used. If its value is other than a blank, this function could start to fail in unusual situations. For this reason, it's always smart to localize ⎕*PR* to the prompting function. That way you can guarantee a proper setting of ⎕*PR*.

Doing Without the Prompt Message

What you really want to do is to have ⍞ return *only* what the user enters. You can do this by replacing the prompt by some character that can be excluded from the input stream, such as backspace or ENTER, and then removing that character with the **without** (~) primitive function:

```
[0]    Z←INPUT MSG;⎕PR
[1]  ⍝ PROMPT FOR INPUT
[2]    ⎕PR←↑⎕TC          ⍝ BACKSPACE IS PROMPT REPLACE
[3]    ⍞←MSG,': '        ⍝ DISPLAY THE PROMPT
[4]    Z←⍞~⎕PR           ⍝ ASSIGN THE RESPONSE
```

In the above illustration, ⎕*PR* is set to a backspace character selected from the system variable **terminal control** (⎕*TC*).

The expression ⍞~⎕*PR* is then used to remove all backspaces. Now you will get the correct result even if the user replaces part of the prompt:

```
      A←INPUT'ENTER STOCK NUMBER'
ENTER STOCK NUMB1135
      A
1135
      ⍴A
4
```

Deleting Excess Blanks

To complete *INPUT*, you delete any excess blanks entered before, between, or after words:

```
[5]   Z←DLTMB Z       ⍝ DELETE EXCESS BLANKS
```

Because this is something that is generally useful, you use a separate function to do it:

```
[0]   Z←DLTMB A
[1]  ⍝ DELETE LEADING, TRAILING, MULTIPLE BLANKS
[2]   Z←' ',A,' '
[3]   Z←(~' '  '⍷Z)/Z
[4]   Z←1↓¯1↓Z
```

Line [2] of *DLTMB* ensures that *Z* has blank "book ends."

DLTMB[3] uses the **find** (⍷) primitive function to find the start of all occurrences of two successive blanks. The resulting boolean vector is then applied as the left operand of the / operator, producing the **compress** (B/) derived function (see Figure 7).

In *DLTMB*[4], the book end blanks are dropped.

Here's a test of *DLTMB*:

```
      DLTMB ' IT  IS   ALWAYS  BEST  TO   PLAN   AHEAD'
IT IS ALWAYS BEST TO PLAN AHEAD
```

Language Note: The Compress Derived Function

If *A* is an array, and *B* is a boolean vector (of the same length as the last axis of *A*), then the expression *B/A* is a derived function of the slash (/) operator. It removes an item of the last axis of *A* wherever a 0 appears in the corresponding position of *B*. This derived function is called *COMPRESS*. For further information about *COMPRESS*, see *APL2 Programming: Language Reference*.

Figure 7. Language Note: The Compress Derived Function

Testing for the Limiting Case

Before you finish writing the *INPUT* section, you should test for the limiting case: What happens if the user responds with the ENTER key only?

```
      A←INPUT 'STOCK'
STOCK:  ◄──────────────── User presses ENTER
      ⍴A
0
```

Just what you wanted — an empty array. This is a useful way to tell if the user no longer wants to enter data.

Because you deleted excess blanks, you'll get the same effect if a row of blanks is entered. You decide this is what you want for this application.

Prompting for Numeric Input

Now that you have a neat little input routine for character data, what do you do about numeric data entry? You want a numeric input function that will protect your users from their entry errors. Your approach is to create a separate function (*NUMIN*) that calls *INPUT*:

```
[0]   Z←NUMIN MSG
[1]  ⍝ PROMPTS FOR POSITIVE NUMBERS
[2]  L1:Z←INPUT MSG        ⍝ PROMPT FOR INPUT
[3]   →(0=⍴Z)/0            ⍝ EXIT IF INPUT IS EMPTY
```

To get valid positive numbers, you must ensure that only the characters that can be components of an acceptable number are allowed:

```
[4]  ⍝ CHECK FOR VALID NUMERICS AND CONVERT
[5]   Z←(∧/Z∈'0123456789. ')/Z  ⍝ NUMERICS ONLY
```

Your list of acceptable numbers excludes negative, complex, and 'E' notation entries, such as -5, 5J0, 1E5. It includes numbers like 5 and 5.0. Including a blank allows more than one number to be entered. If the result of a call to *INPUT* contains *any* character that is not in the allowed set ('0123456789. '), then *Z* becomes an empty vector. Notice that if ⍴*Z* is *really* zero, then an exit is taken at *NUMIN*[3].

One Final Word: Keep Your Prompts Short

Enough said.

Handling Errors

NUMIN[6] can use execute (⍎) to convert *Z* from characters to numbers. You want to signal an error if *Z* is zero at *NUMIN*[6]. Any strange constructions such as 5.5.5 must also be trapped. You can accomplish both by using **execute alternate** (⎕*EA*):

```
[6]   '→L2' ⎕EA 'Z←⍎Z'      ⍝ EVALUATE
[7]   →0                    ⍝ EXIT IF OK
[8]  L2:⎕←'INVALID NUMBER'
```

If *Z*←⍎*Z* succeeds, an exit is taken at *NUMIN*[7]. The explicit result of *NUMIN* is *Z*, which now contains one or more valid positive numbers.

If *Z* contains an error, the left argument of ⎕*EA* is executed, and a branch is taken to *NUMIN*[8].

Language Note: Handling Events

Employ caution in using ⎕*EA* and **event simulation** (⎕*ES*). In particular, it is important to avoid using ⎕*EA* with the assumption that it will trap only a particular error, such as a ⎕*ES*-generated one. Instead, unexpected errors may be masked.

One way to avoid this: Before continuing, check the value of ⎕*ET* to ensure that the error is one of the errors you want to trap. Dyadic ⎕*ES* is useful in this respect, as you can have it set ⎕*ET* with your own error number.

For more information about ⎕*ES* and ⎕*ET*, see *An Introduction to APL2*.

Figure 8. Language Note: Handling Events

Trying *AGAIN* (and *AGAIN*, and....)

You want to give the user a chance to try again, if at all possible. You do this by printing a message and calling *AGAIN*:

```
[9]    →AGAIN/L1                ⍝ RETRY IF DESIRED
```

NUMIN[8] prints a message at the terminal by assigning it to ⎕. If you someday need to send the messages to a printer instead, you will replace ⎕← with a *PRINT* function reference.

What does *AGAIN* do? It prompts the user to see if that user wants to retry, and, if so, returns a 1:

```
       A←AGAIN
RETRY(Y/N)?
Y
       A
1
```

Preventing an Endless Prompting Loop

While coding *AGAIN*, you decide to be extra cautious about the possibility of getting into an endless prompting loop. This can happen if *WS FULL* occurs at *NUMIN*[6]. To protect against this, you check the event type (⎕*ET*) code set by the last error to occur:

```
[0]    Z←AGAIN    ⍝ RETURN 1 TO RETRY
[1]    ⎕ES(⎕ET≡1 3)/⎕ET    ⍝ CHECK FOR WS FULL
```

Here, you match (≡) the *WS FULL* code (1 3) against ⎕*ET*, and use event simulation (⎕*ES*) to signal the same error again if it matches. Because you are not trapping this error with ⎕*EA*, *WS FULL* will print at the terminal. Note that *AGAIN*[1] has the same effect if it causes *WS FULL* itself!

WS FULL? **What's that?:** In your user documentation, you will instruct the user to take certain actions if this message occurs, such as:

```
)SAVE CONTINUE
```

followed by:

```
)RESET
```

or:

```
)LOAD INVENTORY
```

The more helpful you make this documentation, the less likely it will be that a programmer will be needed to troubleshoot the problem.

Back to *AGAIN*

The remainder of *AGAIN* uses "bare bones" prompting. Any response in which the first character is not *Y* will return a zero, and the branch at *NUMIN*[9] will not be taken:

```
[2]    □←'RETRY (Y/N)? '
[3]    Z←'Y'=↑⍞
```

By taking the *AGAIN* code out of *NUMIN*, you have reserved the option to use it elsewhere.

You Only Have to *ASK*

You now have some general prompting functions for character and numeric input. You decide to go a little further, by having an *ASK* function that handles both:

```
[0]    Z←ASK TT;CALL
[1]   ⍝ PROMPT THE USER; CHECK REPLY
[2]   ⍝ IF SOMETHING GOES WRONG, REPROMPT THE USER
[3]   ⍝ TT IS (TYPE OF INPUT) (TEXT OF MESSAGE)
[4]    CALL←(0 ' '=↑TT)/'NUMIN 1↓TT' 'INPUT 1↓TT'
[5]    □ES(0=⍴CALL)/'PROGRAM ERROR - INVALID MESSAGE TYPE'
[6]    Z←⍎↑CALL
```

ASK determines which type of input you want, and calls the appropriate function. It makes this decision at line [4], based on the **first** (↑) character of its right argument, *TT*:

- If (↑*TT*) is 0, *ASK* calls *NUMIN*.
- If (↑*TT*) is ' ', *ASK* calls *INPUT*.

The remainder of the right argument contains the prompt message:

```
      A←ASK 0 'STOCK NUMBER'
STOCK NUMBER: 12
      A
12

      0=↑0⍴A           ⍝ IS A NUMERIC ?
1
```

If (↑*TT*) is neither 0 nor ' ', you generate an error at line [5] (one of your functions has called *ASK* improperly).

Stripping a Layer: In *ASK*[4], a depth 2 array is created, whose only item is either 'NUMIN 1↓*TT*' or 'INPUT 1↓*TT*'. The expression (↑*CALL*) strips away the outer layer. The resulting simple expression is then evaluated, which results in a call to *NUMIN* and *INPUT* with the prompting message.

You now have "all you could *ASK* for" in the way of a general input handling facility for your application, so you are ready to think about controlling the prompting sequences.

Chapter 6. Controlling Prompting Sequences

The next thing to consider is how to do your prompting without coding a lot of detail into your functions.

The Prompting Matrix: ORDERQ

You must prompt for the following items when a clerk places an order:

```
CUSTOMER NUMBER
 STOCK NUMBER
 AMT ORDERED
```

The requested data is all numeric, so you create a matrix, *ORDERQ*, in which each row starts with 0 (as expected by the *ASK* function). You do this by first entering a nested vector:

```
      ORDERQ←(0 'CUSTOMER NUMBER')(0 ' STOCK NUMBER')(0 ' AMT ORDERED')
      DISPLAY ORDERQ
┌→──────────────────────────────────────────────────────────────────────┐
│ ┌→────────────────────┐ ┌→──────────────────┐ ┌→──────────────────┐ │
│ │   ┌→──────────────┐ │ │   ┌→────────────┐ │ │   ┌→────────────┐ │ │
│ │ 0 │CUSTOMER NUMBER│ │ │ 0 │STOCK NUMBER│ │ │ 0 │ AMT ORDERED│ │ │
│ │   └───────────────┘ │ │   └─────────────┘ │ │   └─────────────┘ │ │
│ └∊────────────────────┘ └∊──────────────────┘ └∊──────────────────┘ │
└∊──────────────────────────────────────────────────────────────────────┘
```

and then applying disclose (⊃) to convert it to a matrix:

```
      ORDERQ←⊃ORDERQ
      DISPLAY ORDERQ
┌→──────────────────────┐
↓   ┌→──────────────┐  │
│ 0 │CUSTOMER NUMBER│  │
│   └───────────────┘  │
│   ┌→────────────┐    │
│ 0 │STOCK NUMBER│     │
│   └─────────────┘    │
│   ┌→────────────┐    │
│ 0 │ AMT ORDERED│     │
│   └─────────────┘    │
└∊──────────────────────┘
```

Asking for Each: Each row of *ORDERQ* contains a right argument to the *ASK* function. How do you ask *ASK* to prompt using each of these rows? You can do this by:

1. Converting *ORDERQ* back to a vector of vectors with enclose with axis (⊂[*X*]):

```
      DISPLAY ⊂[2]ORDERQ
.→-------------------------------------------------------------------.
| .→--------------------. .→-----------------. .→-----------------. |
| |   .→--------------. | |   .→-----------. | |   .→-----------. | |
| | 0 |CUSTOMER NUMBER| | | 0 |STOCK NUMBER| | | 0 | AMT ORDERED| | |
| |   '---------------' | |   '------------' | |   '------------' | |
| '∈--------------------' '∈-----------------' '∈-----------------' |
'∈-------------------------------------------------------------------'
```

2. Applying the each (¨) operator to the *ASK* function:

```
      A←ASK¨⊂[2]ORDERQ
CUSTOMER NUMBER: 55
 STOCK NUMBER  : 5613
 AMT ORDERED   : 12      ←————User responses
     A
55 5613 12
```

Why Do Step 1 At All?: If you saved *ORDERQ* as a vector of vectors instead of a matrix, you could do away with Step 1, but the matrix display will make your program documentation more readable.

Because each will have the same effect on a one-column matrix as it does on a vector, you have another way to do away with Step 1: Add a level of nesting to *ORDERQ* and then change it back to a matrix:

```
ORDERQ←,[⍳0]⊂[2]ORDERQ
```

The use of ravel with axis with an empty axis specification (`,[⍳0]`) adds a dimension to the right, creating a one-column matrix:

```
      DISPLAY ORDERQ
┌→────────────────────────┐
↓ ┌→────────────────────┐ │
│ │   ┌→──────────────┐ │ │
│ │ 0 │CUSTOMER NUMBER│ │ │
│ │   └───────────────┘ │ │
│ └∊────────────────────┘ │
│ ┌→─────────────────┐    │
│ │   ┌→───────────┐ │    │
│ │ 0 │STOCK NUMBER│ │    │
│ │   └────────────┘ │    │
│ └∊─────────────────┘    │
│ ┌→─────────────────┐    │
│ │   ┌→───────────┐ │    │
│ │ 0 │ AMT ORDERED│ │    │
│ │   └────────────┘ │    │
│ └∊─────────────────┘    │
└∊────────────────────────┘

      A←ASK¨ORDERQ
CUSTOMER NUMBER: 55
 STOCK NUMBER  : 5613
 AMT ORDERED   : 12
      A
55
5613
12
```

The result is a matrix, because *ORDERQ* is a matrix.

Repeating the Prompts

Because an order form may contain more than one item, you don't want to stop with just one series of prompts. You would like to repeat the questions in *ORDERQ* until all the orders on a given form are entered. You sit down in your armchair and think about program control.

Any APL2 task can be represented as a function, so you can imagine the repetition of any task as the repetitive application of a function:

```
      FOO FOREVER
∘
∘
∘
∘
∘
∘
∘
∘
```

```
∘
∘
∘
```

Well, you *will* want some way to stop it.

```
      FOO UNTIL EMPTY
∘
∘
∘
∘
∘

END OF PROCESSING
```

That's better.

Defining Your Own Operators

In APL2, you can define an operator that will take a function as an argument. Neither *FOREVER* nor *UNTIL EMPTY* will work quite as you imagined them (see Figure 9 on page 45), but something like this will:

```
      EMPTY←,' '
      FOOVAR
∘∘∘∘∘∘

[0]   Z←FOO V
[1]   Z←V↑FOOVAR
[2]   FOOVAR←V↓FOOVAR

      FOO UNTIL EMPTY 1
∘
∘
∘
∘
∘
∘

END OF PROCESSING
```

You can do this by making *UNTIL* a defined operator:

```
[0]   (F UNTIL END)V;X
[1]  ⍝ REPEAT F UNTIL RESULT MATCHES END
[2]  L1:⎕←X←F V
[3]   →(~END≡X)/L1
[4]   'END OF PROCESSING'
```

UNTIL takes two operands, the function *F* and the variable *END*. It also takes an argument, *V*, which is passed to *F*. When the result of the call to *F* matches an *END* condition, it exits.

One Way to *REPEAT:* Using the above musings as a guideline, you can imagine an operator, let's call it *REPEAT*, which continually repeats prompts until the input line is empty, and returns the accumulated results.

Here's one way to use a *REPEAT* operator:

```
     ORD←NUMIN REPEAT 'CUSTOMER NUMBER' 'STOCK NUMBER' 'AMOUNT'
CUSTOMER NUMBER STOCK NUMBER AMOUNT :  55 3569 12
CUSTOMER NUMBER STOCK NUMBER AMOUNT :  7 5613 35
CUSTOMER NUMBER STOCK NUMBER AMOUNT :
```

The result is a 2-item vector of vectors containing 3 items each:

```
      DISPLAY ORD
┌→─────────────────────────┐
│ ┌→─────────┐ ┌→────────┐ │
│ │55 3569 12│ │7 5613 35│ │
│ └~─────────┘ └~────────┘ │
└∈─────────────────────────┘
```

You would prefer, however, to have separate prompts for *each* of the individual items. How can you do that?

Imagine a defined operator, *EACH*, which is like the primitive ¨, but quits when the result of a function call is empty. Because a function operand of an operator can itself be a derived function, you can say something like:

```
(ASK EACH) REPEAT ORDERQ
```

Here, *REPEAT* is an *UNTIL EMPTY* operator. Its right argument, *ORDERQ*, becomes the right argument of the (*ASK* ¨) derived function. Because operators bind first to what is on the left (see the discussion of syntax in *APL2 Programming: Language Reference*), the above statement can be entered without parentheses:

```
ASK EACH REPEAT ORDERQ
```

This statement, which you have “written” without leaving your armchair, executes a loop within a loop.

Here it is in action:

```
       ASK EACH REPEAT ORDERQ          ←——REPEAT calls ASK EACH ORDERQ
CUSTOMER NUMBER : 55              ←——ASK EACH prompts with first row of ORDERQ
STOCK NUMBER    : 3567            ←——ASK EACH prompts with second row of ORDERQ
AMT ORDERED     : 12              ←——ASK EACH prompts with third row of ORDERQ
CUSTOMER NUMBER : 55              ←——REPEAT calls again
STOCK NUMBER    : 1135
AMT ORDERED     : 12
CUSTOMER NUMBER :                  ←——REPEAT exits when empty input is returned
```

The *REPEAT* Operator

REPEAT is a defined operator. In APL2, you can create your own operators as well as functions. However, unlike defined functions, which apply only to arrays, operators apply to arrays or functions (as operands).

In the *REPEAT* operator definition, (*F REPEAT*) is a derived function that acts on the function represented by *F*. The derived function subsequently acts on the array argument represented by *R*. A derived function can be an operand of another derived function.

Notice the parentheses in the header line of the operator definition. This tells APL2 the context in which to recognize the operator. In other words, execute the *REPEAT* operator any time you encounter:

function *REPEAT* array

The parentheses are not needed when the operator is used, but you might include them for clarity.

A defined operator, like a defined function, can be monadic, in which case it takes a function or array operand on its left, or dyadic — in which case it takes a function or array operand on its left and right. Example:

```
[0]    Z←(F REPEAT E) R
```

In the above example, the right operand, *E*, can be an array whose contents indicate the stop condition (an empty vector in the monadic form).

Language Note: Operator Syntax

The header line of a defined operator may contain as many as six different names with special syntactic meaning:

```
[0]  Z←LA (LO DOP RO) RA
```

The name of the above operator is *DOP*. The valence of *DOP* is defined within the parentheses; in this case, *DOP* is dyadic — it takes two operands, *LO* and *RO*.

LO and *RO* can be functions or arrays. They cannot be operators.

The combination of an operator and its operands comprises a **derived function**. Such a function can take one or more arguments. The derived function (*LO DOP RO*) takes two arguments, *LA* and *RA*.

Because a derived function is a function, it may also be an operand of an operator.

A derived function may or may not return an explicit result. In the above illustration, an explicit result, *Z*, is defined.

A function that takes two arguments is **ambi-valent**; in any reference, the left argument may be omitted. An operator that takes two operands can only be dyadic.

A **monadic operator** has a left operand only. A **monadic function** has a right argument only.

Why am I being told all this?

It is useful to keep in mind the distinction between operands and arguments and where they may appear. By way of illustration, consider the following operator headers and their meanings.

A dyadic operator whose derived function is monadic:

```
      (LO DOP RO) RA
```

A monadic operator, *MOP*, whose derived function is ambi-valent:

```
      LA (LO MOP) RA
```

A monadic operator, *MOP*, with a monadic derived function:

```
      (LO MOP) RA
```

The same, but here the operator name is *LO*; *MOP* is the operand:

```
      (MOP LO) RA
```

Not valid:

```
      LA (MOP LO)
```

Not valid:

```
      (MOP LO)
```

For this reason, (*FOO FOREVER*) cannot be written with *FOO* as a function and *FOREVER* as an operator.

Figure 9 (Part 1 of 2). Language Note: Operator Syntax

Note that *FOO UNTIL EMPTY can* be written; if *UNTIL* is an operator, it is interpreted as (*FOO UNTIL*)*EMPTY*. This implies, however, that *FOO* is a monadic function that somehow cares about the argument *EMPTY*. This is not true; such usage is possible, but artificial. Because it is *UNTIL* that wants to know about *EMPTY*, we would like to say:

```
(FOO UNTIL EMPTY)
```

However, this is just like:

```
(LO MOP RO)
```

which we know is not valid.

It is perfectly all right to use instead:

```
FOO UNTIL EMPTY HOWMANY
```

which is interpreted as:

```
(LO DOP RO) RA
```

and *is* valid.

The moral of the story is: Know thy operand.

It is also useful to distinguish the behavior of function and array operands. One distinction involves what happens in the body of the operator. The statement:

```
LO↑RA
```

may be used to call function *LO* with the first item of argument *RA*. If *LO* is an array, however, the expression is a **take** operation. The expression:

```
LO(↑RA)
```

can be used instead. In this case, an array operand is appended to the first item of the argument.

Another distinction involves the question of when operand evaluation takes place. A variable is evaluated *before* it is passed as an operand of the operator. This is a particularly important distinction when the operand is a shared variable, or when it is a niladic function with explicit result, as such a niladic function behaves like a variable. (So operators *can* have niladic functions as operands? No, the niladic functions are syntactically variables.)

Thus, using the *REPEAT* operator, you might expect to be able to say:

```
□ REPEAT 'TAKE DIFFERENT LENGTHS OF THIS'
```

and be continually prompted for □ input, while *REPEAT* accumulates results of the form *LO↑RA*. Instead, since □ is a shared variable, you will be prompted by □ once, your single entry will be evaluated, and, if it is not empty, applied continually. Do not try this.

Figure 9 (Part 2 of 2). Language Note: Operator Syntax

The *EACH* Defined Operator

Returning the Empties: How will empty input be handled? The primitive each defined operator will continue processing even if an empty response comes back from *ASK*. It will return all items, including the empty ones:

```
      A←ASK¨ORDERQ
 CUSTOMER NUMBER :  55
  STOCK NUMBER :   ◄──────────User presses ENTER only
  AMT ORDERED : 12
      A
  55

  12
      ρA
3 1
```

An empty input line should indicate that the user wants to discontinue the current series of prompts. In such a case, the function that is being repeated should exit to the next level. You can achieve this effect by creating an operator that exits when it detects an empty array. For this operator, *EACH* is as good a name as any. You have it return a vector:

```
       A←ASK EACH ORDERQ
 CUSTOMER NUMBER :  55
  STOCK NUMBER :  ◄──────────User presses ENTER only
       A
55
       ρA
1
```

The *REPEAT* operator can now test for completion by comparing the number of items in the result to the number of items in the right argument. It will print a warning message, drop the items for that prompting sequence, and repeat. It will exit when an empty response is given to the first prompt. Example:

```
      A←ASK EACH REPEAT ORDERQ
 CUSTOMER NUMBER :  55
  STOCK NUMBER : 3569
  AMT ORDERED :  12
 CUSTOMER NUMBER :  35
  STOCK NUMBER :            ◄──────User presses ENTER key
***DATA NOT SAVED            ◄──────REPEAT prints a message
 CUSTOMER NUMBER :  55       ◄──────User is prompted again from the top
  STOCK NUMBER : 1135
  AMT ORDERED :  12
 CUSTOMER NUMBER :           ◄──────Now user is finished

       DISPLAY A
┌→──────────────────────────┐
│ ┌→─────────┐ ┌→─────────┐ │
│ │55 3569 12│ │55 1135 12│ │
│ └~─────────┘ └~─────────┘ │
└∊──────────────────────────┘
```

APL2 Data Structures are Control Structures: By structuring your data (in this case, *ORDERQ*) so that it can be driven by common iteration operations like *EACH* and *REPEAT*, you are reducing your program logic. The ability to put program control into the data structures lets you write concise, modifiable functions. Many changes that would require modifications to the program in other languages will require only data changes in APL2, *if* you make an effort to take advantage of this capability.

Write Them Now, Use Them Forever: You will be able to use the *EACH* and *REPEAT* operators, and others like them, in many places. It's worth the effort to develop them now. You'll write them so that they can be read and used by others, in other contexts.

REPEAT Line-by-Line

Here is the complete *REPEAT* operator:

```
[0]    Z←(F REPEAT)R;X
[1]   ⍝ THE DERIVED FUNCTION (F REPEAT) WILL CALL F
[2]   ⍝ REPEATEDLY UNTIL F RETURNS AN EMPTY RESULT
[3]   ⍝ PRINTS 'WARNING' IF (⍴,R) DOES NOT MATCH ⍴(F R)
[4]    ⎕ES(3≠⎕NC 'F')/'ARGUMENT MUST BE A FUNCTION'
[5]    Z←''              ⍝ INITIALIZE RESULT TO EMPTY
[6]   L1:X←F R           ⍝ CALL FUNCTION F WITH R
[7]    →(0=⍴X)/0         ⍝ EXIT IF RESULT IS EMPTY
[8]    →(WARNING ERR(⍴,R)≠⍴X)/L1   ⍝ WARNING MESSAGE
[9]    Z←Z,⊂X            ⍝ APPEND RESULT
[10]   →L1               ⍝ REPEAT UNTIL X IS EMPTY
```

The left operand of *REPEAT* must be a monadic function. In APL2, a single function argument may be a nested array that contains all parameters needed by the function, so you do not need a dyadic form (though sometimes it is handy to have one).

Let's examine the key statements in *REPEAT*. Line [4]:

```
[4]    ⎕ES(3≠⎕NC 'F')/'ARGUMENT MUST BE A FUNCTION'
```

assures that the left operand is a function. If not, a program error is generated in the function that called (*F REPEAT*).

REPEAT[5]:

```
[5]    Z←''              ⍝ INITIALIZE RESULT TO EMPTY
```

initializes the result by setting it to an empty array.

```
REPEAT[6]:

[6]   L1:X←F R              ⍝ CALL FUNCTION F WITH R
```

executes the function that was specified in the argument.

```
REPEAT[7]:

[7]    →(0=⍴X)/0            ⍝ EXIT IF RESULT IS EMPTY
```

ends the operation if the result of the function is empty.

If *REPEAT* had a right operand, *E*, this statement would be →(*X*≡*E*)/0, which ends the operation if the result of the function matches the right operand of *REPEAT*.

```
REPEAT[8]:

[8]    →(WARNING ERR(⍴,R)≠⍴X)/L1   ⍝ WARNING MESSAGE
```

prints a warning message if *X* does not have the expected shape.

To** ERR **Is...: In *REPEAT*[8] the *ERR* function is used to do message handling. *ERR* is a simple general-purpose function for printing messages when errors are encountered. It looks like this:

```
[0]    Z←MSG ERR COND   ⍝ ERROR MESSAGE HANDLER
[1]   ⍝ PRINT MSG IF COND IS 1 ; RETURN COND
[2]    Z←COND           ⍝ RETURN COND
[3]    →(~Z)/0          ⍝ EXIT IF COND IS ZERO
[4]    MSG
```

The left argument of *ERR* is the text of a message; the right argument is a condition (a boolean 0 or 1). If *COND* is 1, and *MSG* is not empty, the message is printed. The explicit result of *ERR* is always *COND*, and is used to control branching in the calling function.

The warning message printed with the *ERR* subroutine is contained in a global variable:

```
      WARNING
***DATA NOT SAVED
```

For some uses, *REPEAT*[8] may be omitted, or its effect negated by setting *WARNING* to an empty vector, as *ERR* will then bypass message printing. Because it is easier to prove the correctness of programs when the effect they produce is dependent only on the arguments passed directly to them, you may prefer to make *WARNING* an argument (see Figure 10 on page 50).

Language Note: Ambi-valence

Any function defined with two arguments may be called either monadically or dyadically. Thus, an attempt to use the left argument may result in a *VALUE ERROR*.

To avoid this, it is advisable to check for the existence of the left argument:

```
[0]   Z←L F R
[1]   ⎕ES(2≠⎕NC'L')/'MISSING LEFT ARGUMENT'
      ∘
      ∘
      ∘
      ∘
```

If missing, you can print an error message, as above, assign a default, or execute a function or piece of code that uses the right argument only.

If the ambi-valent function is a derived function, then this check must be done not only in any ambi-valent functions that may be operands of the operator, but in the body of the operator definition itself, because a value error could occur in an attempt to call a function with a missing left argument.

Figure 10. Language Note: Ambi-Valence

EACH Line-by-Line

Here is the complete *EACH* operator:

```
[0]    Z←(F EACH)R;X;I;N
[1]   ⍝ THE DERIVED FUNCTION (F EACH) WILL PROCESS ALL
[2]   ⍝ OF THE ITEMS IN VECTOR R, APPENDING THE RESULTS.
[3]   ⍝ IF (F R) RETURNS AN EMPTY, (F EACH) EXITS
[4]    ⎕ES(3≠⎕NC 'F')/'ARGUMENT MUST BE A FUNCTION'
[5]    Z←''                  ⍝ INITIALIZE RESULT TO EMPTY
[6]    →(0=⍴R)/0              ⍝ EXIT IF R IS EMPTY
[7]    I←⎕IO                 ⍝ INITIALIZE INDEX
[8]    N←I+⍴R←,R             ⍝ LOOP CONTROL VALUE
[9]   L1:X←F I⊃R             ⍝ CALL FUNCTION F
[10]   →(0=⍴X)/0             ⍝ END - EXIT WITH AN EMPTY
[11]   Z←Z,⊂X                ⍝ APPEND RESULT
[12]   I←I+1                  ⍝ INCREMENT INDEX
[13]   →(N I)/L1              ⍝ PROCESS NEXT ITEM IN R
```

As with *REPEAT*, you must ensure that *F* is a function. If it were not, the statement at *EACH*[9] would be a pick of *R* with a two-item left argument, and might or might not work. You could have said *F*(*I*⊃*R*) instead, which would give *some* sort of an answer with an array, but you don't want to include this possibility in the domain of your definition of *EACH*.

EACH[5] sets an empty initial result and, at the next line, you decide to merely ignore empty right arguments.

In the next line you create an index variable, *I*. ⎕*IO* is used to ensure index independence (you want *EACH* to be used in other workspaces).

Then you set the array you will iterate on, *R*, to a vector, and the loop control variable, *N*, to the length of *R*.

The label at *EACH*[9] signals the beginning of the loop. It will end if one of two conditions is met:

- The result of a call to function *F* is empty.
- All the items of *R* have been processed.

EACH[9] calls function *F* with the *I*th item of *R* as an argument.

EACH[10] exits if *X*, the result of *F I*⊃*R*, is empty. Otherwise, the function result *X* is appended to the derived function result *Z* at *EACH*[11]. Notice that *X* is first enclosed, then catenated. The initial empty *Z* disappears after the first catenation.

Language Note: Building Nested Arrays

The effect of successive catenations of the form:

```
Z←''
Z←Z,⊂X
Z←Z,⊂X
Z←Z,⊂X
```

which produces a 3-item vector, is very different from the following series, which uses juxtaposition to append *Z*:

```
Z←''
Z←Z X
Z←Z X
Z←Z X
```

This produces successively deeper 2-item arrays.

Figure 11. Language Note: Building Nested Arrays

At *EACH*[11], the index variable is incremented. It is compared to the loop control variable at *EACH*[12], and iteration continues until *N* is greater than *I*, or *X* is empty.

Note that *I* is not incremented and compared on the same line. If a failure occurred at *EACH*[12], you would be able to continue with the knowledge that *I* will not be incremented twice. This sort of programming practice eases function debugging.

Chapter 7. Updating Tables

You want to write a function, which you've already named *PUT*, that will update rows of a table. The row is uniquely identified by the first item in it.

The columns of your tables are identified by the arrays you have set up for prompting:

```
     STOCKQ                 CUSTQ                  ORDERQ
0 STOCK NUMBER         0 CUSTOMER NUMBER      0 CUSTOMER NUMBER
  DESCRIPTION            CUSTOMER NAME        0  STOCK NUMBER
0 PRICE                  ADDRESS              0  AMT ORDERED
0 INVENTORY              CITY, STATE, ZIP
0 REORDER LEVEL
```

Let's look at a row of the *STOCKS* table.

```
┌→──────────────────────────────────┐
│      ┌→──────────────┐           │
│ 1135 │FIRST GREAT ITEM│ 9.95 118 50 │
│      └───────────────┘           │
└∈──────────────────────────────────┘
```

How do you store this in *STOCKS*? What you want to do is to replace a row in *STOCKS*, in the position where the value of the first item is the same. If such a row doesn't currently exist, you want to add a row at the end.

What if *STOCKS* contained nothing at all? Indeed, you haven't created it yet. Can you add something to nothing? Yes, if you create *STOCKS* as an empty array with no rows and 5 columns:

```
      STOCKS←0 5⍴0 ' ' 0 0 0
```

Now that *STOCKS* exists, you can add rows to it:

```
      STOCKS←STOCKS,[1]1234 'A THING' 5.25 100 20
      STOCKS
1234  A THING  5.25 100 20
      ⍴STOCKS
1 5
```

Initially, you may create *STOCKS* and your other tables in your workspace as global variables. This requires the user to save all the data. Before going into production, you'll have to use one of the auxiliary processors that APL2 provides for file access, such as AP 127. AP 127 provides access to data base management systems that support the Structured Query Language (SQL).

When it's time to make *STOCKS* an SQL table, you will use an SQL CREATE statement. If data already exists in your table, you add a VALUES clause, and pass all existing rows as the initial values.

See Chapter 12, “Using SQL Tables” on page 89, for a sample CREATE statement. For a more detailed discussion of SQL tables, see *APL2 Programming:*

Using Structured Query Language (SQL) and *IBM DATABASE 2 Application Programming Guide for TSO Users*.

Notice that the empty array assigned above to *STOCKS* mirrors the data types that the clerks will be entering — a number in the first column, characters in the second, and so on. This isn't a requirement; you could have created an empty *STOCKS* by assigning it `0 5⍴' '` or `0 5⍴⍳0`. You *do* need to give *STOCKS* five columns; otherwise, you can't add a five-column row to it.

The PUT and PUTW Functions

You are now ready to write *PUT* and test it with *STOCKS*:

```
[0]   NAME PUT A  ⍝ UPDATE A TABLE
[1]   NAME PUTW⊃A  ⍝ UPDATE WORKSPACE TABLE
[2]  ⍝ FOR SQL UPDATES, USE   NAME PUTSQL A
```

***PUT* Comes in Two Flavors:** You have already decided to have two flavors of *PUT*: one for now, and one for when you add SQL data base access. So that you don't have to change all the references to *PUT*, you use it as a shell that calls *PUTW*, the workspace version of *PUT*, from *PUT*. Later, you will change the *PUTW* reference to a *PUTSQL* reference.

At *PUT*`[2]`, the array to be put, *A*, is disclosed, creating a matrix as input to *PUTW*.

***The PUTW* Function:** Okay, so you still have to code *PUTW*. You write it so that you can both add and replace rows:

```
[0]   NAME PUTW NEW;OLD;TAB;I
[1]  ⍝ UPDATE THE TABLE NAMED 'NAME' WITH 'NEW'
[2]   →('NOT A VALID TABLE NAME' ERR ~(⊂NAME)∈TABLES)/0
[3]   TAB←⍎NAME                ⍝ TABLE
```

PUTW`[2]` makes sure the requested table *NAME* is one the application knows about. *TABLES* is merely a list of these tables:

```
TABLES←'STOCKS' 'CUSTOMERS' 'ORDERS'
```

The error handling at *PUTW*`[2]` treats the error as a user error, not a program error. By doing this, you have conceptually externalized *PUT*, leaving open the possibility of direct use by a sophisticated user. An alternative is to use ⎕*ES*, which would force suspension in the calling function.

PUTW[3] uses ⍎ to extract the table from the name.

```
      TAB←⍎NAME
      TAB
 1135 FIRST GREAT ITEM     9.95  118   55
 9993 HIGH FLYER WIDGET   88.73  240   35
 3569 SECOND MONEYMAKER   24.75    0   30
 5613 MAIL ORDER SPECIAL  14.99  225   95
 2583 A REAL WINNER       49.99   89   10
 9998 NONESUCH FRAMMIS     2.69  440   50
```

You then find where the first column of *NEW*, the matrix of updates, matches the first column in *TAB*, by creating a boolean vector, *OLD*:

```
[4]    OLD←NEW[;1]∈TAB[;1]       ⍝ EXISTING ROWS
```

OLD used at *PUTW*[5] to find the indices of occurrence of the old rows in *TAB*. These indices are then used in *PUTW*[6] to update *TAB* with the corresponding rows of *NEW*:

```
[5]    I←TAB[;1]⍳OLD/NEW[;1]     ⍝ INDICES OF OLD
[6]    TAB[I;]←OLD⌿NEW           ⍝ REPLACE
```

Note that you could have said:

```
      ((TAB[;1]∈OLD/NEW[;1])⌿TAB))←OLD⌿NEW
```

which uses **selective specification** to replace the selected rows of *TAB*. However, this statement could result in the rows of *NEW* being assigned in the wrong order, unless both tables were sorted in column 1 order.

PUTW[7] appends those rows of *NEW* whose key column does not match existing columns in *TAB*.

```
[7]    TAB←TAB,[1](~OLD)⌿NEW     ⍝ APPEND NEW
```

At *PUTW*[8], you assign *TAB* back to the original table whose name is in *NAME*:

```
[8]    ⍎NAME,'←TAB'              ⍝ ASSIGN TO TABLE
```

Here's a test of *PUTW*, using the *EVALIN* function described in “Building a Data Structure” on page 15:

```
      STOCKS
 1135 FIRST GREAT ITEM    9.95  118  55
 9993 HIGH FLYER WIDGET  88.73  240  35
 3569 SECOND MONEYMAKER  24.75   10  30
 5613 MAIL ORDER SPECIAL 14.99  225  95
 2583 A REAL WINNER      49.99   89  10
 9998 NONESUCH FRAMMIS    2.69  416  50
 7777  THINGY             1.89    2   1
 8888 WHEEE               4.33 3011 100
 5555 WHIZBANG           22.22   43   2

      'STOCK' PUTW EVALIN
 2222 'TUTU' 22.22 222 22
NOT A VALID TABLE NAME

      'STOCKS' PUTW EVALIN
 2222 'TUTU' 22.22 222 22
 1135 'FIRST GREAT ITEM' 9.95 150 55
      STOCKS
 1135 FIRST GREAT ITEM    9.95  150  55
 9993 HIGH FLYER WIDGET  88.73  240  35
 3569 SECOND MONEYMAKER  24.75   10  30
 5613 MAIL ORDER SPECIAL 14.99  225  95
 2583 A REAL WINNER      49.99   89  10
 9998 NONESUCH FRAMMIS    2.69  416  50
 7777  THINGY             1.89    2   1
 8888 WHEEE               4.33 3011 100
 5555 WHIZBANG           22.22   43   2
 2222 TUTU               22.22  222  22
```

For a discussion of *PUTSQL*, see Chapter 12, “Using SQL Tables” on page 89.

The GET and GETW Functions

You also need a way to get a row from a table. The arguments to a generalized *GET* function would be the table name and the key field value. For your workspace tables, the key column will always be column 1.

As with *PUT*, you will want to convert to SQL, so *GET* looks like this:

```
[0]   Z←NAME GET N
[1]  ⍝ GET ITEMS N FROM THE TABLE NAMED 'NAME'
[2]   Z←NAME GETW N  ⍝ GET FROM WORKSPACE TABLE
[3]  ⍝ FOR SQL SELECTS, USE  NAME GETSQL N
```

You write *GETW* so that it can obtain a list of items:

```
[0]    Z←NAME GETW N
[1]  ⍝ GET ITEMS N FROM THE TABLE NAMED 'NAME'
[2]    Z←''
[3]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[4]    TAB←⍎NAME
[5]    B←N∈TAB[;1]
[6]    →0⍴('ITEM(S)'((~B)/N)'NOT FOUND')ERR~∧/B
[7]    →(~∨/B)/0     ⍝ EXIT IF NONE FOUND
[8]    Z←(TAB[;1]∈N)⌿TAB
```

GETW obtains a local copy of the table, in the same way that *PUT* does. Then *GETW*[5] creates a boolean vector, *B*, which represents the occurrences of *N* in the first column.

GETW[6] prints a message if any of these items are not found, but do not exit; the →0⍴ applies to the result of the *ERR* function, and merely causes control to be passed to the next line. *GETW*[7] exits if *B* is all zero.

Finally, *GETW*[8] obtains the rows of *TAB* in which items of *N* do occur.

Note that *GETW*[5] through *GETW*[7] are needed only so that a message can be printed for the “not found” items. In writing code for interactive use by others, it is your responsibility to be reasonably explicit about the occurrences of unexpected conditions.

Here is a quick test of *GET*:

```
      'STOCKS' GET⍳0

      'STOCKS' GET  1
ITEM(S)  1  NOT FOUND

      'STOCKS' GET 1 5613 3569
ITEM(S)  1  NOT FOUND
3569 SECOND MONEYMAKER  24.75  10 30
5613 MAIL ORDER SPECIAL 14.99 225 95
```

The two rows of *STOCKS* are returned as an explicit result.

The UPDATE Function

Now that you have *GET* and *PUT*, and the table called TABLES, it is a trivial matter to write a function that can be used to update any of your tables. Such a routine could be added to the menu for direct use, or be used by other routines.

Here's what it looks like:

```
[0]   UPDATE;WHAT;I;MSG
[1]  ⍝ UPDATE AN INVENTORY TABLE
[2]   →(0=⍴WHAT←INPUT 'CHOOSE',3⍕'OR',¯1⍕TABLES)/0
[3]   I←TABLES⍳⊂WHAT  ⍝ TABLE INDEX
[4]   →('REQUEST NOT RECOGNIZED' ERR I>⍴TABLES)/0
[5]   MSG←⍎I⊃PROMPTS          ⍝ SELECT PROMPT TABLE
[6]   A←ASK EACH REPEAT MSG   ⍝ PROMPT FOR INPUT
[7]   →(0=⍴A)/0       ⍝ EXIT IF EMPTY RESPONSE
[8]   WHAT PUT A      ⍝ PUT ROWS 'A' TO TABLE 'WHAT'
```

At *UPDATE*[2], the user selects one of the tables in the global variable *TABLES*:

```
      TABLES
 STOCKS CUSTOMERS ORDERS
```

UPDATE[3] obtains an index, and [4] checks if it is valid. The index is then used at *UPDATE*[5] to select a prompting matrix from the global variable *PROMPTS*:

```
      PROMPTS
 STOCKQ CUSTQ ORDERQ
```

It is assigned to *MSG*.

At *UPDATE*[6], *MSG* is used to prompt for table input. *UPDATE*[8] calls *PUT* to put the changes in the table.

***Changing an Address with** UPDATE:* *UPDATE* gives you some basic capabilities that can be used to handle both specified and unspecified user needs. For example, you will need to make customer name and address changes. You can handle that with a call to *UPDATE*:

```
      UPDATE
CHOOSE STOCKS CUSTOMERS OR ORDERS :  CUSTOMERS
CUSTOMER NUMBER :  55
 CUSTOMER NAME :  NEWNAME
 ADDRESS :  NEW ADDRESS
 CITY, STATE, ZIP :  NEWZIP
CUSTOMER NUMBER :

      CUSTOMERS
  7 CITY TRADERS INC.  41 POSTAGE ROAD      RIMELA, N.Y. 12345
 55 NEWNAME            NEW ADDRESS          NEWZIP
312 MANTUP SALES CORP. RURALIA FARMS, RFD 2 SUBURBIA, WIS. 00000
```

UPDATE can be added to your user menu, as shown in Chapter 9, “Creating the PLACE Function” on page 63.

The DELETE Function

DELETE will handle any table:

```
[0]    NAME DELETE N;T
[1]   ⍝ DELETE THE ROW OF TABLE NAMED 'NAME' WHOSE KEY IS N
[2]    T←(TABLES∈⊂NAME)/⍳↑⍴TABLES
[3]    →('NOT A VALID TABLE NAME' ERR 0=⍴T)/0
[4]    TAB←⍎NAME
[5]    →(('NO ENTRY FOR' N)ERR~N∈TAB[;1])/0
[6]    TAB←(~TAB[;1]∈N)⌿TAB
[7]    ⍎NAME,'←TAB'
```

DELETE ensures that the table name is valid, and that an entry exists. Then *DELETE*[6] removes the first row whose first column is *N*. In the call from *FILL*, *N* will be the invoice number. *DELETE*[7] assigns the temporary table *TAB* back to the global table, in this case, *ORDERS*.

You also add a note about SQL:

```
[8]   ⍝ ** SQL REQUIRES A DELETE STATEMENT
```

Now you're finished with all the building blocks!

Chapter 8. Creating the MENU Function

Now that you've defined the structure of your tables and built tools for inputting, updating, and formatting, you can begin to create and test some of the functions that will drive the application. You decide to call the top level one *MENU*. It will present the user with a list of tasks to choose from. Figure 12 shows the overall structure of the program.

Figure 12. Structure of the Inventory Control Application Program

You start by creating a variable that describes the items on your menu:

```
      ITEMS←'PLACE' 'FILL' 'STOCK'
```

You will use *ITEMS* in more than one way. First it's used for the prompting in *MENU*:

```
[0]   MENU
[1]   ⍝ SELECT AN ITEM
[2]   L1:''
[3]   WHAT←INPUT 'CHOOSE',3⍕'OR',¯1⍕ITEMS
[4]   →(0=⍴WHAT)/0   ⍝ TERMINATE IF EMPTY RESPONSE
[5]   →('REQUEST NOT RECOGNIZED' ERR~(⊂WHAT)∈ITEMS)/L1
[6]   '⎕←⎕EM[1;]' ⎕EA WHAT
[7]   →L1   ⍝ ASK AGAIN
```

At *MENU*[3], *ITEMS* is used to list the choices:

```
      MENU
CHOOSE PLACE FILL OR STOCK :
```

By doing it this way, you can add items to your menu just by appending to *ITEMS*, as in:

```
      ITEMS←ITEMS,⊂'UPDATE'
      ⍴ITEMS
4
      MENU
CHOOSE PLACE FILL STOCK OR UPDATE :
```

Language Note: Appending Arrays to Arrays

It's important to distinguish the effects produced by various ways of appending arrays to arrays. In particular, note the differences between the following expressions:

```
ITEMS,⊂'CHECK'
ITEMS 'CHECK'
ITEMS,'CHECK'
ITEMS (⊂'CHECK')
```

You can use the *DISPLAY* function to observe the differences.

ITEMS looks like this:

```
      DISPLAY ITEMS
 .→-----------------------.
 | .→----. .→---. .→----. |
 | |PLACE| |FILL| |STOCK| |
 | '-----' '----' '-----' |
 '∈-----------------------'
```

In the first expression, *CHECK* is enclosed (⊂) to produce a scalar, which is then catenated to a 3-item vector, producing a 4-item vector of vectors:

```
      DISPLAY ITEMS,⊂'CHECK'          ⍝ A 4 ITEM VECTOR
 .→-------------------------------.
 | .→----. .→---. .→----. .→----. |
 | |PLACE| |FILL| |STOCK| |CHECK| |
 | '-----' '----' '-----' '-----' |
 '∈-------------------------------'
```

Figure 13 (Part 1 of 2). Language Note: Appending Arrays to Arrays

The second expression creates a 2-item vector containing a 3-item vector of vectors and a 5-item vector:

```
      DISPLAY ITEMS 'CHECK'          ⍝ A 2 ITEM VECTOR
 .→-----------------------------------.
 | .→-------------------------. .→----. |
 | | .→----. .→---. .→----. | |CHECK| |
 | | |PLACE| |FILL| |STOCK| | '-----' |
 | | '-----' '----' '-----' |         |
 | '∈-----------------------'         |
 '∈-----------------------------------'
```

The third expression catenates a 5-item vector to a 3-item vector, producing an 8-item vector:

```
      DISPLAY ITEMS,'CHECK'          ⍝ AN 8 ITEM VECTOR
.→-------------------------------.
| .→----. .→---. .→----.         |
| |PLACE| |FILL| |STOCK| C H E C K |
| '-----' '----' '-----' - - - - - |
'∈-------------------------------'
```

In the last expression, a 2-item vector is produced, in which the second item is a scalar containing a 5-item vector:

```
      DISPLAY ITEMS (⊂'CHECK')       ⍝ A 2 ITEM VECTOR
.→-------------------------------------.
| .→-------------------------. .-------. |
| | .→----. .→---. .→----. | | .→----. | |
| | |PLACE| |FILL| |STOCK| | | |CHECK| | |
| | '-----' '----' '-----' | | '-----' | |
| '∈-----------------------' '∈--------' |
'∈-------------------------------------'
```

It is also worth noting that:

```
      ITEMS ('CHECK')
```

is equivalent to the second expression, and not to the fourth.

Figure 13 (Part 2 of 2). Language Note: Appending Arrays to Arrays

Checking for a Valid Response: *MENU*[5] checks for a valid response by comparing what the user entered to the members of items:

```
[5]   →('REQUEST NOT RECOGNIZED' ERR~(⊂WHAT)∈ITEMS)/L1
```

The variable *WHAT* contains the user's menu selection. It is enclosed so that the entire word is treated as a single scalar item:

```
      WHAT←'PLACE'
      DISPLAY ⊂WHAT
.---------.
| .→----. |
| |PLACE| |
| '-----' |
'∈--------'
```

```
        (⊂WHAT) ∈ITEMS
 1
```

If you did not enclose *WHAT*, you would be comparing each of the letters contained in *WHAT*:

```
       WHAT∈ITEMS
0 0 0 0 0
```

Thus, *MENU*[5] prints an error message and branches to *L*1 to prompt again if (⊂*WHAT*) is not a member of *ITEMS*.

After the *MENU* item is selected, you want to execute the appropriate routine. This is easy if the name of the function is the same as the prompt.

Of course, you want to trap errors and give the user a chance to try again:

```
[6]    '⎕←⎕EM[1;]' ⎕EA WHAT
[7]    →L1  ⍝ ASK AGAIN
```

MENU[6] executes the function named in *WHAT*, and prints the first line of the interpreter **event message** (⎕*EM*) if there is a failure.

MENU[7] (the last line of *MENU*) starts the menu prompting sequence over again.

Chapter 9. Creating the PLACE Function

You are now ready to code the first of the functions named in *ITEMS*. Its name is *PLACE*, and it will be used to place an order. In fact, *PLACE* will do a lot of things:

- Prompt for a new order and assign an invoice number
- Check stock availability and reduce the order if stock is low
- Update the *ORDERS* table
- Prompt for a customer number

  If the customer is new, update the *CUSTOMERS* table
- Format an invoice

You sketched out all these steps before you started the building blocks. When you have *PLACE* running, you'll have made use of all the blocks.

Input: Customers, Old and New

You start *PLACE* by prompting for customer number:

```
[0]    PLACE;ORD;CUST;ORDQ
[1]  ⍝ PLACE AN ORDER
[2]    →(0=⍴CUST←ASK↑ORDERQ)/0 ⍝ GET CUSTOMER NUMBER
```

Example:

```
       CUST←ASK↑ORDERQ
CUSTOMER NUMBER : 55
       CUST
55
```

Now the customer number can be compared to the customer list. If it's not there, the clerk can enter new customer statistics:

```
[3]    NEWCUST CUST      ⍝ ADD NEW CUSTOMER
```

NEWCUST will require a prompting matrix. You include the customer number, so that it corresponds with the columns of the *CUSTOMERS* table:

```
       CUSTQ
  0 CUSTOMER NUMBER
    CUSTOMER NAME
    ADDRESS
    CITY, STATE, ZIP
```

Here's a sample call:

```
      NEWCUST CUST
NAME: UNIVERSAL ENTERPRISES
ADDRESS: 3544 CENTRAL AVENUE
CITY, STATE ZIP: METROPOLIS, CA. 99999
NAME:        ←──────────Null entry
 ←──────────────────────Dialog ends
```

NEWCUST is easy to write:

```
[0]   NEWCUST CUST;DATA
[1]  ⍝ ADD A NEW CUSTOMER
[2]   →(CUSTOMERS[;1]∈CUST)/0     ⍝ NOT NEW
[3]   ⎕←⎕TC[2],'NEW CUSTOMER'
[4]   →(0=⍴DATA←ASK EACH 1↓[1]CUSTQ)/0
[5]   →(WARNING ERR (⍴DATA)≠¯1+↑⍴CUSTQ)/0
[6]   'CUSTOMERS' PUT,⊂CUST,DATA  ⍝ UPDATE TABLE
[7]   ⎕←'CUSTOMER RECORD UPDATED',⎕TC[2]
```

NEWCUST[3] prints a message using ⎕*TC*[2], the carriage return character, to space one line before the message is printed. In *NEWCUST*[7], the line spacing comes after the message.

Line [4] of *NEWCUST* uses drop with axis to dispose of the first row of *CUSTQ* before calling (*ASK EACH*). It exits if there is an empty response.

NEWCUST[6] calls *PUT* to update tables. The left argument of *PUT* is a table name. The right argument is the data to be added. It includes the key field that identifies the row of the table the data belongs in. For the customer table, the key is the customer number, which you have placed in *CUST*.

Input: Prompting for Orders

Here are lines [4] and [5] of *PLACE*:

```
[4]   ORDQ←1↓[1]ORDERQ        ⍝ ITEMS TO PROMPT FOR
[5]   →(0=⍴ORD←ASK EACH REPEAT ORDQ)/0
```

The prompting loop omits the repetition of *CUSTOMER NUMBER*:

```
      ORD←ASK EACH REPEAT 1↓[1]ORDERQ
  STOCK NUMBER : 3569
  AMT ORDERED :  12
  STOCK NUMBER : 1133
  AMT ORDERED :
***DATA NOT SAVED
  STOCK NUMBER : 1135
  AMT ORDERED :  12
  STOCK NUMBER :
      ORD
3569 12 1135 12
```

The Invoice Number: Each order form will have a unique invoice number assigned to it. You use a global variable, *INVO*, to contain the current invoice number.

```
[9]   INVO←INVO+1              ⍝ NEW INVOICE NUMBER
```

You make a note somewhere to put *INVO* on a file.

Process: Actual Orders

Before the order can be filled, stock availability is checked. To do this, *PLACE* calls another function, *ACTORD*:

```
[6]  ⍝ ACTORD FINDS ACTUAL ORDERS AND UPDATES 'STOCKS'
[7]   ORD←ACTORD EACH ORD
```

ACTORD is called for each order entry. Here it is:

```
[0]   ACT←ACTORD ORD;STO;NEW
[1]  ⍝ ACTUAL ORDERS BASED ON STOCK AVAILABLE
[2]  ⍝ ORD IS AN ITEM, QUANTITY
[3]   ACT←ORD[1],0
[4]   →(0=⍴STO←'STOCKS' GET↑ORD)/0  ⍝ STOCK RECORDS
[5]   NEW←⌈/0,STO[;4]-ORD[2]        ⍝ MAX NEW IS 0
[6]  ⍝ IF NEW IS ZERO, THE ORDER IS ALL AVAILABLE
[7]   ACT[2]←(1 0=NEW=0)/STO[;4],ORD[2]
[8]   STO[;4]←NEW
[9]   'STOCKS' PUT⊂[2]STO  ⍝ UPDATE STOCK
[10]  →(ACT[2]=ORD[2])/0
[11]  ⎕←'ORDER FOR STOCK ' STO[;1]'REDUCED TO ' ACT[2]
```

ACTORD subtracts the number of parts ordered from the number in stock. In addition, it takes care of the case where there is not enough inventory to fill the order. In this case, the clerk is notified and the amount ordered is reduced as appropriate.

For example, suppose customer number 62 ordered 25 of part number 3569, and suppose only 20 of that part were in stock:

```
ORDER FOR STOCK  3569 REDUCED TO   20
```

At *ACTORD*[4], *ACTORD* calls the *GET* function to get the stock record for the order it is processing.

Reducing the Order: *ACTORD*[5] calculates the new available stock after consignment of stock to this order. If the order quantity is greater than or equal to the stock available, the new available stock is zero. In this case, the order is reduced in *ACTORD*[7] to what was available.

ACTORD[8] updates available stock in the stock record. At *ACTORD*[9], *PUT* is called to update the *STOCKS* table. If the order has been reduced, a message is printed before exiting from *ACTORD*.

When *ACTORD* returns to *PLACE*, some of the processed orders have been reduced to zero, either because of a *GET* failure or because no stock was available. For example, if stock item 3569 has 0 units available, and item 2222 is not in stock, the following could occur:

```
      PLACE
CUSTOMER NUMBER :  55
 STOCK NUMBER :  3569
 AMT ORDERED :  10
 STOCK NUMBER :  2222
 AMT ORDERED :  20
 STOCK NUMBER :
ORDER FOR STOCK   3569  REDUCED TO  0
ITEM(S)  2222  NOT FOUND
```

Handling Empty Orders: *PLACE*[8] deletes the zeroed orders from *ORD*, the vector of order vectors. If they are all zero, we exit from *PLACE*:

```
[8]   →(0=ρORD←(∈0≠2⊃¨ORD)/ORD)/0 ⍝ DELETE ZEROED ORDERS
```

If this exit is taken, the menu will appear again.

The clerk may prefer to be given the opportunity to correct an order that is in error before the invoice is printed. If this is the case, you can add this capability later, perhaps with an optional call to *UPDATE*.

Language Note: Pick, Each, and Chipmunk

Consider the following nested array:

```
      DISPLAY ORD
┌→──────────────────────────────────────────────┐
│ ┌→────────────┐ ┌→──────────┐ ┌→────────────┐ │
│ │121 55 3569 12│ │121 7 5613 35│ │121 312 1135 2│ │
│ └~────────────┘ └~──────────┘ └~────────────┘ │
└⍷──────────────────────────────────────────────┘
```

It is a 3-item vector of 4-item vectors. The second item of this array can be selected using pick (⊃):

```
      2⊃ORD
121 7 5613 35
```

You can also pick at greater depth:

```
      2 4⊃ORD
35
```

To select the second item of *each of the items*, you use ⊃¨:

```
      2⊃¨ORD
55 7 312
```

Similarly, you can pick different items from each vector:

```
      2 3 4⊃¨ORD
55 5613 2
```

What if you want to select the second and third items of *ORD* using pick? You can resort to the **chipmunk idiom** (⊃¨⊂):

```
      2 3⊃¨⊂ORD
121 7 5613 35  121 312 1135 2
```

The same method can be used to pick at greater depth. The following picks the third item of each of the second and third vectors:

```
     (2 3)(3 3)⊃¨⊂ORD
5613 1135
```

Chipmunk? If you take a moment to stare at pick-each-enclose:

```
      ⊃¨⊂
```

you'll see how the chipmunk idiom got its name.

Figure 14. Language Note: Pick, Each, and Chipmunk

The inventory and customer numbers can be appended to the order items by **catenation with each** (,¨):

```
[10]  ORD←INVO,¨CUST,¨ORD    ⍝ APPEND TO ENTRIES
```

Here's how it works:

```
      ORD←ASK EACH REPEAT 1↓[1]ORDERQ
 STOCK NUMBER :  1135
 AMT ORDERED :  2
 STOCK NUMBER :  3569
 AMT ORDERED :  12
 STOCK NUMBER :

     CUST←ASK ↑ORDERQ
CUSTOMER NUMBER :  55

     INVO
179

      DISPLAY ORD
┌→──────────────────┐
│ ┌→─────┐ ┌→──────┐ │
│ │1135 2│ │3569 12│ │
│ └~─────┘ └~──────┘ │
└∊──────────────────┘

      DISPLAY CUST,¨ORD
┌→──────────────────────────┐
│ ┌→────────┐ ┌→─────────┐ │
│ │55 1135 2│ │55 3569 12│ │
│ └~────────┘ └~─────────┘ │
└∊──────────────────────────┘

      DISPLAY INVO,¨CUST,¨ORD
┌→──────────────────────────────────┐
│ ┌→────────────┐ ┌→─────────────┐ │
│ │179 55 1135 2│ │179 55 3569 12│ │
│ └~────────────┘ └~─────────────┘ │
└∊──────────────────────────────────┘
```

The result is a vector of order vectors that can be used to update the order table.

Compare the above example with the following:

```
      DISPLAY INVO,CUST,ORD
┌→──────────────────────────┐
│        ┌→─────┐ ┌→──────┐ │
│ 179 55 │1135 2│ │3569 12│ │
│        └~─────┘ └~──────┘ │
└∊──────────────────────────┘
```

```
      DISPLAY INVO,CUST,¨ORD
.→-------------------------------.
|     .→--------. .→----------.   |
| 179 |55 1135 2| |55 3569 12|   |
|     '~--------' '~----------'   |
'∈-------------------------------'
      DISPLAY INVO CUST,¨ORD
.→---------------------------.
| .→----------. .→----------. |
| |179 1135 2| |55 3569 12| |
| '~----------' '~----------' |
'∈---------------------------'
```

The last expression is equivalent to (*INVO CUST*),¨*ORD*.

Process: Updating the Table

To update *ORDERS*, you merely want to append a matrix, which you do at *PLACE*[11] by using *PUT*.

```
[11]  'ORDERS' PUT ORD          ⍝ UPDATE ORDERS
```

Print: Formatting the Invoice

Finally, you finish *PLACE* by calling functions that create and format an invoice:

```
[12]  FORMAT INVOICE↑↑ORD          ⍝ DISPLAY INVOICE
```

The information from *INVOICE* is kept logically separate from the report formatting so that you can use it, if needed, in other ways. *FORMAT* and *INVOICE* are discussed in Chapter 10, “Formatting the Invoice: The INVOICE and FORMAT Routines” on page 72.

The Complete PLACE Function

We've been looking at *PLACE* piecemeal; now let's view it as a whole:

```
[0]    PLACE;ORD;CUST;ORDQ
[1]   ⍝ PLACE AN ORDER
[2]    →(0=⍴CUST←ASK↑ORDERQ)/0 ⍝ GET CUSTOMER NUMBER
[3]    NEWCUST CUST            ⍝ ADD NEW CUSTOMER
[4]    ORDQ←1↓[1]ORDERQ        ⍝ ITEMS TO PROMPT FOR
[5]    →(0=⍴ORD←ASK EACH REPEAT ORDQ)/0
[6]   ⍝ ACTORD FINDS ACTUAL ORDERS AND UPDATES 'STOCKS'
[7]    ORD←ACTORD EACH ORD
[8]    →(0=⍴ORD←(∈0≠2⊃¨ORD)/ORD)/0 ⍝ DELETE EMPTY ORDERS
[9]    INVO←INVO+1              ⍝ NEW INVOICE NUMBER
[10]   ORD←INVO,¨CUST,¨ORD      ⍝ APPEND TO ENTRIES
[11]   'ORDERS' PUT ORD         ⍝ UPDATE ORDERS
[12]   FORMAT INVOICE↑↑ORD      ⍝ DISPLAY INVOICE
```

By using the functions *ASK*, *NEWCUST*, *ACTORD*, *PUT*, *INVOICE*, and *FORMAT*, as well as the *EACH* and *REPEAT* operators, you have packaged a major portion of your application into ten executable statements. Many of these tools will also simplify the remainder of your task.

***Testing** PLACE:* Here's a quick test of *PLACE*:

Tables Before Changes:

```
      STOCKS
 1135 FIRST GREAT ITEM    9.95  118  55
 9993 HIGH FLYER WIDGET  88.73  240  35
 3569 SECOND MONEYMAKER  24.75   10  30
 5613 MAIL ORDER SPECIAL 14.99  195  95
 2583 A REAL WINNER      49.99   89  10
 9998 NONESUCH FRAMMIS    2.69  416  50
 7777  THINGY             1.89    2   1
 8888 WHEEE               4.33 3011 100
 5555 WHIZBANG           22.22   43   2

      ORDERS
123  55 3569  5
123  55 1135 12
123  55 2583  4
131  55 3569 10
135 312 9998 12
136   7 1135  2
```

Actual Test:

```
      PLACE
 CUSTOMER NUMBER :  55
  STOCK NUMBER :  3569
  AMT ORDERED :  12
  STOCK NUMBER :  55
  AMT ORDERED :
***DATA NOT SAVED
  STOCK NUMBER :  5613
  AMT ORDERED :  12
  STOCK NUMBER :
 ORDER FOR STOCK   3569  REDUCED TO   10
```

Begin Invoice

```
MAIL HOUSE LTD.                                    141
7-11 RAMBLERS LANE
ISLAND CITY, S. DAK. 54321
                                        MAY 22, 1984


3569      SECOND MONEYMAKER 24.75     10     247.5
5613     MAIL ORDER SPECIAL 14.99     12    179.88

                                                427.38
```

End Invoice

Tables After Changes:

```
     STOCKS
1135 FIRST GREAT ITEM     9.95  118   55
9993 HIGH FLYER WIDGET   88.73  240   35
3569 SECOND MONEYMAKER   24.75    0   30   <----Changed
5613 MAIL ORDER SPECIAL  14.99  183   95   <----Changed
2583 A REAL WINNER       49.99   89   10
9998 NONESUCH FRAMMIS     2.69  416   50
7777 THINGY               1.89    2    1
8888 WHEEE                4.33 3011  100
5555 WHIZBANG            22.22   43    2

     ORDERS
123   55 3569   5
123   55 1135  12
123   55 2583   4
131   55 3569  10
135  312 9998  12
136    7 1135   2
141   55 3569  10          <----Added
141   55 5613  12          <----Added
```

Chapter 10. Formatting the Invoice: The INVOICE and FORMAT Routines

Producing an invoice consists of the following steps:

1. Getting the required information
2. Formatting a report suitable for printing

Getting the Invoice Data

Let's say *ORDERS*, the table of active orders, contains the following three entries:

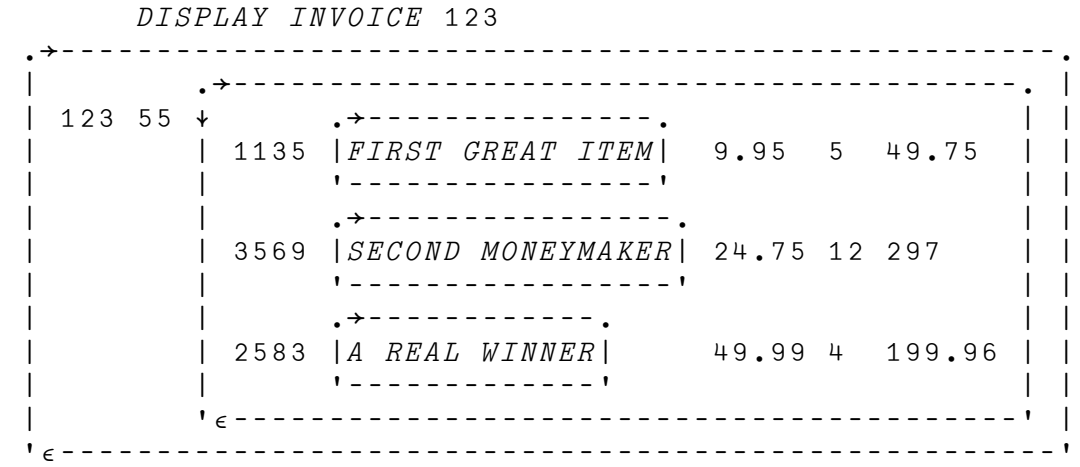
```
123   55 3569   5
123   55 1135  12
123   55 2583   4
```

The columns represent the invoice number, customer number, item, and quantity ordered.

You want to produce an invoice for these orders that contains the following:

- The invoice number
- The customer number
- A single item with the following stock information for each order:
  - Stock number
  - Description
  - Price per item
  - Quantity ordered
  - Price times quantity

Such an invoice would look like this:



Well, you know where all the data is. All you need is an invoice number as an argument, and you can get it:

```
[0]  INV←INVOICE N;STO;ORD
[1] ⍝ CALLED BY PLACE TO GET INVOICE DATA-
[2] ⍝ (INVOICE)(CUSTOMER)(ITEM,DESCRIPTION,PRICE,QUANTITY,COST)
[3]  INV←''
[4]  →(0=⍴ORD←'ORDERS' GET↑N)/0          ⍝ GET ORDERS
[5]  →(0=⍴STO←'STOCKS' GET ORD[;3])/0    ⍝ GET STOCK RECORDS
[6]  INV←(3↑[2]STO),¯1↑[2]ORD     ⍝ STOCK,DESCRIPTION,PRICE,QUANTITY
[7]  INV←INV,×/¯2↑[2]INV          ⍝ COST = PRICE × QUANTITY
[8]  INV←(2↑,ORD),⊂INV            ⍝ INVOICE, CUSTOMER NUMBERS
```

INVOICE[4] gets all orders for a single invoice number.

INVOICE[5] gets all stock records for the orders covered by the invoice. In the above example, *ORD*[3] is 3569 1135 2583.

INVOICE[6] creates records with the stock items, descriptions, and price, followed by the quantity. The expression:

```
      3↑[2]STO
```

takes the first three columns of *STO*. *INVOICE*[7] appends the cost.

INVOICE[8] returns a 3-item vector with the stock information enclosed as a single item, as displayed above.

You do not need to have *INVOICE* handle more than one invoice number: A list of valid invoice numbers could be handled with the primitive operator each (¨).

```
      ⊃INVOICE¨123 131
 123 55    1135 FIRST GREAT ITEM   9.95  5  49.75
           3569 SECOND MONEYMAKER 24.75 12 297
           2583 A REAL WINNER     49.99  4 199.96

 131 55    3569 SECOND MONEYMAKER 24.75 10 247.5
```

Well, what are you going to do with the information now that you've got it? Pass it to a report formatting function. See below.

Formatting the Report

Let's imagine that the output of the function called *INVOICE* is the following three-item vector, *INV*:

```
       INV
 123 55    3569 SECOND MONEYMAKER 24.75  5 123.75
           1135 FIRST GREAT ITEM   9.95 12 119.4
           2583 A REAL WINNER     49.99  4 119.96

      ↑INV
123
      2⊃INV
55
```

```
      3⊃INV
3569 SECOND MONEYMAKER 24.75  5 123.75
1135 FIRST GREAT ITEM   9.95 12 119.4
2583 A REAL WINNER     49.99  4 119.96
```

Now, to fit the preprinted order form shown in Figure 5 on page 24, you need to format *INV* and add some information to it, so that it looks like this:

```
  ADDRESS OF                                  123
  THIS
  CUSTOMER                          TODAY'S DATE


3569       SECOND MONEYMAKER 24.75   5  123.75
1135       FIRST GREAT ITEM   9.95  12  119.40
2583       A REAL WINNER     49.99   4  119.96

                                      TOTAL COST
```

You might think of the formatted invoice as having four parts:

1. The invoice number and date in the top right
2. The customer address in the top left
3. The part number information in the middle
4. The total cost below the part number information

In fact, thinking of the invoice this way suggests the steps you might follow in handling the formatting:

1. Get the invoice number and date and put that information in the top right.
2. Get the customer address and put it in the top left.
3. Get the part number information and put it in the middle.
4. Compute the total cost.
5. Put the total below the part-number information.
6. Put everything together.

And that's exactly what *FORMAT* does.

Here's what *FORMAT* looks like:

```
[0]   FORM←FORMAT INV;TOPR;TOPL;BODY;TOTAL
[1]  ⍝ FORMATS A SINGLE INVOICE
[2]  ⍝ SAMPLE CALL - FORMAT INVOICE 131
[3]   FORM←''
[4]   →(0=⍴INV)/0
[5]   TOPR←6↑[1]⊃¯18↑⍕¨(⍳↑INV)' ' ' ' DATE
[6]   TOPL←6 30↑⊃,ADDRESS(2⊃INV)
[7]   BODY←4 0 23 0 6 2 6 0 9 2⍕3⊃INV
[8]   TOTAL←¯48↑8 2⍕+/(3⊃INV)[;5]
[9]   BODY←BODY,[1]⊃' ' TOTAL
[10]  FORM←' ',[1](TOPL,TOPR),[1]BODY
```

The following section shows how the pieces are put together by *FORMAT*.

Top Right: The Date and Invoice Number

First, *FORMAT* creates *TOPR*, the formatted date and invoice number.

Today's Date: This is done with the niladic defined function *DATE*:

```
[0]    Z←DATE;⎕IO  ⍝ MONTH  DAY, YEAR
[1]    ⎕IO←1
[2]    Z←⎕TS
[3]    Z←(MONTHS[Z[2];]~' '),' 50, 0000'⍕Z[3 1]
```

DATE sets the index origin (⎕*IO*) to 1, so that it can be used in any workspace.

The system variable ⎕*TS* is assigned to *Z* at *FORMAT*[2], so that its value can be referenced more than once (otherwise, you could get the wrong month at midnight of the last day of the month). ⎕*TS* is a seven-item numeric vector that contains the current time and date.

The first three items are the year, month, and day:

```
      ⎕TS[2 3 1]
5 17 1984
```

Everything to the left of *Z*[3 1] puts the day and year into the format you want:

```
      ' 50, 0000' ⍕ ⎕TS[3 1]
 17, 1984
```

This statement uses **format by example** (⍕) to display a comma between the day and year. For detailed discussions of format by example, see *APL2 Programming: Language Reference* and *An Introduction to APL2*.

MONTHS is a matrix that lists the months of the year:

```
      MONTHS
JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER
      ⍴MONTHS
12 9
```

so that, if the month were May:

```
       MONTHS[2⊃⎕TS;]
MAY
```

the result of *DATE* would be:

```
MAY 17, 1984
```

The Invoice Number: The first item in *INV* is the invoice number. The expression ⍕⍬↑*INV* changes it to a three-item character vector:

```
      ⍴⍕⍬↑INV
3
```

You want the order number and date to look like this in the formatted invoice:

```
        123

MAY 7, 1984
```

In other words, you want two blank rows between the order number and the date:

```
      ⊃(⍕⍬↑INV) ' ' ' ' DATE
         123


MAY 17, 1984
```

You also want some blank space to the left of the block:

```
      ⊃¯18 ↑¨(⍕⍬↑INV) ' ' ' ' DATE
             123


    MAY 17, 1984
```

Remember that an “overtake” of a character array fills the extra positions in the result with blanks. Because this is a negative take (¯18↑), the extra positions are in the first positions of the pertinent axis. In other words, blanks are placed in the beginning of short rows, and rows are aligned on the right.

You also want some blank space below the block:

```
      TOPR←6↑[1]⊃¯18↑⍕¨(⍎↑INV) ' ' ' ' DATE
      TOPR
            123


   MAY 17, 1984
```

Because the take is positive (6↑[1]), the filling is done at the end of the pertinent axis. In this case, two blank rows are added to the bottom.

Top Left: The Customer Address

This is done by calling another subfunction, *ADDRESS*, and disclosing the result, to form a simple matrix. Here's *ADDRESS*.

```
[0]   Z←ADDRESS N;B
[1]   ⍝ RETURNS THE ADDRESS(ES) OF CUSTOMER(S) N
[2]   ⍝ IF NO ITEMS ARE FOUND, THE RESULT IS EMPTY
[3]   B←∈(1↑[2]CUSTOMERS)∈N     ⍝ FIND OCCURRENCES OF ORDERS
[4]   Z←B⌿1↓[2]CUSTOMERS        ⍝ SELECT ADDRESS VECTORS


      DISPLAY ADDRESS (2⊃INV)
  MAIL HOUSE LTD.
  711 RAMBLERS LANE
  ISLAND CITY, S. DAK. 54321
```

In *ADDRESS*[3], the expression (1↑[2]*CUSTOMERS*) gets the first column of *CUSTOMERS*. The occurrences of all orders in *N* are then found in the resulting vector. An enlist (∈) is done so that the boolean array *B* is a simple array.

In *ADDRESS*[4], *B* is used to select the needed rows for all columns but the first.

You have written *ADDRESS* to return more than one address, in anticipation of requirements such as creating mailing labels. For this reason, the result of *ADDRESS* is a matrix:

```
      DISPLAY ADDRESS 55
┌→──────────────────────────────────────────────────────────────────┐
↓ ┌→─────────────┐ ┌→─────────────────┐ ┌→────────────────────────┐ │
│ │MAIL HOUSE LTD.│ │711 RAMBLERS LANE │ │ISLAND CITY, S. DAK. 54321│ │
│ └───────────────┘ └──────────────────┘ └─────────────────────────┘ │
└∈──────────────────────────────────────────────────────────────────┘
```

If you disclose it, a rank 3 array is produced:

```
      DISPLAY ⊃ADDRESS 55
┌→─────────────────────────┐
↓↓MAIL HOUSE LTD.         │
│711 RAMBLERS LANE        │
│ISLAND CITY, S. DAK. 54321│
└─────────────────────────┘
```

To get a matrix, you ravel the result of *ADDRESS* first:

```
      DISPLAY ⊃,ADDRESS 55
┌→─────────────────────────┐
↓MAIL HOUSE LTD.          │
│711 RAMBLERS LANE        │
│ISLAND CITY, S. DAK. 54321│
└─────────────────────────┘
      ⍴⊃,ADDRESS 55
3 26
```

Finishing the Top Left: Again, you'd like some space around the result:

```
      TOPL←6 30↑⊃,ADDRESS (2⊃INV)
      TOPL
 MAIL HOUSE LTD.
 711 RAMBLERS LANE
 ISLAND CITY, S. DAK. 54321


      ⍴TOPL
6 30
```

Body: The Orders and Totals

This is the third item of *INV*:

```
      3⊃INV
3569 SECOND MONEYMAKER 24.75  5 123.75
1135 FIRST GREAT ITEM   9.95 12 119.4
2583 A REAL WINNER     49.99  4 119.96
```

You use **format by specification** (⍕) to format the result. Each pair of numbers to the left of ⍕ tells APL2 how many positions to leave for the column and how many decimal place positions to leave:

```
      BODY←4 0 23 0 6 2 3 0 9 2⍕3⊃INV
      BODY
3569   SECOND MONEYMAKER    24.75  5   123.75
1135   FIRST GREAT ITEM      9.95 12   119.40
2583   A REAL WINNER        49.99  4   119.96
```

The Total Cost

That's easy to get. All you have to do is get the price column; it's the fifth column in the third item of *INV*:

```
      (3⊃INV)[;5]
123.75 119.4 199.96
```

and add it up:

```
      +/(3⊃INV)[;5]
443.11
```

You now want to put the total below the part-order information. To do this, you format the total:

```
      8 2⍕+/(3⊃INV)[;5]
  443.11
```

and catenate it, as a new row, to the part number information. But because you want the total to line up underneath the price column, you've got to do some over-taking first:

```
      TOTAL←¯48↑8 2⍕+/(3⊃INV)[;5]
```

Now you can append it to the bottom, with a blank line inserted:

```
      BODY←BODY,[1]⊃' ' TOTAL
      BODY
3569   SECOND MONEYMAKER    24.75  5   123.75
1135   FIRST GREAT ITEM      9.95 12   119.40
2583   A REAL WINNER        49.99  4   199.96

                                        443.11
```

Putting the Invoice Together

At this point you have everything you need to build the formatted invoice:

- *TOPL* — Customer address
- *TOPR* — Invoice and date
- *BODY* — Part-order information and total cost

So let's put them together:

```
      FORM←' ',[1](TOPL,TOPR),[1]BODY
      FORM

  MAIL HOUSE LTD.                              123
  711 RAMBLERS LANE
  ISLAND CITY, S. DAK. 54321

                                     MAY 25, 1984

3569   SECOND MONEYMAKER     24.75   5    123.75
1135   FIRST GREAT ITEM       9.95  12    119.40
2583   A REAL WINNER         49.99   4    199.96

                                           443.11
```

FORMAT can also be used with the primitive each:

```
      ⊃ FORMAT¨ INVOICE¨ 123 131

MAIL HOUSE LTD.                                123
711 RAMBLERS LANE
ISLAND CITY, S. DAK. 54321
                                     MAY 25, 1984


1135   FIRST GREAT ITEM       9.95     5     49.75
3569   SECOND MONEYMAKER     24.75    12    297.00
2583   A REAL WINNER         49.99     4    199.96

                                           546.71



MAIL HOUSE LTD.                                131
711 RAMBLERS LANE
ISLAND CITY, S. DAK. 54321
                                     MAY 25, 1984


3569   SECOND MONEYMAKER     24.75    10    247.50

                                           247.50
```

Chapter 11. Creating the FILL and STOCK Functions

In the previous chapter we showed *PLACE*, the function used to place an order. That was the first of the three tasks on the menu. In this chapter, we'll cover the other two tasks; the filling of orders with the *FILL* function and the stocking of merchandise with the *STOCK* function.

Filling Orders: The FILL Function

To fill an order, the customer brings the invoice to the merchandise area. The clerk takes the invoice, and gets the listed items from the warehouse. After the order is filled, the clerk must delete the order from the list of outstanding orders.

Writing *FILL* is easy:

```
[0]    FILL;N
[1]  ⍝ DELETES AN ORDER WHEN IT IS FILLED
[2]    N←NUMIN 'INVOICE NUMBER'
[3]    →(0=⍴N)/0    ⍝ EXIT IF NO INPUT
[4]    'ORDERS' DELETE↑N
```

FILL[2] uses *NUMIN* to prompt for the invoice number.

FILL[3] exits if no number is entered.

FILL[4] calls a "tool" function, *DELETE*, to delete an order. *DELETE* was discussed in Chapter 7, "Updating Tables" on page 52. *DELETE* is isolated because it is a table-handling routine.

Only the first number entered is handled on *FILL*[4]. You could have chosen instead to modify your input routines so that you can specify how many items are allowed. If you did this with an optional left argument and made the default one item, your existing functions would be unaffected. However, you would have to account for ambi-valence in all your prompting functions and operators (see Figure 10 on page 50).

Testing *FILL*

Here's a test of *FILL*:

```
      ORDERS
131  55 3569 10
135 312 9998 12
136   7 1135  2
179  55 1135 12
      FILL
INVOICE NUMBER: 131
      ORDERS
135 312 9998 12
136   7 1135  2
179  55 1135 12
```

You also test an invalid entry:

```
      FILL
INVOICE NUMBER: 123
 NO ENTRY FOR 123
```

Stocking Merchandise: The STOCK Function

The last item on your menu is stocking merchandise, which will be done by the *STOCK* function. It has the following tasks to perform:

- Provide information for specified stock items.
- Restock existing items, especially those that have a low available inventory.
- Add items to inventory.

You have *STOCK* call separate functions for each of its tasks:

```
[0]   STOCK
[1]  ⍝ UPDATE THE 'STOCKS' TABLE
[2]   CHECK       ⍝ ITEMS TO CHECK
[3]   RESTOCK     ⍝ LIST LOW ITEMS; UPDATE
[4]   NEW         ⍝ NEW ITEMS
```

Checking Stock Items: The *CHECK* Function

The stock clerk will specify one or more item numbers to check, and get a report:

```
      CHECK
STOCK ITEMS TO CHECK: 1135 3569 2222
 ITEM(S)  2222  NOT FOUND
 STOCK NUMBER  DESCRIPTION        PRICE  INVENTORY  REORDER LEVEL
         1135 FIRST GREAT ITEM     9.95        118            55
         3569 SECOND MONEYMAKER   24.75         10            30
```

Because there's no report for item 2222, it is listed “not found.”

CHECK uses *NUMIN* and *GET*:

```
[0]    Z←CHECK;N
[1]  ⍝ RETURN STOCK REPORT FOR SELECTED ITEM
[2]    N←NUMIN 'STOCK ITEMS TO CHECK'
[3]    →(0=⍴Z←N)/0
[4]    Z←'STOCKS' GET N
[5]    →(0=⍴Z)/0
[6]    Z←(,2⊃¨STOCKQ),[1]Z
```

The report headings come directly from the prompting matrix, *STOCKQ*. *CHECK*[6] picks the second item of each item, ravels the result for conformity, and catenates it to the matrix of items selected from the table. The following example demonstrates this, using the entire *STOCKS* table:

```
      DISPLAY STOCKQ
.→------------------------.
↓ .→------------------.    |
| |    .→------------. |    |
| | 0 |STOCK NUMBER| |    |
| |    '------------' |    |
| '∈------------------'    |
| .→------------------.    |
| |    .→------------. |    |
| |    | DESCRIPTION| |    |
| | - '------------' |    |
| '∈------------------'    |
| .→------------.          |
| |    .→-----. |          |
| | 0 | PRICE| |          |
| |    '------' |          |
| '∈------------'          |
| .→----------------.      |
| |    .→----------. |      |
| | 0 | INVENTORY| |      |
| |    '----------' |      |
| '∈----------------'      |
| .→--------------------.  |
| |    .→-------------. |  |
| | 0 | REORDER LEVEL| |  |
| |    '--------------' |  |
| '∈--------------------' |
'∈------------------------'
```

```
      DISPLAY ,2⊃¨STOCKQ
.→-------------------------------------------------------------------------------.
| .→------------. .→------------. .→-----. .→----------. .→--------------. |
| |STOCK NUMBER| | DESCRIPTION| | PRICE| | INVENTORY| | REORDER LEVEL| |
| '------------' '------------' '------' '----------' '--------------' |
'∈-------------------------------------------------------------------------------'
```

```
      (,2⊃¨STOCKQ),[1]STOCKS
 ITEM  DESCRIPTION        PRICE  INVENTORY  REORDER LEVEL
 1135 FIRST GREAT ITEM     9.95        118             55
 9993 HIGH FLYER WIDGET   88.73        240             35
 3569 SECOND MONEYMAKER   24.75         10             30
 5613 MAIL ORDER SPECIAL  14.99        225             95
 2583 A REAL WINNER       49.99         89             10
 9998 NONESUCH FRAMMIS     2.69        416             50
 7777  THINGY              1.89          2              1
 8888 WHEEE                4.33       3011            100
 5555 WHIZBANG            22.22         43              2
```

Restocking Merchandise: The *RESTOCK* Function

We've got two more processing operations to take care of. The first, *RESTOCK*, will be used to add new shipments of a given part to the inventory list.

RESTOCK prints a list of low items, prompts the user for new information, and then updates *STOCKS* based on the user's input. *RESTOCK* also displays information about the part.

The *RESTOCK* dialog looks like this:

```
      RESTOCK                                          ◄———Called by STOCK
 LOW ITEMS:    3569                                    ◄———Program prints list of low items

STOCK NUMBER: 1135
 NOW READS:     1135 FIRST GREAT ITEM 9.95 118 55
INCREMENT: 30                                         ◄———User adds 30 1135s to stock
STOCK NUMBER: 3596                                    ◄———Invalid stock number
 ITEM(S)  3596  NOT FOUND
STOCK NUMBER: 3569
 NOW READS:     3569 SECOND MONEYMAKER 24.75 10 30
INCREMENT: 25
STOCK NUMBER:
```

The reply to *INCREMENT* is added to the current number on hand for the stock item.

CHECK can be used to check the result:

```
      CHECK
STOCK ITEMS TO CHECK: 1135 3569
 ITEM  DESCRIPTION        PRICE  INVENTORY  REORDER LEVEL
 1135 FIRST GREAT ITEM     9.95        148            55
 3569 SECOND MONEYMAKER   24.75         35            30
```

Now the user can do a check only by selecting *STOCK* from the menu. You make a note to add it to the menu if it becomes a frequent need. Or perhaps *RESTOCK* should print the modified lines.

You also make a note that it may be necessary to generate an order form for low items (in the above example, 30 more units of item 3569 had already been ordered and had just arrived). You will talk to the stock clerk again, to see if you can automate the order process.

As a developer of APL2 code, you do not pretend omniscience with regard to the needs of your application. Rather, you develop your application so that it is easily modifiable, with full awareness of the fact that your users may request extensive changes after they start to use the application and get the “feel” of what it can do, and of how they want to interact with it. You are confident that you have the ability to meet the challenge of further user demands, because you have talked to your users, taken a modular tool-oriented approach, and used tables in lieu of coding detail.

RESTOCK ***Line-by-Line:*** Here's what the complete *RESTOCK* function looks like:

```
[0]    RESTOCK;TYPES;MSG;LOW;A
[1]  ⍝ UPDATES THE STOCK TABLE
[2]    LOW←(≤/STOCKS[;4 5])/STOCKS[;1]
[3]    ⎕←'LOW ITEMS: ' LOW
[4]    MSG←,⊂(0,'STOCK NUMBER')(0,'INCREMENT')
[5]    A←RESTOCKIN REPEAT MSG
[6]    →(0=⍴A)/0
[7]    'STOCKS'PUT,¨↑¨A   ⍝ UPDATE STOCK TABLE
```

At lines [2] and [3], all items for which the available stock does not exceed the reorder level are listed as low.

RESTOCK[4] creates a prompting vector of depth 3, for use with the *REPEAT* operator.

RESTOCK[5] repeatedly calls *RESTOCKIN*, which gets and displays a row from *STOCKS* based on the stock number specified:

```
[0]    Z←RESTOCKIN MSG;X
[1]  ⍝ CALLED ITERATIVELY BY RESTOCK
[2]  L1:Z←ASK 1 1⊃MSG    ⍝ GET STOCK NUMBER
[3]    →(0=⍴Z)/0
[4]    X←'STOCKS' GET↑Z   ⍝ GET ROW OF STOCK TABLE
[5]    →(0=⍴X)/L1
[6]    'NOW READS:' X
[7]    Z←ASK 1 2⊃MSG      ⍝ GET INCREMENT
[8]    →(0=⍴Z)/0
[9]    X[;4]←X[;4]+↑Z     ⍝ ADD TO INVENTORY
[10]   Z←,⊂X
```

MSG is a length 1 vector so that the result of *ASK* will pass the shape check in *REPEAT*. In *RESTOCKIN*[2], you pass the first message to *ASK* by selecting the first item of *MSG*:

```
      DISPLAY 1 1⊃MSG
┌→─────────────────┐
│   ┌→───────────┐ │
│ 0 │STOCK NUMBER│ │
│   └────────────┘ │
└∊─────────────────┘
```

RESTOCKIN[6] shows the current row of *STOCKS* for the requested item.

RESTOCKIN[7] prompts for the amount to be added to stock.

RESTOCKIN[9] updates the current line, and line [10] returns it as a length 1 vector.

The result of *RESTOCKIN* is a doubly enclosed vector of vectors. Each inner vector contains a one-column matrix:

```
      DISPLAY A
┌→──────────────────────────────────────────────────────────────────┐
│ ┌→──────────────────────────┐ ┌→──────────────────────────────┐ │
│ │ ┌→──────────────────────┐ │ │ │ ┌→──────────────────────────┐ │ │ │
│ │ ↓      ┌→──────┐        │ │ │ │ ↓      ┌→────┐            │ │ │ │
│ │ │ 7777 │ THINGY│ 1.89 9 1 │ │ │ │ 8888 │WHEEE│ 4.33 3019 100 │ │ │ │
│ │ │      '───────'        │ │ │ │ │      '─────'            │ │ │ │
│ │ '∈──────────────────────' │ │ │ '∈──────────────────────────' │ │ │
│ '∈──────────────────────────' │ '∈──────────────────────────────' │ │
'∈──────────────────────────────────────────────────────────────────'
```

The expression at *RESTOCK*[6], (,¨↑¨*A*) strips away the outer layer and changes the matrices to vectors:

```
      DISPLAY ,¨↑¨A
┌→──────────────────────────────────────────────────────────────┐
│ ┌→──────────────────────┐ ┌→──────────────────────────┐ │
│ │      ┌→──────┐        │ │      ┌→────┐            │ │
│ │ 7777 │ THINGY│ 1.89 9 1 │ │ 8888 │WHEEE│ 4.33 3019 100 │ │
│ │      '───────'        │ │      '─────'            │ │
│ '∈──────────────────────' │ '∈──────────────────────────' │
'∈──────────────────────────────────────────────────────────────'
```

The resulting vector of vectors is suitable for input to *PUT*.

What If the User Makes a Mistake?: You will have to have an update capability for *STOCKS* that allows an entire row to be reentered for an existing item. You have what you need to do this for *any* table. You called it *UPDATE*, and you wrote it when you finished writing *PUT* and *GET*.

Handling New Stock Items: The *NEW* Function

The last function called by *STOCK* is *NEW*. Not surprisingly, it handles new stock items. Writing it is “old hat” to you by now:

```
[0]   NEW;STO
[1]  ⍝ ADD ITEMS TO STOCKS TABLE
[2]   ⎕←'NEW STOCK ITEMS: '
[3]   STO←ASK EACH REPEAT STOCKQ
[4]   →(0=⍴STO)/0
[5]   'STOCKS' PUT STO
```

Testing STOCK

Here's a test of *STOCK*:

```
      STOCK
STOCK ITEMS TO CHECK: 3569
 STOCK NUMBER  DESCRIPTION        PRICE  INVENTORY  REORDER LEVEL
         3569 SECOND MONEYMAKER  24.75         10            30
 LOW ITEMS:   3569
STOCK NUMBER: 3569
 NOW READS:    3569 SECOND MONEYMAKER 24.75 10 30
INCREMENT: 20
STOCK NUMBER:
NEW STOCK ITEMS:
 STOCK NUMBER :  2222
  DESCRIPTION :  TUTUS
  PRICE :  22.22
  INVENTORY :  22
  REORDER LEVEL :  2
 STOCK NUMBER :

      STOCKS
 1135 FIRST GREAT ITEM    9.95  118  55
 9993 HIGH FLYER WIDGET  88.73  240  35
 3569 SECOND MONEYMAKER  24.75   30  30     <-----Updated
 5613 MAIL ORDER SPECIAL 14.99  195  95
 2583 A REAL WINNER      49.99   89  10
 9998 NONESUCH FRAMMIS    2.69  416  50
 7777  THINGY             1.89    2   1
 8888 WHEEE               4.33 3011 100
 5555 WHIZBANG           22.22   43   2
 2222 TUTUS              22.22   22   2
```

Chapter 12. Using SQL Tables

You now have a fairly complete application for use in an APL2 workspace. To ensure that data gets stored and referenced properly, and is available for use by more than one person at one time, you must be able to store the data in files and retrieve it from them. You choose to do this with one of the data base management systems that support SQL, either SQL/Data System (SQL/DS*) or IBM DATABASE 2* (DB2*).

You can use SQL tables through the APL2 auxiliary processor, AP 127. For information on how to use AP 127, see:

APL2 Programming: Using Structured Query Language (SQL)

For information on programming using SQL, see:

IBM DATABASE 2: Introduction to SQL

and:
IBM DATABASE 2: Application Programming Guide for TSO Users

or:
SQL/Data System Application Programming

You will also have to know how to obtain table access authorization. For information about that, see:

IBM DATABASE 2: Data Base Planning and Administration Guide

and:
SQL/Data System Planning and Administration

SQL Tables and Nested Arrays

The structure of an SQL table is totally compatible with the form you have chosen for your tables: Nested matrices containing numbers and character strings, with columns all of one type. For a good example, look at the `STOCKS` array:

```
      DISPLAY STOCKS
┌→──────────────────────────────────────────┐
↓      ┌→──────────────┐                    │
│ 1135 │FIRST GREAT ITEM│   9.95  118  55   │
│      └────────────────┘                    │
│      ┌→─────────────────┐                  │
│ 9993 │HIGH FLYER WIDGET│  88.73 240  35   │
│      └──────────────────┘                  │
│      ┌→──────────────────┐                 │
│ 3569 │SECOND MONEYMAKER│  24.75 10   30   │
│      └───────────────────┘                 │
│      ┌→────────────────────┐               │
│ 5613 │MAIL ORDER SPECIAL│ 14.99 225  95   │
│      └─────────────────────┘               │
│      ┌→─────────────┐                      │
│ 2583 │A REAL WINNER│      49.99 89   10   │
│      └─────────────┘                       │
│      ┌→─────────────────┐                  │
│ 9998 │NONESUCH FRAMMIS│   2.69  416  50   │
│      └──────────────────┘                  │
│      ┌→──────┐                             │
│ 7777 │ THINGY│            1.89  2    1    │
│      └───────┘                             │
│      ┌→────┐                               │
│ 8888 │WHEEE│              4.33  3011 100  │
│      └─────┘                               │
│      ┌→────────┐                           │
│ 5555 │WHIZBANG│           22.22 43   2    │
│      └─────────┘                           │
└←──────────────────────────────────────────┘
```

The SQL CREATE Statement

To initialize your SQL tables, you must use an SQL CREATE statement. You can write one by using EDITOR 2, or by using the *IN* function:

```
      CSTOCKS←IN
CREATE TABLE STOCKS
 (ITEM SMALLINT,
  DESCRIPTION CHAR(20),
  PRICE DECIMAL(6,2),
  QUANTITY SMALLINT,
  REORDER SMALLINT)
```

The CREATE statement can be scheduled using the SQL workspace function *SQL*:

```
      SQL CSTOCKS
0 0 0 0 0              ←——Return code shows success
```

Similarly, CREATE statements can be done for *ORDERS* and *CUSTOMERS* (see the *SQLCREATES* array in the foldout). You then add rows to the tables with SQL INSERT statements, update them with UPDATE statements, and select from them with SELECT statements.

Creating SQL Tables

You want to be able to either update or insert rows of a table. The table will be one of the three named in *TABLES*:

```
      TABLES
 STOCKS CUSTOMERS ORDERS
```

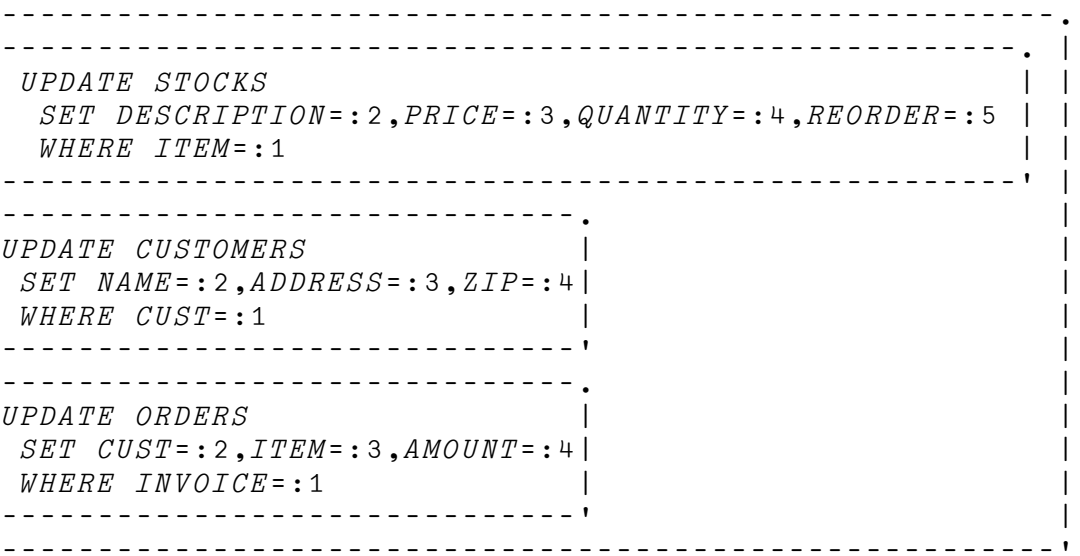
To allow statement selection, you can write a matrix of SQL UPDATE statements:

```
      SQLUPDATES←⊂IN
 UPDATE STOCKS
  SET DESCRIPTION=:2,PRICE=:3,QUANTITY=:4,REORDER=:5
  WHERE ITEM=:1

      SQLUPDATES←SQLUPDATES,⊂IN
 UPDATE CUSTOMERS
  SET NAME=:2,ADDRESS=:3,ZIP=:4
  WHERE CUST=:1

      SQLUPDATES←SQLUPDATES,⊂IN
 UPDATE ORDERS
  SET CUST=:2,ITEM=:3,AMOUNT=:4
  WHERE INVOICE=:1

      SQLUPDATES←,[⍳0]SQLUPDATES
      DISPLAY SQLUPDATES
```



Similarly, you can write a matrix of SQL INSERT statements:

```
      SQLINSERTS
 INSERT INTO STOCKS
  (ITEM,DESCRIPTION,PRICE,QUANTITY,REORDER)
  VALUES(:1,:2,:3,:4,:5)

 INSERT INTO CUSTOMERS
  (CUST,NAME,ADDRESS,ZIP)
  VALUES(:1,:2,:3,:4)

 INSERT INTO ORDERS
  (INVOICE,CUST,ITEM,AMOUNT)
  VALUES(:1,:2,:3,:4)
```

You can select the appropriate statement from a statement table by finding the position of the table name in *TABLES*:

```
      (TABLES⍳⊂'CUSTOMERS')⊃,SQLINSERTS

INSERT INTO CUSTOMERS
 (CUST,NAME,ADDRESS,ZIP)
 VALUES(:1,:2,:3,:4)
```

These statements can then be passed to AP 127. The easiest way to do this is by using the cover functions in the APL2 distributed workspace, SQL. You write *PUTSQL* with the assumption that you have copied *SQLGROUP* from the SQL workspace into your *INVENTORY* workspace.

The *PUTSQL* Function

The arguments of *PUTSQL* are *NAME*, a table name, and *V*, a vector of new or changed table rows:

```
[0]   NAME PUTSQL V;E;S;T
[1]  ⍝ REQUIRES AP127 AND THE SQL WORKSPACE
[2]  ⍝ SQL UPDATE - V IS THE NEW VALUES
[3]  ⍝ NAME IS THE TABLE NAME
[4]   →('INVALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[5]   T←TABLES⍳⊂NAME
```

After ensuring that the table name is a known one, you obtain the corresponding index, *T*.

First you select an update statement and pass it to the *PREP* function along with a statement name, *UP*, which you will use to refer to this statement:

```
[6]   S←T⊃,SQLUPDATES          ⍝ SELECT THE STATEMENT
[7]   E←↑PREP 'UP' S           ⍝ PREPARE - E IS RETURN CODE
```

PREP prepares the statement for a subsequent call. The first item of the result is the return code; you assign it to *E*.

E is a five-item boolean vector. If the first item of *E* is 1, an error has occurred. In such a case the SQL workspace function *MESSAGE* returns the appropriate message:

```
[8]   →((MESSAGE E)ERR↑E)/0  ⍝ ERROR
```

You now use the same process to prepare an INSERT statement:

```
[9]   S←T⊃,SQLINSERTS          ⍝ INSERT STATEMENT
[10]  E←↑PREP 'IN' S           ⍝ PREPARE AN INSERT
[11]  →((MESSAGE E)ERR↑E)/0    ⍝ ERROR
```

To handle the values, you call another function recursively, using *EACH*:

```
[12]  SQLUPIN ¨ V              ⍝ UPDATE OR INSERT
```

The *SQLUPIN* function will schedule SQL calls to update or insert the row of values passed to it:

```
[0]   SQLUPIN V;E
[1]  ⍝ UPDATE USING CURSOR 'UP', VALUES V
[2]  ⍝ IF UPDATE FAILS, INSERT USING 'IN'
[3]   E←↑CALL 'UP' V                ⍝ UPDATE
[4]   →((MESSAGE E)ERR↑E)/0         ⍝ ERROR
[5]   →(~2⊃E)/0    ⍝ EXIT UNLESS NOT FOUND
[6]   E←↑CALL 'IN' V                ⍝ INSERT
[7]   →0⍴MESSAGE E
```

If the second item of *E* is 1 upon return from the update, the record was not found, and must be inserted. This decision is made at *SQLUPIN*[5].

Getting and Deleting Data from SQL Tables

To do a *GET*, you need a table of SQL SELECT statements:

```
      SQLSELECTS
SELECT * FROM STOCKS
 WHERE ITEM = :1

SELECT * FROM CUSTOMERS
 WHERE CUST = :1

SELECT * FROM ORDERS
 WHERE INVOICE =:1
```

The * indicates that all columns are to be selected. The ":1" in the WHERE clause refers to the first column of the array that you pass to AP 127.

The *GETSQL* Function

To get the row corresponding to the key field *N*, you schedule the appropriate SELECT statement, using the SQL workspace function SQL:

```
[0]    V←NAME GETSQL N;E;S;T
[1]   ⍝ REQUIRES AP 127 AND THE SQL WORKSPACE
[2]   ⍝ GET RECORD N FROM THE NAMED SQL TABLE
[3]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[4]    T←TABLES⍳⊂NAME
[5]    S←T⊃,SQLSELECTS          ⍝ SELECT THE STATEMENT
[6]    E←↑V←SQL S N             ⍝ SELECT - E IS RETURN CODE
[7]    V←2⊃V                    ⍝ SECOND ITEM IS RESULT
[8]    MESSAGE E                ⍝ PRINT MESSAGE IF ERROR
```

The second item of the result contains the data.

The *DELSQL* Function

Here is the code to delete rows using the SQL DELETE statement:

```
[0]    V←NAME DELSQL N;E;S;T
[1]   ⍝ REQUIRES AP 127 AND THE SQL WORKSPACE
[2]   ⍝ DELETE RECORD N FROM THE NAMED SQL TABLE
[3]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[4]    T←TABLES⍳⊂NAME
[5]    S←T⊃,SQLDELETES          ⍝ SELECT THE STATEMENT
[6]    E←↑V←SQL S N             ⍝ SELECT - E IS RETURN CODE
[7]    MESSAGE E                ⍝ PRINT MESSAGE IF ERROR

      SQLDELETES
  DELETE FROM STOCKS
   WHERE ITEM = :1

  DELETE FROM CUSTOMERS
   WHERE CUST = :1

  DELETE FROM ORDERS
   WHERE INVOICE =:1
```

Reflections: How Can I Improve the Application?

Now that you have started to use SQL, you can look for ways to improve your application with respect to SQL use.

You certainly want to store *INVO* somewhere, perhaps as a dummy last entry in the *ORDERS* file.

Perhaps you should use an SQL DESCRIBE statement to get the column descriptions and use these to drive your prompting, rather than having to reference prompting matrices in your workspace. See *APL2 Programming: Using Structured Query Language (SQL)* for information about using DESCRIBE.

The type information used for prompting could come from the column prototypes of the data tables. That is, the following are equivalent:

```
      DISPLAY ,↑¨STOCKQ          ⍝ TYPE PORTION OF PROMPTING TABLE
.→--------.
|0    0 0 0|
'+--------'

      DISPLAY ↑¨↑0⍴⊂[2]STOCKS    ⍝ FIRST ITEM OF EACH COLUMN PROTOTYPE
.→--------.
|0    0 0 0|
'+--------'
```

Perhaps rows can be updated directly rather than through separate GET and PUT statements.

Of course, you want to do whatever you do without loss of generality. Above all, you want your code to retain its flexibility and readability. As you go through a production test you will look for opportunities to reduce your logic to the essential, while retaining the ability to respond to your user's needs. You know you can do that, with APL2.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

```
ACTORD
[0]   ACT←ACTORD ORD;STO;NEW
[1]  ⍝ ACTUAL ORDERS BASED ON STOCK AVAILABLE
[2]  ⍝ ORD IS AN ITEM, QUANTITY
[3]   ACT←ORD[1],0
[4]   →(0=⍴STO←'STOCKS' GET↑ORD)/0  ⍝ STOCK RECORDS
[5]   NEW←⌈/0,STO[;4]-ORD[2]        ⍝ MAX NEW IS 0
[6]  ⍝ IF NEW IS ZERO, THE ORDER IS ALL AVAILABLE
[7]   ACT[2]←(1 0=NEW=0)/STO[;4],ORD[2]
[8]   STO[;4]←NEW
[9]   'STOCKS' PUT⊂[2]STO  ⍝ UPDATE STOCK
[10]  →(ACT[2]=ORD[2])/0
[11]  ⎕←'ORDER FOR STOCK ' STO[;1]'REDUCED TO ' ACT[2]

ADDRESS
[0]  Z←ADDRESS N;B
[1]  ⍝ RETURNS THE ADDRESS(ES) OF CUSTOMER(S) N
[2]  ⍝ IF NO ITEMS ARE FOUND, THE RESULT IS EMPTY
[3]   B←∈(1↓[2]CUSTOMERS)∈N     ⍝ FIND OCCURRENCES OF ORDERS
[4]   Z←B⌿1↓[2]CUSTOMERS        ⍝ SELECT ADDRESS VECTORS

CHECK
[0]  Z←CHECK;N
[1]  ⍝ RETURN FIRST STOCK REPORT FOR SELECTED ITEM
[2]   N←NUMIN 'STOCK ITEMS TO CHECK'
[3]   →(0=⍴Z←N)/0
[4]   Z←'STOCKS' GET N
[5]   →(0=⍴Z)/0
[6]   Z←(,2⊃¨STOCKQ),[1]Z

DATE
[0]  Z←DATE;⎕IO  ⍝ MONTH  DAY, YEAR
[1]   ⎕IO←1
[2]   Z←⎕TS
[3]   Z←(MONTHS[Z[2];]~' '),' 50, 0000'⍕Z[3 1]

FILL
[0]  FILL;N
[1]  ⍝ DELETES AN ORDER WHEN IT IS FILLED
[2]   N←NUMIN 'INVOICE NUMBER'
[3]   →(0=⍴N)/0      ⍝ EXIT IF NO INPUT
[4]   'ORDERS' DELETE↑N

FORMAT
[0]   FORM←FORMAT INV;TOPR;TOPL;BODY;TOTAL
[1]  ⍝ FORMATS A SINGLE INVOICE
[2]  ⍝ SAMPLE CALL - FORMAT INVOICE 131
[3]   FORM←''
[4]   →(0=⍴INV)/0
[5]   TOPR←6↑[1]⊃¯18↑¨(⍕↑INV)' ' ' ' DATE
[6]   TOPL←6 30↑⊃,ADDRESS(2⊃INV)
[7]   BODY←4 0 23 0 6 2 6 0 9 2⍕3⊃INV
[8]   TOTAL←¯48↑8 2⍕+/(3⊃INV)[;5]
[9]   BODY←BODY,[1]⊃' ' TOTAL
[10]  FORM←' ',[1](TOPL,TOPR),[1]BODY
```

```
INVOICE
[0]   INV←INVOICE N;STO;ORD
[1]  ⍝ CALLED BY PLACE TO GET INVOICE DATA-
[2]  ⍝ (INVOICE)(CUSTOMER)(ITEM,DESCRIPTION,PRICE,QUANTITY,COST)
[3]   INV←''
[4]   →(0=⍴ORD←'ORDERS' GET↑N)/0          ⍝ GET ORDERS
[5]   →(0=⍴STO←'STOCKS' GET ORD[;3])/0    ⍝ GET STOCK RECORDS
[6]   INV←(3↑[2]STO),¯1↑[2]ORD      ⍝ STOCK,DESCRIPTION,PRICE,QUANTITY
[7]   INV←INV,×/¯2↑[2]INV           ⍝ COST = PRICE × QUANTITY
[8]   INV←(2↑,ORD),⊂INV             ⍝ INVOICE, CUSTOMER NUMBERS
```

```
MENU
[0]  MENU
[1]  ⍝ SELECT AN ITEM
[2]  L1:''
[3]   WHAT←INPUT 'CHOOSE',3⍴'OR',¯1⍴ITEMS
[4]   →(0=⍴WHAT)/0  ⍝ TERMINATE IF EMPTY RESPONSE
[5]   →('REQUEST NOT RECOGNIZED' ERR~(⊂WHAT)∈ITEMS)/L1
[6]   '⎕←⎕EM[1;]' ⎕EA WHAT
[7]   →L1  ⍝ ASK AGAIN

NEW
[0]  NEW;STO
[1]  ⍝ ADD ITEMS TO STOCKS TABLE
[2]   ⎕←'NEW STOCK ITEMS: '
[3]   STO←ASK EACH REPEAT STOCKQ
[4]   →(0=⍴STO)/0
[5]   'STOCKS' PUT STO

NEWCUST
[0]   NEWCUST CUST;DATA
[1]  ⍝ ADD A NEW CUSTOMER
[2]   →(CUSTOMERS[;1]∈CUST)/0    ⍝ NOT NEW
[3]   ⎕←⎕TC[2],'NEW CUSTOMER'
[4]   →(0=⍴DATA←ASK EACH 1↓[1]CUSTQ)/0
[5]   →(WARNING ERR (⍴DATA)≠¯1+↑⍴CUSTQ)/0
[6]   'CUSTOMERS' PUT,⊂CUST,DATA  ⍝ UPDATE TABLE
[7]   ⎕←'CUSTOMER RECORD UPDATED',⎕TC[2]

PLACE
[0]   PLACE;ORD;CUST;ORDQ
[1]  ⍝ PLACE AN ORDER
[2]   →(0=⍴CUST←ASK↑ORDERQ)/0 ⍝ GET CUSTOMER NUMBER
[3]   NEWCUST CUST            ⍝ ADD NEW CUSTOMER
[4]   ORDQ←1↓[1]ORDERQ        ⍝ ITEMS TO PROMPT FOR
[5]   →(0=⍴ORD←ASK EACH REPEAT ORDQ)/0
[6]  ⍝ ACTORD FINDS ACTUAL ORDERS AND UPDATES 'STOCKS'
[7]   ORD←ACTORD EACH ORD
[8]   →(0=⍴ORD←(∈0≠2⊃¨ORD)/ORD)/0 ⍝ DELETE EMPTY ORDERS
[9]   INVO←INVO+1               ⍝ NEW INVOICE NUMBER
[10]  ORD←INVO,¨CUST,¨ORD       ⍝ APPEND TO ENTRIES
[11]  'ORDERS' PUT ORD          ⍝ UPDATE ORDERS
[12]  FORMAT INVOICE↑↑ORD       ⍝ DISPLAY INVOICE

RESTOCK
[0]  RESTOCK;TYPES;MSG;LOW;A
[1]  ⍝ UPDATES THE STOCK TABLE
[2]   LOW←(≤/STOCKS[;4 5])/STOCKS[;1]
[3]   ⎕←'LOW ITEMS: ' LOW
[4]   MSG←,⊂(0,'STOCK NUMBER')(0,'INCREMENT')
[5]   A←RESTOCKIN REPEAT MSG
[6]   →(0=⍴A)/0
[7]   'STOCKS'PUT,¨↑¨A  ⍝ UPDATE STOCK TABLE

RESTOCKIN
[0]   Z←RESTOCKIN MSG;X
[1]  ⍝ CALLED ITERATIVELY BY RESTOCK
[2]  L1:Z←ASK 1 1⊃MSG   ⍝ GET STOCK NUMBER
[3]   →(0=⍴Z)/0
[4]   X←'STOCKS' GET↑Z  ⍝ GET ROW OF STOCK TABLE
[5]   →(0=⍴X)/L1
[6]   'NOW READS:' X
[7]   Z←ASK 1 2⊃MSG     ⍝ GET INCREMENT
[8]   →(0=⍴Z)/0
[9]   X[;4]←X[;4]+↑Z    ⍝ ADD TO INVENTORY
[10]  Z←,⊂X
```

```
STOCK
[0]  STOCK
[1]  ⍝ UPDATE THE 'STOCKS' TABLE
[2]   CHECK       ⍝ ITEMS TO CHECK
[3]   RESTOCK     ⍝ LOW ITEMS
[4]   NEW         ⍝ NEW ITEMS
```

```
AGAIN
[0]  Z←AGAIN  ⍝ RETURN 1 TO RETRY

[1]   ⎕ES(⎕ET ≡1 3)/⎕ET  ⍝ CHECK FOR WS FULL
[2]   ⎕←'RETRY (Y/N)? '
[3]   Z←'Y'=↑⍞

ASK
[0]  Z←ASK TT;CALL
[1]  ⍝ PROMPT THE USER; CHECK REPLY
[2]  ⍝ IF SOMETHING GOES WRONG, REPROMPT THE USER
[3]  ⍝ TT IS (TYPE OF INPUT) (TEXT OF MESSAGE)
[4]   CALL←(0 ' '=↑TT)/'NUMIN 1↓TT' 'INPUT 1↓TT'
[5]   ⎕ES(0=⍴CALL)/'PROGRAM ERROR - INVALID MESSAGE TYPE'
[6]   Z←⍎↑CALL

DELETE
[0]  NAME DELETE N;T
[1]  ⍝ DELETE THE ROW OF TABLE NAMED 'NAME' WHOSE KEY IS N
[2]   T←(TABLES∊⊂NAME)/⍳⍴TABLES
[3]   →('INVALID TABLE NAME' ERR 0=⍴T)/0
[4]   TAB←⍎NAME
[5]   →(('NO ENTRY FOR' N)ERR~N∊TAB[;1])/0
[6]   TAB←(~TAB[;1]∊N)⌿TAB
[7]   ⍎NAME,'←TAB'

DLTMB
[0]  Z←DLTMB A
[1]  ⍝ DELETE LEADING, TRAILING, MULTIPLE BLANKS
[2]   Z←' ',A,' '

[3]   Z←(~' '∈Z)/Z
[4]   Z←1↓¯1↓Z

EACH
[0]   Z←(F EACH)R;X;I;N
[1]  ⍝ THE DERIVED FUNCTION (F EACH) WILL PROCESS ALL
[2]  ⍝ OF THE ITEMS IN VECTOR R, APPENDING THE RESULTS.
[3]  ⍝ IF (F R) RETURNS AN EMPTY, (F EACH) EXITS
[4]   ⎕ES(3≠⎕NC 'F')/'ARGUMENT MUST BE A FUNCTION'
[5]   Z←''                  ⍝ INITIALIZE RESULT TO EMPTY
[6]   →(0=⍴R)/0              ⍝ EXIT IF R IS EMPTY
[7]   I←⎕IO                 ⍝ INITIALIZE INDEX
[8]   N←I+⍴R←,R             ⍝ LOOP CONTROL VALUE
[9]  L1:X←F I⊃R             ⍝ CALL FUNCTION F
[10]  →(0=⍴X)/0             ⍝ END - EXIT WITH AN EMPTY
[11]  Z←Z,⊂X                ⍝ APPEND RESULT
[12]  I←I+1                 ⍝ INCREMENT INDEX
[13]  →(N>I)/L1             ⍝ PROCESS NEXT ITEM IN R

ERR
[0]  Z←MSG ERR COND  ⍝ ERROR MESSAGE HANDLER
[1]  ⍝ PRINT MSG IF COND IS 1 ; RETURN COND
[2]   Z←COND        ⍝ RETURN COND
[3]   →(~Z)/0       ⍝ EXIT IF COND IS ZERO
[4]   MSG

GET
[0]   Z←NAME GET N
[1]  ⍝ GET ITEMS N FROM THE TABLE NAMED 'NAME'
[2]   Z←NAME GETW N  ⍝ GET FROM WORKSPACE TABLE
[3]  ⍝ FOR SQL SELECTS, USE  NAME GETSQL N

GETW
[0]   Z←NAME GETW N;B
[1]  ⍝ GET ITEMS N FROM THE TABLE NAMED 'NAME'
[2]   Z←''
[3]   →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∊TABLES)/0
[4]   TAB←⍎NAME
[5]   B←N∊TAB[;1]
[6]   →0⍴('ITEM(S)'((~B)/N)'NOT FOUND')ERR~∧/B
[7]   →(~∨/B)/0     ⍝ EXIT IF NONE FOUND
[8]   Z←(TAB[;1]∊N)⌿TAB

INPUT
[0]  Z←INPUT MSG;⎕PR
[1]  ⍝ PROMPT FOR INPUT
[2]   ⎕PR←↑⎕TC       ⍝ BACKSPACE IS PROMPT REPLACE
[3]   ⍞←MSG,': '     ⍝ DISPLAY THE PROMPT
[4]   Z←⍞~⎕PR        ⍝ ASSIGN THE RESPONSE
[5]   Z←DLTMB Z      ⍝ DELETE EXCESS BLANKS

NUMIN
[0]  Z←NUMIN MSG
[1]  ⍝ PROMPTS FOR POSITIVE NUMBERS
[2]  L1:Z←INPUT MSG            ⍝ PROMPT FOR INPUT
[3]   →(0=⍴Z)/0                ⍝ EXIT IF INPUT IS EMPTY
[4]  ⍝ CHECK FOR VALID NUMBER AND CONVERT
[5]   Z←(∧/Z∊'0123456789. ')/Z ⍝ NUMERICS ONLY
[6]   '→L2' ⎕EA 'Z←⍎Z'         ⍝ EVALUATE
[7]   →0                       ⍝ EXIT IF OK
[8]  L2:⎕←'INVALID NUMBER'
[9]   →AGAIN/L1                ⍝ RETRY IF DESIRED

PUT
[0]   NAME PUT A   ⍝ UPDATE A TABLE
[1]   NAME PUTW⊃A  ⍝ UPDATE WORKSPACE TABLE
[2]  ⍝ FOR SQL UPDATES, USE   NAME PUTSQL A

PUTW
[0]   NAME PUTW NEW;OLD;TAB;I
[1]  ⍝ UPDATE THE TABLE NAMED 'NAME' WITH 'NEW'
[2]   →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∊TABLES)/0
[3]   TAB←⍎NAME                ⍝ TABLE
[4]   OLD←NEW[;1]∊TAB[;1]      ⍝ EXISTING ROWS
[5]   I←TAB[;1]⍳OLD/NEW[;1]    ⍝ INDICES OF OLD
[6]   TAB[I;]←OLD⌿NEW          ⍝ REPLACE
[7]   TAB←TAB,[1](~OLD)⌿NEW    ⍝ APPEND NEW
[8]   ⍎NAME,'←TAB'             ⍝ ASSIGN TO TABLE

REPEAT
[0]   Z←(F REPEAT)R;X
[1]  ⍝ THE DERIVED FUNCTION (F REPEAT) WILL CALL F
[2]  ⍝ REPEATEDLY UNTIL F RETURNS AN EMPTY RESULT
[3]  ⍝ PRINTS 'WARNING' IF (⍴,R) DOES NOT MATCH ⍴(F R)
[4]   ⎕ES(3≠⎕NC 'F')/'ARGUMENT MUST BE A FUNCTION'
[5]   Z←''              ⍝ INITIALIZE RESULT TO EMPTY
[6]  L1:X←F R           ⍝ CALL FUNCTION F WITH R
[7]   →(0=⍴X)/0         ⍝ EXIT IF RESULT IS EMPTY
[8]   →(WARNING ERR(⍴,R)≠⍴X)/L1  ⍝ WARNING MESSAGE
[9]   Z←Z,⊂X            ⍝ APPEND RESULT
[10]  →L1               ⍝ REPEAT UNTIL X IS EMPTY

UPDATE
[0]   UPDATE;WHAT;I;MSG
[1]  ⍝ UPDATE AN INVENTORY TABLE
[2]   →(0=⍴WHAT←INPUT 'CHOOSE',3⍕'OR',¯1⍕TABLES)/0
[3]   I←TABLES⍳⊂WHAT            ⍝ TABLE INDEX
[4]   →('REQUEST NOT RECOGNIZED' ERR I>⍴TABLES)/0
[5]   MSG←⍎I⊃PROMPTS            ⍝ SELECT PROMPT TABLE
[6]   A←ASK EACH REPEAT MSG     ⍝ PROMPT FOR INPUT
[7]   →(0=⍴A)/0         ⍝ EXIT IF EMPTY RESPONSE
[8]   WHAT PUT A        ⍝ PUT ROWS 'A' TO TABLE 'WHAT'
```

```
CUSTOMERS
      DISPLAY CUSTOMERS
 .→-------------------------------------------------------------------------.
 ↓     .→----------------.  .→--------------.      .→------------------.     |
 | 7   |CITY TRADERS INC.|  |41 POSTAGE ROAD|      |RIMELA, N.Y. 12345|     |
 |     '-----------------'  '---------------'      '------------------'     |
 |     .→----------------.  .→------------------.  .→--------------------.  |
 | 55  |MAIL HOUSE LTD.|     |711 RAMBLERS LANE |   |ISLAND CITY, S. DAK. 54321|  |
 |     '----------------'    '------------------'   '--------------------'  |
 |     .→------------------. .→--------------------. .→--------------------.  |
 | 312 |MANTUP SALES CORP.| |RURALIA FARMS, RFD 2| |SUBURBIA, WIS. 00000|     |
 |     '------------------' '--------------------' '--------------------'     |
 '←------------------------------------------------------------------------'

CUSTQ
      DISPLAY CUSTQ
 .→--------------------------.
 ↓ .→--------------------.   |
 | | .→---------------. |   |
 | | 0 |CUSTOMER NUMBER| |   |
 | | '---------------' |   |
 | '←--------------------'   |
 | .→--------------------.   |
 | | .→--------------. |     |
 | | | CUSTOMER NAME| |     |
 | | - '--------------' |   |
 | '←--------------------'   |
 | .→-------------.          |
 | | .→--------. |           |
 | | | ADDRESS| |           |
 | | - '--------' |          |
 | '←-------------'          |
 | .→------------------------.|
 | | .→-----------------. | |
 | | | CITY, STATE, ZIP| | |
 | | - '-----------------' | |
 | '←----------------------' |
 '←--------------------------'

INVO
     DISPLAY INVO
 140

ITEMS
      DISPLAY ITEMS
 .→-----------------------.
 | .→----. .→---. .→----. |
 | |PLACE| |FILL| |STOCK| |
 | '-----' '----' '-----' |
 '←-----------------------'

MONTHS
      DISPLAY MONTHS
 .→---------.
 ↓JANUARY   |
 |FEBRUARY  |
 |MARCH     |
 |APRIL     |
 |MAY       |
 |JUNE      |
 |JULY      |
 |AUGUST    |
 |SEPTEMBER |
 |OCTOBER   |
 |NOVEMBER  |
 |DECEMBER  |
 '----------'

ORD
      DISPLAY ORD
 .→-------------.
 ↓131 55 3569 10|
 '~--------------'

ORDERQ
      DISPLAY ORDERQ
 .→--------------------------.
 ↓ .→----------------------. |
 | | .→----------------. | |
 | | 0 |CUSTOMER NUMBER| | |
 | | '----------------' | |
 | '←---------------------' |
 | .→-------------------.    |
 | | .→-------------. |     |
 | | 0 |STOCK NUMBER| |     |
 | | '-------------' |     |
 | '←-----------------'      |
 | .→-------------------.    |
 | | .→-------------. |     |
 | | 0 | AMT ORDERED| |     |
 | | '-------------' |     |
 | '←-----------------'      |
 '←--------------------------'

ORDERS
       DISPLAY ORDERS
 .→---------------.
 ↓123  55 3569  5|
 |123  55 1135 12|
 |123  55 2583  4|
 |131  55 3569 10|
 |135 312 9998 12|
 |136   7 1135  2|
 '~---------------'

PROMPTS
      DISPLAY PROMPTS
 .→---------------------------.
 | .→------. .→-----. .→------. |
 | |STOCKQ| |CUSTQ| |ORDERQ| |
 | '-------' '-----' '-------' |
 '←---------------------------'

STOCKQ
      DISPLAY STOCKQ
 .→------------------------.
 ↓ .→--------------------.  |
 | | .→---------------. |  |
 | | 0 |STOCK NUMBER| |  |
 | | '-------------' |  |
 | '←-----------------'  |
 | .→--------------------.  |
 | | .→-------------. |  |
 | | | DESCRIPTION| |  |
 | | - '-------------' |  |
 | '←-----------------'  |
 | .→------------.          |
 | | .→------. |           |
 | | 0 | PRICE| |          |
 | | '------' |           |
 | '←------------'          |
 | .→-----------------.     |
 | | .→----------. |       |
 | | 0 | INVENTORY| |      |
 | | '-----------' |       |
 | '←-----------------'     |
 | .→--------------------.  |
 | | .→--------------. | |
 | | 0 | REORDER LEVEL| | |
 | | '--------------' | |
 | '←--------------------' |
 '←-------------------------'

STOCKS
      DISPLAY STOCKS
 .→-----------------------------------------.
 ↓     .→----------------.                   |
 | 1135 |FIRST GREAT ITEM|   9.95  118  55   |
 |      '----------------'                   |
 |      .→----------------.                  |
 | 9993 |HIGH FLYER WIDGET|  88.73 240  35   |
 |      '-----------------'                  |
 |      .→-----------------.                 |
 | 3569 |SECOND MONEYMAKER|  24.75 10   30   |
 |      '-----------------'                  |
 |      .→-------------------.               |
 | 5613 |MAIL ORDER SPECIAL| 14.99 225  95   |
 |      '-------------------'                |
 |      .→-------------.                     |
 | 2583 |A REAL WINNER|      49.99 89   10   |
 |      '-------------'                      |
 |      .→----------------.                  |
 | 9998 |NONESUCH FRAMMIS|   2.69  416  50   |
 |      '-----------------'                  |
 |      .→-------.                           |
 | 7777 | THINGY|            1.89  2    1    |
 |      '--------'                           |
 |      .→-----.                             |
 | 8888 |WHEEE|              4.33  3011 100  |
 |      '------'                             |
 |      .→--------.                          |
 | 5555 |WHIZBANG|           22.22 43   2    |
 |      '---------'                          |
 '←-----------------------------------------'

WARNING
      DISPLAY WARNING
 .→-----------------.
 |***DATA NOT SAVED|
 '-----------------'
```

```
DELSQL
[0]   V←NAME DELSQL N;E;S;T
[1]   ⍝ REQUIRES AP 127 AND THE SQL WORKSPACE
[2]   ⍝ DELETE RECORD N FROM THE NAMED SQL TABLE
[3]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[4]    T←TABLES⌷⍨⊂NAME
[5]    S←T⊃,SQLDELETES        ⍝ SELECT THE STATEMENT
[6]    E←↑V←SQL S N           ⍝ SELECT - E IS RETURN CODE
[7]    MESSAGE E              ⍝ PRINT MESSAGE IF ERROR
```

```
GETSQL
[0]   V←NAME GETSQL N;E;S;T
[1]   ⍝ REQUIRES AP 127 AND THE SQL WORKSPACE
[2]   ⍝ GET RECORD N FROM THE NAMED SQL TABLE
[3]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[4]    T←TABLES⌷⍨⊂NAME
[5]    S←T⊃,SQLSELECTS        ⍝ SELECT THE STATEMENT
[6]    E←↑V←SQL S N           ⍝ SELECT - E IS RETURN CODE
[7]    V←2⊃V                  ⍝ SECOND ITEM IS RESULT
[8]    MESSAGE E              ⍝ PRINT MESSAGE IF ERROR
```

```
PUTSQL
[0]   NAME PUTSQL V;E;S;T
[1]   ⍝ REQUIRES AP127 AND THE SQL WORKSPACE
[2]   ⍝ SQL UPDATE - V IS THE NEW VALUES
[3]   ⍝ NAME IS THE TABLE NAME
[4]    →('NOT A VALID TABLE NAME' ERR~(⊂NAME)∈TABLES)/0
[5]    T←TABLES⌷⍨⊂NAME
[6]    S←T⊃,SQLUPDATES         ⍝ SELECT THE STATEMENT
[7]    E←↑PREP 'UP' S          ⍝ PREPARE - E IS RETURN CODE
[8]    →((MESSAGE E)ERR↑E)/0   ⍝ ERROR
[9]    S←T⊃,SQLINSERTS         ⍝ INSERT STATEMENT
[10]   E←↑PREP 'IN' S          ⍝ PREPARE AN INSERT
[11]   →((MESSAGE E)ERR↑E)/0   ⍝ ERROR
[12]   SQLUPIN~V               ⍝ UPDATE OR INSERT
```

```
SQLINSERTS
       DISPLAY SQLINSERTS
 .→-------------------------------------------.
 ↓ .→--------------------------------------.  |
 | ↓ INSERT INTO STOCKS                    |  |
 | |  (ITEM,DESCRIPTION,PRICE,QUANTITY,REORDER)|  |
 | |  VALUES(:1,:2,:3,:4,:5)               |  |
 | '---------------------------------------'  |
 | .→-------------------------.               |
 | ↓ INSERT INTO CUSTOMERS   |               |
 | |  (CUST,NAME,ADDRESS,ZIP)|               |
 | |  VALUES(:1,:2,:3,:4)    |               |
 | '-------------------------'               |
 | .→-----------------------------.          |
 | ↓ INSERT INTO ORDERS           |          |
 | |  (INVOICE,CUST,ITEM,AMOUNT) |          |
 | |  VALUES(:1,:2,:3,:4)        |          |
 | '-----------------------------'          |
 '⊂-----------------------------------------'
```

```
SQLSELECTS
       DISPLAY SQLSELECTS
 .→--------------------------.
 ↓ .→--------------------.    |
 | ↓ SELECT * FROM STOCKS|    |
 | |  WHERE ITEM = :1    |    |
 | '---------------------'    |
 | .→-----------------------. |
 | ↓ SELECT * FROM CUSTOMERS| |
 | |  WHERE CUST = :1       | |
 | '------------------------' |
 | .→--------------------.    |
 | ↓ SELECT * FROM ORDERS|    |
 | |  WHERE INVOICE =:1  |    |
 | '---------------------'    |
 '⊂---------------------------'
```

```
SQLCREATES
       DISPLAY SQLCREATES
 .→-------------------------------.
 ↓ .→---------------------------.  |
 | ↓CREATE TABLE STOCKS        |  |
 | | (ITEM SMALLINT,           |  |
 | |  DESCRIPTION VARCHAR(20),|  |
 | |  PRICE DECIMAL(6,2),      |  |
 | |  QUANTITY SMALLINT,       |  |
 | |  REORDER SMALLINT)        |  |
 | '---------------------------'  |
 | .→-----------------------------.|
 | ↓ CREATE TABLE CUSTOMERS      | |
 | |  (CUST SMALLINT,            | |
 | |   NAME VARCHAR(30),         | |
 | |   ADDRESS VARCHAR(30),      | |
 | |   ZIP VARCHAR(30))          | |
 | '-----------------------------' |
 | .→-----------------------.     |
 | ↓CREATE TABLE ORDERS     |     |
 | | (INVOICE SMALLINT,     |     |
 | |  CUST SMALLINT,        |     |
 | |  ITEM SMALLINT,        |     |
 | |  QUANTITY SMALLINT)    |     |
 | '------------------------'     |
 '⊂--------------------------------'
```

```
SQLDELETES
       DISPLAY SQLDELETES
 .→----------------------------.
 ↓ .→--------------------.     |
 | ↓ DELETE FROM STOCKS  |     |
 | |  WHERE ITEM = :1    |     |
 | '---------------------'     |
 | .→------------------------. |
 | ↓ DELETE FROM CUSTOMERS  | |
 | |  WHERE CUST = :1       | |
 | '------------------------' |
 | .→-------------------.     |
 | ↓ DELETE FROM ORDERS|      |
 | |  WHERE INVOICE =:1|      |
 | '-------------------'      |
 '⊂---------------------------'
```

```
SQLUPDATES
       DISPLAY SQLUPDATES
 .→-----------------------------------------------------.
 ↓ .→---------------------------------------------------. |
 | ↓  UPDATE STOCKS                                    | |
 | |   SET DESCRIPTION=:2,PRICE=:3,QUANTITY=:4,REORDER=:5 | |
 | |   WHERE ITEM=:1                                   | |
 | '---------------------------------------------------' |
 | .→-------------------------------.                    |
 | ↓ UPDATE CUSTOMERS               |                    |
 | |  SET NAME=:2,ADDRESS=:3,ZIP=:4|                    |
 | |  WHERE CUST=:1                |                    |
 | '-------------------------------'                    |
 | .→-------------------------------.                   |
 | ↓ UPDATE ORDERS                  |                   |
 | |  SET CUST=:2,ITEM=:3,AMOUNT=:4|                    |
 | |  WHERE INVOICE=:1             |                   |
 | '-------------------------------'                    |
 '⊂-----------------------------------------------------'
```

```
SQLUPIN
[0]   SQLUPIN V;E
[1]   ⍝ UPDATE USING CURSOR 'UP', VALUES V
[2]   ⍝ IF UPDATE FAILS, INSERT USING 'IN'
[3]    E←↑CALL 'UP' V              ⍝ UPDATE
[4]    →((MESSAGE E)ERR↑E)/0       ⍝ ERROR
[5]    →(~2⊃E)/0   ⍝ EXIT UNLESS NOT FOUND
[6]    E←↑CALL 'IN' V              ⍝ INSERT
[7]    →0⍴⍥MESSAGE E
```

History Sheet

Date:	03/02/92
Title:	APL2 Programming: Guide
Order No./TNL#/Activity:	SH20-9217-00
File Prefix:	G20P1
EA/Writer/Editor:	Dana Marsh/Janet Walters/Laura Nystrom
Development Book Owner	David Liebtag
Graphics Consultant	Kathy Holland
ISIL Version:	BookMaster
Output Device:	3820 4250
Support:	1.5 1.25

Common art?	Yes No
Board Art?	Yes No
IPG Art?	Yes No
Common files?	Yes No
Common File Name(s):	None
Special Style File Name:	IBMXAGD

Comments:

Changes to this book were minor - there were no formal information inspection reviews.

There are four special fold out pages at the back that are reference and do not get listed in the table of contents. A special file was created as place holders for the printer. There was a miscommunication with the printer and the pages are listed in the table of contents. This will need to be corrected on the next revision of this book.

Writer's Signature: Janet L. Walters (electronically)

cc: Lead EA - Dana Marsh

We'd Like to Hear from You

APL2 Programming:
Guide
Version 2 Release 1

Publication No. SH21-1072-00

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: (408) 463-4488.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB6JN8
  - Internet: apl2@vnet.ibm.com

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

APL2 Programming:
Guide
Version 2 Release 1

Publication No. SH21-1072-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

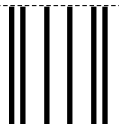
Company or Organization

Phone No.

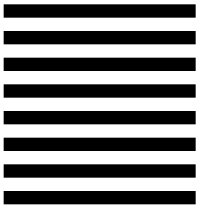
Readers' Comments
SH21-1072-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department M46/D12
PO Box 49023
San Jose, CA 95161-9023

Fold and Tape **Please do not staple** Fold and Tape

SH21-1072-00

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5688-228

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SH21-1072-00