

Programming in APL:

A Beginner's Guide

for

Dyalog APL

Dr. Raymond Polivka

This DRAFT is being distributed
for the purpose of getting feedback
prior to publication.

Please send your comments to
Polivka@APLclass.com

Use a subject line: Book
Thank you!

© 2022, Polivka Associates. All rights reserved.

No part of this material may be reproduced, transmitted, transcribed, stored on a retrieval system, or translated into any language or digital format, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Polivka Associates, 60 Timberline Drive, Poughkeepsie, New York 12603–5546 USA

Contents

Preface	6
Introduction.	7
APL Programming	8
The Basics	9
Vectors	12
Sum and Product $+/R$ \times/R	14
Error Messages	15
Assignment	16
Legal Names	17
A Word of Caution	19
A Bit of Grammar	21
A Mechanical Procedure for Expression Evaluation	23
Character Data	26
Primitive Functions	27
Shape of ρR	28
Catenation L,R	29
Relational Functions $L < \leq = \geq > \neq R$	30
Summary 1: The Basics	32
Programming	33
Editing	35
Comments	36
Direct Function Guards	38
Summary 2: Programs	40
Primitive Scalar Functions.	41
Tack $L\uparrow R$ $L\rightarrow R$ and Same $\uparrow R$ $\rightarrow R$	42
Maximum and Minimum $L\uparrow R$ $L\downarrow R$	43
Floor and Ceiling $\lfloor R$ $\lceil R$	43
Absolute $ R$	44
Residue $L R$	45
Boolean Functions $L \wedge \vee \tilde{\wedge} \tilde{\vee} R$ and $\sim R$	46
Greatest Common Divisor and Least Common Multiple	49
Power $*R$ $L * R$	51
Compound Interest.	52
Present Value	53
System Facilities	54
Some Non Scalar Functions	55
Index Generator $\imath R$	55
Roll and Deal $?R$ $S?R$	57
A Review	59
Summary 3: Scalar Functions	61
Matrices.	62
Reshape $\vee \rho R$	65
Tally $\neq R$	66
Array Transposition	68
Reverse First $\ominus R$	69
Transpose ΦR	70
Summary 4: Matrices	74

Array Selection	75
Selective Assignment	79
Take $L \uparrow R$ and Drop $L \downarrow R$	81
Grade Up and Grade Down ΔR $L \Delta R$ ΨR $L \Psi R$	84
Summary 5: Array Selection	90
Array Inquiry	91
Match $L \equiv R$ and Not Match $L \neq R$	92
Index of $L \wr R$	93
Where $\underline{L} R$	97
Unique $\cup R$	98
Union $L \cup R$	99
Unique Mask $\neq R$	100
Find $L \subseteq R$	101
Membership $L \in R$	102
Interval Index $L \underline{L} R$	103
Summary 6: Array Inquiry.	106
Arrays Revisited	107
Depth $\equiv R$	109
Display Facilities	110
Ravel $, R$	113
Enlist ϵR	114
Table $\bar{;} R$	115
Catenate L, R	117
Enclose $\subset R$	118
Partition $L \subseteq R$	120
Mix $\uparrow R$ and Split $\downarrow R$	123
Pick $V \supset R$	127
Index $L \sqcap R$	130
Selective Assignment Revisited Again	132
Summary 7: Arrays Revisited	133
Mathematical Facilities	134
Logarithms $\otimes R$ $L \otimes R$	135
Trigonometric Functions	137
Decode $L \perp R$	139
Encode $L \top R$	144
Elementary Scalar Functions: Factorial $!R$, Binomial, $L!R$, Residue $L \mid R$	148
Format: ΦR and $L \Phi R$	151
Execute ΔR	154
Matrix Inverse $\boxminus R$	155
Matrix Divide $L \boxdiv R$	156
Summary 8: Mathematical Facilities	158
Operators	159
Axis Specification $f[I]R$ $L f[I]R$	160
Reduction $f/[I] R$	168
N-wise Reduction: $N f/[I] R$	173
Replication $V/[I]R$	175
Scan $f \backslash [I] R$	177
Expansion $N \backslash [I]R$	179

Each $f^{\cdot\cdot}R - Lf^{\cdot\cdot}R$	182
Outer Product $L \circ.f R$	186
Inner Product $L f.g R$	190
Summary 9: Operators	195
A Return to Programming	197
Control Constructs	200
Session Input/Output	208
Procedural Programming	212
Conclusion	213
What is over the Horizon?	213
Shirley	215
Complex Number Translation	216
Grade Point Average	217
Check Balancing	218
A Series Expansion	219
Adjacent Removal	220
Falling Factorials	221
Inner Vowel Removal	224
An Unusual Paragraph.	226
Finding Pythagorean Triples	228
Data Smoothing with an Adjustable Weight Window	230
The FBI Checksum	232
Generating Prime Numbers	234
Cardinal to Ordinal.	235
The Totient/Phi function	236
Palindromes.	237
Pillow Problem #21 (07/04/1889)	238
Fibonacci Numbers	239
The Golden Ratio	241
The United States Reapportionment Algorithm	242
Multiple Replacement	245
Partitioned Matrices	247
Postscript: Some Debugging Facilities	249
Appendix A: The Dyalog Windows Working Environment	251
A User Convenience	254
Shifting Between Keyboards	254
The Workspace	255
Accessing a Workspace	255
Setting Default Settings	256
System Commands	257
Appendix B: A Functional Summary	258
Appendix C: The Computer System	263
Vita	267

Preface

“Programming in APL, A Beginner’s Guide” offers a first time entry into the programming discipline using Dyalog APL. Its intended audience is the rank beginner. Consequently, in introducing the various features of APL some of the more advanced features and details are omitted. As students advance beyond this material and become more proficient, they should view the documents on the Dyalog.com website. In some cases, footnotes will guide them directly to pages in the *Dyalog Language Reference Guide*.

In addition to addressing the beginner, this presentation could also interest the much larger audience of people, the domain experts, who do not wish necessarily to be computer scientists but rather wish to solve their problems as easily as possible. And too, this material should be useful for teachers who were to introduce APL.

The best way to study programming and APL with this material is to have APL active on their computer where they may pause to try something out and do the exercises. The emphasis here is to learn by doing. The exercises are meant to reinforce understanding. Do them. These exercises are intended to drill on the current and previously introduced material.

The following material is intended to introduce APL as a programming language tool. Other aspects of application development are not addressed. Such topics are the role of Computer Science.

The Dyalog working environment found in *Appendix A* is intended to acquaint you with the Session Manager of Dyalog APL for Windows. This is your working environment. *If you are not familiar with the Session Manager, begin your introduction by reading it.*

Many introductions to a programming language begin with an introduction, perhaps historically, of the computer system environment into which a programming language fits. This presentation does not. We begin immediately to introduce the APL programming language. An appendix *C* exists for those readers that may wish a brief summary of the organization of a computer system and how APL fits into the software of a computer system.

Introduction

Most of our lives involve solving problems, some very simple to some very difficult, even as yet unsolved. For example, “What shall I wear today?”, “What job shall I take?”, or “Where are the car keys?” We hardly consider such questions as problems, but they are.

Solving a problem must involve knowledge of facts and information that are related to the problem. With such knowledge one can, produce a series of steps to reach a solution. In moving toward a solution, we plan and execute a series of activities to reach this solution. This series of steps when explicitly described is often called an *Algorithm*. When working on a computer, we implement algorithms in a language created to be understood by both a person and the computer. The creations that we write are called *programs* and the languages we use are called *programming languages*. The programs that we create offer a degree of permanency when we save them in the computer. Such programs can then be reused repeatedly without re-creating their contents.

Why do we need a programming language, of which there are hundreds? We wish to control the computer using human terminology. Computers only understand “0” and “1”s. A programming language forms a bridge between the computer using “0” and “1”s and the human being using the computer.

There are many programming languages just as there are many spoken languages. Let’s look at one programming language: APL.

APL Programming

“By relieving the brain of all unnecessary work,
a good notation
sets it free
to concentrate on more advanced problems,
and, in effect,
increases the mental power of the race.”

—Alfred N. Whitehead, British mathematician (1861–1947)

Throughout this text we will be using *Dyalog APL*¹. *APL* appropriately stands for “*A Programming Language*.” It is the earliest commercially-successful interpretive language, unique in that it was designed by a mathematician, Kenneth E. Iverson, and not by computer scientists. *APL* is a symbolic array-oriented programming language distinguished from other programming languages by a large set of powerful mathematical like symbols. In its continuous evolution, today *APL* exists as a multi-platform general-purpose array programming language that incorporates procedural, functional, and object-oriented capability.

First, let us focus on the concept of a *language*. A spoken language starts with a set of symbols, in English the alphabet, and develops a vocabulary and a grammar through which meaningful statements are created to communicate with other people. So, it is with the *APL* programming language. Here the communication is between a human being and a computer. *APL*, with the spoken language symbols of the user, contains other symbols. These symbols are listed in the language bar on the session manager. These *APL* symbols are more powerful than the alphabetic symbols of a spoken language since these symbols act as words or phrases to be meaningful to both the user and the computer.

The vocabulary of a spoken language, its words, are defined in a dictionary. A programming language also has a dictionary which defines the primitive facilities of the language that allows you to manipulate data. In fact, as one reads through a programming language manual one is essentially reading through the “dictionary” of the language. (The story line of any dictionary is not too exciting but nevertheless important.)

The role of a programming language is to manipulate data. To do this, the programmer will take the primitive facilities of the language and fashion them into a series of statements called *expressions*. Building such sets of statements is called *Coding*. Collected together this series of expressions, when named, becomes a *program*. In *APL*, its symbols are called *primitive functions*. A collection of expressions, the *program*, is called a *defined function*.

¹ For more detail, go to the *Dyalog APL Language Reference Guide*.

The Basics

The basic data of APL are numbers² and characters.

Data is often assigned a name. This named data can be considered the *nouns* of APL. The *verbs* of APL are the *functions* that manipulate the data to produce a result. In addition, in *Dyalog APL* there exist *operators* which behave like *adverbs* modifying the behavior of functions. Functions take data and produce data results but operators take functions and produce functions. Let us begin.

Whenever you wish to have the computer recognize what you entered, you tell it by depressing the *Enter* key.

Type **3.14159** and press the *Enter* key.

After you type **3.14159**, upon pressing the *Enter* key, APL gives back to you just what you had entered. Notice that when APL is waiting for further input activity, the blinking cursor is indented 6 spaces. Whatever APL returns to you begins at the left edge of the screen. This way it is easy to distinguish between your entry and APL's response. There is a general term for this action it is called *REPL* which stands for *Read, Execute, Print, Loop*. This is how the APL interpreter behaves. A great deal of the time it may be waiting to read what you will enter.

Exercise 1

Type:

- a) 123
- b) 3.14159

APL is a very symbolic language, as is mathematics. Consider this chronology of when the introduction these mathematic symbols occurred:

1-9	874
+	-	1489
=	1557
()	1610
×	1631
÷	1659
.	(decimal point) .	1671
!	1808
 a 	1841
€	1889
v	1909
^	1933

APL is merely following this pattern of introducing new useful symbols. Dyalog APL has made the use of them more convenient by introducing the Language Bar.

² Including complex numbers

The *Language Bar* in the session manager contains all the symbols with which you can manipulate data in APL. Here is the language bar broken into parts.



It may appear as one line or several in your workspace.

At this point, you can think of APL on your computer as a very powerful pocket calculator. If APL can act as a calculator, you should, at least, be able to add, subtract, multiply, and divide. Can you find the symbols for these functions? The symbols $+$, $-$, \times , \div are very familiar, aren't they?

(Note: In most other programming languages there are different symbols for multiplication and division.)

You may enter \times and \div symbols by clicking on them in the language bar in which case the symbol appears where the cursor is in your workspace. Alternatively, by holding down together the Control and minus key, (*Ctrl*+ $-$),³ for \times and the Control and plus keys, (*Ctrl*+ $+$), for \div .

Let try them out.

Exercise 2

- a) $32+47$
- b) $123-5.25$
- c) 7.5×21
- d) $512 \div 8$

The symbols represent functions or operators. For example, the action of adding two numbers is referred to as the function of addition. The nature of operators will be covered later.

There is a shortcut to entering all the APL function symbols using the language bar. Suppose you wish to multiply 32 by 84 . After typing 32 , left click on the times symbol on the language bar. This will cause the \times symbol to appear right after 32 . Then enter 84 and press *Enter*.

Also, when you place your cursor on a symbol on the language bar, you get a drop-down window that defines the keyboard combination you need to use to enter the symbol. In addition, this window also states the names associated with this symbol, its definition, and examples of its behavior.

Exercise 3

Repeat the previous exercise twice, once using the proper control key combinations for the symbols $+$ $-$ \times \div and then by left clicking the symbol on the language bar.

³ Statements like (*Ctrl*+ $-$) imply that the Control key and the symbol following the plus sign are to be depressed together. Throughout this text, the various key combinations will be stated in parenthesis.

Primitive APL function symbols may have two different meanings depending on whether the function has one argument or two arguments. The data presented to a function are called *arguments*. A function with one right argument is called a *monadic* function and a function with two arguments is called a *dyadic* function. The argument of a monadic function appears to the right of the function symbol. The function symbol for a dyadic function appears between the two arguments.

The important point is:

Any primitive function will always have a right argument.

For example, the symbol \div is called *divide* when used in a dyadic fashion as in $10 \div 2$. When the symbol is used in a monadic fashion, it is called *reciprocal* as in $\div 2$.

Thus,

	$10 \div 2$
5	
	$\div 2$
0.5	

For real numbers, the symbols $+$, $-$, and \div are as learned in school. But *monadic* \times is different. It is called *Signum*: “*Sign of the number.*” Signum produces a positive 1 for positive numbers, -1 for negative numbers, and 0 for 0.

Note that there is a special symbol $\bar{-}$ used in indicating a negative number. It is called *high minus* (Ctrl+2) The need for it will be explained shortly (page 12) when vectors are introduced.

Exercise 4

- a) $1 \div 4$
- b) $\div 4$
- c) $123 - 25$
- d) $1.5 + 16$
- e) $12.7 - 3.5$
- f) $10 \div 2$
- g) $10 \div \div 2$
- h) $10 \div 0$
- i) $+5$
- j) -5
- k) $\div 5$
- l) $\times 5$
- m) $\times -5$
- n) $\times 5 - 5$
- o) If a person slept seven hours a night, write an expression to determine the number of hours that person slept in a year.
- p) Write an expression to determine the number of minutes in a year.
- q) The Library of Congress has about 850 miles of books. If the average thickness of a book there is an inch thick, how many books would the Library contain?

Vectors

APL is array oriented. An *array* is an ordered collection of data whose items can be selected by their location in the array. Various names are associated with an array. For example, a *vector* is also referred to as a list and a *matrix* as a table. In the first part of this text we will only be using vectors.

What is a *vector*? For our purposes, it is just an ordered list of numbers or characters. In entering a set of numbers, you enter them horizontally one after another separating each from the other by at least one space. A single number or character is called a *simple scalar*. A vector containing just simple scalars is called a *simple vector*.

An *expression* is any combination of functions and data that is meant to be executed by APL to produce a desired result.

If you wish to work with a vector of numbers, a new situation arises. You need to distinguish between items in a vector. This is done by putting at least one space between the numbers.

Exercise 5

Write the APL expression for creating a vector of numbers from the following numbers:

Twelve, five, six hundred and four, and seventy-three.

Now suppose you wish to create a two-item vector consisting of 6, and negative 4 and the expression 6 minus 4. The minus sign cannot both represent the function of subtraction and the negativeness of a number. A special symbol is needed to indicate that a number is negative. This new symbol, $\bar{}$, called *high minus*, is created via the key combination (*Ctrl* + 2). It is *not* a function symbol.

Exercise 6

- a) Create a two item vector consisting of six and a negative four.
- b) Create the expression to subtract 4 from 6.
- c) Create the expression 10 minus negative 5.

Scroll back to that set of four digit numbers; 12 5 604 73. Type 10+ in front of that vector. What was the result? 10 was added to each of the numbers. Retrieve those for numbers again and type +10 on the right end of the vector of numbers. Did you get the same result? You should.

There is a very important feature illustrated here. It is that of *scalar extension*. Many of the functions of APL will take a scalar argument or a one item vector and apply it to each of the data items in a vector argument. Such functions are called *scalar functions*.

Pictorially, where S represents a scalar, V represents a vector and fn is an APL function, you may think of it as:

S fn S	5 + 12
S fn V	5 + 3 7 2
V fn S	3 7 2 + 5
V fn V	3 7 2 + 8 4 9

This is a very important feature of APL.

Now you may wonder what if I add two vectors together. Let's see.

Exercise 7

- a) $2\ 10\ 5 + 3\ 4\ 7$
- b) $2\ 10 + 3\ 4\ 7$

In the first exercise the addition was carried out with the 2 added to the 3, 10 added to the 4, and 5 added to the 7. In the vector function vector situation, the corresponding elements are added together, first with the first, second with the second and so forth. However, the second exercise gave you an error message. APL did not know what number to add to what number. So it produced the **Length Error** message.

Thus, for scalar functions, vectors may be combined with vectors provided the vectors are the same length or one of the arguments is a scalar.

All of the arithmetic and logic functions that we will encounter later are *scalar functions*.

Exercise 8

Evaluate the following expressions, but first evaluate them with paper and pencil and then go to the computer to check your answer.

- a) $.5 + 10\ 3.5$
- b) $6\ 5\ 4 + 3\ 2\ 1$
- c) $5 \times 2\ 4\ 6$
- d) $9\ 18\ 36 \div 9$
- e) $^{-}1\ 2\ 3^{-}\ 5$
- f) $^{-}5 - ^{-}3\ 2\ 1$
- g) $2 + 3 - 4\ 5$
- h) $h) 2 + 3\ ^{-}4\ 5$
- i) $2 + 3 - ^{-}4\ 5$
- j) $2 + 2\ 2 + 2$
- k) $+2\ ^{-}2\ 0$
- l) $-2\ ^{-}3\ 0$
- m) $\div 3^{-}\ 2\ 0$
- n) $\times 2\ ^{-}3\ 0$
- o) Your hair grows about a half a millimeter in a day. If it does, how many inches does it grow in a day or a week? (1 meter = 3.28084 feet)

Sum and Product $+/R$ \times/R

With a set of numbers, a common desire is to add them up or perhaps less frequently multiply them. In mathematics these actions are so common that special symbols are created for these actions. They are:

$$\sum \text{ and } \prod$$

APL has comparable symbols; they are

$$+/\text{ and } \times/$$

Thus for a list of numbers such as

$$12 \ 10 \ ^{-}5 \ 2.5$$

to find the sum

$$\begin{array}{r} +/12 \ 10 \ ^{-}5 \ 2.5 \\ 18.5 \end{array}$$

or to find the product

$$\begin{array}{r} \times/12 \ 10 \ ^{-}5 \ 2.5 \\ ^{-}1800 \end{array}$$

Exercise 9

Determine the values of the following expressions.

- $+/ \ 23 \ 45 \ 67$
- $+/23 \ ^{-}45 \ 67$
- $+/^{-}23 \ 45 \ ^{-}67$
- $\times/10 \ 5 \ 12$
- $\times/10 \ ^{-}5 \ 12$
- Determine how many seconds there in one day and in one week.
- If your heart beats at about 69 beats per minute, on your next birthday how many times has your heart beat since you were born?
- If someone told you that her heart rate was 69 beats per minute and that her heart has beaten for 906,660,000 to date, how old is she?

Note: As we encounter them, other dyadic functions may be used to the left of the $/$. Functions such as $f\mathbf{n}/$ are known as Reduction functions. In their execution, what happens is that the function is distributed between the data arguments. Thus the expression $+/12 \ 10 \ ^{-}5 \ 2.5$ is evaluated as $12+10+^{-}5+2.5$.

Error Messages

Probably by now you have encountered an error message. Actually one occurred in a previous exercise. Here it is again. Let us examine it.

```

2 10 + 3 4 7
LENGTH ERROR: Mismatched left and right argument shapes
2 10+3 4 7
^

```

The first line is a capitalized generalized the error message followed by more detail if possible. In this case, the 2 arguments were not the same length. The second line contains the whole statement in which the error occurred. The third consists of a single █. It indicates how far along moving from right to left the APL interpreter proceeded before it found an error. Knowing this is useful since it helps you look for the error. Looking to the left of the carrot is unnecessary.

As you enter an expression, color tries to prevent you from making some common errors. Whenever you enter a (, [, or {, the symbol appears in red. Only when the matching),], or } is entered do the symbols turn to black. When you type a quote, ', both the quote and what follows appears in red until the matching ' is typed. Generally should there be any color in the expression you typed, there is an entry error within it.

Some of the most common error messages are LENGTH, VALUE, DOMAIN, and SYNTAX.

For example,

```

AAAA
VALUE ERROR: Undefined name: AAAA
AAAA
^
25÷0
DOMAIN ERROR Divide by zero
25÷0
^
235÷
SYNTAX ERROR Missing right argument
235÷
^
(1 2 3+4 5)÷0
LENGTH ERROR: Mismatched left and right argument shapes
(1 2 3+4 5)÷0
^

```

Note that the last expression above had two errors. If there are several errors in a statement, only the first one that the APL interpreter encounters will be indicated.

Exercise 10

Determine each error message and its cause:

- a) 10 × yyy
- b) 10 +
- c) 1 2 3 + 4 5
- d) 12 ÷ 0

Assignment

Suppose those four numbers 12 5 604 73 are important to you. Later you may wish to subtract 5 from them and again much later you may wish to divide them by 2. Retrieving them from the session manager screen may not be possible. Your only option is to enter the numbers again. That is not very pleasant. A better way would be to assign the numbers a name and thereafter referred to those numbers by its assigned name.

Thus enter:

```
Nom←12 5 604 73
```

The symbol ← is created by the combination (*Ctrl+]*) or clicking on the symbol on the language bar. Now you may type **Nom** to retrieve the data.

```
Nom
12 5 604 73
```

The new symbol, ←, called *Assignment*, takes whatever was evaluated to its right and associates it with the name to its left. This name representing data is called a *Variable*, short for Variable name. It may be used to represent that vector of numbers in any expression. Note also whenever you type an expression without the assignment symbol the APL interpreter will return the result of the expression to you on the screen.

With the introduction of assignment, another type of error can occur. Consider:

```
AAAA
VALUE ERROR: Undefined name: AAAA
AAAA
^
```

VALUE ERROR occurred because the variable name was used before it was assigned data.

Exercise 11

Enter:

- a) `Nom← 12 5 73`
`Nom`
- b) `Nom+5`
- c) `100-Nom`
- d) `Nom ÷ 10`
- e) `.5×Nom`
`Nom←10`
`.5×Nom`

Note in exercise e) the value associated with *Nom* has been changed. *The only way you can change what the variable represents is to reassign new data to it.*

Legal Names

What can become a legal name in APL?

A name may be made up of alphabetic characters, upper- or lower-case, the integers 0 through 9, Δ , $\underline{\Delta}$, but it may not begin with an integer. While $\underline{\Delta}$ and underlined characters are acceptable in naming characters, they are legacy characters, maintained for backward compatibility, and should not be used.

The name may be of any length (of course using it, remember you may have to enter it in again.) Since names may be of any length, you are able create and use meaningful names wherever names are needed. “Grades” and “Names” are much more meaningful than “X” and “Y”.

There are *no reserved words*⁴ in APL. You may create and use any name that APL find acceptable.

It is also possible to assign data to several variables in one expression. This is done by placing the variable names separated by a space to the left of assignment and the matching data to the right of assignment.

```
a b c ← 10 20 30
a b
10 20
c+b+a
60
```

As you work within your workspace you will be creating many named variables. In this case the system command **)Vars** will produce listing a listing of all of them. Those you wish to remove can be done by issuing the system command **)Erase**. **)Erase** will remove any named object from within your workspace. Later as you create named functions (programs) you have the system command **)Fns** to get a listing of them.

⁴ A reserved or “Key” word is a word that a programming language claims for itself for special language purposes. The user cannot use such words to represent data in a program.

Exercise 12

- a) Write two ways to assign to the variable **X** the value **25** and the variable **Y** the value **34**.
- b) Determine which of these letter number sequences would be acceptable as a variable name.
 - c-1) **9Abb** _____
 - c-2) **2A ÷1** _____
 - c-3) **_ab2** _____
 - c-4) **A+5** _____
 - c-5) **Price** _____
 - c-6) **Bad no** _____
- d) What happens when you type the following?
 - d-1) **a b c←5**
 (then examine **a b c**)
 - d-2) **a b c←5 10**
 (then examine **a b c**)
 - d-3) **a b c←1 5 10**
 (then examine **a b c**)
 - d-4) **a b←2 4 6**
 (then examine **a b c**)

If there are separate vectors of data to be assigned to separate names, each vector is separated from the other vector by parentheses. For example,

```

a b←(34 56) (14 -23 89)
a
34 56
b
14 -23 89

```

Exercise 13

- a) Assign your phone number to the variable **Phone_no**.
- b) Assign in one expression the two grades eighty-eight and ninety-two to the variables **English** and **Math**.
- c) Separate your phone number into two variables, **area** and **local**.
- d) In one expression, assign the two grades **78** and **84** to the variable **French** and the three grades **80**, **85**, and **92** to the variable **Physics**.

A Word of Caution

All of the work you do is done in a *workspace*. What you see on your computer screen in the session manager is the active part of your workspace in which you create variables and functions. To leave APL you issue the system command

```
)Off
```

however, unless you had previously issued the system command

```
)Save
```

you will have lost the variables and functions that you just have created.

Before you save for the first time, give your currently active workspace a name. Do this via

```
)Ws id your chosen name
```

Note it is probably a good idea to save several times during your active session. Certainly one general rule to follow would be “*Save before signing off*,” as well as “*Save often*”.

Recall when you return later, you can retrieve your saved workspace with your work in it via the system command

```
)Load your chosen workspace name.
```

Refer to appendix A for more details on system commands.

When you issue the `)save` command, you are saving named objects; variables, defined functions, and operators. The unnamed expression that you previously created is not saved. However, Dyalog APL maintains a session log. This contains all of your previous keystrokes. Thus after you sign on you may scroll up past your sign-on indication and examine your yesterday’s keystrokes. By scrolling back into yesterday’s keystrokes, you may, by clicking on a prior keyed expression, execute it again and bring it into your current active workspace.

For example, execute a `)Save`, then

```
v←3.24259
foo←'A test'
```

Sign off via `)off`

Sign on again:

Type `v` and `foo`

Scroll up beyond the sign-on material until you see where the variables `v` and `foo` were assigned. Click on them. This brings them into your active workspace.

Type `v` and `foo`

It’s always a good idea to save your active workspace before you sign off. It is also a good idea to load your active workspace that you have been working in when you sign on.

Suppose in your excitement when you signed on you forgot to load your working workspace. By the time you realized that you had not loaded your working workspace you had created some valuable work but it’s not in that prior working workspace. Do not despair. Here’s a procedure that will enable you to save your current active work into your prior working workspace that you forgot to save when you signed off.

The procedure:

- a) Type `)LOAD` your active workspace *but...*
- b) *Do not* depress the *Enter* key
- c) Rather change the `)Load` to `)Copy` or `)Pcopy`
where `)Copy` and `)Pcopy` will both bring the named workspace contents into your currently active workspace. The `)Pcopy` will not bring in any named item that has the same name of a named item in the current active workspace. And then *Enter*.
- d) Issue a `)WSID` with the name of your active working workspace
- e) Issue a `)SAVE` having merged the previous work into your active workspace.

Let's do this exercise:

Right after signing on, do these steps:

- 1) `)wsid test`
- 2) `Phone1←914 555 4567`
- 3) `)save`
- 4) Sign off.
- 5) Sign on again.
- 6) `Phone2←914 555 9876`
- 7) Type `Phone1`, getting a `VALUE ERROR`
- 8) `)Pcopy test`
- 9) Type `Phone1 Phone2`
- 10) `)wsid test`
- 11) `)save`

A Bit of Grammar

So far you have been entering only a single function along with data. Can you create expressions involving more than one function in an expression, you ask? You certainly can. Try entering this,

```
10-2+5
3
```

Three, did you expect that? Were you expecting 13?

The reason APL does not agree with what you thought the answer should be is that APL does *not* follow the PEMDAS order of evaluation (*Please Excuse My Dear Aunt Sally*) .

This letter sequence defines an order in which the arithmetic functions may be evaluated reading from left to right. This sequence is, translating the letters,

```
Parenthesis
Exponentiation
Multiplication
Division
Addition
Subtraction
```

That order of evaluation may be fine for a small number of primitive functions, but if you have *many* primitive functions, as APL does, defining a hierarchical ordering would be difficult to define, let alone to remember.

All of this is prologue to show you how APL evaluates APL expressions. Recall that different spoken languages have different rules of grammar. APL as a language also has its own rule of grammar. The rule of grammar for evaluating an APL expression is *the right-most function, whose data arguments are available to the function, is evaluated first*.

This is often stated as “*The Right to Left Evaluation Rule*”.

Now this may seem upsetting to you, since it seems that even a simple expression like 10-2+5 does not get the “right” answer you’ve received in school math. “Can I even get the answer 13 from it?” you ask. Of course you can, by introducing parentheses:

```
(10-2)+5
13
```

Thus, especially for beginners, there is a corollary to this right to left APL evaluation rule. The corollary is

Whenever in doubt use parentheses.

Parentheses overrule the normal order of evaluation.

Within the parenthetical expression the normal right to left evaluation also occurs.

Until you become more familiar with APL, use parentheses when you are not sure. Of course, there are places where you need to use parentheses to delineate a function's arguments within an expression.

APL's rule of grammar allows you to read expressions from left to right and more importantly it allows you to create expressions with a mixture of primitive and user created functions.

It is time for some exercises.

Exercise 14

In the following expressions indicate the order of execution of the functions within it by writing under each function symbol a sequence integer beginning with one for the first function that is evaluated. Then on paper evaluate the expression. Finally entered it into APL and check your answer.

- a) $55-15\times 10-4+1$
- b) $(55-15)\times 10-4+1$
- c) $((55-15)\times 10)-4+1$
- d) $(55-15\times 10)-4+1$
- e) $(55-15)\times (10-4)+1$
- f) $55-(15\times 10)-4+1$
- g) $55-15\times (10-4)+1$
- h) $(55-15\times 10-4)+1$
- i) $55-(15\times (10-4))+1$
- j) $(55-15)\times (1+(4-10$
- k) $((55-15)\times (+1((4-10$

A Mechanical Procedure for Expression Evaluation

To help you get comfortable reading APL expressions here is a rote sequence of steps that you can follow that will lead you to the same answer that APL provides.

1. Position yourself at the extreme right hand end of the expression.
2. Moving from right to left, look for a function.
3. Determine if the function is monadic or dyadic.
4. If the function is monadic, that is, there is no data to its left or just another function, execute the monadic function and return to statement 2.
5. If the function is dyadic, determine the value of the left argument and execute the function. Remember the left argument is *all* the data up to the next function or closing right parenthesis.
6. Return to the statement 2) above and continue evaluating the expression moving from right to left.

Recall that if there is no final assignment symbol, the value of the expression is displayed.

Since the arguments of a function may be vectors, it is important to remember that the left argument of a dyadic function is *all* the data up to the next function symbol. If the dyadic function encounters a right parentheses before the left argument then the execution of that function must wait for the evaluation of the parenthetical expression.

Exercise 15

Evaluate the following expressions.

- a) $+ / 6 \ 5 \ 4 \ + \ 3 \ 2 \ 1$
- b) $(+ / 6 \ 5 \ 4) \ + \ 3 \ 2 \ 1$
- c) $\times / 6 \ 5 \ 4 \ + \ 3 \ 2 \ 1$
- d) $+ / 6 \ 5 \ 4 \ ++ / 3 \ 2 \ 1$
- e) $(+ / 6 \ 5 \ 4) \ ++ / 3 \ 2 \ 1$
- f) $((+ / 6 \ 5) \ 4) \ ++ / 3 \ 2 \ 1$
- g) $((+ / 6 \ 5) + \ 4) \ ++ / 3 \ 2 \ 1$
- h) $(+ / 6 \ 5 \ ^{-}4) \ ++ / 3 \ ^{-}2 \ 1$
- i) $(+ / 6 \ 5) \ 4) \ ++ / 3 \ 2 \ 1$

As was stated earlier, *the function **fn** in **fn/** is distributed between the data arguments and the resulting expression is evaluated in a right to left fashion.* Recall that

$+ / 12 \ 10 \ ^{-}6 \ 2.5$ evaluates as $12 + 10 + ^{-}6 + 2.5$

Try it with non-commutative functions.

Exercise 16

Evaluate the following expressions.

- a) $- / 10 \ 5 \ 2$
- b) $(- / 10 \ 5) \ 2$
- c) $\div / 10 \ 5 \ 2$
- d) $(\div / 10 \ 5 \ 2) \div 2$
- e) $\div / (10 \ 5 \ 2) \div 2$

Let us pause moving into new material and see what we can do with what we have learned thus far.

Exercise 17

- a) John Von Neumann is reported to have said that the optimum lecture is *one* micro century, how many minutes is that?
- b) Write an expression to calculate the cost of four shirts at \$10.95 per shirt, including a sales tax of 8.25%
- c) Given that the temperature in Celsius is five-ninths of the Fahrenheit temperature minus 32. Create a variable Celsius that represents the Celsius temperature given the Fahrenheit temperature.⁵ Note: A good test for temperatures is the vector:

32 50 212
- d) A light year is the distance light travels in one year. Write an expression to determine the number of miles that a light year can travel in 1, 2, and 5 years, if light travels 186,282 miles per second.
- e) Each hair on your head grows a half a millimeter a day. A full head of hair consists of approximately 100,000 hairs. (1 meter equals 39.36 inches)
 - e-1) Write an expression to determine the total number of meters and inches a full head of hair grows in a day.
 - e-2) How long would it take to grow a hair 1 m longer? Write an expression to determine this.
 - e-3) How long would it take to grow a hair 1 inch longer? Write an expression to determine this.
 - e-4) Write a single expression that would determine the amount of time it would take for a hair to grow a meter, a foot, and an inch.

⁵ Note the ambiguity in the problem statement. It could be read in two ways; take five ninths of just Fahrenheit or five ninths of Fahrenheit minus 32. Here is a good example of having good test data.

Character Data

To this point, the data used has been numeric but APL handles both numeric and character data. Character data are the items enclosed in a pair of quotation marks.

Type `'Hello, World'`

APL's immediate response is `Hello, World`, a 12-item character vector. Each item within the quotation marks is a separate vector item. That also includes space or blank as a character.

You may also assign a name to this character vector.

`World←'Hello, World'`

When you created a vector of numbers, it generally took more than one graphic symbol, digit, to represent a simple number. For example, the number `512` is not represented as a three-item vector consisting of the digits `5`, `1`, and `2`. It seen as a single number separated by at least one space from another number. But in representing the number as a character string each of the digits making up the number is a separate character. Thus `'512'` is a three-item character vector, but as a number it is a single scalar.

Exercise 18

a) How many separate items are there within each of the following vectors?

a-1) `'Hello'`

a-2) `'Hello, World'`

a-3) `'512'`

a-4) `'12 5 -3.1'`

a-5) `317 'a' 'b'`

a-6) `317 'a' 'b'`

b) Enter as a character string the word `don't`.

The answer to problem *b)* is found in problem *a-6)*. In order to include the quote mark within a character string you must issue two quote marks for it.

In working through these exercises, did you notice something new in exercises 5 and 6? In those exercises the vectors containing both numeric and character data. APL allows you to intermixed character and numeric data in an array.

Such an array is called a *mixed* or a *heterogeneous* array. We shall encounter more such arrays later.

Let us now begin to investigate more of the primitive functions.

Primitive Functions

APL has many primitive functions represented by symbols. As we begin to examine them we need to understand the nature of the data as the functions create it. All data have certain attributes. For example, is it numeric or character, a scalar or vector. In APL, as data is created, these attributes are carried along with it. These attributes are

Data (usually)

Type

Shape

Rank (to be covered later)

The various functions may affect all these attributes or only a portion of them. For example, a function may only rearrange the data into a different shape but does not change the data.

Note that the phrase “usually” appears after *Data*. Seems strange, but it is indeed possible to have a named variable that has type, shape, and rank but no data associated with it. The concept of emptiness is important in APL. This situation must be made specific in APL. At this point in the presentation `A←'` is a way to create variable. With no data, `A` is called an *empty vector*. Such a vector could be employed where you begin to build a list of names. There will be other ways to create emptiness as we encounter other functions.

Shape of ρR

When working with vectors, we often wish to know how many items make up the vector. The monadic function *Shape of* ρ , (*Ctrl+r*), applied to its right argument, returns, as a vector, the number of items the right argument contains. For example,

```

       $\rho$ 12 4.5 ^6.78
3
       $\rho$ '12 4.5 ^6.78'
12

```

In the second example, each of the items that appear between the quote marks is a character including the blanks. In the first example the blanks are cosmetic and are not part of the data.

Exercise 19

Determine the *Shape of* the following:

- a) 'Hello'
- b) 'Hello, World'
- c) '512'
- d) '12 5 ^3.1'
- e) '12 5 3.1'
- f) 12 5 - 3.1
- g) '12 5' ^3.1
- h) 12 5 ^3.1
- i) 317 'a' 'b'
- j) 317 'a' 'b'
- k) 23

Did something strange happen with problem k)? Recall that a simple scalar namely a single number or character has no shape but *Shape of*, ρR , always returns a vector. In this case, it returned an *empty vector*. Taking the *Shape of* again of an empty array will be a zero indicating that there is no data in the empty vector. Thus:

```

       $\rho\rho$ 23
0

```

Catenation L , R

On occasion given two separate vectors, there may be a need, to combine them into a single vector. The symbol “,” representing the dyadic function of *Catenation*, L , R will combine a vector or scalar left argument with a scalar or vector right argument producing a single vector. For example, given a vector `grades←87 91` and `newgrad←95`,

```
      grades←grades,newgrad
      grades
87 91 95
```

Exercise 20

Display and determine the shape of the following:

- a) 'OK ' '512'
- b) 'OK' , 512
- c) 'OK' 512 '678'
- d) 'OK' , 512 '678'
- e) 'OK' 512 , '678'
- f) 'OK' , 512 , '678'
- g)⁶ x,20, x←2 10 1
- h) a←10 2 × b←5
- i) Write an expression that will create the character vector:

Team1 vs Team2

given t1←'Team1' and t2←'Team2'

⁶ Note that assignment, ←, is a function and, as such, does not have to appear as the last executed function in an expression.

Relational Functions $L < \leq = \geq > \neq R$ ⁷

A *relational function* is a binary function that returns either a 0 or a 1. The function symbols for the dyadic Relational functions are $< \leq = \geq > \neq$, representing *less than*, *less than or equal*, *equal*, *greater than or equal*, *greater than*, and *not equal*. The Relational functions return a 1 if a relationship is true or a 0 if a relationship is not true. They are *scalar* functions.

For example,

```

10=11
0
5≤2 5 7
0 1 1
5=2 5 7
0 1 0
'F'='FfC'
1 0 0

```

Illustration

Let $v \leftarrow 3.2 \ -7.6 \ 0 \ 15 \ -12 \ 0 \ 4.7$

Convert all the negative numbers in v to zero.

```

      v×v≥0
3.2 0 0 15 0 0 4.7

      v
3.2 -7.6 0 15 -12 0 4.7

      v←v×v≥0
      v
3.2 0 0 15 0 0 4.7

```

Recall the only way to change the data associated with a variable name is to reassign it with the new data.

⁷ $< (Ctrl+3); \leq (Ctrl+4); = (Ctrl+5); \geq (Ctrl+6); > (Ctrl+7); \neq (Ctrl+8)$

Exercise 21

What will the following expressions return?

- a) `'A' = 'a'`
- b) `'Ray' = 'Roy'`
- c) `'5' = 5`
- d) `2 5 10 < 10 5 2`
- e) `2 5 10 ≥ 10 5 2`
- f) `5 > 10 5 2`
- g) `'Ray' ≠ 'Roy'`
- h) For `v←3.2 -7.6 0 15 -12 0 4.7`, find the number of positive numbers.
- i) Starting with `v←3.2 -7.6 0 15 -12 0 4.7`, replace the zeros by 100.
- j) *The Hamming Distance:*

When binary information is transmitted, there is a need to determine if any of the bits were changed during the transmission. The number of bits changed is called *The Hamming Distance*. For example, for

```
sent←0 0 1 1 0 0 0 1 1 1
recv←0 1 1 1 0 0 0 1 0 1
```

The Hamming Distance is 2.

Write an expression for the Hamming Distance between `sent` and `recv`.

It is time to summarize what has been presented (and understood?).

Summary 1: The Basics

1. APL is a highly symbolic language with all of its primitive symbols displayed on the language bar.
2. APL will respond to anything you entered when you depress the Enter key. It responds to you in a REPL manner.
3. The symbols for addition, subtraction, multiplication, and division are $+$ $-$ \times \div .
4. Negative numbers are indicated through the use of the “high minus”, $\bar{}$.
5. A name may be assigned to data only via the assignment symbol, \leftarrow .
6. Most function symbols are defined either monadically with just a right argument or dyadically with a left and right argument.
7. Thus, *all primitive functions will have a right argument*.
8. There is *no hierarchical ordering* among functions.
9. The single rule of grammar for functional expressions is *the right most function whose arguments are available is evaluated first*. This is sometimes referred as the *right to left scan*.
10. The use of parentheses overrules the normal order of scanning.
11. An APL expression may contain several functions.
12. APL data consists of numbers and characters.
13. Character data is enclosed by quotation marks.
14. A single number or character is called a *Simple Scalar*.
15. APL will accept linearly ordered data, called *vectors* containing either numeric or character data where the numbers are separated from each other by at least one blank and the characters are not.
16. The functions $+/V$ and \times/V produce the sum and produce of a numeric vector.
17. A *Scalar* function executes on the individual scalar data items in its argument.
18. A *Scalar* function behaves symbolically as follows:
 - $S \text{ fn } V$ or $V \text{ fn } S$, meaning the scalar argument S is applied via the function to each of the arguments of V .
 - For $V \text{ fn } V$, the *Scalar* function fn is applied to matching left and right arguments so both arguments must be the same length.
19. The monadic function *Shape of*, ρ , returns the number items in a vector.
20. Dyadic *Catenation*, joins together the left and right arguments into a vector.
21. *Relational* dyadic functions $<$ \leq $=$ \geq $>$ \neq return zeros or ones.
22.)Fns and)Vars display the names of the created functions and variables in the currently active workspace.
23.)Erase will remove any named object within the workspace.

Programming

So far we have been using APL much like a pocket calculator. Now it is time to consider active programming within APL. At this point you can create and name data; either numeric or character. You have also encountered a set of primitive functions.

Exercise 22

List the primitive functions you have encountered thus far.

Note whenever you created an expression upon depressing the *Enter key*, it was immediately executed. There are going to be times when that is not satisfactory. Many times, more than one expression will be needed to reach the solution of your problem or you may wish to execute those expressions repeatedly on different data. This is where programming enters the picture.

Programming is the process of translating a given algorithm into a sequence of instructions executable by a computer. The term *algorithm* represents a set of explicit steps to achieve a solution to a problem. A good analogy to the nature of an algorithm is the recipe used in cooking. This sequence of instructions, often called *statements*, collectively become a *program*. Using a programming language to create a program is called *coding*. Coding is done after the more challenging process of establishing an algorithm is created.

The very important point of programming is *a program separates the creation of statements from their execution*.

Let us create a program to determine the Celsius temperature given a Fahrenheit temperature as an example. An algorithm for it would be to multiply $\frac{5}{9}$ times the Fahrenheit temperature from which 32 has been subtracted. Here is an APL representation of that statement:

```
(5÷9)×f-32
```

Of course, if you executed this statement immediately you would get a Value error since you have not defined *f*. Therefore define *f* and re-execute the expression:

```
f←32 50 212 -40
(5÷9)×f-32
0 10 100 -40
```

Here you have executed this expression and found the Celsius temperatures for four Fahrenheit temperatures. Since we may need many other temperatures at other different times, we need to turn this expression into a program rather than retyping the expression again.

There are two approaches to creating a program in APL.

One is through a *Direct Function*, called a “*Dfn*” (sometimes called a “Dynamic Function”), which we will first introduce. It provides a simple straightforward path to the creation of a program.

The other is the *Procedural* approach that has existed in APL from its beginning. It will be introduced later.

Using the *Direct Function*, *Dfn*, approach to programming, the program **F to C** for the Fahrenheit to Celsius conversion is:

$$\mathbf{F\ to\ C} \leftarrow \{ (5 \div 9) \times \omega - 32 \}$$

You use it just like the primitive monadic functions.

```

      F to C 32 212 ^40
0 100 ^40
      F to C 50 60 70
10 15.55555556 21.11111111

```

The *Dfn* is unique to Dyalog APL.

Let's examine the structure of a *Dfn* and how to create one.

The structure of the above *Dfn* **F to C** program consists of:

- A name assigned to the program followed by,
- the assignment symbol \leftarrow , then
- the left curly brace, { begins the definition,
- statements line by line, then
- the right curly brace, } ending the definition.
- A *Dfn* may be monadic or dyadic.

In defining a *Dfn* you are separating the definition of statements within the *Dfn* from their actual execution. Therefore, you need placeholder names indicating where the arguments are to be used within the body of the *Dfn* when it is executed. They are ω and α .

ω (*Ctrl+w*) represents the right argument,

α (*Ctrl+a*) represents the left argument, if it has one.

Any names created within the definition statements are local.

*Finally, the first unassigned statement in a multi statement *Dfn* is the result of the *Dfn*, and any expression thereafter is ignored.*

Editing

If the *Dfn* consists of just one line, it may be built directly in the session manager. Multi-line *Dfns* are built via the system command `)ED`. Should more lines be needed, then issue:

`)ED name`

This will open an *edit window* which appears in the session window. The chosen “*name*” in the `)ED` expression is the name of the *Dfn* to be used when executing it. Then you entered the statements of the function that will be executed later.

To create a *Dfn*,

1. Establish a name for it. E.g. `)ED FtoC`
2. In the edit window this chosen name appears after which enter `←{`
3. Followed by lines of executable statements and comments,
4. Close the *Dfn* with the `}`

To exit the edit window and save your work, use the *Esc* key;

to exit without saving the information in the edit window, use the *Esc* and *Shift* keys together.

Comments

One of the important aspects of programming is that of commenting your work. That can be done in several ways after the program is running, you should create a document describing the algorithm used and how to use the program. Aside from a description of how to use the program, further documentation is often missing or out of date. Hence a second form of commenting can and should be used; that is providing comments within the body of the program. APL permits comments within the function using the *comment symbol*, **␣** (*Ctrl*+,). When it is encountered on a line of code in a program everything to its right is ignored. Returning to the *Dfn FtoC* function, one can insert a comment as follows:

```
FtoC←{(5÷9)×ω-32 ␣ Fahrenheit to Celsius}
```

Or

```
[0] FtoC←{ ␣ Fahrenheit to Celsius
[1]   ␣ω: Fahrenheit temperatures
[2]   (5÷9)×ω-32 }
```

In commenting, state what is intended by the expression rather than a word translation of the functions in it. In addition to commenting on specific lines of a function, a good practice is to comment on the purpose of the function and the nature of the expected arguments.

For an argument, where important or specific, state

- values: integer, nonnegative,
- type: numeric, character, mixed,
- shape: scalar, vector, matrix, array,
- depth: simple, nested.

Stating more specifically what the function is expecting may prevent errors because the wrong data was given.

Again recall the result of a multiline *Dfn* is the result of the first unassigned statement. Any statements following the unassigned one are ignored. Following the unassigned expression could be a good place to put general comments.

Recall a *Dfn* can be used like a primitive function:

- a) its result displayed immediately,
e.g., `FtoC 50`
10
- b) assigned a name,
e.g., `Celcius←FtoC 50`
- c) used in a larger expression including other primitive functions or *Dfns*,
e.g., `10+FtoC 50`
20

Note that creating expressions containing a mixture of primitive and defined functions is possible because of the right to left scanning rule.

In any case, meaningful names improve readability.

Exercise 23

- a) Create the expression for converting Celsius temperatures to Fahrenheit temperatures.
- b) Using the editor, create the *Dfn* program **CtoF** to get Fahrenheit temperatures given Celsius temperatures. Use the Celsius temperatures of 0, 10 100 -40 as test cases.
- c) Create a *Dfn* that given a three item vector defining hours, minutes, and seconds, returns the total number of seconds.
- d) Create a *Dfn* that given a kilometer distance will produce the labeled kilometer and mile values. (1 kilometer=0.62 mile)
- e) Create a *Dfn* that, given a price and a vector of prices changes as percentages will produce the price change values.
- f) Create a set of *Dfns* that allow a person to enter the expression
`WHAT IS 1.2 TIMES 5.1`
and receive the proper answer.
- g) There are several possible solutions to the previous problem. Provide at least one other solution.

Direct Function Guards

A *guard* is an expression in a *Dfn* that provides the ability to change the next statement to be executed. The guard expression is defined to be a single valued Boolean expression producing either a 1 or a 0 followed by a colon. For example within a *Dfn* one could write

$\omega > 0 :$

This guard statement can be read as “If ω is greater than zero then execute what follows the : ”

This guard expression may then be followed by the statement to be conditionally executed. This statement could be a *Dfn*. The guard expression could stand alone but then followed on the next line by the statement to be conditionally executed. This is the only control construct available with *Dfns*.

Let us return to the *Dfn* *FtoC*, the Fahrenheit to Celsius conversion program. With a guard, a more general temperature function can be built:

```
fccf←{a conversions between Fahrenheit and Celsius
  A  $\alpha$ ='f' indicates a Celsius to Fahrenheit conversion
  A  $\alpha$ ='c' indicates a Fahrenheit to Celsius conversion
  A  $\omega$  either Fahrenheit or Celsius temperatures
   $\alpha$ ='f': CtoF  $\omega$ 
  FtoC  $\omega$ 
}
```

Note that a *Dfn* may contain calls to other existing *Dfns*, in this case *CtoF* and *FtoC*.

```
'f' fccf 100
212
'c' fccf 212 32
100 0
```

Guards applies only to *Dfns*. It is the only conditional control accepted by a *Dfn*. All the other control word structures are not accepted.

Exercise 24

- a) Modify the *Dfns fccf* to accept 'f' or 'F' and 'c' or 'C' as the left argument.
- b) Modify the *Dfns fccf* such that if 'f', 'F', 'c', or 'C' are not used properly, an error message such as "Improper letter used."
- c) When binary information is transmitted, there is a need to determine if any of the bits were changed during the transmission. The number of bits changed is called the *Hamming Distance*. For example for


```
sent←0 0 1 1 0 0 0 1 1 1
recv←0 1 1 1 0 0 0 1 0 1
```

 the Hamming Distance is 2.
- d) Write a *Dfn* that, given two equal length bit strings, would return the Hamming Distance for the pair of bit strings.
- e) Create a *Dfn* that when given Miles will return the matching Kilometers
or
 when given Kilometers will return the matching Miles.
- f) Create a dyadic *Dfn* which will accept a numeric vector of positive, negative and zeros and either a 1 or $\bar{1}$ or 0. It returns the sum of positive, or negative or zeros based on receiving a 1 or $\bar{1}$ or 0.

Summary 2: Programs

1. Programs encapsulate APL statements that are meant to be used on many sets of data and to avoid the need of reentering APL expressions repeatedly.
2. Creating a program allows one to separate the creation of statements from their actual execution.
3. Programs may be created by either
 the direct function, *Dfn*, form or
 the traditional function form (presented later).
4. *Dfns* may be defined either monadically or dyadically.
5. The body of the *Dfn* is enclosed by the braces { }.
6. In the *Dfn* statements ω is the placeholder for the right argument and α is the placeholder for the left argument if it has one.
7. All variables created in the body of the *Dfn* are strictly local.
8. The result of a *Dfns* is the first unassigned statement in it.
9. Multiline *Dfns* are created via the system command `)ed` or the system function `⌈ed`.
10. The comment symbol `⌈` is to be used in a program to document the behavior of a portion of the program or define the expected nature of ω and α .
11. The Data Guard expression, " a single valued Boolean expression : ", provides the ability to change the sequence of execution.

Primitive Scalar Functions

At this point you have been exposed to enough of APL to code. You understand the single rule of grammar, you can create variables, write *Dfns*, and save and reload your work.

The primitive functions to which you have been exposed only allow you to do rather simple things. It is time to introduce you to more of APL's primitive functions. The primitive functions play different roles in the language.

- The *mathematical functions* transform numeric data.
- The *structural functions* relate to the structure of arrays, creating or rearranging them.
- The *array inquiry functions* provide logical inquiry and comparison ability between data items.
- The *data selection and selector functions* address retrieving and replacing items within an array.
- The *search and sort functions* permit finding and rearranging data within arrays.

To completely understand any APL function, you must understand how it reacts with all of the data attributes— data, type, shape, and rank.

.

Much of the following presentation will follow the function-by-function pattern introduced in the context of their use with interesting Illustrations.) The definition of each new function will include exercises and illustrations. If the function or material is familiar to you move on but first do the exercises. They will strengthen your understanding of the material.

Now some of the following functions may not be familiar to you if you have not had more advanced mathematics. For those that are familiar with the functions you should find their behavior in APL straightforward. If you are not familiar with them, one of the actions that is encouraged in this document is that the you should frequently answer the question, “*What if I did....?*” Explore— you can't harm the APL interpreter.

Well, if you did, you have found a bug!

But before we move into more scalar functions the following function is so useful as you create named variables that it is introduced here.

Tack $L \vdash R$ $L \dashv R$ and Same $\vdash R$ $\dashv R$

The symbol \vdash (*Ctrl+)* is called *Right*. When used, it produces whatever is to its right ignoring what may be to its left.

The symbol \dashv is called *Left* (*Ctrl+shift+)*. When used, it produces whatever is to its left ignoring what may be to its right. This is the case for either monadic or dyadic case.

Not too exciting, is it?

Here is a nice use of them. When you create a variable name for an expression, you often then on the next line enter the variable name to see its value. With *Same* you may both assign a name and display the result of the assignment on the same line. For example:

```
\st←'How about that '  
How about that
```

Currently, the symbol \dashv *Left Tack* (*trl+shift+)* behaves the same way when used monadically.

```
\w←'OK '  
OK
```

When used dyadically, they can replace a pair of parentheses.

Exercise 25

Determine the result of each expression before keying it in:

- a) $\vdash v \leftarrow 1 \ 2 \ 3 \ 4$
- b) $\dashv v$
- c) $10 \times 1 \ 2 \ 3 \ 4$
- d) $10 \times \vdash 1 \ 2 \ 3 \ 4$
- e) $10 \times \dashv 1 \ 2 \ 3 \ 4$
- f) $10 \times 1 \ 2 \dashv 3 \ 4$
- g) $10 \times 1 \ 2 \vdash 3 \ 4$
- h) $1 \ 2 \vdash 3 \ \vdash 4$
- i) $1 \ 2 \dashv 3 \dashv 4$
- j) $1 \ \dashv 2 \ \vdash 3 \ 4$
- k) $1 \ \vdash 2 \ \dashv 3 \ 4$
- l) $\vdash v \leftarrow 1 \ 2 \vdash 3 \ 4$
- m) $\vdash v \leftarrow 1 \ 2 \dashv 3 \ 4$

Let's begin by looking at some of the scalar mathematical functions.

Maximum and Minimum \lceil R \lfloor R

The dyadic use of \lceil (*Ctrl+s*) is called *Maximum*. It does what the name implies; it finds the larger value. The dyadic use of \lfloor (*Ctrl+d*) is called *Minimum* and similarly it finds the smaller value.

Exercise 26

Determine the result of the following expressions. First determine the values before executing the statements on the computer.

- a) $12 \lceil 23$
- b) $12 \lceil 23 \ 5 \ 15$
- c) $12 \ 6 \ 20 \lceil 23 \ 5 \ 15$
- d) $2.3 \ ^{-12}.1 \ ^{-12} \lceil 2.35 \ ^{-12}.5 \ ^{-12}$
- e) $23 \ 5 \ 15 \lfloor 10$
- f) $12 \ 6 \ 20 \lfloor 23 \ 5 \ 15$
- g) $2.3 \ ^{-12}.1 \ ^{-12} \lfloor 2.35 \ ^{-12}.5 \ ^{-12}$

Floor and Ceiling \lfloor R \lceil R

The monadic use of \lceil and \lfloor are called *Ceiling* and *Floor*.

Putting the behavior of *Ceiling* (*Ctrl+s*) into words: *Ceiling*, \lceil R, returns the *smallest integer greater than or equal to* the number R.

Exercise 27

- a) $\lceil 3.56$
- b) $\lceil 3.63 \ 6.2 \ 6.9 \ 7$
- c) $\lceil 3.4 \ ^{-3}.4 \ 6 \ ^{-6}$
- d) $\lceil ^{-10}.32 \ 0 \ ^{-1}.73 \ 1.73 \ 12$

Similarly, *Floor*, \lfloor R (*Ctrl+d*), returns the *largest integer less than or equal to* the number R.

Exercise 28

- a) $\lfloor 3.56$
- b) $\lfloor 3.63 \ 6.2 \ 6.9 \ 7$
- c) $\lfloor 3.4 \ ^{-3}.4 \ 6 \ ^{-6}$
- d) $\lfloor ^{-10}.32 \ 0 \ ^{-1}.73 \ 1.73 \ 12$

Absolute |R

How about the function *Absolute*, |v (Ctrl+m)? Even the name helps.

In words then, |R, returns the absolute value of the number *S*; its magnitude; its positive value.

Exercise 29

- a) |v+2.3 -2.3 0.36 -.36 -12.7 12.7
- b) v**v

Are v**v and |v equivalent?

Yes, but only for real numbers— not for complex numbers.

.

Residue L | R

The dyadic use of | (*Ctrl+m*) represents the *Residue* function. When L and R in L | R are positive numbers, the *residue* is an integer remainder when R is divided by L.

The major use of Residue is usually with positive L and R.

Exercise 30

- a) 10 | 45 12 123 6 0
- b) 5 | 45 12 123 6 0
- c) 2 | 45 12 123 6 0
- d) 7 | 45 12 123 6 0
- e) 12 | 45 12 123 6 0
- f) 10 | 45.3 12.6 123.45 6.2 0.89
- g) 2.5 | 45.3 12.6 123.45 6.2 .089
- h) 25 | 453 126 1234.5 62 8.9
- i) 1 | 45.3 12.4 123.45 .0897 What do you get here?

If the left argument L is a negative number, the residue is the negative complement of the left argument... unless a remainder is zero. For example:

```

      ⍒x←7|45 14 123 6 0 ⍒45 ⍒14 ⍒123 ⍒6
3 0 4 6 0 4 0 3 1
      ⍒y←-7|45 14 123 6 0 ⍒45 ⍒14 ⍒123 ⍒6
-4 0 -3 -1 0 -3 0 -4 -6
      x-y
7 0 7 7 0 7 0 7 7

```

Illustration

The *Residue* function, using 2 as the left argument, can determine if an integer is even or odd in that even integers return a zero and odd numbers return a 1.

```

      2|3 12 -4 5 -7 123 54
1 0 0 1 1 1 0

```

Exercise 31

- a) 1 | -45.3 12.4 123.45
- b) -123.45 12.4 45.3 - | 1
- c) -123.45 12.4 45.3 | 1
- d) 10 | 45.3 -45.3
- e) Write an expression to replace each number divisible by 3 in a vector by a 0.
- f) Write a *Dfn* to extract the fractional part of a number.

Boolean Functions⁸ $L \wedge v \tilde{\wedge} \tilde{v} R$ and $\sim R$

There are some other functions within APL that only return zeros or ones⁹. Collectively they can be called the query functions. Does this seems strange? The query functions are involved in decision-making in programs. The name query implies asking a question. For example you may have been asked “Did you do your homework?” The answer expected is either “Yes” or “No”. Of course this is in a “No Excuse” environment. You are able to ask questions of your computer too. The Relational functions introduced earlier also allow you to ask such questions.

However, questions are often are paired together. Have you ever said “I wish to call my friend and talk to her”. You are stating that you wish to do two things: call *and* talk. Or you could have said “I will have either chocolate *or* strawberry ice cream *or* maybe both in my sundae.” The *and* and the *or* choices are representative of logic decisions. Such functions are also used in decision-making and in selecting items in programs. A program could ask a question like “Is the number greater than ten or less than five?”

The *Boolean functions* in APL are more formal. These functions, except for the extensions to \wedge and v , accept only zeros and ones as arguments and return only zeros and ones. When a program issues a $L \wedge R$, it is asking if both L and R are true and relates the answer true with the digit 1 and false with the digit 0.

The Boolean functions are:

Dyadic	Symbol	Monadic	Entry
And	\wedge	—	(Ctrl+0)
Or	v	—	(Ctrl+9)
Nand	$\tilde{\wedge}$	—	(Ctrl+Shift+0)
Nor	\tilde{v}	—	(Ctrl++Shift+9)
—	\sim	Not	(Ctrl+t)

They are all *scalar* functions.

⁸ The name come from George Boole who created an algebraic system involving these logical statements.

⁹ In some programming languages, the language returns True or False instead of 1 or 0. That response is less useful, unless you want the computer to talk to you.

Since the input to these functions are only binary, their definitions can be given as tables.

- For *And* $A \wedge B$:

		B	
		\wedge	
A	0	0	0
	1	0	1

$A \wedge B$ can only produce a 1 only if both A and B are 1's.

- For *Or* $A \vee B$:

		B	
		\vee	
A	0	0	0
	1	1	1

$A \vee B$ produces a 1 if either A or B or both are 1

- For *Nand* $A \tilde{\wedge} B$:

		B	
		$\tilde{\wedge}$	
A	0	1	1
	1	1	0

$A \tilde{\wedge} B$ produces a 0 only if both A and B are 1

- For *Nor* $A \tilde{\vee} B$:

		B	
		$\tilde{\vee}$	
A	0	1	0
	1	0	0

$A \tilde{\vee} B$ produces a 1 only if both A and B are 0.

- For *Exclusive Or* $A \neq B$:

		B	
		\neq	
A	0	0	1
	1	1	0

$A \neq B$ produces a 1 only if A and B are not the same.

- For *Not* $\sim A$:

		B	
		\sim	
	0	1	
	1	0	

\sim changes a 0 to a 1 and a 1 to a 0.

They are all *scalar* functions.

Exercise 32

Determine the result of the following expressions:

- a) $0\ 0\ 1\ 1\ \wedge\ 1\ 0\ 1\ 0$
- b) $0\ 0\ 1\ 1\ \vee\ 1\ 0\ 1\ 0$
- c) $0\ 0\ 1\ 1\ \tilde{\wedge}\ 1\ 0\ 1\ 0$
- d) $0\ 0\ 1\ 1\ \tilde{\vee}\ 1\ 0\ 1\ 0$

With $a \leftarrow 7\ -3\ 12\ 8\ -5\ 0\ -2$

- e) $a \leq 0$
- f) $a > -4$
- g) $(a \leq 0) \wedge a > -4$
- h) $(a \geq 0) \wedge a < 10$
- i) $a \geq 0$
- j) $a < 10$
- k) $(a \geq 0) \tilde{\wedge} a < 10\ 12$

For $v \leftarrow 1\ 5\ 2\ 9\ 7\ 3\ 2\ 3\ 1$

- l) Using v , Generate a binary vector with 1's only for 2 and 3.
- m) Replace all the items of v by zero except for the two's and three's.
- n) Given a pair of numbers, $x\ y$, create the *Dfns*'s that will determine if another number z , lies between the numbers x and y .
 - n-1) excluding x and y .
 - n-2) including x and y .

Greatest Common Divisor *and* Least Common Multiple

The dyadic symbols \wedge (*Ctrl+0*) and \vee (*Ctrl+9*) were given extended meanings. The acceptable arguments for these two functions was extended to accept numeric data beyond just zeros and ones.

Greatest Common Divisor

The function $L \vee R$ produces the largest value that divides both L and R without a remainder. For example:

$$18 \vee 24 \\ 6$$

If you consider the integer divisors of **18** and **24**,

for **18**: 1 2 3 6 9 18

for **24**: 1 2 3 4 6 8 12 24

You see that **6** is the largest divisor common to both **18** and **24**.

While the Greatest Common Divisor definition is usually given for positive integers, the function \vee will also accept negative integers and decimal numbers.

Exercise 33

- a) $72 \vee 90$
- b) $-72 \vee 90$
- c) $7.2 \vee 90$
- d) $7.2 \quad 90 \div 3.6$
- e) $7.2 \vee 9.0$
- f) $60 \vee 75 \vee 300$
- g) $17 \vee 37$ **A** Why?
- h) Given a rectangular area 35 inches by 28 inches to be subdivided into identical squares, what would be the largest possible length of the side of a square that would divide the rectangle exactly?

Least Common Multiple

The function $L \wedge R$ produces the smallest number that is divisible by both L and R , without producing a remainder. For example:

$$\begin{array}{r} 24 \wedge 90 \\ 360 \end{array}$$

This could be calculated manually by multiplying together the highest powers of all the prime factors of A and B .

$$\begin{array}{l} 24 = 2^3 \times 3 \\ 90 = 2 \times 3^2 \times 5 \end{array}$$

Hence, the LCM is $2^3 \times 3^2 \times 5 = 360$.

While the Least Common Multiple definition is usually given for positive integers the function \wedge will also accept negative integers and decimal numbers.

Exercise 34

- a) $36 \wedge 25$
- b) $13 \wedge 7$
- c) $72 \wedge 90$
- d) $72 \wedge 90 \wedge 25$
- e) $^{-}36 \wedge 25$
- f) $^{-}36 \wedge ^{-}25$
- g) $3.6 \wedge 25$
- h) $3.6 \wedge 2.5$
- i) $^{-}3.6 \wedge 2.5$
- j) Consider the front and back wheels of a tricycle. The front wheel is 32 cm in circumference and the back wheel is 24 cm in circumference. If the current contact with the ground were marked for both wheels, how many revolutions going forward would there have to be made so that the marks of both would be contacting the ground together again?

Power *R L*R

Exponentiation is provided via the *Power* symbol $*$ (*Ctrl+p*), for both monadic and dyadic uses. When the power symbol is used as a dyadic function the left argument is the base to which the power is applied.

Exercise 35

- a) $10 * 2 \ 0 \ ^{-2}$
- b) $2 * 2 \ 0 \ ^{-2}$
- c) $^{-4} * 2 \ 0 \ ^{-2} \ ^{-3} \ 3$
- d) $16 * .5$
- e) $2 * 1 \ 2 \ 5$
- f) $2 * .1 \ .2 \ .5$
- g) $2 * ^{-1} \ ^{-2} \ ^{-5}$
- h) $2 * ^{-.1} \ ^{-.2} \ ^{-.5}$
- i) $^{-2} * 1 \ 2 \ 5$
- j) $^{-2} * .1 \ .2 \ .5$
- k) $^{-2} * ^{-1} \ ^{-2} \ ^{-5}$
- l) $^{-2} * ^{-.1} \ ^{-.2} \ ^{-.5}$

Recognize problem *d*) as the square root of sixteen?

- m) Write the expression for the cube root of 27.
- n) Write the expression for the two thirds root of 125.
- o) When *Power* is used in a monadic fashion, it raises the number *e* (2.718281828...) to the stated power.

Exercise 36

Can you state in words what these expressions do?

- a) $* \ 1$
- b) $* \ 2$
- c) $* \ 1 \ 2 \ ^{-2} \ 5 \ ^{-5}$
- d) Write the expression for the reciprocal of the $^{4/3}$ power of *e*.

Compound Interest

The Compound Interest formula:

$$S = P(1+i)^n$$

where

P is the current value, (the principal).

i is the rate of interest per conversion period.

n is the number of interest periods.

S is the compounded amount.

In other words, the compound interest formula can determine the growth, S , of an investment, P , for the rate of interest per period, I , over the number of periods, n . For example, investing \$1000 at 4% interest annually for one and two years would yield \$1040 and \$1081.60.

Exercise 37

- a) Write the expression that will generate these values.
Often the interest rate is stated as “earning 4% compounded semiannually”. Here than the interest rate, I , must be determined for the smaller period of time, Were 4% interest evaluated semiannually, then I in the formula would be $\frac{1}{2}$ of 0.04. Also, the number of periods, n , is multiplied by 2. For example, investing \$1000 at 4% interest compounded semiannually for one and two years would yield \$1040.4 and \$1082.43216. In APL that would be:

$$1000 \times (1 + .04 \div 2) * 2 \quad 4$$

$$1040.4 \quad 1082.43216$$

- b) Write the expression for investing \$1000 at 4% interest compounded monthly for one and two years. It should yield \$1040.741543 and \$1083.142959.
- c) Create a *Dfn* to determine the compound amount for one dollar given the annual interest rate, the period over which it is compounded, and the number of periods compounded. For example, investing \$1000 at 4% interest compounded semiannually for two years,

$$1000 \times (.04 \div 2) \text{ Compound } 4$$

$$1082.43216$$

- d) Having created the *Dfn* for compound interest determine how much you would gain if 2% interest were compounded daily rather than annually on a \$1000 in a year.

Present Value

The *Compound Interest* formula allows you to determine the growth of an investment, P , at a later time. There is another question that frequently arises— namely, how much must I invest today to earn a given amount at a later time?

The *Present Value* formula will do that for you:

$$P = S(1+i)^{-n}$$

where

S is the desired amount,

i is the rate of interest per conversion period.

n is the number of interest periods

P is the original investment to be made to achieve S .

For example, if \$1000 were desired in 3 years during which time the rate of interest was 4% compounded semiannually, \$862.30 would have needed to be invested. In APL the expression to achieve that would be

```
1000*(1+.04÷2)*^-2×3
862.296866
```

Exercise 38

- You can check your Present Value answer by putting the value received into the Compound Interest equation. Do that for the above answer.
- Create a *Dfn* to determine the Present Value of one dollar, given the annual interest rate, the period over which it is to be evaluated and the number of periods. For example how much would you have to invest to earn \$1000 in 2 years where the annual interest rate was 3% compounded monthly.
- As you have observed the formula for Compound Interest and Present Value are very similar. What requirement would enable the Compound Interest function (or Present Value function) to be able to return either the compounded amount or the original investment?

System Facilities

Before moving to some non-scalar functions of the APL language, it is timely to be aware of other system facilities that support the APL language. namely *System Commands* and *System Variables* and *System Functions*.

System Commands were introduced In Appendix A. They all begin with a `)`. They cannot be used within APL programs.

System Variables and *System Functions* give the APL user more access to the system software that supports the APL language. They begin with `⍵` (*Ctrl+L*), and may be used with expressions and defined functions. Specific ones will be introduced as needed.

The System Variable Index Origin, `⍵io` is one of the most frequently used. It may be assigned only *zero* or *one*. It determines the index value of the first element of a non-empty array. The need for using zero or one can be explained by noting the actions of counting and measuring. When you count you begin with one but when you measure you begin with zero. It is an implied argument to functions where counting or generating indices occur. It will be explicitly noted in the definition of functions influenced by `⍵io`.

The system variables `⍵A` or `⍵a` and `⍵D` or `⍵d` are also convenient and will be used throughout this text. Both produce character vectors.

```
⍵A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
⍵D
0123456789
```

Some Non Scalar Functions

All of the functions introduced thus far except for *Shape* of “ ρ ” and *Catenation* “ $,$ ” have been scalar functions. The scalar functions allow the very important feature of scalar extension:

$S \ f \ V$
 $V \ f \ S$

Let us consider some other very useful non-scalar functions where scalar extension does not occur.

Index Generator ιR

Generating a sequence of numbers is often useful. The monadic function *Index Generator*, ιR (*Ctrl+i*), can help doing that. It will generate the first R integers, starting at the index origin ($\Box io$).

The argument of ι is restricted to nonnegative integers.

For example:

```

       $\Box io \leftarrow 1$ 
       $\iota 4$ 
1 2 3 4
       $\Box io \leftarrow 0$ 
       $\iota 4$ 
0 1 2 3
    
```

Used within an expression it can produce a variety of integer expressions.

```

       $-4 + \iota 7$ 
-3 -2 -1 0 1 2 3
    
```

Exercise 39

- a) 15
- b) $2+15$
- c) 2×15
- d) $2-15$
- e) $(15)\div 5$
- f) $^{-}2+3\times 15$ (Note this is an arithmetic sequence.)
- g) Create a *Dfn* for the first R even numbers.
- h) Create a *Dfn* for the first R odd numbers.
- i) Create a *Dfn* for the first R multiples of L .
- j) Create the vector of odd integers from one to nine.
- k) Create the vector of multiples of 5 from $^{-}10$ to 10.
- l) Create a *Dfn* to create a arithmetic sequence given the number of terms, the distance between the numbers, and a starting value.
- m) Having created the *Dfn* of problem i), use it to create the arithmetic sequences
 - 1) $5\ 8\ 11\ 14\ 17\ 20$
 - 2) $^{-}6\ ^{-}2\ 2\ 6\ 10$
- n) Let $V\leftarrow 037037037$. Create a *Dfn* that will multiple V by the first n multiples of 3. Test your function for n equal to at least $3\ 5\ 10\ 15$. (Interesting isn't it?)
- o) In entering a set of numbers it would be more convenient to simply state the starting number 4 and the final number 9 to produce $4\ 5\ 6\ 7\ 8\ 9$.
- p) Define a *Dfn* TO to produce a string of numbers given a starting number and a final number. Test the function on the following test cases;
 - 1) $4\ TO\ 9$
 - 2) $^{-}2\ TO\ 6$
 - 3) $.5+2\ TO\ 7$
 - 4) $^{-}7\ TO\ ^{-}2$
 - 5) $4\ TO\ 4$
 - 6) $9\ TO\ 4$
- q) How would you use the TO function to generate a set of integers:

$$^{-}2\ ^{-}3\ ^{-}4\ ^{-}5\ ^{-}6$$

Roll and Deal ?R S?R

Often when you create *Dfns* you need to generate test data. The *monadic* “?” can help.

?J, where *J* is an non-negative integer vector,
will generate random integers selected from ιJ .

This function is also called *Roll*. Why? Issue **?6 6** several times. Doesn’t it look like you are rolling a pair of dice?

If you do **?6 6** several times you will come upon a case where the two integers are the same. Each selection is independent.

Note: in Dyalog Version 14 and beyond **?0** will generate a random ten place fraction from the open range 0 and 1. For example:

```
?0
0.01472753354
```

There are also times when you wish to generate a set of random integers that are never the same. The dyadic **S?R** will do that.

S?R where *S* and *R* are single scalar non-negative integers and the right argument *R* must be greater than or equal to the left argument *L*. Here *S* integers will be selected from the set of integers ιR , *without replacement*.

This function is called *Deal*. Why? If you issued **4?52**, that could be seen as getting four cards from a deck of 52 cards.

Note: *Roll*, **?R**, is a *scalar* function and *Deal*, **S?R**, is a *non-scalar* function.

Exercise 40

Determine in the following groups, which expression satisfies the request:

- a) Select with replacement three random integers from the integers $\{1, 2\}$:
 - a-1) $?12 \ 12 \ 12 \ 12$
 - a-2) $?13 \ 13 \ 13$
 - a-3) $3?12$
 - a-4) $?12 \ 12 \ 12$
- b) Select without replacement three random integers from the integers $\{1, 2\}$:
 - b-1) $?12 \ 12 \ 12$
 - b-2) $3?13$
 - b-3) $3?12 \ 12 \ 12$
 - b-4) $12?3$
 - b-5) $3?12$
- c) Select with replacement two integers between -4 and 4 , excluding both -4 and 4 :
 - c-1) $-4+2?9$
 - c-2) $-4+2?7$
 - c-3) $-4+?9 \ 9$
 - c-4) $-5+?7 \ 7$
 - c-5) $-4+?7 \ 7$
- d) Select without replacement two integers between -4 and 4 including both -4 and 4 :
 - d-1) $-5+?10 \ 10$
 - d-2) $-6+2?9$
 - d-3) $-5+2?10$
 - d-4) $-4+2?9$
 - d-5) $-5+2?9$
- e) Write the expression that will generate two one-decimal place random numbers between 1.5 and 2.5 , inclusive.
- f) Write the expression that will roll four dice.
- g) Write the expression that will deal a five-card hand from a standard 52-card deck.

A Review

Let us pause here and see if we can put some of the functions that have been introduced to some use. How do you round off a number? Let us round off the two numbers

10.324 and 10.378

to the hundredths position. A common way would be to add .005 to each number getting

10.329 and 10.383

and then discarding the digits to the right of the hundredths position getting

10.32 and 10.38.

In other words, if the digit in the thousandths position is greater than or equal to 5, add one to the hundredths position and truncate; otherwise just truncate.

Let us build a *Dfn* RND to do this simple rounding. RND is going to be dyadic with α indicating the decimal position at which rounding is to occur and ω being the number to be rounded. In developing the algorithm for RND let us use a for α and w for ω .

With $a \leftarrow 2$ and $w \leftarrow 10.378$

The first to 10.378 add .5 to the position one beyond the rounding position:

$w + (.1 * a + 1) \times 5$

getting 0.383

By multiplying the number by $10 * a$, the decimal point moves to the right two places:

$(10 * a) \times w + (.1 * a + 1) \times 5$
1038.3

Taking the floor of the number removes the fractional part:

$\lfloor (10 * a) \times w + (.1 * a + 1) \times 5$
1038

Finally, moving the decimal point two places back to the left again, by multiplying by $.1 * a$ yields the solution

$(.1 * a) \times \lfloor (10 * a) \times w + (.1 * a + 1) \times 5$
10.38

Hence a *Dfn* to round many numbers becomes

```
RND ← { ⍺⍺ rounding program
⍺ ⍺: integer decimal position to which to round
⍺ ⍻: decimal nos.
    (.1 * ⍺) × ⌊ (10 * ⍺) × (5 × 0.1 * 1 + ⍺) + ⍻
}
```

Consider:

What is easy for a human being to do, may take a lot of work for a computer to do. While it is easy to round off a number or two, the program comes into play when there are a large set of numbers to round.

Exercise 41

- a) 1 RND 75.63 75.69
- b) 2 RND 123.456
- c) ~ 1 RND 175.23 174.84
- d) 0 RND 175.23 174.84
- e) 5 RND 175.23 174.84
- f) ~ 2 2 RND 9876.543 1234.567

Summary 3: Scalar Functions

Here is the listing of the primitive APL functions introduced thus far.

The Arithmetic Functions:

<i>Dyadic</i>		<i>Monadic</i>	
Addition	$a+b$	Identity	$+b$
Subtraction	$a-b$	Negation	$-b$
Multiplication	$a \times b$	Signum	$\times b$
Division	$a \div b$	Reciprocal	$\div b$
Minimum	$a \lfloor b$	Floor	$\lfloor b$
Maximum	$a \lceil b$	Ceiling	$\lceil b$
Residue	$a b$	Absolute	$ b$
Power	$a * b$	Exponential	$* b$
Greatest Common Divisor	$a \wedge b$	Roll	$? b$
Least Common Multiple	$a \vee b$	—	

The Relational Functions:

Less than	$a < b$
Less than or equal	$a \leq b$
Equal	$a = b$
Great than or equal	$a \geq b$
Great than	$a > b$
Not equal	$a \neq b$

The Boolean Functions:

<i>Dyadic</i>		<i>Monadic</i>
And	\wedge	
Or	\vee	
Nand	$\tilde{\wedge}$	
Nor	$\tilde{\vee}$	
Not		\sim

Nonscalar functions:

Index Generator	ιR
Deal	$L ? R$
Same (right)	$\vdash R$
Same (left)	$\lhd R$

The very prominent mathematical variables:

π $\circ 1$ and e $* 1$

Index Origin	$\square IO$
Alphabet	$\square a$
Digits	$\square d$

Matrices

To this point we had been dealing with vectors. Many other arrangements of data exist. An especially useful arrangement is a tabular one where the data items are arranged in rows and columns.

For the readers who may not be too familiar with matrices the following material is presented as a brief introduction to them and their terminology. Readers who are comfortable with matrices may go directly to the APL facilities that support matrices and other arrays.

This material is here to give you a brief introduction to the nature of matrices. This is not a full treatment of matrices. Such treatments are covered in courses on Linear Algebra. Nonetheless, knowledge of the introductory material on matrices will help you understand APL's use of them. There is much APL notation associated with them.

A *matrix* is a rectangular arrangement of data. In layman terms, it is often spoken as a table. Often a matrix contains numeric data however in APL, characters may also occur intermixed with numbers. Matrices can be addressed in terms of its rows ,*r*, and columns ,*c*, where the row number appears before the column number. Number of rows and columns together define the shape of the matrix. A matrix which has the same number of rows and columns is called a *square matrix*. Consider the matrices **m** in which the integers 1–12 are arranged into a matrix of 3 rows and 4 columns and the matrix **n** which has 3 rows and 4 columns of 11–22:

m				n			
1	2	3	4	11	12	13	14
5	6	7	8	15	16	17	18
9	10	11	12	19	20	21	22

A matrix can be considered as a single item and, as such, be manipulated arithmetically with a single scalar number. The scalar number, when applied to a matrix, is appended to each of the individual items within the matrix. For example,

10×m				n+10			
10	20	30	40	1	2	3	4
50	60	70	80	5	6	7	8
90	100	110	120	9	10	11	12

Two matrices are *identical* if they have the same shape with identical elements in matching locations.

Matrices themselves can be arithmetically manipulated too, provided they are the same shape, each having the same number of rows and columns. Thus,

m+n			
12	14	16	18
20	22	24	26
28	30	32	34

In this addition, as well with other arithmetic operations, the adding is between the corresponding items within the two matrices. For example, 1 in **m** is added to 11 in **n** and corresponding to 12 added to 22.

Since often it is necessary to address individual items within a matrix, each item within the matrix is associated with a pair of integers. This pair of integers define the row and column in that order for each item in the matrix.

In \mathbf{m} , for example, 1 1 is associated with 1, 2 3 with 7, and 3 2 with 10. These pairs of integers are often called the *subscripts* or *indices* of the matrix.

Here are the set of subscripts for both \mathbf{m} and \mathbf{n} ,

```
1 1 1 2 1 3 1 4
2 1 2 2 2 3 2 4
3 1 3 2 3 3 3 4
```

Beside addressing individual rows or columns within a matrix other actions may be needed. The *diagonal* of a matrix, are those items whose subscripts are the same. For the matrix \mathbf{n} , the diagonal is 11 16 21. This diagonal is called the *main diagonal*. When this diagonal is all ones and all the other items are zero, the matrix is called the *Identity matrix*. A 4 by 4 identity matrix is

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

A transposed matrix is one where the rows become columns and the columns become rows. The *transpose* of matrix \mathbf{m} is

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

This can be viewed as flipping of the matrix along the main diagonal.

When a transposed square matrix is identical to the original matrix, the matrix is called a *symmetric matrix*.

\mathbf{s}	<i>transposed \mathbf{s}</i>
2 -6 8	2 -6 8
-6 7 5	-6 7 5
8 5 3	8 5 3

On occasion, the rows could be *reversed*, for \mathbf{m} :

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

Or by the columns:

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

Most of the manipulation of matrices follow closely the manipulation of single numbers.

However, there is one very important manipulation of a pair of matrices called *Matrix Multiplication*. It is different than just multiplying the matrices element by element. (Note in APL this will be introduced in the section on operators as *Inner Product*.)

In mathematical matrix multiplication both multiplication and addition are applied in that order. More detail will be presented in the presentation of *Inner Product* since other functions may be used as well.

With this brief introduction on matrices, but let us now see how to create them, what effect they have on the APL operations, and how to put them to use.

Reshape $\mathbf{V\rho R}$

While matrices can often arise through calculations, the dyadic use of *Reshape* (*Ctrl+R*) can create a matrix. The dyadic form $\mathbf{V\rho R}$ is called *Reshape*, where V may be a vector of non-negative integers or a scalar and R is the data to be reshaped. Consider:

```

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

```

It builds a matrix with 3 rows and 4 columns from the vector of integers 1 to 12. The first item, 3, in the vector left argument \mathbf{V} defines the number of rows and the second item, 4, defines the number of columns in the matrix. Using a three item vector left argument builds a three-dimensional array where the new dimension is the leftmost. As the number of items in the left argument increases so does the number of dimensions in an array. *Reshape* always builds an array starting row by row.

In a rank 3 array, we often speak about the data having planes containing rows and columns.

In creating an array, *Reshape* uses as many data elements of the right argument as it needs. If there are too many it ignores the rest and if there are not enough it cycles through them again.

The function *Shape of*, $\rho\mathbf{R}$, returns the dimensions of the array \mathbf{R} . As the dimension of an array increases, the new dimension values appear first in \mathbf{V} . Consider \mathbf{A} :

```

      ρA
3 2 4
Where A was defined as
      A←3 2 4⍳a
      ABCD
      EFGH

      IJKL
      MNOP

      QRST
      UVWX

```

In all cases, *Shape of* returns the left argument of *Reshape*.

The term *Rank* defines the number of dimensions an array has. Thus:

$\rho\mathbf{R}$ returns the value of each dimension of an array,

$\rho\rho\mathbf{R}$ returns the *Rank* of an array.

Tally $\neq R$

The function *Tally $\neq R$* (*Ctrl+Shift+)* returns the value of the first dimension of an array. It always returns as a simple scalar. This in contrast where *Shape of* a vector returns its value as a one element vector. *Shape of* always returns a vector.

```
For    A←3 2 4ρ124
      ρA
3 2 4
      ≠A
3
```

Tally $\neq R$ is more convenient to use than writing an expression to extract the first value of ρR as well as being a simple scalar.

```
      ≠2 -5 9 12
4
      ≠6
1
      ρ≠6
      ρρ≠6
0
```

Exercise 42

- a) `3 2p'Hi '`
- b) `4p5`
- c) `3p-12`
- d) `3 3p1 0 0 0`
- e) `3p'Yes or No '`
- f) `2p'OK '`
- g) `2 4p'OK '`
- h) `4 2p'OK '`
- i) `≠ 4 2p'OK '`
- j) `3p'ABCD '`
- k) `2 3p'ABCD '`
- l) `≠ 2 3p'ABCD '`
- m) `2 10p'APL is Fun '`
- n) Create an expression that would generate your name three times row-wise as a matrix.
- o) Create a *Dfn* that would accept a name and create it *n* times row-wise in a matrix.
- p) Using the Reshape function build the following picture:

$$\begin{array}{c} \text{N} \\ \uparrow \\ \text{W} \leftarrow \text{O} \rightarrow \text{E} \\ \downarrow \\ \text{S} \end{array}$$
- q) Write an expression that will generate a 10 by 10 matrix containing a random set of zeros, positive and negative integers less than or equal to 10.

Array Transposition

Rearranging the data orientation of a whole array can be useful. Three monadic functions, closely related, capable of rearranging an array are:

ϕR (*Ctrl+Shift+5*) Reverse
 $\ominus R$ (*Ctrl+Shift+7*) Reverse First
 $\oslash R$ (*Ctrl+Shift+6*) Transpose

Even their names describe what these functions do:

Reverse ϕR

The *Reverse* ϕR function reverses the elements of an array along the last dimension of the array. Shape, Rank, and Depth remain unchanged.

```

V←'ABCDEF'

⊢M←3 4ρ⊞A
ABCD
EFGH
IJKL

⊢A←2 3 4ρ⊞A
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

ϕV
FEDCBA

ϕM
DCBA
HGFE
LKJI

ϕA
DCBA
HGFE
LKJI

PONM
TSRQ
XWVU

```

Reverse First $\ominus R$

The *Reverse First* $\ominus R$ function reverses the elements of an array along the first dimension of the array. Shape, Rank, and Depth remain unchanged.

```

V←'ABCDEF'
⊢M←3 4ρ⊂A

ABCD
EFGH
IJKL

⊢A←2 3 4ρ⊂A

ABCD
EFGH
IJKL

MNOP
QRST
UVWX

⊢V
FEDCBA
⊢M
IJKL
EFGH
ABCD

⊢A
MNOP
QRST
UVWX

ABCD
EFGH
IJKL

```

Transpose ΦR

Transpose ΦR rearranges the arrays such that the shape of the original array R is reversed. The rank and depth do not change.

$V \leftarrow 'ABCDEF'$

$V \ (\Phi V) (\rho V) (\rho \Phi V)$

ABCDEF	ABCDEF	6	6
--------	--------	---	---

$M \ (\Phi M) (\rho M) (\rho \Phi M)$

ABCD	AEI	3 4	4 3
EFGH	BFJ		
IJKL	CGK		
	DHL		

$A \ (\Phi A) (\rho A) (\rho \Phi A)$

ABCD	AM	2 3 4	4 3 2
EFGH	EQ		
IJKL	IU		
MNOP	BN		
QRST	FR		
UVWX	JV		
	CO		
	GS		
	KW		
	DP		
	HT		
	LX		

Illustration: Depreciation

There are several methods of depreciating an asset. One of them is called "Sum-of-the-year's digits". This method allows for an accelerated depreciation. For example, if the asset had a five year life, the base of the depreciation would be $15 = 1 + 2 + 3 + 4 + 5$ and the yearly depreciations would be $\frac{5}{15}, \frac{4}{15}, \frac{3}{15}, \frac{2}{15}, \frac{1}{15}$.

What would the sum-of-the-year's-digits depreciation over 5 years on \$15000 be?

```
      5 SOYRS 15000
5000 4000 3000 2000 1000
```

```
      SOYRS←{⍵: investment to be depreciated
[1]  ⍵α: no. of depreciated periods- pos. integer
[2]  ⍵ The Depreciation values over α periods
[3]  ⍵×(⊖1α)÷+/1α
[4]  }
```

Exercise 43

With the following variables evaluate the following expressions.

```
v←'abcdefg'
m←3 4⍴'abcdABCD'
```

a) Apply ϕ , θ , \boxtimes individually to v and m .

Determine the value of the following expressions:

- b) $\phi\boxtimes m$
- c) $\boxtimes\phi m$
- d) $\theta\boxtimes m$
- e) $\boxtimes\theta m$
- f) $\phi\theta m$
- g) $\theta\phi m$
- h) $\theta\boxtimes\phi m$

Now with $m←2\ 3\ 4⍴124$, repeat exercises a) through h) again

In addition to the monadic functions of Reverse, Reverse First, and Transpose, they each have dyadic forms.

$L \ \phi \ R$ Rotate
 $L \ \Theta \ R$ Rotate First
 $L \ \bowtie \ R$ Dyadic Transpose

For *Rotate* and *Rotate First*, the left argument L is an integer, either positive or negative.

For *Rotate*, the change is along the last dimension of the array and

For *Rotate First*, the change is along the first dimension of the array.

If L is a positive integer, it can be viewed as being subtracted from the value of the proper subscript. (For *Rotate*, it is the last subscripts and for *Rotate First*, it is the first subscripts.)

For *Rotate*:

If it is positive, it produces a rotation to the left.

If it is negative, it produces a rotation to the right.

With:

$v \leftarrow \iota 6$

$m \leftarrow 4 \ 5 \rho \iota 20$

$v \ (1\phi v) \ (-1\phi v)$

1	2	3	4	5	6	2	3	4	5	6	1	6	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$m \ (1\phi m) \ (-1\phi m)$

1	2	3	4	5	2	3	4	5	1	5	1	2	3	4
6	7	8	9	10	7	8	9	10	6	10	6	7	8	9
11	12	13	14	15	12	13	14	15	11	15	11	12	13	14
16	17	18	19	20	17	18	19	20	16	20	16	17	18	19

$m \ (1\Theta m) \ (-1\Theta m)$

1	2	3	4	5	6	7	8	9	10	16	17	18	19	20
6	7	8	9	10	11	12	13	14	15	1	2	3	4	5
11	12	13	14	15	16	17	18	19	20	6	7	8	9	10
16	17	18	19	20	1	2	3	4	5	11	12	13	14	15

The *Dyadic Transpose* comes into play with rank three or higher arrays and will not be covered in this text. There is one useful use for dyadic transpose with matrices. If L is a two-element vector of identical positive integers, $L \bowtie R$ will produce the main diagonal.

$m \ (1 \ 1\bowtie m)$

1	2	3	4	5	1	7	13	19
6	7	8	9	10				
11	12	13	14	15				
16	17	18	19	20				

Exercise 44

Evaluate the following expressions, for $m \leftarrow 4 \ 4\rho 16$

- a) $2\phi 16$
- b) $-3\phi 16$
- c) $\bar{3}\phi 6$
- d) $10\phi 16$
- e) $\bar{10}\phi 16$
- f) $2\Theta 16$
- g) $1\phi\phi 16$
- h) $2\phi m$
- i) $2\Theta m$
- j) $\bar{2}\Theta m$
- k) $1\phi m$
- l) $1 \ 2 \ 3 \ 1\phi m$
- m) $1 \ 2 \ 3 \ 1\Theta m$
- n) $\bar{1} \ \bar{2} \ \bar{3} \ 1\phi m$
- o) $\bar{1} \ \bar{2} \ \bar{3} \ 1\Theta m$
- p) Create a vector of length $n > 10$ that would consist of a series of alternating 1 and 2's where the rightmost value of the vector is always 2.

Summary 4: Matrices

Arrays are fundamental to APL. First, a formal definition of an array:

An *Array* is a function
that maps
a set n of non-negative integer-tuples (indices)
to data items.
“ n ” is the rank of the array.
For example, for a vector $n=1$
and
for a matrix $n=2$.¹⁰

Building an array, determining its shape, rearranging the whole array or portions of it are possible. The following functions support these activities.

Functions

Shape of	ρR
Reshape	$V\rho R$
Tally	$\#R$
Reverse	ϕR (<i>Ctrl+Shift+5</i>)
Reverse First	ΘR (<i>Ctrl+Shift+7</i>)
Transpose	ΦR (<i>Ctrl+Shift+6</i>)
Rotate	$L\phi R$
Rotate First	$L\Theta R$
Transpose	$L\Phi R$

Definitions

As more of the power of APL is encountered more special terminology also occurs. These special terms and phrases need to be defined, reviewed, and emphasized.

Scalar Extension means a single argument of a dyadic function may be used with each of the items of its other argument.

Rank defines the number of dimensions of an array.

In later presentations,

Cells of an array define the manners in which an array may be examined and considered.

Major Cells of an array are the sub arrays whose dimensions are derived by dropping the first dimension of the original array.

For a vector, its major cells are its scalar items.

For a matrix, its major cells are its rows.

For a 3-dimensional array, its major cells are its plane

¹⁰ The definition - Roger Hui

Array Selection

In addition to creating matrices, and manipulating them as a single object, there is a great need to select, change, replace elements within an array. Probably the most widely operation to extract elements from an array is called *Indexing*.

Indexing

The next few functions address handling individual elements within an array.

You have been working with lists of data that in APL we call vectors. You have been able to name vectors, for example,

```
Nos←17 250 5.2
Lets←'How now'
Grades←90 85 97 88
```

As the name represents all the data, you are able to manipulate all the data just using its name. For example,

```
3×Nos-50
99 600 165.6
```

However, there comes a time to extract one or more of the data items within an array. Consider the vector **Grades**. Perhaps we would like to extract the largest number, the best grade, **97**, from the vector **Grades**. One process of extracting an item from a vector is called *Indexing*. Implicitly associated with the items in the vector are a set of nonnegative integers called *indices*. These integers describe the position of the items within the vector. Recall Roger Hui's previous definition of an array. Thus, for **Grades**, **1** is associated with **90**, since it is the first number in the vector. Similarly, the integer **2** with **85**, the integer **3** with **97**, and so forth. If you take the integer associated with an item in the vector and place it in square brackets to the right of the vector name, you will select that item from the named vector.

Thus **Grades[3]** yields **97**, the third item within Grades.

Suppose you wish to extract the last two items in Grades. What would you put into the square brackets?

```
Grades [3 4]
97 88
```

You are *not limited* via indexing to ask for just a single item from the vector. In fact, you are able to use any expression or defined function that produces valued indices within the brackets.

```
Grades[ 2 2ρ14]
90 85
97 88
```

Exercise 45

With $V \leftarrow 17 \ 9 \ -5 \ 1.23 \ 0 \ 6$, evaluate the following:

- a) $V[3]$
- b) $V[1 \ 3 \ 5]$
- c) $V[5 \ 3 \ 1]$
- d) $V[\iota 3]$
- e) $V[2 \times \iota 2]$
- f) $V[I \leftarrow 2 \times \iota 2]$
- g) $V[5.0]$
- h) $V[3 \ 1 \ 3]$
- i) $V[1 \ 7 \ 5]$

From these exercises, you should note that

The order of the data in the result derives from the order of the indices.

The Shape of the result of indexing is the shape of the indices.

And,

Any expression or function that generates legal indices may be used as index values.

- j) Given a three item numeric vector, Create a *Dfn* that will produce both values of the quadratic formula.

An array is an ordered set of data from which one may select individual items by specifying their location in the array. At this point, two types of arrays have been presented, vectors and matrices. Consider the vector *V* and the matrix *M*.

```
V ← 'abcdef'
V
abcdef
M ← 3 4ρ'abcdefghijkl'
M
abcd
efgh
ijkl
```

Now their shapes are

```
ρV
6
ρM
3 4
```

and their ranks are

```
ρρV
1
ρρM
2
```

The term *Rank* defines the integer number of dimensions an array has. *Rank* tells us the number of integers that an array needs to select a single item from the array.

A vector requires a single integer, and a matrix requires two integers to select an item. Such integers are called *indices*. With both vectors and matrices these index values are used within square brackets, [] to do the selection. For example,

```
V[3]
c
V[2 5]
be
```

Now select the item “h” from the matrix *M*. As *Rank* tells us, two integers are needed. For example, with the matrix one can speak of the item in the second row and fourth column as *h*. To accomplish this, one must separate the row indices from the column indices. Placing a semicolon between them in the square brackets does this separation. For example, to select the letter “h” from *M*,

```
M[2;4]
h
```

The integers before the semicolon indicate the rows desired and the integers following the semicolon indicate the columns desired. A whole row or column can be selected by omitting integers before or after the semicolon.

For example,

```

      M[2;]
efgh
      M[;4]
dh1
      M[1 3;2 4]
bd
j1

```

In this last expression, the letters states that they lie within the first or third rows *and* the second or third columns.

Exercise 46

Determine the result of the following expressions,

- a) $M[2\ 3;]$
- b) $M[3\ 2;]$
- c) $M[2\ 3;1]$
- d) $M[2\ 3;1\ 4]$
- e) $M[2\ 3;4\ 1\]$
- f) $M[3\ 2\ 1;]$
- g) $M[;2\ 1]$
- h) $M[2\ 3;1\ 1\]$
- i) $M[2\ 3\ 2\ ;1\ 4\ 1]$
- j) $M[;4\ 3\ 2\ 1]$
- k) $M[3\ 2\ 1;4\ 3\ 2\ 1]$
- l) $M[4;]$
- m) Write the expression that will produce the four corner items of the matrix M as a vector.

Selective Assignment

The opposite of selecting items from an array is replacing items within an array.

Consider

```
Grades←90 85 97 88 0 91
```

Suppose the zero grade in `Grades` needs to be replaced by the make-up score *96*. Where does the zero grade appear in the `Grades` vector? Select it.

```
Grades[5]
```

To do a replacement, place this expression which did the selection to the left of assignment and its replacement value to the right of assignment.

```
Grades[5] ←96
```

In order to see your change, you must type `Grades` again.

```
Grades
90 85 97 88 96 91
```

Selective assignment is a two-step process. The first step is to create the expression that selects the date to be replaced. Placing this expression to the left of assignment makes it the replacement expression. The new replacing items are placed to the right of assignment.

To view the replacement the variable must be displayed again.

Exercise 47

For `V←17 9 -5 1.3 0 6`, issue the following expressions and then type `V` after each exercise,

- a) `V[3]←5`
- b) `V[1 4]←8 7`
- c) `V[5]←-12`
- d) `V[2 3 6]←100`
- e) `V`
- f) Given `v←'abcd'`, write the expression to interchange *a* and *c*.
- g) Replace the “O” in “ROY” by “A”.
- h) In the matrix `M←2 3⍲6`:
 - h-1) Replace the first row by zeros.
 - h-2) Replace the number in the second row and second column by 100.
 - h-3) Replace the second column by `-13 -16`

Now, there are other ways of modifying the contents of matrices. Suppose you had a simple numeric matrix *M* such as

```

      M←?5 5ρ20
7  12  6  7  8
2  19  1 14 17
13 14  2 16 18
10 17  4  7 13
1  11 19 10  2
    
```

And you wished to replace all numbers less than 10 by zero. Thus

```

      M×M≥10
0 12  0  0  0
0 19  0 14 17
13 14  0 16 18
10 17  0  0 13
0 11 19 10  0
    
```

Also as in Selective Assignment the change is not complete until the result is reassigned to the original matrix. Remember the only way a variable name changes what it represents is by reassignment.

```

      M←M×M≥10
    
```

Exercise 48

For exercises a-c do not assign your solution. Given a simple matrix such as *M* above;

- a) Add 102 to all the numbers in *M* less than 10.
- b) Change all the numbers in *M* less than 10 to 100.
- c) Change all the numbers in *M* less than 10 to their negative values.
- d) Create *Dfn*s that will replace all numbers greater than a specific number by a given number.
 - d-1) Let the *Dfn* assign the resulting solution to the original array.
 - d-2) Let the *Dfn* not assign the resulting solution to the original array.

Take $L \uparrow R$ and Drop $L \downarrow R$

The previous material discussed how one would extract data from an array via indexing. Often there is a need to take or remove elements from either end of an array. That is the role of the dyadic functions *Take* and *Drop*.

For *Take*, $L \uparrow R$ (*Ctrl+Y*), L must be a simple integer or integer vector. R may be any array. L defines the amount of data taken from R . If L is positive, L items are taken from the beginning of R . If L is negative, $|L|$ items are taken from the end of R . The length of L must be less than or equal to the rank of R .

Informally, for a vector left argument in taking from a matrix or higher rank array, it can be read as an “and” situation.

	$v \leftarrow 'abcdef'$	
	$2 \uparrow v$	
ab		
	$\bar{3} \uparrow v$	
def		
	$\leftarrow m \leftarrow 4 \quad 4 \rho \square a$	
ABCD		
EFGH		
IJKL		
MNOP		
	$\bar{1} \uparrow m$	
MNOP		
	$\bar{1} \quad 3 \uparrow m$	Take the elements in the last row <i>and</i> the first three columns
MNO		
	$2 \uparrow m$	Take the first two rows
ABCD		
EFGH		

The function *Drop* $L \downarrow R$ (*Ctrl+U*) as the name implies, causes data to be removed from an array. If L is positive, L items are dropped from the beginning of R . If L is negative, $|L|$ items are dropped from the end of R . The length of the left argument L must be less than or equal to the rank of R .

	$2 \downarrow v$	Drop the first two elements in v .
cdef		
	$\bar{3} \downarrow m$	Drop the last 3 rows in m .
ABCD		
	$2 \quad \bar{1} \downarrow m$	Drop the elements in either the first two rows OR the last column.
IJK		
MNO		

Illustration: A Range of Equal Distant Numbers

Given that you wish to have a set of numbers they are the same distant apart is straight forward. For example, to create five numbers ten units apart beginning with the number 13 is

$$3 + 10 \times 15$$

13 23 33 43 53

However, suppose , a first number and a second number are given in which you wish to insert four numbers each equidistant apart between them.

Here is a function that will do that.

```

▽ RANGE0←{Rω:2 numbers
[1]  Rα:no. of numbers between ω
[2]      d←(¬1↑ω)-1↑ω
[3]      s←(1↑ω)+(d÷α+1)×ια
[4]      (1↑ω),s,¬1↑ω
[5]  }
```

```

4 RANGE0 5 21
5 8.2 11.4 14.6 17.8 21
```

```

4 RANGE0 ¬5 ¬20
¬5 ¬8 ¬11 ¬14 ¬17 ¬20
```

Exercise 49

For $v \leftarrow 'tapes'$, determine the result of the following expressions:

- a) $4 \uparrow v$
- b) $\neg 4 \uparrow v$
- c) $\neg 2 \downarrow v$
- d) $1 \uparrow v$
- e) $3 \uparrow 4 \uparrow v$
- f) Find, just using \uparrow and \downarrow , several different ways of extracting 'ape' from v . There are at least seven.
- g) $5 \uparrow \iota 3$
- h) $\neg 5 \uparrow \iota 3$

For $m \leftarrow 3 \ 4 \rho \square a$

ABCD
EFGH
IJKL

- i) $2 \ 2 \uparrow m$
- j) $2 \ 2 \downarrow m$
- k) $\rho 2 \ 2 \downarrow m$
- l) $2 \uparrow m$
- m) $\rho 2 \uparrow m$
- n) $2 \downarrow m$
- o) $\rho 2 \downarrow m$
- p) $\neg 1 \ \neg 2 \uparrow m$
- q) $\rho \neg 1 \ \neg 2 \uparrow m$
- r) $\neg 1 \ \neg 2 \downarrow m$
- s) $\rho \neg 1 \ \neg 2 \downarrow m$
- t) $1 \uparrow 1 \downarrow m$
- u) $\rho 1 \uparrow 1 \downarrow m$
- v) Determine just using \uparrow and \downarrow several different ways of extracting 'EFGH' from m .
- w) The following *Dfns* have no comments. Fill in comments and state what they do. Assume ω is a numeric vector.
 - w-1) $f1 \leftarrow \{0 = + / \neg 4 \uparrow \omega\}$
 - w-2) $f2 \leftarrow \{0 = \div \times / \neg 4 \uparrow \omega\}$
- x) Create a *Dfn* that will produce the absolute distance between adjacent numbers in a simple numeric vector.
- y) Create a *Dfn* to indicate the sums of the distances between adjacent numbers that are in a positive direction and in a negative direction.

Grade Up and Grade Down $\Uparrow R$ $L\Uparrow R$ $\Downarrow R$ $L\Downarrow R$

One of the very frequent actions in computing is to select data from an array in an orderly fashion often called sorting. Two function symbols provide sorting capability. They are *Grade Up*, \Uparrow (*Ctrl+Shift+4*) and *Grade Down*, \Downarrow (*Ctrl+Shift+3*), in both monadic and dyadic form.

Let's look at the *monadic* functions.

- R must be a simple numeric or character array
- *Grade Up*, $\Uparrow R$, produces the *indices* of the arrays in ascending order, and
- *Grade Down*, $\Downarrow R$ produces the *indices* of the arrays in descending order.
- R must be a numeric or character array.

The result of a *Grade Up* is the rearrangement of the indices of R such that when used as indices on R would produce R in ascending order

The result of a *Grade Down* is the rearrangement of the indices of R such that when used as indices on R would produce R in descending order

For example,

```
nos←15 -15 4.2 0 15 0

      ⍷nos
2 4 6 3 1 5

      nos[⍷nos]
-15 0 0 4.2 15 15

      ⍶nos
1 5 3 4 6 2

      nos[⍶nos]
15 15 4.2 0 0 -15
```

For character arrays, the implied *collating sequence* is the numeric order of the Unicode code points, $\square AV$. That order with respect to numbers and the alphabet is:

```
abcdefghijklmnopqrstuvwxyz-0123456789-ΔABCDEFGHIJKLMNQRSTU...
(note this not all of  $\square av$ )
```

```
⍵Nams←'ROY' 'RAY' 'JACK' 'SUE'
JACK  RAY  ROY  SUE

      ⍷Nams
3 2 1 4

      Nams[⍷Nams]
JACK  RAY  ROY  SUE
```

If R is of rank >1 , then the indices along the first dimension are used. (In APL terms $\iota \neq R$.)

$\leftarrow T \leftarrow \uparrow Nams$

ROY
RAY
JACK
SUE

$T[\uparrow T;]$

JACK
RAY
ROY
SUE

One might ask why not have these two function just produce the sorted numbers directly. Getting just the indices is more versatile. Consider the following illustration.

Illustration: Array rearrangement

$CDS \leftarrow 4 \ 2\rho 'Jazz' \ 670 \ 'Classic' \ 970 \ 'Folk' \ 725 \ 'Rock' \ 325$

CDS

Jazz	670
Classic	970
Folk	725
Rock	325

$\psi CDS[;2]$

2 3 1 4

$CDS[\psi CDS[;2];]$

Classic	970
Folk	725
Jazz	670
Rock	325

Illustration: Ordinality of a Sorted Vector

For

```
V ← 2 5 6 2 4
V[ΔV]
5 2 2 4 6
```

Here the ordinality of the numbers of **V** were changed after the Grade Up as follows:

```
For 2: 1 became 2
     5: 2 became 1
     6: 3 became 5
     2: 4 became 3
     4: 6 became 4
```

These new ordinal numbers for **V** can be determined without actually doing the grade up by issuing:

```
ΔΔV
2 1 5 3 4
```

Illustration: Merging Two Vectors

Let

```
A ← '-Yds -Ft '
B ← '3912 '
```

And the goal is

```
39-Yds 12-Ft
```

Determining a binary vector to determine the decided order will provide the merge.

Here

```
V ← 1 1 0 0 0 0 0 1 1 0 0 0
```

i.e., first the 1 1 indicates choose 39 from **B**. Then the 5 zeros indicates choose '-Yds' from **A**. This is followed by 1 1 to choose 12 from **B** and finally the 0 0 0 will choose '-Ft' from **A**.

```
Z ← A, B
  Z[ΔV] ← Z
39-Yds 12-Ft
```

Exercise 50

For `n←46 7 43 12.7 0.5 12.3 25 12.6`

- a) `Δn`
- b) `∇n`
- c) `n[∇n]`
- d) `n[Δ]`

For `W←'NEAT' 'TOO' 'APL' 'TOO'`

- e) `W[ΔW]`
- f) `W[∇W]`

For `W1←'What' 'About' 'lower' 'case'`

- g) `W1[ΔW1]`
- h) `W1[∇W1]`

For `W2←'How' 'about' 'Nos.' '21' '12' 12 12 21.4 21`

- i) `W2[ΔW2]`
- j) `W2[∇W2]`

For `T←↑'JOE' 'ROY' 'RAY' 'SUE'`

- k) `T[ΔT;]`
- l) `T[;ΔT]`

For `T←↑'JOE' 'ROY' 'RAY' 'SUE' 'BETTY'`

- m) `T[ΔT;]`
- n) `T[;ΔT]` ...*Why?*
- o) Given a vector of numbers, for example,
`np←-15+?20ρ30`
- p) State in words what the following expression does.
`n[∇n←(np≤0)/np]`
- q) Write an expression that will extract the positive numbers from `np` rearranging them in ascending order.
- r) Create a *Dfn* that will accept an arbitrary set of vectors of the same length and merge them in an arbitrary defined manner.
- s) What happens when you *Grade up* `v←12 0 21.5 3j4 6` ...*Why?*
- t) Given

`Items←'B12' 'A72' 'D92' 'C58'`

`Amts←35 18 27 48`

Write the expression to generate:

A72	18
B12	35
C58	48
D92	27

Grade Up, $A \uparrow B$, *Grade Down*, $A \downarrow B$ produce an ascending (\uparrow) or descending (\downarrow) set of indices for the character array **B** based on the collating array **A**. The indices are with respect to the first dimension of **B**. For example:

```

27      pa1pa1 ← ' ', ␣a

      ␣a
      ABCDEFGHIJKLMNOPQRSTUVWXYZ

      ␣Nams ← ↑ 'ROY' 'RAY' 'JACK' 'SUE'
      ROY
      RAY
      JACK
      SUE

      Nams[ a1pa1 Ⓢ Nams ; ]
      JACK
      RAY
      ROY
      SUE

```

But $\uparrow a$ applies for the monadic *Grade Up* and *Down*:

```

      Nams[ Ⓢ Nams ; ]
      JACK
      RAY
      ROY
      SUE

```

Let

```

      c ← 'abcdefghijklmnopqrstuvwxyz'
      nos ← '1234567890'
      ␣amix ← ↑ '12Fr' 'cd24' 'T21' 'cd31' 'T6' '7Cr'

      12Fr
      cd24
      T21
      cd31
      T6
      7Cr

      amix[ Ⓢ amix ; ]
      12Fr
      7Cr
      T21
      T6
      cd24
      cd31

```



```

    amix[⊂a⊣amix;]
T21
T6
7Cr
12Fr
cd24
cd31

    amix[(c,nos,⊂a)⊣amix;]
cd24
cd31
12Fr
7Cr
T21
T6

```

Exercise 51

With

```

c←'abcdefghijklmnopqrstuvwxyz'
no←'1234567890'
⊢amix←↑'12Fr' 'cd24' 'T21' 'cd31' 'T6' '7Cr'

12Fr
cd24
T21
cd31
T6
7Cr

```

- a) Evaluate `amix[(c,no)⊣amix;]`
- b) Evaluate `amix[(no,c)⊣amix]`
- c) Write the expression to produce from `amix`:


```

T21
T6
cd24
cd31
12Fr
7Cr

```

Summary 5: Array Selection

Manipulating arrays often involves extracting items from within them or replacing items in them. Here are some of the ways APL allows you to do that:

Functions

Vector Indexing	$V[I]$
Matrix indexing	$M[I;J]$
Take	$L \uparrow R$
Drop	$L \downarrow R$
Grade Up (Monadic)	$\Uparrow R$
Grade Up (Dyadic)	$L \Uparrow R$
Grade Down (Monadic)	$\Downarrow R$
Grade Down (Dyadic)	$L \Downarrow R$
Vector Replacement	$V[I] \leftarrow \dots$
Matrix Replacement	$M[I;J] \leftarrow \dots$

Array Inquiry

When working with arrays, many questions could be asked about an individual array or the relationship between arrays.

Such questions as:

- How many numbers are positive, negative, or zero?
- Are elements of an array unique?
- Are there any negative numbers in the array and where are they?
- Are two arrays identical?
- Where are specific elements located in an array?

There are many additional questions that could be asked.

The following functions will address such questions.

Match $L \equiv R$ and Not Match $L \neq R$

The *Match* function (*Ctrl + Shift + ;*) returns a scalar 1 if the two arrays are identical; otherwise it returns a scalar 0. The *Not Match* function (*Ctrl + Shift + '*) returns a 1 if the two arrays are not identical. They are opposites of each other.

To be identical not only does all the data positionally match but also all of the attributes of the arrays must match; Shape, Depth, Rank.

Two non-empty arrays are identical *if and only if*

1. They have the same structure (Shape and Depth), and
2. They have identical values in all of the corresponding locations (type and value).

Exercise 52

Determine which expressions produce a 1 or 0 and then enter to check;

- a) $5 \equiv (5)$ _____
- b) $5 \neq, 5$ _____
- c) $(5=5) \neq 5 \div 5$ _____
- d) $5=5 \neq 5 \div 5$ _____
- e) $2 \ 4 \ 5 \neq 2 \ 5 \ 4$ _____
- f) $1 \ 2 \ 4 \equiv 1 \ 2, \ 4$ _____
- g) $1 \ 2 \ 4 \equiv 1 \ (2 \ 4)$ _____
- h) $1 \ 2 \ 4 \equiv 1, (2 \ 4)$ _____
- i) $1 \ 2 \ 4 \equiv 1 \ (2, 4)$ _____
- j) $1 \ 2 \ 4 \equiv 1, (2, 4)$ _____
- k) $(1 \ 2) \ 'abc' \neq \ '1 \ 2' \ 'abc'$ _____
- l) $(1 \ 2) \ 'abc' \neq (1 \ 2) \ 'abc'$ _____

Determine which function should be inserted in the following expressions to produce a 1:

- m) $2 \ 4 \ 1 \ 3$ _____ $(2 \ 4) \ (1 \ 3)$
- n) $2 \ 4 \ 1 \ 3$ _____ $(2 \ 4), (1 \ 3)$
- o) $2 \ 4 \ 1 \ 3$ _____ $(2, 4), (1 \ 3)$
- p) $2 \ 4 \ 1 \ 3$ _____ $(2, 4), (1, 3)$
- q) $'abc' \ (1 \ 2)$ _____ $(1 \ 2) \ 'abc'$
- r) $2 \ 4 \ 1 \ 3$ _____ $4\rho 2 \ 4 \ 1 \ 3$
- s) $2 \ 4 \ 1 \ 3$ _____ $5\rho 2 \ 4 \ 1 \ 3$

Index of L ⌈ R

To understand the role of the *Index of*, \lceil (*Ctrl+i*) function, consider a text book. If you wished to find a specific item in it you would turn to the pages in the back of the book called the *index of* the book. There you would find the page numbers of the information you are seeking in the book. This is the same role that the *Index of* function, $V \lceil A$, plays with data. Let us see it in action.

```
V ← 'abcde'
V ⌈ 'debated'
4 5 2 1 6 5 4
```

The integers returned are the index values of 'debated' with respect to the vector left argument v . Now the letter t was not in v . In this case, all of the right argument items not found in the left argument are given the integer one greater than the length of v , i.e., $1 + \rho v$.

The left argument must be a vector.

Also *Index of* will yield the index values of the *first occurrence* of the data in the left argument. Note the values for d in the above example.

Index of is also \lceil io dependent.

Illustration: Grade Point Average

Suppose there is a set of letter grades, A, B, C, D, and F. If their numeric values are 4, 3, 2, 1, and 0 respectively, the GPA function will determine the numeric grade point average:

```
GPA ← {⍵: ⍵ a vec of letter grades from set 'A,B,C,D,F'
[1]  n ← 1 + 'FDCBA' ⌈ ⍵
[2]  (+/n) ÷ ⍵
[3]  }

3      GPA 'C' 'B' 'B' 'A'

3      GPA 'CBBA'
```

Exercise 53

- a) 'abcd' ı 'bad'
- b) 'abcd' ı 'avenue'
- c) 'how now' ı 'oh no! wow'
- d) 'abcdef' ı 'abcdef'
- e) 'edcba' ı 'abcde'
- f) 'aabbcc' ı 'abc'

Here is another useful system variable. It is the capital alphabet `A`.

```

A
ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

- g) Using the upper case alphabet vector, what single number can represent your first name? (e.g. 'RAY' is 44).

Illustration: Simple Encoding

Between *Index of* and *Indexing* one can develop a simple way to encode words into numbers and numbers into letters.

Thus,

```
alpha←'abcdefghijklmnopqrstuvwxyz '
pw←'apl is fun'
⍒numbers←alpha ⍒ pw
1 16 12 27 9 19 27 6 21 14
alpha[numbers]
apl is fun
```

Exercise 54

- Using the above illustration, what is your name as a number vector?
- Can you derive a single number from this number vector?
- Create a *Dfn* that will accept an alphabetic phrase and a proper alphabet returning the numeric vector equivalent of the phrase. Try it on '*Apl is Great 2!*'

Note if you wish to include capital letters, the system variable `⍳A` represents the capital letters and `⍳D` the integers.

Illustration: DNA to RNA

A strand of DNA consists of a long sequence of the four nucleotides: adenine, cytosine, guanine, and thymine, abbreviated as A, C, G, and T. Organisms use DNA to construct a complementary structure called RNA to create proteins. The transcription of RNA from DNA is via the following substitutions:

A is replaced by U

C is replaced by G

G is replaced by C

T is replaced by A

Thus

```
v←'AACGTTGAGC'
```

```
DNAtoRNA v
UUGCAACUCG
```

A solution:

```
DNAtoRNA←{n ω: a DNA char vector
[1]  n Result: corresponding RNA vector
[2]  'UGCA'['ACGT'ω]
[3]  }
```


Where $\underline{1}R$ ¹¹

The monadic function *Where*, $\underline{1}R$, (*Ctrl+Shift+I*), must take as its right argument *R* a *simple binary array*. *Where* produces the indices of all the 1's in *R*. If *R* is all zeros, the result is an empty vector¹¹.

```

      1 1 0 1 1 0 1
    1 3 4 6

```

The *Where* function creates a bridge between an array of binary numbers and index values that can be used to extract the values associated with the ones.

Illustration

Find the locations of the letter 'e' in *'Fourscore and seven years ago'*.

```

      1 'e'='Fourscore and seven years ago'
    9 16 18 22

```

Exercise 55

- a) For the expression `a2←?3 4p15`, what will the expression `$\underline{1}2|a2$` produce?

Let `v←15?50`

- b) Write an expression to find the locations of the numbers greater than 25 in *v*.
- c) Write an expression to select out the numbers greater than 25 in *v*.
- d) Consider the following *Dfn*:

```

zeroby←{Rω: simple numeric array
[1]  Rα: the zero replacement
[2]  Rresult all 0s replaced by α
[3]  z←ω
[4]  z[ $\underline{1}\omega=0$ ]←α
[5]  z}

```

Modify this *Dfn* so that it can replace any item in *ω* by another item.

¹¹ The concept of an object that has data but no shape is possible. For example a simple scalar has no shape. Thus when you ask for its shape you see an empty vector, nothing. For example, type `p12`.

Unique $\cup R$

Much of data manipulation involves in searching. In some situations, the goal is to determine unique items in the array. The function *Unique*, $\cup R$ (*Ctrl+Shift+V*), provides this ability. *Unique* provides the unique items of the major cells of its argument in the shape of its right argument. That is, it returns the unique scalar items of a vector as a vector and the unique rows of a matrix as a matrix.

```

      ⍵ 'Mississippi'
Misp
      ⍵ 5 4 ⍵ 'abcdABCDabcdABCDabcc'
abcd
ABCD
Abcc

```

Exercise 56

- \cup 'onomatopoeia'
- \cup 2 2 2.1 2 3 2 3
- \cup 'abc' 'defg' 'abc' 'def'
- \cup 'abc' 'def' 'abc' 'def'
- \cup 4 3 ⍵ 'abc' 'def' 'abc' 'def'
- \cup 4 3 ⍵ 'abc' 'def' 'abc' 'def'
- \cup 3 2 3 ⍵ 6
- With $\leftarrow m \leftarrow 6$ 3 ⍵ 4, issue $\cup m$

⚠ Watch out for spaces— they count, too.

Union $L \cup R$

In *Unique*, the goal was to find the unique elements within an array. *Union* permits the combination of two vectors without duplicates. The arguments of both **L** and **R** must be vectors or scalars. If a scalar is used, it is treated as a one-element vector. The result is built by adding to **L** all the non-duplicate elements of **R**.

```

      2 4 6 ∪ 8 6 5 4
2 4 6 8 5
      8 6 5 4 ∪ 2 4 6
8 6 5 4 2

```

Exercise 57

- a) 'abc' ∪ 'f dcba'
- b) 'abc' ∪ 'c'
- c) 'abc' ∪ 'abc' 'ab' 'c'

Unique Mask ≠R

Should the locations of the unique items in an array be needed, the *Unique Mask*, $\neq R$ (*Ctrl*+8), can be an aid. It returns a Boolean vector indicating the first occurrence by a 1 of the items in the array.

```

      ≠ 'mississippi'
1 1 1 0 0 0 0 0 1 0 0

      m←4 4p'abcdefgh'
abcd
efgh
abcd
efgh

```

```

      ≠ m
1 1 0 0

```

To find the locations of the unique items in an array, *Unique Mask* in conjunction with *Where* can provide that information.

```

      u≠ 'mississippi'
1 2 3 9

      'mississippi'[u≠ 'mississippi']
misp

```

Exercise 58

- \neq 'onomatopoeia'
- \neq 2 2 2.1 2 3 3 2
- \neq 'abc' 'defg' 'abc' 'def'
- \neq 'abc' 'def' 'abc' 'def'
- \neq 'abc' 'def' 'abc' 'def'
- With $m \leftarrow 6$ $3p \leftarrow 4$, determine $\neq m$
- Create a *Dfn*, given two vectors, that would list the duplicate elements between them.
- Create a *Dfn* that will extract the unique elements from vectors or matrices and provide their locations.
(One suggestion using bracket indexing would be to design separate *Dfns* and switch between them.)

Find $L \in R$

Besides looking for simple scalar items within an array, finding a sequence of data items within an array could be desired. The function Find, $L \in R$ (*Ctrl+Shift+E*), searches for all collective occurrences of L in R . It returns a binary array in the shape of R with a 1 where L begins in R .

```

sb←'he said I said she said'

'said'∈sb
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0

⌈m24←2 4 ρ'abcde'
abcd
eabc

'abc'∈m24
1 0 0 0
0 1 0 0

```

Exercise 59

For $sd1 \leftarrow 'He said, I said'$

- a) $'said' \in sd1$
- b) $'said,' \in sd1$
- c) $'said ' \in sd1$

For $\uparrow sd2 \leftarrow 3 \ 12 \rho 'Right now?--Yes now now!OK it''s now.'$

```

Right now?--
Yes now now!
OK it's now.'

```

- d) $'now' \in sd2$
- e) $'now ' \in sd2$
- f) $'now?' \in sd2$

Membership $L \in R$

Membership $L \in R$ (*Ctrl+e*) produces a binary array the shape of L indicating which elements of L are members of the array R .

```

      ⌊ A←3 4p7 5 2 4 9 8
7 5 2 4
9 8 7 5
2 4 9 8
      ⌊ B←2 2p2 4 6 8
2 4
6 8
      (A∈B) (B∈A)
0 0 1 1 1 1
0 1 0 0 0 1
1 1 0 1

```

Exercise 60

Predict the result of each expression before executing it.

- a) $'b' \in 'a' \text{ 'bc'}$
- b) $'b' \in 'a', 'bc'$
- c) $'b' \in 'ab' \text{ 'c'}$
- d) $'b' \in \in 'ab' \text{ 'c'}$
- e) $'b' \in \in \in 'ab' \text{ 'c'}$
- f) $'bc' \in 'b'$
- g) $'bc' \in 'b' \text{ 'ac'}$
- h) $'bc' \in 'b', 'ac'$
- i) $'bc' \in 'bc' \text{ 'ac'}$
- j) $(\in 'bc') \in 'bc' \text{ 'ac'}$
- k) $'bc' \in 'bc', 'ac'$
- l) $(\in 'b') \in 'b'$
- m) *A Selfie*: The Cardinality of a finite set of integers is simply the number of elements in the set. The definition of a *Selfie* set of integers is defined to be a set which has its own cardinality as a member of the set.

Write a *Dfn*, given a finite set of integers, will determine if the set of integers is a *Selfie* set.

```

      ⌊ x←8 14 4 10 13
8 14 4 10 13
      ⌊ y←10 14 12 5 7
Selfie x
0
      ⌊ Selfie y
1

```

Interval Index $L \underline{I} R$

There are situations where given data it needs to be distributed within separate groupings. For example, assume a set of pound weights that are to be divided into five different groups: *very small* (less than 100 pounds), *small* (less than 150 pounds), *large* (less than 250 pounds), and *very large* (greater than 250 pounds). The dyadic *Interval Index* function enables this type of data subdivision.

The *Interval Index* function, $L \underline{I} R$ (*Ctrl+Shift+i*), takes a set of ordered scalar intervals as the left argument, L , in *ascending order* and returns the set of indices determining into which interval the right argument data R items lie.

L is an *ordered* non-scalar array representing a set of intervals. Consider:

```

      L←10 25 30 50
and
      R←12 26 9 30 61 37

```

The left argument L defines five intervals numbered¹² from 0 to 4, where *Interval Index* returns:

```

0 for all numbers where  $R < 10$ 
1 for all numbers in the half open interval  $[10 \ 25]$ , i.e., including 10 and excluding 25,
2 for all numbers in the half open interval  $[25 \ 30]$ ,
3 for all numbers in the half open interval  $[30 \ 50]$ ,
4 for all numbers  $R \geq 50$ .

```

Thus,

```

      L I R
1 2 0 3 4 3

```

Interval Index is origin dependent. Thus,

```

      ⍶IO←0
      L I R
0 1 -1 2 3 2

```

Also, the left argument, L , may be an alphabetic character array sorted in ascending order. The blank character comes before *a* or *A* in the English alphabet.

For example,

```

      'aeiou' I 'dyalog'
1 5 1 3 4 2

      'eauiou' I 'dyalog'
DOMAIN ERROR: Left argument must be sorted ascending
      'eauiou' I 'dyalog'
      ^

      'aeiou' I 'dyalog'
1 5 1 3 4 2

```

¹² $\square IO$ is an *implicit argument* of *Interval Index*.

Exercise 61

If the left argument, L, were 'AEIOU', the half-open interval applies as follows:

For 'AEIOU' 1 ...Evaluate the following:

- a) Letters ABCD
- b) Letters EFGH
- c) Letters IJKLM
- d) Letters OPQRST
- e) Letters UWXYZ

Illustration: Grade Ranges

Given a set of numeric inclusive grade ranges. For example:

100–95: A
 94–84: B
 83–70: C
 69–60: D
 59–0: F

A function that will accept a vector of Grades and an numeric interval vector that defines the letter grade ranges is:

```
GradeRanges←{A ω: vector Of numeric grades
[1] A α: an Interval Index vec
[2] Aresult a matrix of the letter grade and its numeric value
[3] (2,ρω)ρω,'FDCBA'[1+α1ω] }
```

For the above set;

```
ix←60 70 84 95

Grades←98 94 93 84 83 70 69 60 55

ix GradeRanges Grades
98 94 93 84 83 70 69 60 55
A B B B C C D D F
```


Exercise 62

If the left argument, *L*, were 'AEIOU', the half-open interval applies as follows.

For 'AEIOU' 1 ...Evaluate the following:

- a) Letters ABCD
- b) Letters EFGH
- c) Letters IJKLM
- d) Letters OPQRST
- e) Letters UWXYZ

Determine value of the following expressions.

- f) `0 20 50 101 1 10 5 50 33 20 100 101`
- g) `10 5 0 5 1 1 6 6 12 12`
- h) `10 5 0 5 10 50 1 10 12 50 55 4 10 12`
- i) Create the Interval Index for this set of grade ranges;
 100–93: A where the values are inclusive
 92–84: B
 83–75: C
 74–60: D
 59–0: F

Check with `Grades←100 93 92 85 84 83 75 74 60 55` using the *Dfn* in the illustration on the previous page.

- j) `'aeiou' 1 'diallog'`
- k) `'aeiou' 1 ' diallog '`
- l) `'aeiou' 1 ' diallog '`
- m) `'aeiou' 1 ' diallog!'`
- n) `'st' 1 'mississippi'`
- o) `'stpq' 1 'mississippi'`
- p) `'abcd' 1 'bill' 'anne' 'bob' 'frank' 'ann' 'david'`

Summary 6: Array Inquiry

There are many aspects of exploring the nature of arrays in terms of the data within them. Working with array data often involves inquiries and comparisons. The following functions, even by name, provide such support.

Functions

Match	$L \equiv R$
Not Match	$L \neq R$
Index of	$L \iota R$
Where	$\underline{\iota} R$
Index Generator	ιR
Unique	$\cup R$
Union	$L \cup R$
Unique Mask	$\neq R$
Find	$L \underline{\epsilon} R$
Membership	$L \epsilon R$
Interval index	$L \underline{\iota} R$

Now it is time to examine further richness of APL data.

Arrays Revisited

There is much more to say about the data organization of an array. Previously we had built vectors and matrices whose data were either character or numeric scalars. However, any array may contain both character and numeric data. Such arrays are called *Heterogeneous Arrays*.

In addition, any element within an array may also be another array. Such arrays are called *Nested Arrays*. If an array does not contain other arrays, it is called a *Simple Array*.

A function that penetrates the structure of an array to execute its simple scalars is called a *pervasive function*. All scalar functions are pervasive.

Heterogeneous nested arrays present great flexibility. Also, however, additional functions are necessary to work with such arrays. Also, parentheses have a new role to play. Previously they were used to change the order of execution. Parentheses also now have a grouping role to play. Let's consider some examples

```

      ⍵v1←1 2 3 10 20
1 2 3 10 20

      pv1
5

      ⍵v2←(1 2 3) 10 20
1 2 3 10 20

      pv2
3

      ⍵v3←(1 2 3) (10 20)
1 2 3 10 20

      pv3
2

      ⍵v4←1 2 3 (10 20)
1 2 3 10 20

      pv4
4

```

Here the parentheses are saying that the data within the parentheses is to be considered to be a single item with respect to the other data with it. For character information, quotation marks can also play the role that parentheses do. Such grouping is not limited to vectors.

```

      ⍵m←2 3⍲6
1 2 3
4 5 6

      'The matrix is ' m
The matrix is 1 2 3
               4 5 6

```

Consider a more realistic vector:

```
Name1←'Roy' 'Jones' 914 555 6289
ρName1
5
```

Here the first item of this five item vector **Name1** is the vector **Roy** and the second is a vector **Jone**. Now the last three numbers independently make up the rest of the five-item vector.

Where the numbers are to be considered together as perhaps a phone number, then they should be parenthesized:

```
Name1←'Roy' 'Jones' (914 555 6289)
ρNname1
3
```

Notice that in creating *Name1* the function *catenation* was not used. Separating *Roy* and *Jone* by a space indicates to *APL* that they are to be considered separate items. Using the *catenation* function joins *Roy* and *Jone* together as a single vector:

```
ρ'Roy' 'Jones'
2

ρ'Roy', 'Jones'
7
```

With the introduction of multidimensional heterogeneous nested arrays, APL must provide additional support. *Shape*, ρA , and *Rank*, $\rho\rho A$, already provide some information, but more needs to be known about the nesting of arrays.

Depth \equiv R

The function *Depth*, \equiv (*Ctrl+shift+;*), provides information as to the nature of nesting. *Depth* returns an integer that defines the maximum depth that APL has to go through the structure of an array to get to a single simple element in the array. The depth of the scalar is zero. This maximum integer will be negative if all of the nested items are not of the same Depth.

For

```
Name1 ← 'Roy' 'Jones' (914 555 6289) 123 45
≡Name1
-2
```

With the advent of the function *Depth*, the complete nature of an APL array can be defined.

All APL data is self describing.

Any established array has associated with it:

Data: derived from:

- a character set
- the number set (including complex numbers)

Type: considered as:

- numeric, character, or heterogeneous

Structure: consisting of

Shape: the array dimensions

Rank: the number of dimensions

Depth: the nature of nesting

Exercise 63

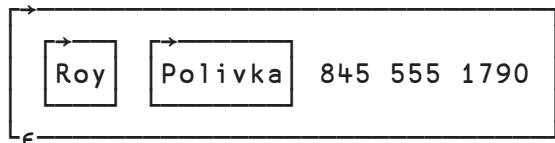
- a) Create a *Dfn* called **SRD**¹³ that will name and display the *shape*, the *rank*, and the *depth* of an APL expression.
- b) Evaluate the following expressions for *shape*, *rank*, and *depth*; (or use the function just created. Note that it will also be used in further problems.)
 - b-1) 'APL'
 - b-2) 'APL' 2
 - b-3) 'APL' 2 3
 - b-4) 'APL',2 3
 - b-5) 'APL' (2 3)
 - b-6) ('APL' 2 3) 'abcd'
 - b-7) ('APL' (2 3)) 'abcd'
 - b-8) 2 2ρ'abcd'
 - b-9) (2 2ρ'abcd') 'abcd'
- c) Establish the situation such that either **SRD** or **srd** produce the same result.

¹³ The **SRD** *Dfn* will be referenced in oncoming exercises. It should also be helpful in your future APL work.

Display Facilities

The nature of nesting arrays within an array may be difficult to see when displaying the array. However, the APL system provides several visual aids to visualize an array. One of these is the *User Command* `]display`.

```
Name1←'Roy' 'Polivka' 845 555 1790
]display Name1
```



The function *Depth* just introduced returns to you the max depth of the array. You can relate that to the picture that `]display` yields. One way to visually understand *depth* from the `]display` of it, would be to determine the maximum number of horizontal lines you would need to cross to get to a single number or character. In **Name1**, it is *two*.

The user command `]display` also provides a great deal of information pictorially. However, it must be specifically issued for each array to be viewed in this boxed fashion.

A second user command, `]Box`, when enabled, provides the box structure for all nested arrays.

To turn it on, issue:

```
]box on
```

...and to turn it off issue

```
]box off
```

```
]box on
```

```
Name1←'Roy' 'Polivka' 845 555 1790
Name1
```

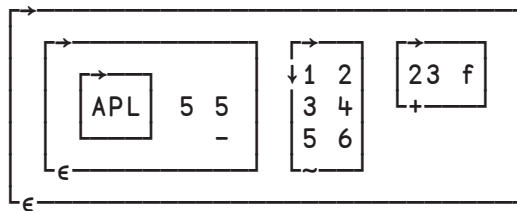
Roy	Polivka	845	555	1790
-----	---------	-----	-----	------

Create

```
n←('APL' 5 5')(3 2⍲6)(23 'f')
```

then issue

```
]display n
```



and examine the symbols on the edges. The following table defines them:

<i>Symbol</i>	<i>Meaning</i>
_	Under a character . . . A scalar character
→	Top edge A vector or higher rank array
↓	Left edge A matrix or higher rank array
~	Lower left Numeric data in the box
_	Lower left Character data in the box
+	Lower left Mixed type data in the box
€	Lower left A nested array
⊖	Top edge An empty vector or higher rank array
⊘	Left edge An empty matrix or higher rank array

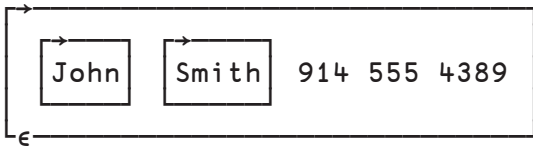
For:

- `pNamea`
- `]display Namea`
- `]display Nameb`
- `]display Names`
- `pNames`
- `≡Names`

Thus,

Nameb				
John	Smith	914	555	4389

```
)pcopy DISPLAY
DISPLAY Nameb
```



Ravel ,R

The function *Ravel*, “,R” applied to its argument, returns a vector regardless of its previous shape. The raveling is done along the last dimension. It reduces the rank of an array but not its depth. The nesting remains but now the nested items are elements of the vector.

```
⍣ a2←2 3⍴'ab' 'cde' 5 10 20
ab cde 5
10 20 ab
,a2
ab cde 5 10 20 ab
```

Exercise 65

Determine the *shape*, *rank*, *depth*, and `⍎isplay` of the following:

- a) `v←1 10`
- b) `v,v`
- c) `v,⊖v`
- d) `(⊖v)(⊖v)`
- e) `(⊖v),⊖v`
- f) `⊖v,v`
- g) `⊖v v`
- h) `(⊖v),v`
- i) `(⊖v) v`
- j) `'ABC' 25 50 100`
- k) `, 'ABC' 25 50 100`
- l) `'ABC'(25 50 100)`
- m) `, 'ABC'(25 50 100)`
- n) `25`
- o) `,25`

With `m←2 5⍴'hello' 19`

- p) `m`
- q) `,m`

Enlist ϵR

When working with data and its structure, changing it into a simple vector could be very convenient. The *Ravel* function $,R$ always returned a vector. However, it did nothing to change the depth of the elements in the vector. That is the role of the function *Enlist*, ϵR (*Ctrl+e*). Not only does it return to its argument as a vector, it returns it as a simple vector. This allows one to get to a *simple vector* regardless of the previous structure.

The enlist function ϵR (*Ctrl+e*) reduces both rank and depth of an array. Each number and character are returned as simple scalars.

For example,

```
Names1 ← 'Sue' 'Jones' 845 555 1790
(ρNames1) (≡Names1)
```

5 2

```
εNames1
```

```
SueJones 845 555 1790
```

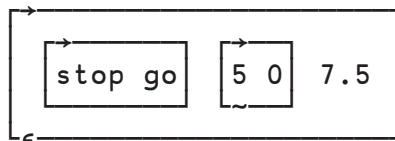
```
(ρεNames1) (≡εNames1)
```

11 1

```
m ← 'stop go' (5 0) 7.5
(ρm) (≡m)
```

3 2

```
DISPLAY m
```



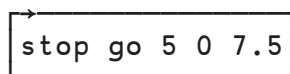
```
εm
```

```
stop go 5 0 7.5
```

```
(ρεm) (≡εm)
```

10 1

```
DISPLAY εm
```



Exercise 66

Given the following variables, determine the *shape*, *rank*, *depth* and **DISPLAY** of each.

- `a ← 'Roy' 'Polivka' 845 555 1790`
- `b ← 'Roy' 'Polivka' (845 555 1790)`
- `c ← 'Roy' 'Polivka', (845 555 1790)`
- `d ← 'Roy', 'Polivka', (845 555 1790)`
- `e ← 'Roy' 'Polivka', 845 555 1790`
- Now take any pair from a)–e) and apply ϵ to each and test to see if they match.

Table ⍤R

Where the *Ravel* function will convert any array into a vector, the *Table* function $\text{⍤}R$ (*Ctrl+shift+*;) will convert an array into a two-dimensional array.

If R is a matrix, *Table* makes no change. If R is a higher rank array, *Table* converts it into a matrix, preserving the first dimension of the array and raveling the array in row major fashion. *Table* is especially useful in converting a vector into a matrix.

```

      ⍵←⍵ 4
1 2 3 4

      ⍵v
1
2
3
4

      ⍵v
4 1

      ⍵a3←3 2 4⍵a
ABCD
EFGH

      IJKL
      MNOP

      QRST
      UVWX

      ⍵a3
ABCDEFGH
IJKLMNOP
QRSTUVWX

```

Illustration: Vectors into a matrix

```

      ⍵v1←3⍵a
ABC

      v2←⍵ 3
      v1,⍵v2
A 1
B 2
C 3

      v2,⍵v1
1 A
2 B
3 C

```

Exercise 67

Determine the result and shape, rank, and depth of each:

- a) $\bar{\gamma}$ 'abcd'
- b) $\bar{\gamma}3 \ 4\rho\Box a$
- c) $\bar{\gamma}3 \ 2 \ 4\rho\Box a$
- d) $\bar{\gamma}2 \ 3 \ 2 \ 4\rho\Box a$
- e) Suppose as an instructor, you have the vectors,

$$\text{students} \leftarrow \text{'Student1' 'Student2' 'Student3'}$$

$$\text{grades} \leftarrow (86 \ 91 \ 88) \ (78 \ 88 \ 0) \ (90 \ 89 \ 93)$$
and you wish to post them as

$$\begin{array}{l} \text{Student1} \quad 86 \ 91 \ 88 \\ \text{Student2} \quad 78 \ 88 \ 0 \\ \text{Student3} \quad 90 \ 89 \ 94 \end{array}$$
Create a *Dfn* to do this.
- f) Suppose that $\bar{\gamma}$ is 'broken' (i.e., unavailable), create an APL expression that would create the same result for two vectors that $\bar{\gamma}$ does.

Catenate L , R

Earlier, the catenate function had been introduced to catenate two vectors together into a single vector. The function *Catenate*, will join together arrays along the *last* dimension. The prior dimensions must match.

```

      n← 3 2 ρ⍳8
1 2
3 4
5 6

```

```

      m←3 4 ρ⍳a
ABCD
EFGH
IJKL

```

```

      m,n
ABCD 1 2
EFGH 3 4
IJKL 5 6

```

```

      m n
┌───┬───┐
│ ABCD │ 1 2 │
│ EFGH │ 3 4 │
│ IJKL │ 5 6 │
└───┴───┘

```

More will be said about catenation after the operator *axis specification*.

Enclose ⌵R

Now, this function extends the important concept of scalar extension. This function is called *Enclose*, ⌵R (*Ctrl+Z*). It encapsulates the array **A**, creating a *scalar* in the eyes of the APL interpreter. Thus permitting scalar extension. Recall

S f A B ↔ (S f A)(S f B) where S is a scalar and A and B are arrays.
A B f S ↔ (A f S)(B f S)

```
(⌵3) +⌵3
2 4 6
```

```
(⌵3) +⌵⌵3
2 3 4 3 4 5 4 5 6
```

Enclosure in an expression increases *depth*.

```
≡1 2+3 4
1
≡1 2+⌵3 4
2
```

It is often useful with the forthcoming operator *Each*.

Illustration: Repeated powers

Given a simple numeric vector, **V**, create a matrix of initial powers of each the values of **V**.

For example,

```
V←.2 4 -5
      4 Rp .2 4 -5
0.2  0.04  0.008  0.0016
4    16    64    256
-5   25   -125   625

▽ Rp←{Aω a simple numeric vec.
[1]      A← number of initial powers
[2]      A result matrix of repeated powers of ω
[3]      ⍳IO←1
[4]      ↑ω*⌵⌵A A for function ↑ see page 123
[5]      }
▽
```

Exercise 68

Create a function **Rp1** that accept a vector of powers to which **V** is to be raised.

Exercise 69

Let

```
a←2 3 π 6
b←2.5
c←'ABCD'
d←a b c
```

Determine the *shape*, *rank*, *depth* and *display* of the following expressions;

- a) `b c`
- b) `b,c`
- c) `d`
- d) `c d`
- e) `c b c`
- f) `c b,c`
- g) `2 2ρ'Hello' 23 ¯59 'OK'`
- h) `,2 2ρ'Hello' 23 ¯59 'OK'`
- i) `ε2 2ρ'Hello' 23 ¯59 'OK'`
- j) `c2 2ρ'Hello' 23 ¯59 'OK'`
- k) `2 4 6+ 5 3`
- l) `2 4 6+ c5 3`
- m) `(c2 4 6)+5 3`
- n) `c2 4 6+c5 3`
- o) Create a 2 by 3 matrix where each item of it is 'APL'.

Exercise 70

Combining two vectors can occur in several ways. In the following determine the *shape*, *rank*, and *depth*.

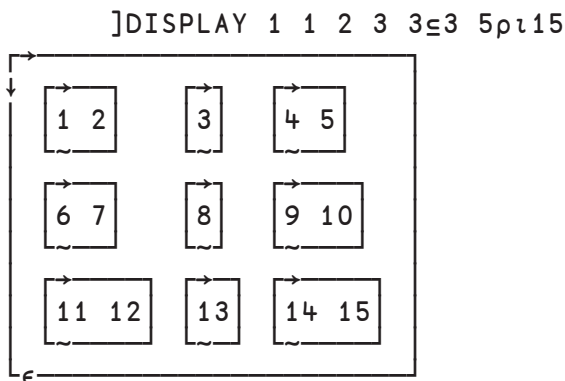
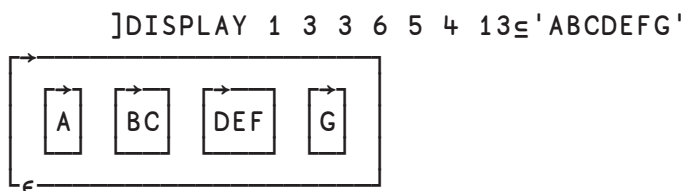
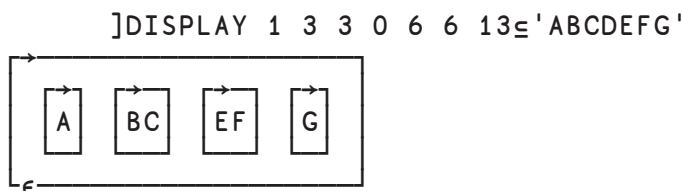
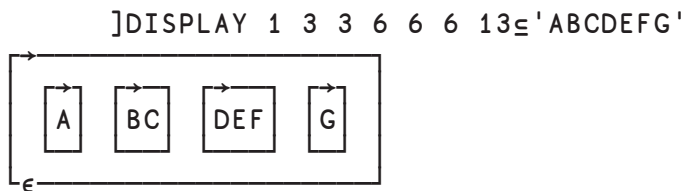
For

- a) `v←1 10`
- b) `v v`
- c) `v,v`
- d) `v,cv`
- e) `v`
- f) `(cv)(cv)`
- g) `(cv),cv`
- h) `cv,v`
- i) `cv v`
- j) `(cv),v`
- k) `(cv) v`

Partition $L \subseteq R$

Partition, $L \subseteq R$ (*Ctrl+Shift+Z*), is another form of enclosing data. Where \mathbf{cR} encloses the complete array, *Partition* enables one to subdivide the enclosures. The left argument is a simple vector of nonnegative integers whose shape matches the last dimension of the array. The values of the integers are immaterial—only the adjacency of the integers determines the partitioning. An enclosing partition ends at the i^{th} location of R when the $i+1^{\text{th}}$ integer of L is greater than the i^{th} integer of L . If a zero appears in L , the matching item of R is omitted and the current partition is closed.

For example,

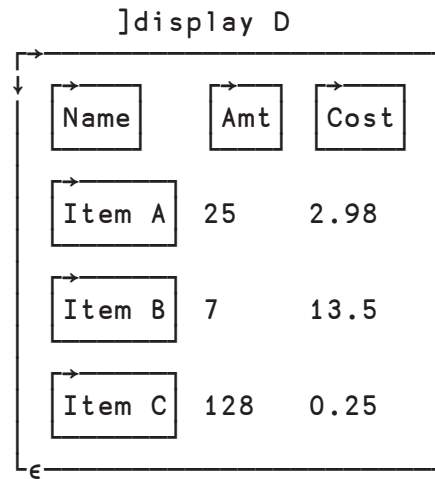


Exercise 71

- Given the vector $v \leftarrow 12 \ 345 \ 67 \ 89$, determine $0 \ 1 \ 0 \ 1 \subseteq v$
- What is the Shape, Rank, and Depth of $0 \ 1 \ 0 \ 1 \subseteq v$?
- Convert $0 \ 1 \ 0 \ 1 \subseteq v$ into a vector of simple scalars.
- Evaluate the expression $(0 \neq 2 | M[1;]) \subseteq M$
- Tracking the Structure of Data:
Given a titled matrix containing numerical data in which you wish to extract the numeric data as a simple numeric matrix. Consider this matrix:

```
⎕← D←⎕ 3⎕ 'Name' 'Amt' 'Cost' 'Item A' 25 2.98 'Item B' 7
13.50 'Item C' 128 0.25
```

Name	Amt	Cost
Item A	25	2.98
Item B	7	13.5
Item C	128	0.25



Using the Shape-Rank-Depth function created earlier,

```
srd D
Shape 4 3 Rank 2 Depth 2
```

Enter the following expressions

```
0 1 1 ⊆ D
srd 0 1 1 ⊆ D
]display 0 1 1 ⊆ D
⌵ 3 1↑ 0 1 1 ⊆ D
```

Here you have just the numbers of D ...but are they in a simple matrix?

```
srd  ^3 1† 0 1 1 ⊆D
```

```
]display  ^3 1† 0 1 1 ⊆D
```

```
3 2ρ ∈ ^3 1† 0 1 1 ⊆D
```

```
srd  3 2ρ ∈ ^3 1† 0 1 1 ⊆D
```

```
Shape  3 2  Rank  2  Depth  1
```

Finally a matrix of simple scalars. There are other ways to achieve the original problem.

The purpose of these exercises is help you to remember to check the Shape, Rank, and Depth of data.

Mix $\uparrow R$ ¹⁴ and Split $\downarrow R$

These two functions *Mix* (*Ctrl+y*) and *Split* (*Ctrl+u*) are paired since their performance on arrays are inter-related. *Mix* increases the rank of an array while *Split* decreases the rank of an array. Let's start with *Mix*.

Enclosing data is very useful in that all the data can be viewed as a scalar. Having enclosed data implies the need to disclose data. This can be done via *Enlist*, ϵ , which removes all depth and returns a simple vector. The monadic function *Mix* offers another alternative, a less dramatic process.

Mix removes only one level of depth, but in so doing increases the rank:

```
⊃y←(1 2 3)(7 8) 'ab'
1 2 3 1 2 ab
```

```
⊃y
1 2 3
7 8 0
A b
```

```
srd y      ⌘ the function you created in the exercise on Depth
Shape: 3 Rank: 1 Depth: 2
```

```
srd ⊃y
Shape: 3 3 Rank: 2 Depth: 1
```

In the case that the individual items in the array do not have the same shape, *Mix* will fill out each item to have same shape with either a zero or a blank depending on the type of the data.

¹⁴ For $\square m < 2$, the symbol for Mix is \uparrow .
For $\square m \geq 2$, the symbol for Mix is \supset .

Illustration

One of the frequent uses of *Mix* is converting a vector of vectors into a simple depth-one matrix:

```
⌈ Stks←'AAPL 172.55' 'MSFT 299.5' 'T 23.94' 'GOOG 2749.75'
AAPL 172.55 MSFT 299.5 T 23.94 GOOG 2749.75
```

```
⌈Stocks←↑Stks
AAPL 172.55
MSFT 299.5
T 23.94
GOOG 2749.75
```

```
srd Stocks
Shape 4 12 Rank 2 Depth 1
```

```
]display Stocks
```

```
→
↓ AAPL 172.55
MSFT 299.5
T 23.94
GOOG 2749.75
```

Exercise 72

With **]Box On**, determine the *Shape*, *Rank*, and *Depth* of the expressions then apply *Mix* to them and redetermine *Shape*, *Rank*, and *Depth*.

- 10 20 30 40
- (10 20)(30 40)
- (10 20 30) 40
- 'abc' (10 20 30)
- 'abc' 10 20 30
- 'abcd' 'ok'
- 2 2ρ'Joe' 27 'Roy' 31
- 'abc' (10 20) 'wxyz'
- What do you get if you applied *Mix* twice to expression h)?
- Write an expression that would align 93 with the other numbers in **Stocks**.

With the *Mix* function the rank of an array was increased. Vectors were changed into matrices. It is the opposite with *Split* (*Ctrl+u*). The rank is decreased. Consider:

```

      m←3 5⍴a
ABCDE
FGHIJ
KLMNO

      ↓m
ABCDE  FGHIJ  KLMNO

      srd ↓m
Shape  3 Rank  1  Depth  2

```

where

```

      srd m
Shape  3 5 Rank  2  Depth  1

```

The process of applying *Split* to the matrix results in a vector of vectors.

```

      a3←2 3 4⍴a
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      ↓a3
ABCD  EFGH  IJKL
MNOP  QRST  UVWX

      srd ↓a3
Shape  2 3 Rank  2  Depth  2

      srd a3
Shape  2 3 4 Rank  3  Depth  1

```

With *Split*, the enclosure is along the last dimension.

Exercise 73

With `]box` on and `m1←3 4p⍵a` and `v1←'ABCD'`, determine the value of the expression and its `srd`.

- a) `m1`
- b) `↑m1`
- c) `↓m1`
- d) `]display ↑↑m1`
- e) `↑↓↓m1`
- f) `m1≡↑↓m1`
- g) `m1≡↓↑m1`
- h) `]display ↓m1`
- i) `v1≡↓v1`
- j) `↓↓v1`

With `v2←'abc' 'defg' 'hi'`

- k) `↓ v2`
- l) `↓↓v2`

Pick V→R

With the advent of nested arrays, the problem of recovering data from a nested array is an issue. Indexing by itself will not work. Consider the variable **Names**:

```
Namea←'Ray' 'Polivka' (845 555 1790)
Nameb←'John' 'Smith' (914 555 4389)
Names←Namea Nameb
Names
```

Ray	Polivka	845 555 1790	John	Smith	914 555 4389
-----	---------	--------------	------	-------	--------------

≡Names

3

The function *Pick V→A* (*Ctrl+x*) provides a path through the structure to a *single item* at a particular level. The left argument are integers that define the path through the levels of structure to an item in the nested array. Suppose we wish to extract Mr. Smith's first name from the variable **Names**. A good way to understand constructing a path is to look at the display of **Names**:

Names

Ray	Polivka	845 555 1790	John	Smith	914 555 4389
-----	---------	--------------	------	-------	--------------

If we open up the top level of structure, we see two vectors. We choose the second vector since it contains information for John Smith. Thus, 2 becomes the first member of **V**. If we look at that structure we see that the name John is the first item in that vector. Therefore, the second member of **V** is 1.

Thus:

```
2 1→Names
```

John

The path to John Smith's area code is

```
2 3 1→Names
```

914

Also:

```

      3>'APL' 'IS' 'GREAT!'
GREAT!

      3 6>'APL' 'IS' 'GREAT!'
!

      ⍵←3 3ρ'abcdefghijk'
abc
def
ghi

      (⊃3 2)⊃M
h

```

Note the enclosure of 3 and 2. In an unnested situation `M[3;2]` would also have produced `h`.

However, in a nested situation, the indices to address the subscripts of an array need to be enclosed:

```

      ⍵←2 2ρ(8 0.5) 98 'XYZ' ¯12.3
8 0.5    98
XYZ      ¯12.3

      ((⊃1 1),2)⊃A
0.5

```


Exercise 74

Given:

```
L←'LPS' 8.95
T←'TAPES' 12.50
C←'CDS' 14.95
PRD←L T C
```

- a) 2⊃PRD
- b) 2 1⊃PRD
- c) 2 1 3⊃PRD
- d) 1 2⊃PRD
- e) 1 2 3⊃PRD

Given `dt←(1 2 'jkl')(3 4 5 'mno')((6 7 8 9) 'pqrs')`
Determine the *value*, *shape*, *depth* and `⌈display` of the following:

- f) 3⊃dt
- g) 3 2⊃dt
- h) 3 2 3⊃dt
- i) 2⊃dt

Given `v←'vwxyz' (1 2) ((3 4 5)(6 7))`
Determine the paths that select the stated values:

- j) _____⊃v
vwxyz
- k) _____⊃v
1 2
- l) _____⊃v
2
- m) _____⊃v
z
- n) _____⊃v
6 7
- o) _____⊃v
6

Given `names ← 'David' 'Anne' 'Ray'`

- p) Select `Anne` and `David` in that order.
- q) Select `e` in `Anne`
- r) Select `Ray` from `names`

Index L [] R

This is an appropriate time to discuss selecting data from an array via indexing again. Up to this point selecting data from an array was accomplished via brackets with embedded semicolons. This construct is not a function. We shall see brackets behaving as an operator shortly..

Being able to select random items (also called *scatter indexing*) from within an array and to use the index facility as operand to an operator are desirable features. The introduction of the enclosure function is essential here.

```

      m←4 5ρa
ABCDE
FGHIJ
KLMNO
PQRST
      m[⊖2 3]
H
      m[⊖2 3]≡m[ ,2;3]
1
      m[(⊖1 3)(⊖1 1)(⊖4 5)]  ⌘ This is not possible with semicolon indexing
CAT

```

Trading *Enclosure* and parentheses for semicolons allows *scatter indexing*. Care must be taken. Recall that *the shape of the result of bracket indexing is the shape of the indices*.

```

      m[⊖2 3]≡m[2;3]
1
      m[⊖2 3]≡m[ ,2;3]
0

```

Still bracket indexing cannot be used as an operand to an operator hence the introduction of the function *Index*, L[]R (*Ctrl+Shift+L*). Informally it is often called *squad indexing* since the symbol can be imaged as [and] being merged together.

The right argument R is a data array and the left argument L are the integer selectors. The left argument is a vector of integers ≥ 1 , the length of which must be less than or equal to the rank of the array R:

```

      v←'abcdef'
abcdef
      4[]v
d
      1 4[]v
LENGTH ERROR
      1 4[]v
      ^
      (⊖1 4)[]v
ad
      (⊖6 1 4)[]v
fad

```

```

      ⍒ a2←10×⍳4 5
10 10 10 20 10 30 10 40 10 50
20 10 20 20 20 30 20 40 20 50
30 10 30 20 30 30 30 40 30 50
40 10 40 20 40 30 40 40 40 50

```

```

      2⍳a2
20 10 20 20 20 30 20 40 20 50

```

```

      2 4⍳a2
20 40

```

```

      (← 2 4)⍳a2
20 10 20 20 20 30 20 40 20 50
40 10 40 20 40 30 40 40 40 50

```

Illustration: Complex number Translation

In APL, complex numbers are written as xjy where x is the real part and y is the complex part;

For example, $2j3$, $-2j3$, $2j^{-3}$, or $-2j^{-3}$

In mathematics, some complex numbers are written as $(x+yi)$

The *Dfn Complex* converts a mathematical complex representation $x+yi$ into a real APL complex number xjy for x and $y \geq 0$.

```

      ▽ Complex←{⍒Converts a math complex no. x+yi (x&y>0)
[1]      ⍒to APL complex no. xjy
[2]      ⍒a chac. vec. 'x+yi'
[3]      w←w~'i'
[4]      ((⍳'+ '=w)⍳w)←'j'
[5]      ⍳w
[6]      }
      ▽

```

Exercise 75

- $3⍳a2$
- $3 1⍳a2$
- $(←3 1)⍳a2$
- $4 (3 1)⍳a2$
- $(3 1) 4⍳a2$

Selective Assignment Revisited Again

Earlier we saw how through indexing we were able to replace items within a vector. Such selective assignment can be done through other functions as well. It is a two-step process.

First, you create an expression that selects the items you wish to replace. Then you place that selective expression in parentheses to the left of the assignment symbol.

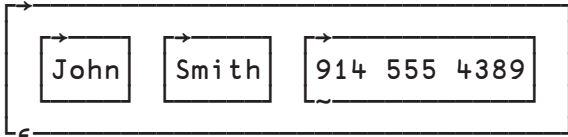
Figuratively,

(a selective expression) ← the replacement expression or data.

Consider the variable:

```
Nameb←'J.' 'Smith' (914 555 4389)
1>Nameb
J.
      (1>Nameb)←'John'
      Nameb
John  Smith  914 555 4389

      ]display Nameb
```



The diagram shows a large rectangular box representing a vector. Inside this box, there are three smaller rectangular boxes arranged horizontally. The first box contains the text 'John', the second contains 'Smith', and the third contains '914 555 4389'. Arrows point from the left and right sides of the large box to each of the three smaller boxes, indicating they are elements of the vector.

Note that the replacement item need not have the same shape, rank or type.

Exercise 76

- In `grades← (86 91 88) (78 88 0) (90 89 93)`, replace the 0 grade by the makeup grade 85.
- Given `q←12 0 3.5 0 0 ^56`, replace the zeros by blanks.
- Replace 914 in `Nameb` by 845.
- Given the matrix `m←3 4p'ABCDEFGHIJK'`, replace the second row by the letter z.

Summary 7: Arrays Revisited

The nature of an array has been greatly extended. No longer must a homogeneous array contain just numbers or just characters. An array may now contain both. Such an array is called a *heterogeneous* array. In addition, the elements of an array may in turn be other arrays. Such arrays are called *Nested* arrays. Heterogeneous nested arrays have more functional support.

The functions and system services introduced are:

<i>Depth</i>	$\equiv R$	Returns the maximum amount of nesting within the array.
<i>Ravel</i>	$, R$	Converts an array into a vector.
<i>Enlist</i>	ϵR	Converts an array into a simple vector.
<i>Table</i>	$\bar{,} R$	Converts an array into a matrix
<i>Catenate</i>	L , R	Combines two arrays into one.
<i>Enclose</i>	$\subset R$	Enables an array to be treated as a scalar.
<i>Mix</i>	$\uparrow R \quad (\square m1 < 2)$	Increases the rank of the array.
<i>Disclose</i>	$\subset R \quad (\square m1 \geq 2)$	Increases the rank of the array.
<i>Split</i>	$\downarrow R$	Decreases the <i>rank</i> of the array
<i>Partition</i>	$L \subseteq R$	Enables the subdividing of an array into separate nested items
<i>Pick</i>	$V \supset R$	Moves through a nested array to get to a single element. To the left of assignment, it replaces an item within a nested array.
<i>Index</i>	$L \square R$	Selects L items from R

Selective assignment is extended:

(a selective expression) \leftarrow the replacement expression or data.

The System functions `]display` and `]box` provide a pictorial view of an array.

Mathematical Facilities

As we have worked through the APL functions most of them in general manipulate numbers. But there are also some that perform more specific advanced mathematical functions. They are grouped here. Several, you have already met.

For example,

The monadic function \circ represents the function '*Pi times*'.

```

      ○1 2
3.141592654 6.283185307

```

```

      ○.5      ♂ one half pi
1.570796327

```

Then the monadic function $*$ represents the function '*e raised to the power of*'

```

      *1 2 3
2.718281828 7.389056099 20.08553692

```

Then the dyadic function $L * R$ represents the power function where the L is raised to the power R .

```

      36 2 ^4 4*0.5 4 0.5 5
6 16 0J2 1024

```

Note the third item produces a complex number. APL handles complex numbers, too.

Now let's look at some more mathematically-oriented primitive functions.

Logarithms $\otimes R$ $L \otimes R$

Logarithms are related to exponentiation. Consider

$$A^B = C$$

- With exponentiation, you have the values A , the base, and B , the exponent, and you wish to find C .
- With logarithms, you have the values A and C and you wish to find the exponent B of A , the *base* of the logarithm. The symbol for the logarithm is \otimes (*Ctrl+Shift+8*).

In taking logarithms, the base A is the left argument of the function symbol \otimes , and the right argument C is value to be derived from the base by raising this base A to the to-be-determined exponent for the data C . Pictorially,

$$L \otimes ? = R \text{ where you wish to replace the exponent } ? \text{ with a number.}$$

This number is called the logarithm of R with respect to the base of L .

If the left argument is omitted, then the base is e (2.7182818284...).

Most of the time, the base for logarithms has been either 10 or e . In mathematics, *base 10* logarithms use the symbol *log* and are called *common logarithms*. *Base e* logarithms use the symbol *ln* and are called *natural logarithms*.

You are *not* restricted in APL to using only base 10 or e .

Exercise 77

- a) $10 \otimes 1 \ 10 \ 100 \ 1000$
 - b) $10 \otimes 1 \ 2 \ 3 \ 10$
 - c) $10 \otimes 100 \times 0 \ 1 \ 2 \ 3$
 - d) $10 \star 10 \otimes 100 \times 1 \ 2 \ 3$
 - e) $2 \otimes 2 \ 4 \ 6 \ 8$
 - f) $2 \star 2 \otimes 2 \ 4 \ 6 \ 8$
 - g) $3 \otimes 3 \ 9 \ 27$
 - h) $3 \star 3 \otimes 3 \ 9 \ 27$
 - i) Create a *Dfn* to find the number of interest payment periods, n , given S, P , and i . (N.B.: using the compound interest formula, $S=P(1+i)^n$, solve for n)
 - j) Find the number of interest payment periods that would be necessary for $S=\$200$, starting with $P=\$100$ at 4% annually. (You may check your answer by substituting it in the compound interest formula on page 52.)
- Theorem: Whenever a number N is multiplied by 10, the \log_{10} of it is increased by 1.
- k) Show that for $V \leftarrow 2.345 \ 15 \ 0.00987$
 - l) Is that still true for natural logarithms? Show what happens for V and $10 \times V$.
 - m) What would increase the characteristic by 1 for the natural logarithm? Show with V .

Benford's Law

Benford's Law concerns the probability of the digits, d , (1, 2, ...9) being the leading digits of numbers in a data set. It assumes that the logarithms of the numbers in the data set are uniformly and randomly distributed. Benford's Law has been used as a tool to detect fraud.

This probability, $P(d)$, is

$$\log_{10}(d+1) - \log_{10}(d)$$

Exercise 78

Create a *Dfn* that will accept any integer 1 to 9 and determine its percentage-wise distribution as the first digit of a number in a given data set.

Trigonometric Functions¹⁵

Also among the mathematical functions are the trigonometric functions referred to in *APL* as the *Circular* functions. Their symbol is \circ (*Ctrl*+ *O*).

The monadic function represents the function *pi times*:

```

      ○ 1 2
3.141592654 6.283185307

```

The dyadic form, $L \circ R$, yields the trigonometric functions where L is an integer ranging from -12 to 12 and R is an angle in radians:

```

1○y represents Sine y
2○y represents Cosine y
3○y represents Tangent y

```

This is the table defining the full set of trigonometric functions.

Circular Functions		
$(-X) \circ Y$	X	$X \circ Y$
$(1-Y*2)*.5$	0	$(1-Y*2)*.5$
Arcsin Y	1	Sine Y
Arccos Y	2	Cosine Y
Arctan Y	3	Tangent Y
$(-1+Y*2)*.5$	4	$(1+Y*2)*.5$
Arcsinh Y	5	Sinh Y
Arccosh Y	6	Cosh Y
Arctan Y	7	Tanh Y
$-8 \circ Y$	8	$(-1+Y*2)*0.5$
Y	9	Real part
+Y	10	Y
$Y \times 0J1$	11	Imaginary part
$*Y \times 0J1$	12	The phase of Y

¹⁵ If in your mathematical experience you have not been exposed to the trigonometric functions, skip this page until you encounter them.

Exercise 79

Should you wish to use the trigonometric functions, the following functions may be useful. The measurement of angles are usually give in degrees or radians. The relation between them is $\text{Pi radians} = 180 \text{ degrees}$.

- a) Create a *Dfn* to convert degrees to radians.
- b) Create a *Dfn* to convert radians to degrees.
- c) Create a *Dfn* that given degrees will produce the matching radians or given radians will produce the matching degrees.
- d) Write the expression for cosine of 45 degrees.
- e) Write the expression for sine of one-half pi.
- f) Write the expression for the tangent of 45 degrees.

Decode L⊥R

Polynomials such as $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ often appear in mathematics. In such equations the a 's are constants and x is the unknown whose value is to be found. In replacing the x with a number in the polynomial, it may be involved in answering such questions as

- Evaluating a polynomial,
- Determining if a given number is the zero of the polynomial,
- Determining how many inches are there in 2 yards, 7 feet, and 3 inches?

Of course, in APL one can evaluate the equation directly by replacing the unknown in the polynomial with a numeric value. For example, for $3x^2 - 5x + 7$ and x is 4:

```
X←4
(3×X*2) + (-5×X) + 7
35
```

The *Decode* function $L⊥R$ (*Ctrl+b*) can accomplish this directly;

```
4 4 4⊥3 -5 7
35
```

Let's examine how *Decode* works. Both arguments may be numeric arrays. The right argument R is the data to be decoded. The Left argument L determines the nature of the decoding.

Both R and L must meet conformability conditions:

The last dimension of L matches the first dimension of R or either L or R is a scalar or 1-element array.

```
4⊥3 -5 7
35
```

When L is a scalar and R is a vector, L is expanded into a vector the length of R .

```
4 4 4⊥3 -5 7
35
```

In the example $3x^2 - 5x + 7$, L becomes the vector $(4*2) (4*1) (4*0)$.

that is, $16 4 1$, which is multiplied by the vector $3 -5 7$.

```
16 4 1× 3 -5 7
48 -20 7
```

These intermediate products are then summed to get **35**, the value of the polynomial $3x^2 - 5x + 7$ for $x=4$.

Provided that the digits of that base number are proper. To find the decimal equivalent of the base 8 number, 175, type:

```
8⊥1 7 5
125
```

Where **8** is shorthand for the vector **8 8 8**.

```
8 8 8⊥1 7 5
125
```

Which is $(1*8*2) + (7*8*1) + 5*8*0$.

When the components of the vector left argument **L** are different, the polynomial evaluation can be viewed as a mixed-base representation. For example, if you wished to find how many inches are there in 2 yards, 7 feet, and 3 inches, you create the relationship vector **1 3 12** representing 1 yard, 3 feet per yard, and 12 inches per foot, the relationship of the items in the mixed base, for **L** and then apply via **Decode** to the date as **R**.

1 3 12 2 7 3
159

Here the left argument becomes **(3×12) 12 1**.

L and **R** may be arrays of any shape and rank. Again, the conformability is either

- The last dimension of **L** must match the first dimension of **R** *or*
- **L** or **R** is a scalar *or*
- The last dimension of **L** or the first dimension **R** must be a 1.

The shape of the result of **Decode** is the catenation of all but the last dimension of **L** with all but the first dimension of **R**.

Illustration: Several Values for a Polynomial Evaluation

Often a polynomial may need to be evaluated for several values. To do this each value for x must appear as a row in a matrix.

Suppose the above equation:

$$3x^2 - 5x + 7$$

needs to be evaluated individually for 4, -2, and 3. Then

$$\begin{pmatrix} 3 & -5 & 7 \\ 4 & -2 & 3 \end{pmatrix} \begin{pmatrix} 13 \\ -5 \\ 7 \end{pmatrix} = \begin{pmatrix} 35 & 29 & 19 \end{pmatrix}$$

But

$$\begin{pmatrix} 4 & -2 & 3 \\ 13 & -5 & 7 \end{pmatrix} = \begin{pmatrix} -26 \end{pmatrix}$$

Illustration: Multiple Data to be Decoded

Find the number of seconds for:

- 1 hour, 2 minutes, and 3 seconds
- 3 hours and 5 minutes
- 2 hours, 14 minutes, and 37 seconds

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 5 & 0 \\ 2 & 14 & 37 \end{pmatrix} \begin{pmatrix} \text{Times} \\ 3600 \\ 60 \end{pmatrix} = \begin{pmatrix} 3723 \\ 11100 \\ 8077 \end{pmatrix}$$

Again decoding is by column. Therefore, the first row of R are to be the hours, the seconds the minutes, and the third the seconds.

$$\begin{pmatrix} 24 & 60 & 60 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} \text{Times} \\ 3600 \\ 60 \end{pmatrix} = \begin{pmatrix} 3723 \\ 11100 \\ 8077 \end{pmatrix}$$

Illustration: Multiple Values for Multiple Data

Find the four decimal representations for the two vectors (1 7 5) and (2 5 1) as if both were to be represented both in base 8 and 16. Both L and R must be arrays.

$$\begin{pmatrix} 1 & 7 & 5 \\ 2 & 5 & 1 \end{pmatrix} \begin{pmatrix} \text{Base} \\ 8 \\ 16 \end{pmatrix} = \begin{pmatrix} 125 & 328 \\ 373 & 1296 \end{pmatrix}$$

Illustration: Name to Number

Given a character string like perhaps a name, “RAY” it can be converted to its representative number as follows:

$$26 \perp \Box a \text{ } \tau \text{ 'RAY'}$$

$$12219$$

Of course, with 26 being raised to various powers the number can get pretty big for larger names.

$$26 \perp \Box a \text{ } \tau \text{ 'DIALOG'}$$

$$58975989$$

Illustration: A Two Item Vector to a Single Number

Give a pair of positive integers, they can be converted to a single integer. First you have to the determine the largest number for the power of 10 as the left argument of Decode.

$$v \leftarrow 234 \text{ } 1567$$

$$\lceil 10^{\otimes} \rceil / v$$

$$4$$

$$(10 * \lceil 10^{\otimes} \rceil / v) \perp v$$

$$2341567$$

Illustration: Replacing Sum Reduction

Suppose that you were not able to use +/ to sum up a numeric vector V, what can you do?

One answer is

$$1 \perp V$$

$$V \leftarrow 12 \text{ } 0.5 \text{ } \neg 10 \text{ } 13$$

$$+ / V$$

$$15.5$$

$$1 \perp V$$

$$15.5$$

$$(+ / V) \equiv 1 \perp V$$

$$1$$

And for minus reduction,

$$\neg 1 \perp \phi V$$

$$\neg 11.5$$

$$\neg / v \neg 11.5$$

$$\neg 11.5$$

Exercise 80

Evaluate the following expressions:

a) $10 \perp 1 \ 9 \ 2 \ 5$

b) $2 \perp 1 \ 1 \ 0 \ 1$

c) $2 \perp 2 \ 1 \ 4 \ 3$

d) $1.5 \perp 1 \ 5 \ 1.5$

e) $\neg 3 \perp 1 \ 2 \ 3$

f) $.5 \perp 1 \ 2 \ 3$

g) $1 \ 2 \ 3 \perp 5$

h) $(\neg 1 \ 2 \ 3) \perp 5$

i) $1 \ 2 \ 3 \perp 1 \ 2 \ 3$

With $m \leftarrow 3 \ 3 \rho 19$

j) $10 \perp m$

k) $1 \ 2 \ 3 \perp m$

l) $(\neg 1 \ 2 \ 3) \perp m$

m) $m \perp m$

n) Create a *Dfn* that will evaluate a given polynomial for each value in a given vector of values.

o) Given a base 8 number such as 3611, determine the equivalent decimal number.

Encode L T R

With the symbol of *Encode* an inversion of *Decode*'s symbol, you would expect a relationship. Indeed, in *Decode* you can determine the decimal equivalent of a number stated in a different number base. With *Encode* you can determine a different number base representation of a decimal number.

Consider the base 8 number 251:

$$\begin{array}{r} 8 \downarrow 2 \ 5 \ 1 \\ 169 \end{array} \quad \# \text{ The decimal equivalent of the 251 (base 8)}$$

Now with *Encode*:

$$\begin{array}{r} 8 \ 8 \ 8 \uparrow 169 \\ 2 \ 5 \ 1 \end{array} \quad \# \text{ The Octal digits Of the decimal number 169}$$

Since *Encode* cannot determine the number of 8's for the non-decimal representation, the user must supply the necessary number of them.

The general form of *Encode* is L T R.

Where R is the data to be encoded and L provides the radix conversion details.

L and R may be any rank array. The shape of *Encode* is

$$(\rho L), \rho R$$

And the rank is

$$(\rho \rho L), \rho \rho R.$$

The complete encoding of R depends on the number of digits in L. An incomplete encoding will occur. Consider the following sequence:

$$\begin{array}{r} 8 \uparrow 169 \\ 1 \\ 8 \ 8 \uparrow 169 \\ 5 \ 1 \\ 8 \ 8 \ 8 \uparrow 169 \\ 2 \ 5 \ 1 \\ 8 \ 8 \ 8 \ 8 \uparrow 169 \\ 0 \ 2 \ 5 \ 1 \end{array}$$

The resulting digits of the *Encode* function are produced by repeated applications of the residue function on right argument R. Consider this sequence that the *Encode* performs:

$$\begin{array}{r} 8 \mid 169 \quad \# \text{ the rightmost digit of result} \\ 1 \\ 169 \div 8 \quad \# \text{ the rightmost } 8 \ 8 \ 8 \\ 21.125 \\ \lfloor 169 \div 8 \quad \# \text{ need only the integer} \\ 21 \\ 8 \mid 21 \quad \# \text{ 2nd digit of result} \\ 5 \end{array}$$


```

      | 21÷8      R again, only need the integer
2
      8|2        R 3rd digit of result
2
      8 8 8⌈169
2 5 1
      10⌈8 8 8⌈169
251

```

If a zero were the leftmost component of L before the necessary number of components of L , then the final value would be the result of the last residue. This then may not be the complete representation of the number.

```

      0 8⌈169
21 1
      8 8⌈169
5 1
      0 8 8⌈169
2 5 1
      0 8 8 8⌈169
0 2 5 1

```

As with the *Decode* function, the values of L may be different.

For example,

```

      0 1760 3 12⌈1000
0 27 2 4

```

returns the number of yards, feet, and inches in 1000 inches.

It is important that the leftmost argument of L be a zero when the right argument R is a negative number.

```

      8 8 8 8⌈-169
7 5 2 7
      8⌈ 8 8 8 8⌈-169
3927
      0 8 8 8⌈-169
-1 5 2 7
      8⌈ 0 8 8 8⌈-169
-169

```

Essentially, the *Encode* function executes by repeated use of the *Modulo* function.

```

Consider      7 7 7⌈194
3 6 5        ↓
              7|194
5            ...where |194÷7 is 27
              ↓
              7|27
6            ...where |27÷7 is 3
              7|3
3

```

Illustration: Time Conversion

Suppose a satellite with two Terabytes of data memory can transmit this data at the rate of a byte per millisecond, how long would it take to transmit the two Terabytes of data?

$$\begin{array}{r} 0 \ 24 \ 60 \ 60 \ 1e3 \tau 2e12 \div 1e3 \\ 23 \ 3 \ 33 \ 20 \ 0 \end{array}$$

Illustration: Integer and Factional Parts of a Decimal Number

With a decimal number like 12.345, the ceiling and floor functions produce the integers immediately surrounding the number.

$$\begin{array}{r} \lceil 12.345 \\ 13 \\ \lfloor 12.345 \\ 12 \end{array}$$

The one-residue of the number produces the fractional part of the number:

$$\begin{array}{r} 1 \mid 12.345 \\ 0.345 \end{array}$$

Now the 0 1 Encode produces both the integer and fractional part of a decimal number:

$$\begin{array}{r} 0 \ 1 \tau 12.345 \\ 12 \ 0.345 \end{array}$$

Illustration: The number of Digits for the Left Argument

Placing a zero as the left argument of Encode acts like a catch all. Using the Ceiling of the logarithm in the base to which you wish to go. For example, what is the base 8 representation of the decimal number 1929?

$$\begin{array}{r} ((\lceil 8 * 1929 \rceil) \rho 8) \tau 1929 \\ 3 \ 6 \ 1 \ 1 \\ 0 \ 8 \ 8 \ 8 \ 8 \ \tau 1929 \\ 0 \ 3 \ 6 \ 1 \ 1 \\ 8 \downarrow ((\lceil 8 * 1929 \rceil) \rho 8) \tau 1929 \\ 1929 \\ 8 \downarrow 0 \ 8 \ 8 \ 8 \ 8 \ \tau 1929 \\ 1929 \end{array}$$

Illustration: An Integer into Vector of its Digits

The conversion of a positive integer into a vector representing its individual digits is as follows

$$\begin{array}{r} v \leftarrow 5749 \\ ((1 + \lfloor 10 * v \rfloor) \rho 10) \tau v \\ 5 \ 7 \ 4 \ 9 \end{array}$$

Illustration: Binary Vector Representation of a Positive Decimal Number

The number of twos needed will depend upon the size of the decimal number.

$$\begin{array}{r} 2 \ 2 \ 2 \ 2 \ \tau 13 \\ 1 \ 1 \ 0 \ 1 \end{array}$$

Exercise 81

Evaluate the following expressions:

- a) $8 \ 8 \ 8 \tau 55$
- b) $8 \perp 8 \ 8 \ 8 \tau 55$
- c) $8 \ 8 \ 8 \tau^{-} 55$
- d) $8 \perp 8 \ 8 \ 8 \tau^{-} 55$
- e) $0 \ 8 \ 8 \ 8 \ \tau^{-} 55$
- f) $8 \perp 0 \ 8 \ 8 \ 8 \tau^{-} 55$
- g) $0 \ 2 \ 2 \ 2 \tau 14$
- h) Create a *Dfn* to convert a positive integer into a vector of its digits.
- i) Create a *Dfn* to determine the binary representation of a positive decimal number.

Elementary Scalar Functions: Factorial !R, Binomial L!R, and Residue L|R

The APL *Factorial*, !R, and *Binomial*, L!R, functions behave as they do in mathematics.

When R is a non-negative integer, *Factorial* !R produces $\times / \iota R$. *Factorial* is a scalar function.

```
      !4 3 5
24 6 120
```

```
      !0
1
```

```
      !-4
DOMAIN ERROR
      !-4
      ^
```

If R is not an integer, the result is the value of the *Gamma* function evaluated at R+1.

```
      !.5
0.8862269255
```

Illustration: Permutations

A permutation is any arrangement of a set of different objects. For example for the three letters A, B, and C there are six different permutations ABC, BAC, ACB, CAB, CBA. For the vector V is any arrangement of a set of different objects. For example for the three letters A, B, and C, there are six different permutations: ABC, ACB, BAC, BCA, CBA, and CAB. For the vector V←'abcd', the the number of permutations is:

```
      !pV
24
```

The number of permutations that can be formed from N distinct elements taking only R of them at a time is

```
      (!N)÷!N-R

      N←10
      R←4
      (!N)÷!N-R   where 1≤R≤N
5040
```

Illustration: Combinations

A *Combination* is a selection of items from a set of distinct items where the order of the elements in each item selected does not matter and is not counted. Only one of the various arrangements in one selected item is counted. For example, if “ABC” were selected, then “BCA”, “BAC”, “CAB”, “CBA”, and “ACB” would not be members of the *Combinations*.

The mathematical equation for determining the combination from a set n of r length items is, in APL terminology, $(!n)÷(!r)×!n$...however, the dyadic L!R also returns the same Combination value.

```

n←10
r←3
(!n)÷(!r)×!n-r
120

```

```

3!10
120

```

The *Binomial* $L!R$ answers the question of how many ways can L items be taken from a set of R items.

```

4!8
70

```

```

4 2!8 10
70 45

```

The expression $L!R$ is called *Binomial* because the coefficients of the binomial expression $(x+1)^n$, where n is a position integer

Can be determined through $L!R$ via $(0, 1n)!n$

Exercise 82

Create a *Dfn* that will produce the coefficients of the binomial expression $(x+1)^n$ where n is a position integer.

The Scalar *Residue* function was first introduced on page 146. In summary, the residue function produces the remainder when the right argument is divided by the left argument.

```
2 3 4|27
1 0 3
```

Extending the earlier presentation consider the following illustrations. The situation addressed is that when *Residue* presented a negative number the compliment of the fractional part is produced.

Illustration: The Fractional part of Number

```
1|12.34 ^12.34
0.34 0.66
+ / 1|12.34 ^12.34
1.000
1|12.34 ^12.34
0.34 0.66
^1|12.34 ^12.34      n ^1 doesn't work
^-0.66 ^0.34
(xv)×1||v←12.34 ^12.34
0.34 ^0.34
```

Exercise 83

Create a *Dfn* that will produce the proper fractional part of a decimal number..

Illustration: Vector Number Separation

When given a number such as 123.456 or ^78.19, the goal is to return separately the integer and fractional part of the number properly signed. The number 123.456 is to return 123 and 0.456, and ^78.19 is to return ^78 and ^0.19.

Let us step through a solution using ^78.19.

```
(0 1)|n←^78.19
^78.19 0.81

(0 1)||n←^78.19
78.19 0.19

z←(×n)×(0 1)||n←^78.19
^78.19 ^0.19

(-/z) ,1↓z←(×n)×(0 1)||n←^678.98
^78 ^0.19

(-/z) ,1↓z←(×n)×(0 1)||n←123.45
123 0.45
```

Exercise 84

Create a *Dfn* that splits a decimal number into its whole and fractional parts.

Format: $\text{f}R$ and $L\text{f}R$

The format functions do not manipulate data mathematically; rather they control the appearance of the output. Normally, much of mathematical output is displayed with the fractional part given up to nine decimal places. Often this extended precision is not necessary. In the process of doing this, the Format functions convert their right argument, R , into character information. The symbol for format is f (*Ctrl*+'').

```

      v←12 ^3 4.5
12 ^3 4.5
      ρv
3
      f v
12 ^3 4.5
      ρf v
9

```

In the conversion to character data every symbol such as a negative sign, spacesNFC, negative sign, and others are counted as they are transformed into characters. The character transformation occurs for any rank array.

```

      A←3 2ρ 25.3 ^12.6 2e^3 1234 (,9.1)
25.3      ^12.6
0.002 1234
9.1      25.3
      ρA
3 2
      f A
25.3      ^12.6
0.002 1234
9.1      25.3
      ρf A
3 14

```

The left argument L of the dyadic *Format* function, $L\text{f}R$, gives one more control over the nature of the characterization of the data. It may be a zero or a positive or negative integer. A zero implies that no fractional part should appear in the formatting

```

      0f v←12.345 ^67.8904
12 ^68

```

A positive integer indicates the number of digits to the right of the decimal point in the format of the data.

```

      2f v←12.345 ^67.8904
12.35 ^67.89

```

A negative integer indicates that the data should be represented in exponential form. The integer indicates the number of digits of the multiplier in exponential form.

```

      ^2f v←12.345 ^6712.8904 9.75 789
1.2E1 ^6.7E3 9.8E0 7.9E2

```

The left argument of format may be a pair of integers where the first integer indicates the character field into which the formatted number is to be placed and the second digit characterizes the representation of the data in either decimal or exponential form.

In arrays this may even be done column-wise.

```

      v
12.345 -6712.8904 9.75 789
      pv
4      10 2fv
      12.35 -6712.89      9.75      789.00

      p 10 2fv
40

      ]display 10 2fv
→ [ 12.35 -6712.89      9.75      789.00 ]

```

Formatting also can be applied to arrays. A single pair of integers would apply to each column or individual pairs can be associated with the individual columns of an array.

```

      m
1234.56 -1234.5678
987.2    0
5.217    -35
123      0.456
      pm
4 2
      10 2fm
1234.56 -1234.57
987.20  0.00
5.22    -35.00
123.00  0.46
      p10 2fm
4 20
      5 2fm
*****
***** 0.00
5.22*****
***** 0.46
      10 2 12 -2fm
1234.56 -1.2E3
987.20  0.0E0
5.22    -3.5E1
123.00  4.6E-1
      p 10 2 12 -2fm
4 22

```

Note that when the numbers do not fit into the defined character field they are replaced by asterisks.

Exercise 85

Apply the Shape, Rank, and Depth function after executing each expression. For example:

```

⌈12.345
12.345

srd ⌈12.345
Shape 6 Rank 1 Depth 1

v1←12.3456
v2←12.3456 ⌈12.3456
v3←(?3 ⍥1e6)÷1e5
m43←(?4 3⍥1e5)÷1e4
a234←(?2 3 4⍥1e5)÷1e4

```

- a) $0 \times v1$
- b) $2 \times v1$
- c) $5 \times v1$
- d) $^{-2} \times v1$
- e) $^{-5} \times v1$
- f) $10 \ 2 \times v1$
- g) $10 \ 2 \times v2$
- h) $10 \ 2 \ 10^{-2} \times v2$
- i) $5 \ 2 \times v2$
- j) $10 \ 2 \times v3$
- k) $10 \ 2 \ 10^{-2} \ 12 \ 3 \times v3$
- l) $10 \ 2 \times m43$
- m) $10 \ 2 \ 9 \ 1 \ 10^{-3} \times m43$
- n) $8 \ 2 \times a234$
- o) $8 \ 2 \ 10 \ 3 \times a234$

Execute ⌘R

Unlike the function which turns its argument into character information, the monadic *Execute* function, ⌘R (*Ctrl*+;), receiving either a scalar or vector character data, figuratively removes the quote marks and presents the expression to the interpreter to execute.

```

      ⌘ '⌈u←⌈4'
1 2 3 4
      u
1 2 3 4

      u×10
10 20 30 40

```

If the unquoted expression is not executable in a workspace, then an error message will occur.

```

      ⌘')erase u'
⌘VALUE ERROR: Undefined name: erase
      )erase u
      ^

      u
1 2 3 4

      v←'5 -5'

      v+10
DOMAIN ERROR
      v+10
      ^

```

Illustration: A Number from a Character Vector

Often it may be necessary to extract a number from a character vector.

Given the sentence, `sen←'The loss is ($1,234.56)'`, the *Dfn* NFC will create the APL value 1234.56.

```

      ⌈num←NFC 'The loss is ($1,234.56)'
1234.56

      num×10
12345.6

      ∇ NFC←{⌈ extracting a number from a character vector.
[1]   ⌈ ω a character vector
[2]       ⌘(∈(ω∈⌈D,','.'')⊆ω)~',''
[3]   }
      ∇

```

Exercise 86

Modify NFC to handle a negative number, too.

Matrix Inverse

In the real number system, we all work with mathematical inverses. The additive inverse of a number is its negative; 7 and -7 . The multiplicative inverse of a number is its reciprocal; 7 and $\div 7$. In Linear Algebra, where the basic element is not a single number but a matrix, we need to address the matrix inverse of square matrices, that is matrices with the same number of rows and columns. Here, if a matrix and its inverse are multiplied together the result is the identity matrix. The identity matrix is a matrix with ones down the main diagonal and zeros elsewhere.

```

      4 4p1 0 0 0 0
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

```

The identity matrix plays the same role in Linear Algebra that the number 1 does in the Real Number system. They are both multiplicative inverses. Not all matrices have inverses. The matrices that do not have inverses are called *singular* matrices and those that do have inverses are called *non-singular* matrices. The nature of matrix multiplication will be covered in the presentation of the operator *Inner Product*.

Matrix Inverse is a primitive monadic function in APL. Its syntax is R (*Ctrl+shift+=*).

```

      ⍒ a←2 2p4 -9 0 5
4 -9
0 5

      ⍒a
0.25 0.45
0 0.2

      ⍒ b←2 2p6 -9 -4 6
6 -9
-4 6

      ⍒b
DOMAIN ERROR
      ⍒b
^

```

Exercise 87

As mathematicians began using computers in their work, they established sets of test matrices. One of them is the 4×4 Wilson Matrix. It is:

```

      ⍒ Wilson←4 4p5 7 6 5 7 10 8 7 6 8 10 9 5 7 9 10
5 7 6 5
7 10 8 7
6 8 10 9
5 7 9 10

```

- Show that it is symmetric and extract its diagonal and its inverse.
- Show that its inverse is symmetric too.

Matrix Divide $\mathbf{L} \div \mathbf{R}$

Matrix Division, \div (*Ctrl* + *+*), enables one to solve systems of linear equations. In mathematics, presenting a system of equations to be solved are presented like this example:

$$\begin{aligned} 2X_1 - X_2 + 5X_3 &= 13 \\ X_1 + 2X_2 + X_3 &= 0 \\ 4X_1 - X_3 &= 11 \end{aligned}$$

Using Matrix Divide to solve such sets of these equations, the coefficients of the unknowns are created as a matrix:

$$\begin{aligned} &\mathbf{M} \leftarrow \begin{bmatrix} 2 & -1 & 5 \\ 1 & 2 & 1 \\ 4 & 0 & -1 \end{bmatrix} \\ &\mathbf{C} \leftarrow \begin{bmatrix} 13 \\ 0 \\ 11 \end{bmatrix} \end{aligned}$$

and the constants are set as a vector:

$$\mathbf{C} \leftarrow \begin{bmatrix} 13 \\ 0 \\ 11 \end{bmatrix}$$

The vector of constants, \mathbf{C} , becomes the left argument to matrix divide and the matrix \mathbf{M} becomes the right argument:

$$\mathbf{C} \div \mathbf{M}$$

The values of the unknowns in the three original equations follows.

The expression $\mathbf{L} \div \mathbf{R}$ is executed *if*

- a) The first dimensions of \mathbf{L} and \mathbf{R} are equal.
- b) The columns of \mathbf{R} are linearly independent.

The left argument, \mathbf{L} , may be a matrix, in which case the columns of \mathbf{L} are used independently in Matrix Divide with the matrix right argument to yield multiple independent solutions:

$$\begin{aligned} &\mathbf{M} \leftarrow \begin{bmatrix} 3 & 1 & -2 & 1 & 0 & 2 & -8 & -4 & 5 & 9 \end{bmatrix} \\ &\mathbf{L} \leftarrow \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix} \\ &\mathbf{v1} \leftarrow \begin{bmatrix} 0 & 8 & -9 \end{bmatrix} \\ &\mathbf{v2} \leftarrow \begin{bmatrix} 2 & -8 & 8 \end{bmatrix} \\ &(\mathbf{v1} \div \mathbf{M} \quad \mathbf{v2} \div \mathbf{M}) \\ &\mathbf{v1} \div \mathbf{M} \\ &\mathbf{v2} \div \mathbf{M} \\ &(\mathbf{v1} \div \mathbf{M} \quad \mathbf{v2} \div \mathbf{M}) \end{aligned}$$

Illustration: Nickels, Dimes, and Quarters

Give that you have \$3.00 in nickels (5 pennies), dimes (10 pennies), and quarters (25 pennies) and that:

- 1) The number of dimes equals the number of nickels and quarters plus 8.
- 2) The number of dimes and quarters equals the numbers of nickels by 18

How many nickels, dimes, and quarters are there?

Let x represent quarters; y, dimes; and z, nickels.

There exist three equations:

$$25x+10y+5z=300$$

$$y=x+z+8$$

$$y+x=z+18$$

Rearranging these equations,

$$25x+10y+5z=300$$

$$x \quad -y \quad +z = -8$$

$$x \quad y \quad -z = 18$$

In APL:

```

      m←3 3ρ25 10 5 1 1 1 1 1
25 10 5
1 1 1
1 1 1

      300 8 18m
5 16 3

      +/25 10 5×5 16 3
300
    
```

Summary 8: Mathematical Facilities

This section contains the functions that are specifically mathematical. The functions introduced or re-introduced are:

Mathematical Functions	
Pi times	$\circ R$
Exponencial	$* R$
Natural log	$\otimes R$
Logarithms	$L \otimes R$
Trigonometric Functions	$L \circ R$
Decode	$L \perp R$
Encode	$L \top R$
Scalar Factorial	$! R$
Scalar Binomial	$L ! R$
Scalar Residue	$L R$
Format	$\mp R$
Dyadic Format	$L \mp R$
Exécute	$\pm R$
Matrix Inverse	$\boxtimes R$
Matrix Divide	$L \boxtimes R$

Also presented earlier were these mathematically-useful functions:

Mathematically-Useful Functions	
Reverse	ϕR
Rotate	$L \phi R$
First-Axis Reverse	$\ominus R$
First-Axis Rotate	$L \ominus R$
Transpose	$\boxdot R$
Axis Reordering	$L \boxdot R$

This extensive set will be further enhanced through the use of forthcoming *Operators*.

Operators

The concept of operators is unique to APL. They extend the power of APL significantly. The role of the operator is similar to that of the adverb in the English language. There, the adverb modifies the behavior of the verb. For example, “run quickly”, where “quickly” defines the nature of the run. Similarly, operators modify the behavior of functions and produce new functions called *Derived Functions* that manipulate data.

It is important to note that *operators produce functions and functions produce data*.

The *functional* arguments of an operator are called *operands*. The operators may also be either monadic or dyadic. However, in the case of a monadic operator the operand appears to the left of the operator. Hence:

All operators will have a left operand.

As with the primitive functions the primitive operators are symbolic. The operator symbols to be presented are `[]`, `/`, `\`, `.`, `∘.`, `¨`, and `⋆`.

The operators that we will initially consider are, in terms of the derived functions;

Operators	
Axis specification	<code>f[I]</code>
Reduction	<code>f/—</code>
Compression/Replication	<code>c/—</code>
N-wise Reduction	<code>f/—</code>
Scan	<code>f\—</code>
Outer Product	<code>—∘.f—</code>
Inner Product	<code>—f.g—</code>
Each	<code>f¨—</code>
Dyadic Each	<code>—f¨—</code>

Axis Specification $f[I]R$ L $f[I]R$

Earlier you encountered $[I]$ and $[I;J]$ in the process of data selection. The notation was associated with the indexing of data, but it also can act as a monadic operator modifying functional operands. The notation $[I]$ is called *Axis Specification*. I must be either a scalar or vector, and is origin dependent. It exists for arrays of rank two or higher. It provides the choice of which dimension you wish to apply the operand function.

Consider,

```

stores←?3 4p100
43 79 61 18
86 93 77 44
69 29 16 95

```

...which could represent the number of the four identical items that each of the three stores contains.

In this case, one could determine the total number of items each store has via:

```

+/[2]stores
259 250 139

```

or the total number of each item in the three stores via:

```

+/[1]stores
201 178 156 113

```

For all functions that take *axis specification*, not specifying an axis implies that the function is to be applied along a default axis. This default axis is either the first or the last.

For example:

```

m←3 4p12
+ /m
10 26 42
+/[2]m
10 26 42

```

The following are the functions previously introduced to which axis specification may apply. Axis Specification may be applied to *all primitive scalar dyadic* functions.

<div> <div>m</div> <div>1 2 3 4</div> <div>5 6 7 8</div> <div>9 10 11 12</div> </div>	<div> <div>m+[1]2 5 10</div> <div>3 4 5 6</div> <div>10 11 12 13</div> <div>19 20 21 22</div> </div>
<div> <div>m</div> <div>1 2 3 4</div> <div>5 6 7 8</div> <div>9 10 11 12</div> </div>	<div> <div>2 1 -1 ×[1]m</div> <div>2 4 6 8</div> <div>5 6 7 8</div> <div>-9 -10 -11 -12</div> </div>

Exercise 88

Evaluate the following and determine their shape, rank, depth, and `]display`

- $(m2)$ $(m2*[2]0\ 3\ 2\ 1)$
- $(m2)$ $(m2*[1]3\ 2\ 1)$
- $(m2)$ $(m2×[1]10\ 100\ 2)$
- $(m2)$ $(m2÷[2]2\ 3\ .5\ -2)$

Axis Specification may be applied to these *specific non-scalar monadic* functions:

ϕ or Θ	Reverse
\uparrow	Mix
\downarrow	Split
$,$	Ravel
\Leftarrow	Enclose

For Reverse:

	$\Leftarrow a \leftarrow 3 \ 4 \rho \Box a$		
a	(ϕa)	($\phi[1]a$)	($\phi[2]a$)
ABCD	DCBA	IJKL	DCBA
EFGH	HGFE	EFGH	HGFE
IJKL	LKJI	ABCD	LKJI

For Mix:

b	($\uparrow b$)	($\uparrow[1]b$)	($\uparrow[2]b$)	($\uparrow[.3]b$)
abc defg	abc defg	ad be cf g	abc defg	ad be cf g

For Split:

d	($\downarrow d$)	($\downarrow[1]d$)	($\downarrow[2]d$)
1 2 3 4 5 6	1 2 3 4 5 6	1 4 2 5 3 6	1 2 3 4 5 6

For Enclose:

c	($\Leftarrow c$)	($\Leftarrow[1]c$)	($\Leftarrow[2]c$)
ABCD	ABCD	AEI BFJ CGK DHL	ABCD EFGH IJKL
EFGH	EFGH		
IJKL	IJKL		

For Ravel:

c	($,c$)	($,[1]c$)	($,[2]c$)
ABCD	ABCDEFGH IJKL	ABCD	ABCD
EFGH		EFGH	EFGH
IJKL		IJKL	IJKL

($,[.1]c$)	($,[1.1]c$)	($,[2.3]c$)
ABCD	ABCD	A
EFGH		B
IJKL	EFGH	C
	IJKL	D
		E
		F
		G
		H
		I
		J
		K
		L

Exercise 89

- a) In the axis expressions for each of the monadic functions above, determine the *shape*, *rank*, *depth*, and `]display`.

Axis Specification may be applied to these specific non-scalar dyadic functions:

Partition	\subseteq
Rotate	ϕ or θ
Catenate	, or $\bar{,}$
Laminate	, or $\bar{,}$
Take	\uparrow
Drop	\downarrow

For *Partition*:

<div style="text-align: right; margin-right: 10px;">m3</div> ABCD EFGH IJKL MNOP QRST UVWX	<div style="text-align: center;">srd m3</div> Shape 2 3 4 Rank 3 Depth 1	
<div style="text-align: right; margin-right: 10px;">m3</div> ABCD EFGH IJKL MNOP QRST UVWX	<div style="text-align: center;">1 0 2 1\subseteqm3</div> A CD E GH I KL M OP U WX	<div style="text-align: center;">1 0 2 1\subseteq[3]m3</div> A CD E GH I KL M OP U WX
<div style="text-align: right; margin-right: 10px;">m3</div> ABCD EFGH IJKL MNOP QRST UVWX	<div style="text-align: center;">1 0 2\subseteq[2]m3</div> A B C D I J K L M N O P U V W X	<div style="text-align: center;">1 1 2\subseteq[2]m3</div> AE BF CG DH I J K L MQ NR OS PT U V W X

Note that *Partition* always does the partitioning along the last dimension. The axis specification affects the other dimensions.

Exercise 90

In the axis expressions for each of the functions above, determine the *shape*, *rank*, *depth*, and `]display`.

For *Rotate*:

<div style="text-align: right;">m3</div> ABCD EFGH IJKL MNOP QRST UVWX	<div style="text-align: center;">srd m3</div> Shape 2 3 4 Rank 3 Depth 1			
<div style="text-align: right;">m3</div> ABCD EFGH IJKL MNOP QRST UVWX	<div style="text-align: right;">1ϕm3</div> BCDA FGHE JKLI NOPM RSTQ VWXU	<div style="text-align: right;">1ϕ[3]m3</div> BCDA FGHE JKLI NOPM RSTQ VWXU	<div style="text-align: right;">1ϕ[2]m3</div> EFGH IJKL ABCD QRST UVWX MNOP	<div style="text-align: right;">1ϕ[1]m3</div> MNOP QRST UVWX ABCD EFGH IJKL

Exercise 91

In the axis expressions for each of the functions above, determine the shape, rank, depth, and `display`:

For *Catenate*:

$\begin{matrix} & a \\ ABCD \\ EFGH \\ IJKL \end{matrix}$	$\begin{matrix} & a,12 \\ ABCD\ 12 \\ EFGH\ 12 \\ IJKL\ 12 \end{matrix}$	$\begin{matrix} & a,[2]12 \\ ABCD\ 12 \\ EFGH\ 12 \\ IJKL\ 12 \end{matrix}$	$\begin{matrix} & a,[1]12 \\ A\ B\ C\ D \\ E\ F\ G\ H \\ I\ J\ K\ L \\ 12\ 12\ 12\ 12 \end{matrix}$	
$\begin{matrix} & a \\ ABCD \\ EFGH \\ IJKL \end{matrix}$	$\begin{matrix} & 1,[1]a \\ 1\ 1\ 1\ 1 \\ A\ B\ C\ D \\ E\ F\ G\ H \\ I\ J\ K\ L \end{matrix}$	$\begin{matrix} & 1,[2]a \\ 1\ ABCD \\ 1\ EFGH \\ 1\ IJKL \end{matrix}$	$\begin{matrix} & a,[1]1 \\ A\ B\ C\ D \\ E\ F\ G\ H \\ I\ J\ K\ L \\ 1\ 1\ 1\ 1 \end{matrix}$	$\begin{matrix} & a,[2]\ 1 \\ ABCD\ 1 \\ EFGH\ 1 \\ IJKL\ 1 \end{matrix}$
$\begin{matrix} & a \\ ABCD \\ EFGH \\ IJKL \end{matrix}$	$\begin{matrix} & 1,[1]a \\ 1\ 1\ 1\ 1 \\ A\ B\ C\ D \\ E\ F\ G\ H \\ I\ J\ K\ L \end{matrix}$	$\begin{matrix} & 1,[2]a \\ 1\ ABCD \\ 1\ EFGH \\ 1\ IJKL \end{matrix}$	$\begin{matrix} & a,[1]1 \\ A\ B\ C\ D \\ E\ F\ G\ H \\ I\ J\ K\ L \\ 1\ 1\ 1\ 1 \end{matrix}$	$\begin{matrix} & a,[2]\ 1 \\ ABCD\ 1 \\ EFGH\ 1 \\ IJKL\ 1 \end{matrix}$

Exercise 92

- In the axis expressions for each of the functions above, determine the *shape*, *rank*, *depth*, and `display`.
- Write at least two different expressions using *catenate* to generate the following result:

```

1 1 1 1 1 1
1 A B C D 1
1 E F G H 1
1 I J K L 1
1 1 1 1 1 1

```

For *Laminate*:

Laminate, $L[I]R$ is the name associated with L and R where the axis specifier I is *not* an integer. It increases the rank with a new dimension of two. Also, it requires that $\rho L = \rho R$.

```
v1←⍳4
v2←'abcd'
```

v1,[.3]v2				v2,[1.2]v1			
1	2	3	4	a	1		
a	b	c	d	b	2		
				c	3		
				d	4		

```
a2←3 4⍬a
m2←3 4⍬12
```

a2,[.6]m2				a2,⍤[.6]m2			
A	B	C	D	A	B	C	D
E	F	G	H	E	F	G	H
I	J	K	L	I	J	K	L
1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12

a2,[1.6]m2				a2,[2.6]m2			
A	B	C	D	A	1		
1	2	3	4	B	2		
				C	3		
E	F	G	H	D	4		
5	6	7	8				
				E	5		
I	J	K	L	F	6		
9	10	11	12	G	7		
				H	8		
				I	9		
				J	10		
				K	11		
				L	12		

Exercise 93

In the axis expressions for each of the functions above, determine the *shape*, *rank*, *depth*, and `]display`.

For *Take*:

$a2$ ABCD EFGH IJKL	$2\uparrow a2$ ABCD EFGH	$2\uparrow[2]a2$ AB EF IJ	$^{-1}\uparrow[2]a2$ D H L	$1\uparrow[1]a2$ ABCD	$^{-1}\uparrow[1]a2$ IJKL
------------------------------	--------------------------------	------------------------------------	-------------------------------------	--------------------------	------------------------------

Exercise 94

In the axis expressions for each of the functions above, determine the *shape*, *rank*, *depth*, and `]display`.

For *Drop*:

$a2$ ABCD EFGH IJKL	$2\downarrow a2$ IJKL	$2\downarrow[2]a2$ CD GH KL	$^{-1}\downarrow[2]a2$ ABC EFG IJK	$1\downarrow[1]a2$ EFGH IJKL	$^{-1}\downarrow[1]a2$ ABCD EFGH
------------------------------	--------------------------	--------------------------------------	---	------------------------------------	--

Exercise 95

In the axis expressions for each of the functions above, determine the *shape*, *rank*, *depth*, and `]display`.

Reduction $f/[I] R$

Reduction was introduced earlier to do vector addition and multiplication. Now that we know that it is an operator, let us see what more we can do with it. It is one of the most widely-used monadic operators. This operator is called *Reduction*, because the result it produces is always of *rank* one less than its operand. The reduction of a vector is a scalar, and the reduction of a matrix is a vector.

Let us see it in action again.

```
Nom←120 50 27.2 202
```

To sum the numbers in *Nom* via reduction, the left operand is the function $+$ producing the derived function $+/$ with the right argument *Nom*.

```
+/Nom
299.2
```

Similarly, to multiply all the numbers the times function \times would be used as the left operand.

```
×/Nom
32966400
```

It is important to note that *any result-producing dyadic function may be the operand of the reduction operator*.

This includes primitive functions, defined functions, and derived functions. For example,

```
Add←{Addition
      α+ω}
Add/2 10 12 5
29
```

What the reduction operator is doing is taking its functional operand and distributing it between the items of the data argument. For example,

```
+/2 10 12 5
29
```

...is seen by the interpreter as $2+10+12+5$ and, following the rule of grammar, evaluated from right to left. Similarly,

$\times/2\ 10\ 12\ 5$ is shorthand for $2\times10\times12\times5$

Also, the symbol \neq (*Ctrl*+ $+/$) is equivalent to $/[1]$.

```
⌈ m←2 3p16
1 2 3
4 5 6
  ×/[1]m
4 10 18
  ×≠m
4 10 18
```


Illustration: Number of Negative Numbers

Given a numeric array such as,

```
⍵←⍵+?4 4p10
5 6 6 -2
5 -1 5 2
6 -2 1 6
-3 2 0 -1
```

the expression

```
+/⍵=⍵∈A
5
```

states the number of negative numbers in A.

A word of caution is necessary when the operand is not commutative. Consider:

```
-/2 10 12 5 which represents
2-10-12-5 yielding
-1
```

Perhaps you were expecting -25 which is the result of a left to right evaluation; thus

```
((2-10)-12)-5
-25
```

Remember you use parenthesis to overrule the normal right-to-left evaluation.

For fun, try *Rsub* as an operand:

```
▽ Rsub←{⍵ reverse subtraction
[1] ⍵-⍵
[2] }
▽
Rsub/2 10 12 5
-19
```

Exercise 96

Evaluate

- a) $+/10 \ 2 \ ^{-}4 \ 5$
- b) $-/10 \ 2 \ ^{-}4 \ 5$
- c) $+/5$
- d) $-/5$
- e) $+/\imath 4$
- f) $-/\imath 4$

For $\mathbf{Nom} \leftarrow 120 \ ^{-}50 \ 27.2 \ 2e2$

- g) Write the expression to find the largest number in **Nom**.
- h) Write the expression to find the smallest number in **Nom**.
- i) Write the expression to find the difference between the smallest and the largest number in **Nom**, the range.
- j) Given a vector of numbers, create a *Dfn* that will return the sum, product, and the range (the difference between the largest and smallest number).
- k) As a student gets numeric grades often there is an interest in the average. Create a *Dfn* to compute the average grade.

Illustration: Heron's Formula

While finding the areas of right angled, isosceles, and equilateral triangles are easily derived or remembered, finding the area of any triangle is more evolved.

Around 50 AD, the mathematician Heron derived a formula for finding the area of any triangle given the length of its three sides.

This formula for the area of any triangle with sides of length a, b, and c is

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

Where

$$s = \frac{a+b+c}{2}$$

Exercise 97

Create a *Dfn* to compute the area of a triangle given the length of each side.

For example, for the 3 4 5 right triangle:

```
triarea 3 4 5
6
```

And for the 13 13 10 isosceles triangle:

```
triarea 13 13 10
60
```

Illustration: Finding Pythagorean Triples

Recall that for a right triangle with sides a , b , and c where c is the longest side, the equation $a^2+b^2=c^2$ is true. When these numbers a , b , c are positive integers they are called *Pythagorean Triples*. Around 250 A.D., the mathematician Diophantus found how to find all possible integer number solutions to the equation $a^2+b^2=c^2$.

His algorithm begins by taking any two whole numbers, and then do the following three steps:

- take the difference of their squares,
- take twice their product,
- take the sum of their squares.

This algorithm, starting with any pair of whole numbers, will lead to three numbers that form a Pythagorean triple.

Exercise 98

- a) Given any three numbers, create a *Dfn* that will determine if they are a Pythagorean triple.
- b) Following the Diophantus algorithm create a *Dfn* to produce a Pythagorean Triple.

With the advent of matrices and higher-rank arrays, axis specification permits functions to apply along different dimensions. With vectors there was only one dimension along which to apply the function. For example, with a matrix there are 3 common choices:

- apply the function to the rows,
- apply the function to the columns,
- apply the function to the whole matrix.

```

      m←3 4 p12
      m
1  2  3  4
5  6  7  8
9 10 11 12

      +/[1]m
15 18 21 24

      +/[2]m
10 26 42
    
```

N-wise Reduction: $N \text{ f } / [I] \text{ R}$

The dyadic version of the reduction operator, $N \text{ f } / \text{R}$ is called *N-wise Reduction*. The left argument N is to be an integer which indicates a moving window of N items of R upon which the operand f is applied. A negative integer will cause the operand to be applied to the elements in reverse order.

```
3 [ / 10 5 9 8 12 13
10 9 12 13
```

For non-commutative functional operands the results may be reversed from what a positive integer would yield.

```
2 ÷ / 1 2 3 5 10
0.5 0.6666666667 0.6 0.5
```

```
¯2 ÷ / 1 2 3 5 10
2 1.5 1.666666667
```

Illustration: A Moving Average

A moving average generally smooths out a given set of data. It is accomplished by taking an average over fixed adjacent portions of the given data

For

```
Sales←100 274 401 730 920 510
(4 +/Sales) ÷ 4
376.25 581.25 640.25
```

Exercise 99

- a) 2 +/ 1 13 5 9 15
- b) 3 +/ 1 13 5 9 15
- c) -2 +/ 1 13 5 9 15
- d) 2 -/ 1 13 5 9 15
- e) -2 -/ 1 13 5 9 15
- f) 3 -/ 1 13 5 9 15
- g) -3 -/ 1 13 5 9 15
- h) 2,/'a' 'b' 'c'
- i) -2,/'a' 'b' 'c'
- j) 0+/ 1 13 5

Did you notice that when the integer in the n -wise reduction is negative the data to be scanned is reversed before the reduction is done.

- k) Create a *Dfn* for the moving average example. If you wish, include the previous *RND* function to control the number of places to the right of the decimal point.
- l) In computational biology DNA sequencing, k -mers are subsequences of length $k < n$, in biological sequences of length n . For *GCAT*, the 2-mers and 3-mers are:

2-mers GC CA AT

3-mers GCA CAT

Create a *Dfn* for a vector DNA sequence of length n that will produce the set of k -mers of a specific length k .

In problem)m of the exercises for *Membership* on page 102, you were asked to build a *Selfie*. Modify it so that returns a sentence:

```
x←8 14 4 10 1
Selfie x
The vector is not a "Selfie" set.
```

```
y←10 14 12 5 7
Selfie y
The vector is a "Selfie" set.
```

Replication V/[I]R

The reduction operator symbol / has an additional role. The symbol / can also represent a function if **V** is numeric data. In this case **V/A** is called *Replication/Compression*. The data argument **V** may be either a scalar or a vector of integers of the same length as the argument **R**. The argument **V** acts as a mask against the argument **R**.

Replication is another way of selection items from an array. Where indexing selects out items from an array, *Replication* does the selection by discarding elements of **R** whose matching element in **V** is zero. **V** may contain positive, negative integers and zero. A zero would cause the matching element of the right argument **R** to be discarded.

Positive integers would cause the matching element of the right argument **R** to be replicated, items matching negative integers are replaced and replicated by the integer number of blanks for character data and zeros for numeric data. If the operand consists only of ones and zeros, the derived function is conventionally referred to as a *Compression* operator. When other integers for the left argument occur, the operator is called *Replication*.

```

      1 0 1 0 1 0/'ABCDEF '
ACE

      3 1 0 0 2 1/'ABCDEF '
AAABEEF

      1 -1 2 0 -2 2/'ABCDEF '
A CC FF

      2/'ABCDEF '
AABBCCDDEEF

      1 -1 2 0 3 -2/10×16
10 0 30 30 50 50 50 0 0

```

If the left argument is a single scalar, it is applied to each of the elements in the right argument.

Given a vector,

```

      ⍒←?10ρ100
90 68 98 66 72 27 74 44 1 46

      (v<50)/v
27 44 1 46

      ((v≥40)∧v<70)/v
68 66 44 46

```

The first compression selects all the numbers less than 50 and the second compression selects all the numbers in the half open interval 40 between 70 including 40 but not including 70; often indicated mathematically as [40 70).

Exercise 100

- a) $1\ 1\ 0\ 1\ 1\ 0\ 1 / 'abcdefg'$
- b) $1\ 1\ 0\ 1\ 1\ 0\ 1 / \iota 7$
- c) $1\ 1\ 0\ -1\ 1\ 0\ 1 / 'abcdefg'$
- d) $1\ -1\ 0\ 1\ -2\ 0\ 1 / 'abcdefg'$
- e) $1\ -1\ 0\ 1\ -2\ 0\ 1 / \iota 7$
- f) $2 / 'abcd'$
- g) $-2 / \iota 3$
- h) $(\iota 4) / \iota 4$

For the vector $v \leftarrow 2\ -2\ 0\ 3.1\ -4.5\ 0\ 13\ -2.8$:

- i) Write an expression listing all of the negative numbers in v followed by the positive numbers excluding zeros.
- j) Write an expression listing all of the negative numbers in v followed by the positive numbers including zeros.
- k) Write an expression to extract the odd numbered letters of $a \leftarrow 'abcdefghij'$

For the vector $v \leftarrow 2\ -5\ -0.7\ 9\ -12\ 12\ 0.7$:

- l) Write an expression to extract the negative numbers from v .
- m) Write an expression for the product of the negative numbers in v .
- n) Write an expression to produce the indices of the negative numbers in v .
- o) Given a vector of salaries, for example:

$$\text{salaries} \leftarrow 15560 \times ?12 \rho 10$$
- p) Create a *Dfn* to display which salaries exceed a given salary value.
- q) Write a *Dfn* that will replace all numbers less than five by zero.
- r) Write a *Dfn* that will replace all numbers between five and ten.
- s) Write a *Dfn* that will replace even numbers by zero.
- t) In problem m) in the exercises for *Membership* on page 102, you were asked to create a *Dfn* for a *Selfie* that would return a 1 or a 0 depending on whether the vector was a *Selfie* or not. Change the *Dfn* to return a proper sentence.

```

      x ← 8 14 4 10 13
8 14 4 10 13

```

```

      y ← 10 14 12 5 7
10 14 12 5 7

```

```

Selfie x
The vector is not a "Selfie" set.

```

```

Selfie y
The vector is a "Selfie" set.

```


Scan f\ [I] R

The *Scan* operator produces an array by applying the *Reduction* operator to successively more elements of its argument. Thus,

```

      ×\2 3 4 5
2 6 24 120

```

which is the same as

```

      2,(×/2 3),(×/2 3 4),×/2 3 4 5
2 6 24 120

```

```

      ,\ (1 2)(3 4)(5 6)
1 2 1 2 3 4 1 2 3 4 5 6

```

```

      B←0 0 1 1 0 1 0 1 0 1
      v\B
0 0 1 1 1 1 1 1      ⌘ All 1's after the first 1
      <\B
0 0 1 0 0 0 0 0      ⌘ The first occurrence of a 1
      ≤\B
1 1 0 1 1 1 1 1      ⌘ All 1's except the first 0

```

Illustration: Monthly Start Date

```

      Start_days←{⌘ ω:start day for Jan 1st as integer 0-6
      ⌘ α:1 for leapyear else 0
      7|+ω,31,(α+28),31 30 31 30 31 31 30 31 30
      }

      '2021:',0 Start_days 5
2021: 5 1 1 4 6 2 4 0 3 5 1 3

```

Also, f\ R will do the scan along the first dimension.

```

      r
1 1 6 2 1
2 3 5 2 6
5 3 3 2 6
8 2 5 4 5

      +\[1]r
1 1 6 2 1
3 4 11 4 7
8 7 14 6 13
16 9 19 10 18

      +\r
1 1 6 2 1
3 4 11 4 7
8 7 14 6 13
16 9 19 10 18

```

Exercise 101

a) $+\backslash 2 \ 3 \ 4 \ 5$

b) $-\backslash \iota 8$

c) $4 \ 2\rho -\backslash \iota 8$

d) $\wedge \backslash 2 \ 16 \ 32 \ 14$

e) $\times \backslash \ 2 \ 16 \ 32 \ 14$

$\vdash e \leftarrow 3 \ 4\rho \iota 6$

$1 \ 2 \ 3 \ 4$

$5 \ 6 \ 1 \ 2$

$3 \ 4 \ 5 \ 6$

f) $+\backslash e$

g) $+\backslash e$

h) $, \backslash e$

i) $, \backslash e$

Expansion N\[I]R

The symbol \ has a dual role, similar to that of /. It can represent a function if N is numeric data, a scalar or integer vector. The elements in R are expanded as determined by the left argument N.

The number of positive integers in N must match the length of R.

```

      t←3 2 1\5 10 15
5 5 5 10 10 15
      ρt
6
      3 2\'AB'
AAABB

```

Each positive integer determines the number of replications of the matching element in R. The length of the result is defined by N.

If 0's are intermixed in the left argument N, zeros or blanks are placed in the result matching their location in N:

```

      0 3 0 0 2\8 9
0 8 8 8 0 0 9 9

      0 3 0 0 2\'AB'
AAA  BB

      ρ0 3 0 0 2\'AB'
8

```

A single negative integer indicates the number of zeros or blanks to be placed in the result.

```

      -2 3 -3 2\8 9
0 0 8 8 8 0 0 9 9

      -2 3 -3 2\'AB'
AAA  BB
      ρ-2 3 -3 2\'AB'
10

      1 0 1 0 1 \3 4ρ12
1 2 3 4
0 0 0 0
5 6 7 8
0 0 0 0
9 10 11 12
      1 0 1 0 1 0 1\3 4ρ12
1 0 2 0 3 0 4
5 0 6 0 7 0 8
9 0 10 0 11 0 12
      1 0 1 0 1 0 1\3 4ρ'ABCDEFGHIJKL'
A B C D
E F G H
I J K L

```

Illustration: Integer Separation

Convert a positive integer into a numeric vector of its digits

```

0+ Isep 35597
3 5 5 9 7

▽ Isep←{A converts a positive integer into a vector
[1]      A of its digits
[2]      d←⌊ω
[3]      ⌊((2×pd)ρ1 0)\d
[4]      }
▽

```

Exercise 102

What is another function that will convert an integer into a vector of its digits.

Convert the vector of digits back into the decimal integer.

Illustration: Inserting character data between digits of a numeric vector

Now here is one using Expansion. This illustration illustrates how often an particular approach may not be the most effective way of solving the problem. The first solution was focused on using *Expansion* to insert the colon symbol.

For

```

x← 12 ^2.5 8.76 123
xe[⊔0=xe←((^1+(2×ρx))ρ1 0)\x]←':'
12 : ^2.5 : 8.76 : 123

```

Looking at the problem from an array point of view

```

^1↓ ∈(⌊x), ':'
12 : ^2.5 : 8.76 : 123

```

Exercise 103

Using either expression modify or create a different expression to allow more than one character to be inserted. For example:

```
12 -:- ^2.5 -:- 8.76 -:- 123
```

Exercise 104

- a) `3 2 3\10 20 30`
- b) `3 ^2 3\10 20 3`
- c) `3 ^2 2 3\10 20 3`
- d) `3 ^2 2 0 3\10 20 3`
- e) `3 2 3\'ABC'`
- f) `3 ^2 2 3\'ABC'`
- g) `ρ3 ^2 2 3\'ABC'`
- h) `3 ^2 2 0 3\'ABC'`
- i) `ρ3 ^2 2 0 3\'ABC'`
- j) `3 ^2 2\'A' 12`
- k) `ρ3 ^2 2\'A' 12`
- l) `3 ^2 2\12 'A'`
- m) `ρ3 ^2 2\12 'A`
- n) `2 ^2 2 0 2 0\12 'A' 5`
- o) `ρ 2 ^2 2 0 2 0\12 'A' 5`
- p) `2 ^2 2 0 2 0\'A' 12 5`
- q) `ρ2 ^2 2 0 2 0\'A' 12 5`
- r) `1 0 1 0 1 0 1 \3 4ρι12`
- s) `1 0 1 0 1\3 4ρι12`
- t) Create a *Dfn*, which given a character matrix, will return the matrix with its columns separated by a blank column.

Each $f \ddot{R} - L f \ddot{R}$

This appropriately-named monadic *Each* operator, $\ddot{}$ (*Ctrl+I*), is one of the most powerful and popular operators. It applies its derived function to the elements of its array argument one level down in the array structure. This an extremely useful operator.

For example, with many sets of items often one wishes to apply the same function to each of them. Consider the three following vectors:

```

      v1←?3p25
17 22 16
      v2←?4p20
15 1 11 4
      v3←?5p50
28 40 17 38 34

```

Given these three vectors, to find the largest element in each of them, max reduction, $\lceil /$, could be applied to each of them,

```

      (⌈/v1), (⌈/v2), (⌈/v3)
22 15 40

```

Or one could apply the monadic operator *Each* $\ddot{}$ (*Ctrl+I*) with the derived function, $\lceil /$, as its operand to the vectors $v1$, $v2$, and $v3$.

```

      ⌈/⍥ v1 v2 v3
22 15 40

```

The monadic operator *Each*, $f \ddot{R}$, accepts any result producing monadic function f as its left operand and treats f as a scalar function applying it to “each” of the items of R *one level down in the structure* of the argument R . Again, for example:

```

      p⍥v1 v2 v3
3 4 5
      p v1 v2 v3
3

```

The operator *Each* can also take on a dyadic form, $L f \ddot{R}$.

Here, the dyadic operand f must be a result-producing function.

For example,

```

      3 2 p⍥ 'AB'
AAA BB

```

Again,

- *The operand of any operator may be any result producing primitive, defined, or derived function.*
- *In all situations, the operator *Each* executes its operand ONE level down in the data structure.*

To penetrate deeper into a structure, one may need a further *Each*.

Dyadic *Each* extends *scalar conformability* to non-scalar functions, too. In this case, one of the arguments must be a scalar or $(\rho L) = \rho R$. The scalarization of an argument can be done via the function *Enclose*. Often then one of the arguments L or R must be enclosed.

Consider,

```

      3 2ρ'AB'
AB
AB
AB

      3 2 ρ''AB'
AAA BB

      (⊃3 2) ρ''AB'
AA BB
AA BB
AA BB

      3 2 ρ''⊃'AB'
ABA AB

      3 2 4ρ''⊃'AB'
ABA AB ABAB

      3 2 4ρ'' 'AB'
LENGTH ERROR
      3 2 4ρ''AB'
      ^

      v←((2 5) 'abc') ((2 3ρ⊂a)(⊔5))
      ρv
2

      ρ''v
⎕⎕

      ρ''''v
⎕⎕⎕⎕⎕⎕

```

Illustration: The number of 1s, -1s, and 0s in an array

Given an array v such as

```

      v ← [-5, 4, 4, 10]
      [-2, 2, 0, 0]
      [3, -3, 5, 3]
      [1, -3, 4, -3]
      [-1, -1, -3, 5]

```

The expression

```

      +/'1 -1 0 = c × v
      7 7 2

```

will determine the number of 1s, -1s, and 0s in an array.

Illustration Appending 'Mr. '

The goal is to append the title 'Mr.' to the names 'Smith' 'Chan' 'Prez'.

```

      'Mr.', ''Smith' 'Chan' 'Prez'
      MSmith rChan .Prez

      'Mr. ', ''Smith' 'Chan' 'Prez'
      LENGTH ERROR
      'Mr. ', ''Smith' 'Chan' 'Prez'
      ^

      p ('Mr. ') ('Smith' 'Chan' 'Prez')
      4 3

      (c 'Mr. '), ''Smith' 'Chan' 'Prez'
      Mr. Smith Mr. Chan Mr. Prez

```

Enclosing 'Mr. ' makes it a scalar, and thus $S \neq V$.

Illustration: Sentence Construction

```
[0] Sentence←{⍺ ω:builds a sentence from a vector
[1]      ⍺ of words; subject verb noun
[2]      (?ρω)⊃ω
[3]      }
      subjects ← 'Ray' 'Anne' 'David' 'Jo'
      verbs ← 'likes' 'eats' 'dislike'
      nouns ← 'fish' 'oatmeal' 'apples' 'beets'
      Sentence subjects verbs nouns
Ray likes beets
Sentence subjects verbs nouns
Jo likes beets
Sentence subjects verbs nouns
David eats fish
```

Exercise 105

- a) Determine the value, shape, depth and rank
 - a-1) `ι3`
 - a-2) `ι"4 3`
 - a-3) `]display ι"4 3`
 - a-4) `ι 3 4`
- b) Determine the value, shape, depth and `]display`
 - b-1) `+/10 20 30 40`
 - b-2) `+/(10 20)(30 40)`
 - b-3) `+/"(10 20)(30 40)`
 - b-4) `+/(10(20 30))(40 50)`
 - b-5) `+/"(10(20 30))(40 50)`
 - b-6) `+/(10 20)30 40`
 - b-7) `+/"(10 20)30 40`
- c) Given the daily number of copies three copy machines made on each of the five work days of the week,


```
copya←345 401 388 391 300
copyb←397 423 422 450 387
copyc←345 321 367 381 376
```

Write an expression for the following

 - c-1) The maximum of daily copies each copy machine made during the week.
 - c-2) The total number of copies each machine made in the week.
 - c-3) The grand total number of copies made that week.
 - c-4) The daily percentages of the total weekly output for each machine.

Outer Product $L \circ . f R$

With the introduction of matrices and other higher rank arrays it is time to introduce another operator, *Outer Product*. We have already determined that an expression like

```
5 10 20×4 3 2 will produce
20 30 40
```

Now suppose you wish to multiply 4 3 2 by just 5 and 10.

```
5 10×4 3 2
LENGTH ERROR
5 10×4 3 2
^
```

It didn't work did it?

The operator *Outer Product*, $L \circ . f R$, will help. The symbol for Outer Product is the *jot* \circ (*Ctrl+J*) followed by a *period*. This combination $\circ .$ is referred to as “*jot dot*”.

This operator applies its dyadic operand f between the two arguments L and R in an “*Each with Every*” manner. In doing so, it builds an array where the first dimension is the shape of L and the other dimension is the shape of R .

This operator is often referred to as a table builder.

```
5 10 \circ . \times 4 3 2
20 15 10
40 30 20
```

where here the result is a 2 row 4 column matrix. This operator is often associated with the phrase “*Each with Every*.” The dyadic Outer Product operator takes the elements of the left argument and applies them individually via the dyadic functional operand f in a row-wise fashion to each of the elements of the right argument. In the previous example, the first row of the matrix results from 5 multiplying each item of the right argument, 4 3 2. The second row results from 10 multiplying each item of the right argument. Formally, the shape of the Outer Product of $L \circ . f R$ is:

$(\rho L), \rho R$.

Illustration: Number of Vowels in a Sentence

```
text←'For score and seven years ago.'
+/'aeiou'\circ . =text
3 4 0 3 0
```

Illustration: Evaluating a Polynomial

Consider the equation $z = x y^2 \div w^{1/3}$

Where you wish to find the set of values for z, given

```
x ← 1 2 3
y ← 2 4 6 8
z ← 10 100
```

Then

```
z ← x ◦ . × ( y * 2 ) ◦ . ÷ w * ÷ 3
```

and

```
]display z
```

1.856635533	0.861773876
7.426542134	3.447095504
16.7097198	7.755964884
29.70616854	13.78838202
-3.713271067	-1.723547752
-14.85308427	-6.894191008
-33.4194396	-15.51192977
-59.41233707	-27.57676403
5.5699066	2.585321628
22.2796264	10.34128651
50.1291594	23.26789465
89.11850561	41.36514605

Illustration: The number of 1s, -1s, and 0s in an array

Now, using the outer product:

```
r ← -5 + ? 4 4 p 10
```

1	5	4	1
-1	0	3	-2
-4	0	0	-2
2	-2	2	-3

```
+ / 1 -1 0 ◦ . = , × v
```

7 6 3

Illustration: A Histogram

A histogram for a simple vector can be generated as follows:

```
Histo←{⌈ω; simple numeric vector
      ⌈α:Char symbol to be used
      (' ',α)[1+(⌈⌈ω)∘.≤ω]
      }

⌈ v← ?6ρ8
4 5 3 4 8 4

' ' Histo v
  □
  □
  □
  □
□ □
□ □ □
□ □ □ □
□ □ □ □
□ □ □ □
```

Illustration: Repeated Powers

Given a simple numeric vector, **V**, create a matrix of initial powers of each the values of **V**.

For example,

```
V←.2 4 ^5

V∘.*⌈4
0.2  0.04  0.008  0.0016
4    16    64    256
^5    25   ^125  625
```

Compare with the *Repeated Powers* illustration on page 118 under the function *Enclose*.

Often in this language there several different ways of solving the problem.

Exercise 106

- a) `5 10 ◦.+ 1 5 10`
- b) `ρ 5 10 ◦.+ 1 5 10`
- c) `23 5 ¯12 ◦.⌊ 25 2 ¯16`
- d) `3 7 8 ◦.⌈ 5 (6 4)`
- e) `1 2 ◦., 'abc'`
- f) `1 2 ◦., 'abc' 'xy'`
- g) `2 4 ◦.ρ 'ABC'`
- h) `ρ 2 4 ◦.ρ 'ABC'`
- i) Write the expression to produce all nine pairing of the letters 'APL'.
- j) An n by n *Vandermonde Matrix* is a matrix that using n arbitrary numbers, x_1, x_2, \dots, x_n , creates an n -by- n matrix established by successively raising the n numbers column-wise to the powers $0, 1, 2, \dots, n-1$.
For example, if the numbers were 3 2 4 5, the Vandermonde Matrix would be:

```
1 3 9 27
1 2 4 8
1 4 16 64
1 5 35 175
```

Inner Product $L \mathbf{f} . \mathbf{g} R$ **“Matrix Product”**

The *Inner Product* operator is a very powerful operator, since it combines any two result-producing dyadic functions sequentially. The functional right operand \mathbf{g} combines the arguments of L and R in a scalar fashion, element by element, then \mathbf{f} *reduction* applies to that intermediate result. First, consider the simple vector case,

$F \leftarrow 3600 \ 60 \ 1$ and $G \leftarrow 5 \ 10 \ 20$

where F represents a vector of the number of seconds in an hour, minute, and seconds and G is the vector of hours, minutes, and seconds.

$F + . \times G$
 18621

which represent the number of seconds in 5 hours, 10 minutes, and 20 seconds.

For the situation where F and G are vectors, this could also be formally represented as

$c \leftarrow F / F \times G$

For arrays, it is slightly more involved. With matrices, the function \mathbf{g} is applied in a scalar fashion, rows of L versus columns of R followed by a \mathbf{f} reduction. In general then, for arrays,

The rightmost dimension of L must match the leftmost dimension of R or L or R is a scalar.

For example,

$\mathbf{F} \leftarrow 2 \ 3 \ 1 \ 6$
 1 2 3
 4 5 6

$\mathbf{G} \leftarrow 2 \ 3 \ 5 \ 10 \ 100 \ 2 \ 4 \ 6$
 5 2
 10 4
 100 6

$\mathbf{F} + . \times \mathbf{G}$
 325 28
 670 64

In mathematics, $L + . \times R$ is called an *Inner Product*.

The dyadic function \mathbf{f} does not have to be a scalar function.

$'ab' \ 'cd' \circ . \cup 10 \ 20 \ 30$

ab 10	ab 20	ab 30
cd 10	cd 20	cd 30

Illustration: Best Price

Two departments have different requirements on the same items,

```
DA←36 144 230
DB←85 410 751
```

The purchasing department has three vendors with the following package prices for the items needed by the departments DA and DB:

```
V1←4.25 3.25 .75
V2←4 3.5 .55
V3←4.7 3 .5
```

Create the matrices:

```
Demand←2 3p DA,DB
Vendors←V1,V2,⋮V3
```

Recall that **V1**, **V2**, and **V3** must appear as columns of **Vendors** for inner products.

The inner product

```
Prices ← Demand +.× Vendors
```

produces the costs for each department against the vendors.

```
Prices
793.5 774.5 716.2
2257 2188.05 2005
```

Now suppose that the departments had delivery needs for their items.

```
Need←2 3p(100 6 15)(12 7 11)
100 6 15
12 7 11
```

and the vendors had delivery schedules:

```
Deliver←(10 5 12),(8 7 9),⋮(15 6 11)
Deliver
10 8 15
5 7 6
12 9 11
```

Then:

```
Need∧.≥ Deliver
1 0 0
0 1 0
```

The ones in the binary matrix indicate which vendors can the delivery demands of the departments.

There are several more useful examples

Mat ∨.= ' '	⌞ Rows containing blanks
0 ∨.= Mat	⌞ Columns containing zeros
Mat ∨.≠ ' '	⌞ Rows with at least one non-blank character
Vec <.> 1φ Vec	⌞ Tests a numeric vector if it is in ascending order
No. <.> Limit	⌞ Test for a number being between a 2-item limit

Illustration: Pillow Problem #32 (07/04/1889)

Charles Dodgson (better known as Lewis Carroll) created a book entitled “*Pillow Problems Thought Out During Wakeful Hours*”.

His problem #32 states

Sum the series $1 \times 5 + 2 \times 6 + 3 \times 7 + \dots$ (1) to n terms and (2) to 100 terms.

A *Dfn* that would solve his problem is:

```

      LC32a←{aLewis Carroll prob. 32
[1]      aω: pos. integer
[2]      □IO←1
[3]      a +/v×4+v←1ω
[4]      v+.×4+v←1ω
[5]      }
```

Note that line [3] illustrates the definition of inner product in the simple vector case:

```

      LC32a 4
70
```

```

      LC32a 100
358550
```


Illustration: A product series evaluation

In *Mathematics Magazine*, Vol. 95, No. 3, June 2022, pg 242, problem 2147 stated:

Evaluate:

$$\prod_{n=2}^{\infty} \frac{n^4+4}{n^4-1}$$

An APL solution is:

```
pr2147←{⎕ prob2147 math mag52-1jun22
(4+ω*4)×.÷~1+ω*4
}

pr2147 1+ι10
1.468826763

pr2147 1+ι100
1.470428821
```

Exercise 107

Approximately how many terms in the product will it take to get to 8 places of accuracy?

Exercise 108

- a) If x y were symbols in mathematics, write the ordinary symbolic mathematical statement representing this APL expression: `(v+. *2)*.5` with `v←5 12`.
- b) With `a←3 4p1 0 0 0 0` and `b←4 3p12`, evaluate `a+. *b`.
- c) The expression `0v.= mat` indicates the columns that contain zeros. What does the expression `matv.= 0` do? Try it with `⌈mat←-2+?3 4p9`.
- d) What if you replaced the `0` in the previous problem by a blank, `' '`, what could you be testing for?
- e) Create a *Dfn* that will return the row numbers of the rows that contain zeros.
- f) The sum of the reciprocals of 5 raised to successive powers of \mathbb{N} is said to be .25; that is:

$$\sum_{i=1}^{\infty} \frac{1}{5^i} = 0.25$$

How many terms in the series will it take?

- g) Given:

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

- g-1) What single APL function would produce the same result as `k+. *h` ?
- g-2) What single APL function would produce the same result as `h+. *k` ?

Summary 9: Operators

The introduction of operators introduces unique power to the language.

- a) Operators in general have functional operands.
- b) Operators produce derived functions.
- c) Operators may be either monadic or dyadic.
- d) All operators will have a left operand; a monadic operator thus will have *only* a left operand.
- e) Operators will accept any result producing primitive, user defined, or derived function.
- f) The *Reduction* operator, $f /$, distributes its operand, f , between the items of its argument and then scans right to left.
- g) The *N-wise Reduction* operator is dyadic. Its integer left argument defines the width of a moving window in which the operand is applied. If the integer left argument is negative, the data in the window is reversed before the operand is applied.
- h) The *reduction* operator symbol, $/$, also may take an integer vector as its operand and is called *Compression/Replication*.
- i) The integer vector for *Compression/Replication* can act as a mask applied to the argument with 0s discarding and 1s retaining its matching item in the argument. Positive integers replicate its matching argument. Negative integers replace matching items in the argument by zeros or blanks based on the type of the item.
- j) *Axis specification* considered as an operator changes the axis along which its functional operand executes.
- k) *Outer Product* $L . f R$ performs in an “Each with Every” fashion. The right operand executes between all combinations of the left data with all combinations of the right data.
- l) *Inner Product* $L f . g R$ performs an f -reduction on the result of a $L g R$.
- m) *Each* $L f '' R$ executes its operand on the data one level in the data structure.

For further information on all of the current operators reference Chapter 2 in the *Dyalog APL Language Reference Guide*.

Now you, the reader, may be impatiently asking what can I do with such capability. Patience is being asked of you. At this stage of the presentation, the objective was to make you aware of the existence of some of the many powerful primitive functions and operators. You see, if you are not aware of them you would not use them. In many cases you may be able to achieve your solution with more lengthy elementary means, since in APL, there are usually several ways of achieving a correct solution. Now you have a rich APL vocabulary.

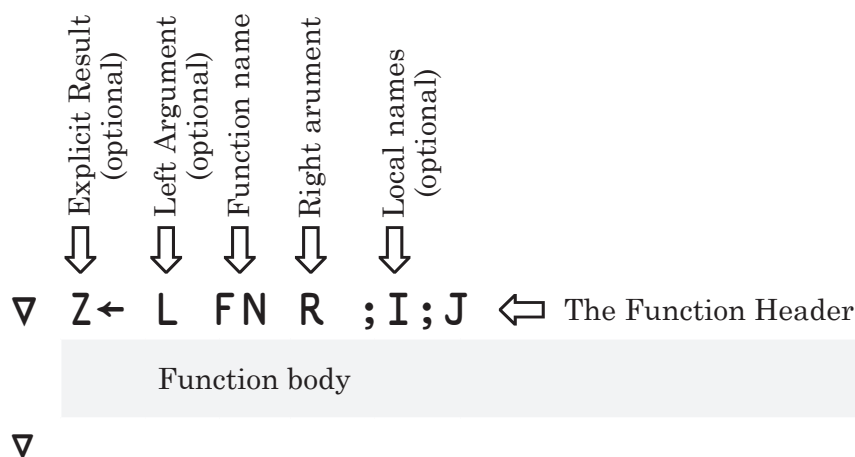
But there is always more.

Now let us look at Procedural Programing.

A Return to Programming

Up to this point, we have been introducing a variety of functions and operators that you may use in creating programs. Earlier, *Dfns* functional programming was introduced. While it is simple to create and use, it is unique to Dyalog APL. There is another form of creating programs that is more traditional across the programming world. We shall refer to this type of program creation as *Procedural Programming*. Such programs will also be called *Traditional Functions* and often referred as *tradfns*. This form of building programs may be more suitable to larger-scale problems. In this form of programming the defined functions is in two parts;

- a) The *header* which contains the name of the defined function and determines the nature of its arguments and how it may be used.
- b) The *body* which contains the executable code. Pictorially,



The `∇` symbol is a way of getting to an editor from earlier time. Using `)ed` is more recommended.

For example:

```

∇ z←Seconds ms
[1]  A ms: 2-item vector of minutes seconds
[2]  A z: the equivalent no. of seconds
[3]  z←ms[2]+ms[1]×60
∇
    Seconds 3 5
185

```

This function was created by going to the editor via `)ed Seconds`.

Exercise 109

- a) Create this function *Seconds* in your workspace.
- b) Suppose there is a set of letter grades A, B, C, D, and F. If there numeric values are 4, 3, 2, 1, and 0 respectively, create a *defined function* to determine the grade point average of a set of letter grades. For example,

```

gpa1 'CBCBA'
2.8
gpa1 'C' 'B' 'C' 'B' 'A'
2.8

```

The header of the defined function, the first line, comes in several forms. The most useful forms are

<code>Z ← L FName R</code>	The dyadic form
<code>Z ← FName R</code>	The monadic form

For example, this is how the `FtoC Dfn` would look as a defined function;

```
[0]   z←fTOc t
[1]   #Fahrenheit to Celsius
[2]   z←(5÷9)×t-32
```

The inclusion of the `z←` in the header is important since it states that the defined function will have any result. It requires that somewhere in the body of the defined function there will be a statement containing `z←` indicating that this is to be the output of the function. There are several other forms of the header which will not be covered here.

The inclusion of the assignment symbol, `z←`, in the header of the function is necessary for it to be used like a primitive function. This is automatic for *Dfns*. For either case, being able to use the function like a primitive function means its result may be:

- Displayed immediately,
- Assigned a variable name,
- Used in a larger expression,
- Used as an operand of an operator.

One extra item needs to be included in the header. It is possible within the body of a defined function to change the value of a variable outside of the function from within the defined function.

Suppose you write in the workspace

```
V← 'YOU'
```

and within the body of the defined function `FOO` on some line there exists

```
V← 'ME'
```

After executing `FOO`, typing `V` is `ME` not `YOU`. Let's do it. Go to the editor with

```
)ed FOO
```

In the edit window, type

```
z← x FOO y
V← 'ME'
z←x+y and then close the function.
```

The line numbers on the left side in the editor are placed there by the editor.

As with *Dfns* the `Esc` key closes the function.

Now execute the following sequence:

```
V← 'YOU'
ans ←5 FOO 3
V
```

and now `YOU` becomes `ME`.

Such behavior can cause confusion. To prevent such behavior. It is wise to declare any variable used within the body of a *defined function* as local to the function. Then it will exist having meaning only within the execution of the function.

Declaring a variables *local* to a function is done by including the variable name in the header of the defined function. Such variable names are placed to the right of the right argument and are separated from it by a semicolon. Also, each local variable is separated from other local variables by semicolons.

In *Dfns* this is not necessary since any variable created within it is automatically local.

Exercise 110

Localize **V** in the function **FOO**.

When it comes to making a choice as to which form of program to create, it is an open choice. Note that one may embed in the body of a *Dfn* calls to *Defined Functions* and conversely one may call *Dfns* within the body of a *Defined Function*.

Control Constructs

At this point whatever programs you create its statements will be executed sequential. Each statement within the program will be executed one after the other; the first followed by the second followed by the third and so on. Not all programs can act that way. Often there are situations where rather than executing the next statement in sequence, a different set of statements needs to be executed. This requires that the program requires to go to a different location in the program. Such an action is referred to as *Conditional Branching*. For example, the program may need to treat positive numbers differently from negative numbers. To do this, there needs to be a conditional test that will allow this conditional branching. However, with the array nature of APL much can be accomplished without conditional branching.

Computer instructions are either

data transforming or
sequence controlling.

A control construct is that portion of a programming language that determines the order of execution of the program's instructions. This is often referred to as "*branching*". To this point in this material, the programs have been "straight line programs"; one instruction is executed sequentially after another. Sequence controlling is especially important in non-array languages or languages that are very close to controlling the hardware of the computer.

All programming languages have a form of branching. One form is involved in decision-making. It is called *conditional branching*. Based on the nature of the data or condition occurring, the program has to decide which set of instructions it is to execute.

For example: If a number is negative then take its absolute value, otherwise (Else) continue on.

A second form of branching is called *iterative branching*. Here you wish to step through a set of data and carry out the same set of operations on each item of the data. Such branching is necessary for sequential programming languages such as Python. In these languages the concept of branching is introduced early in the presentation of the language. Only very elementary actions are possible without the introduction of some form of branching.

Where is branching useful or necessary?

Algorithms where the execution of each step toward the solution depends upon the previous step often uses a control construct. This type of activity is referred to as *iteration* in which each time the program steps repeatedly through the algorithm as it approaches the solution.

Another common situation for branching occurs when there exists a large amount of data needing to be updated and all of the data will not fit in the computer's memory.

Finally errors occur in the data. In such situations the program has to detect the erroneous data and react differently.

From the very beginning of APL, there has been a branch symbol, \rightarrow , which could do unconditional (‘go to’) branching and conditional (“if this statement is true, go to”) branching. In this form of branching, the branch is to a labeled line in the function. One may still use it today; however there is a large set of more efficient APL control constructs which are more efficient. These APL control constructs match similar control constructs found in other sequential programming languages. These control constructs described here are for use within procedural programs.

Direct functions, Dfns, only have *guards*. See page 38.

A large set of control constructs exist in APL that may be used within Procedural Programming were sequential programming may be necessary.

Let us examine the control constructs available in APL.

Branching and Iterating

The control word structure consist of four parts:

- A beginning Control word prefaced by a colon :,
- A binary control expression,
- A block of executable. Statements
- A matching ending control word prefaced by a colon.

The control words are case insensitive. The binary control expression must evaluate to a 0 or 1.

- If it is 1, the block of statements are executed.
- If it is 0, the block of statements are not executed.

The block of statements are delineated by an initiating control word and a terminating matching control word. For example

```
:If ...
    (Block of statements)
:Endif
```

Primary Control Words	
<i>Initiating</i>	<i>Terminating</i>
:If	:Endif
:For :In	:Endfor
:Select	:EndSelect
:Repeat	:Until or :EndRepeat
:While	:Until or :EndWhile
:Trap	:EndTrap
:With	:EndWith
:Hold	:EndWith

Supportive Complementary Control Words
:Else
:ElseIf
:AndIf
:OrIf
:Case
:CaseList
:Until

Exception Control Words
:GoTo
:Return
:Leave
:Continue

Illustration: Control Word Formats

The **:If** **:Else** control situation:

```
:If cond1
|
|   executed if cond1 is satisfied
|
:Else
|
|   executed if cond1 is not satisfied
|
:EndIf
|
```

The **:If** **:OrIf** control situation:

```
:If cond1
|
|   executed if cond1 is satisfied
|
:OrIf cond2   executed if cond1 is not satisfied
|
|   executed if cond2 (or cond1) is satisfied
|
:EndIf
|
```

The **:While** *control situation*

```

:While conditional function limit
|
|   code executed repeatedly until
|   the relationship is not satisfied
|
:EndWhile

```

The **:Repeat** *control situation*

```

:Repeat
|
|   code is executed repeatedly until
|   the relationship is satisfied
|
:Until conditional function limit
|

```

The **:Select** *control situation*

```

|
:Select itemx
:Case a
|
:Code list b c d
|
|   code is executed if itemx
|   is equal to b, c, or d
|
:Else
|
|   code is executed if itemx
|   is not equal to a,b, c, or d
|
:EndSelect

```

Let us view some of these control word formats in action. Remember that these illustrations are here to show how one might use them in an APL procedural program. There are more direct ways using the *array orientation* of the language.

Illustrations

- 1) This is a problem often used in sequential programming to introduce branching. Given a vector (list) of positive integers **V** and a positive integer **N**, modify all the integers in **V** such that they are all less than **N**.

One iterative solution would be

```

      ∇ z←n LessThan v;i
[1]   An: a number
[2]   Av: vector of numbers
[3]   A for no≥n,the modulo of it is taken
[4]   z←0
[5]   ⍳IO←0
[6]   i←0
[7]   :While I<pv
[8]       :If v[I]≥n
[9]           z←z,n|v[I]
[10]      :Else
[11]          z←z,v[I]
[12]      :EndIf
[13]      i←i+1
[14]  :EndWhile
      ∇

```

```

      test←2 5 12 10 123 45 9
      10 LessThan test
2 5 2 0 3 5 9

```

Of course, the *Dfn* to do this is:

```

      lessthan←{α|ω}
      10 lessthan test
2 5 2 0 3 5 9

```

2) *Triangular Numbers*

Preface: Triangular numbers are the sequence of integers which, when representing objects, allow the objects to be arranged in a triangular fashion. Thus 6, a triangular number, could represent the following triangle :

```

0
0 0
0 0 0
```

A partial sequence of such triangular numbers is

1 3 6 10 15 21 28 36 45 55 ...

These integers are generated by adding sequentially the integers, thus

1 1+2 1+2+3 1+2+3+4 1+2+3+4+5 ...

Problem: Create a *Dfn* that will generate the first *n* terms of a triangular integer sequence.

This first solution uses the **:Repeat** construct:

```

      triang 5
1 3 6 10 15

      ▽ z←triang n;j;k;t
[1]  Az: sequence of triangular nos
[2]  An: number of terms
[3]  z←''
[4]  (j k)←1
[5]  t←0
[6]  :Repeat
[7]      :Repeat
[8]          t←t+j
[9]          j←j+1
[10]      :Until (j>k)
[11]      k←k+1
[12]      z←z,t
[13]  :Until (k>n)
      ▽
```

A second solution is an array solution.

```

      +/'i''i5
1 3 6 10 15
```

A third array solution uses the primitive function scan.

```

      +\i5
1 3 6 10 15
```

Exercise 111

- a) Modify the *Dfn fccf* so that should α not be an ‘f’ or ‘F’ and a ‘c’ or ‘C’, the function produces the message “Not a valid request”:

```
'a' fccf1 212 32
Not a valid request
```

- b) Write the expression for the radian equivalent of one degree. Angles are given in degrees or radians. The two are related in that π radians = 180 degrees

The number π , called Pi, is a nonending nonrepeating decimal number that appears frequently in scientific work and as such appears in APL as a primitive monadic function whose symbol is \circ (*Ctrl+o*). $\circ n$ represents *Pi times n*. For example:

```
o1
3.141592654
o2
6.283185307
```

- c) Create a function that will, given an angle in degrees, produced the equivalent radian value or given the angle in radians produce the equivalent angle in degrees using the relationship that π radians = 180 degrees.

Session Input/Output

Many APL programs are presented data at the time of their execution. However, there are situations where data may be requested during the execution of a function. For example, the function may be forced to wait for a name or password. There are two functions—`⎕`, *Quad evaluated I/O*, and `⎕`, *Quote-Quad character I/O*—that are useful here.

They may be used either to the right or left of assignment. When `⎕` is used independent of assignment or to the right of it, APL pauses and issues the prompting symbol `⎕:` and waits for data entry. If the response is to be character data, the data must be enclosed by quotation marks.

Exercise 112

Issue and respond to the `⎕:` with some numbers:

- a) `Result←⎕`
- b) `Result`
- c) `a←10× ⎕ +5`
- d) `a`

Issue and respond to the `⎕:` with characters:

- e) `data←⎕`
- f) `data`
- g) `b←'Hi ',⎕`

When `⎕` appears to the left of assignment the data to the right of assignment is displayed. For example,

```

      ⎕←x←3+⎕←10×25⌈12
250
253
      x
253

```

Note that when there are repeated quote quads as in the above expression each quote quad is displayed on a separate line.

Observed that interspersing quote quads in a lengthy expression could be a useful scheme for seeing the intermediate results.

`⎕` may also be used either to the right or left of assignment. When `⎕` is used independent of assignment or to the right of it, APL pauses but does not exhibit any prompting symbol. It simply pauses and waits for input which will always be character information. For example,

```

      x←'Your name is ',⎕
ray
      x
Your name is ray

```

In the case of replying to repeated *quads* each response appears on a different line. In the case of repeated *quote-quads*, the responses appear on the same line. Thus,

```

      x←⎕⎕
⎕:
      45
⎕:
      12
      x
      12 45

      x←⎕⎕
Hi
Jim

      x
Jim Hi

```

Both `⎕` and `⎕←` are often used with control constructs.

Illustration

```

      ▽ z←textentry x;t
[1]  A A function to enter a long character string-'text'
[2]  A the exit is via just the ENTER key
[3]  Ax: text to be entered (appended to previous text)
[4]  A first entry for x should be ''(empty) thus textentry ''
[5]  Az: the new character vector.
[6]  z←x
[7]  'Exit from the input loop via the ENTER key alone'
[8]  :Repeat
[9]  t←⍬
[10] z←z,t
[11] :Until 0=ρ,t
      ▽

```

```

      data←textentry ''
Exit from the input loop via the ENTER key alone
Hello
how are
today

```

```

      data
Hello how are today?

```

Observe that the *Enter* key response to `⍬` on line 9 is interpreted as an empty vector, a vector with no data. Then in line 11 when APL asks for the shape of it, response correctly is 0; it has no data associated with it.

The symbol \emptyset represents the empty numeric vector.

Consider the following illustration for its use.

Illustration

This is a simple function illustrating prompting to enter a set of numbers resulting a vector.

```

      ▽ z←NumInput1;t;x
[1]   ⍵←'To exit type  $\emptyset$  (ctrl+shift+)]'
[2]   A ⍵← is not really necessary
[3]   z← $\emptyset$  A  $\emptyset$  is the empty numeric vector
[4]   over:
[5]   t←⍵
[6]   :If 0≠t
[7]       :Return
[8]   :EndIf
[9]   z←z,t
[10]  :GoTo over
      ▽

```

```

      data←NumInput1
To exit type  $\emptyset$  (ctrl+shift+)]
⍵:
    12
⍵:
    345 67 89
⍵:
     $\emptyset$ 

      data
12 345 67 89

```

Procedural Programming

- The nature of Procedural Programming,
- A large set of Control Constructs for Procedural programs,
- The functional guard for Dfns,
- Session Input/Output via `⎕` and `⎕`,
- `⊖` for an empty numeric vector.

Before we begin reading APL functions and operators in context of problems, let us pause and review the functions and operators that have been covered. In doing so, we could arbitrarily subdivided into how they are involved with data. Suppose the categories are

- Data Transformation
- Array Construction
- Array Inquiry
- Data Selectors and Selection
- Data Rearrangement
- Data Replacement

Exercise 113

Using the Table of Contents, distribute the functions and operators that have been introduced among the above categories.

Conclusion

As an author, I find it difficult to stop without sharing one more thing with you, but this is a proper place to stop. As in any spoken language, one needs to establish one's vocabulary. You only use the verbs, adverbs and nouns that you have learned. This the same with the APL programming language. Most of the primitive verbs (functions) and adverbs (operators) have been covered. But there is much to be covered especially in the APL system that supports the language. Many of the things not covered especially support productive aspects of APL.

What is over the Horizon?

Here are further aspects of Dyalog APL that can be explored.

- Several of the more powerful operators:
Power \star ; Rank \circ ; Commute \sim ; Beside&Bind \circ ; Over $\overleftarrow{\circ}$; A $@$; Key \equiv ; Stencil \boxtimes
- The ability to define your own Operators
- The User Commands
- The System Functions and Variables
- Namespaces
- Graph features
- File handling ability
- Interfaces with R and Python
- Using RIDE
- Running on the Raspberry Pi
- Features communicating with the non-APL computing world

Yes, there is much more there to meet your needs.

In summary, DAPL is a very beautiful and powerful programming language within an excellent supporting APL environment. I hope you have found it so.

Reading APL

A large part of the APL language has been covered. At this point you should be able to do significant APL programming. As with many languages, gaining the ability to write in a language is more difficult than reading the language. This is also the case with programming. Gaining skills in creating a program can be aided by reading existing programs. This section will give you the opportunity to read program code.

The pattern to be followed is the following:

- A statement of the problem is given.
- A solution to the problem follows.

In fact, there are probably several different solutions to many of the problems. The author would be pleased to know of them.

This material may also help answer such questions as “What can I do with APL?”

Note: It is intended that the Reading examples be selective depending on the level of the students. Perhaps a different set could be provided.

(For many problems, further comments will be forthcoming after each of them.)

Shirley

In the 1950s, the University of Illinois was building the Illiac and the Ordvac computers. A programming class was offered in 1953 to a limited number of faculty and graduate students. “Shirley” was one of the problems in this first Illiac problem set. Here is the verbatim problem statement:

“Shirley’s weight at birth was a ton. Her weight increased linearly at 8 oz. per week for 20 weeks and thereafter the rate (of increase) decreased by the factor b (<1) every 4 weeks. Find her weight after 16 weeks and after 52 weeks and put the results in 0000 and 0001 respectively. (16oz=1 lb, 2000lbs=1 ton)”

Shirley still waits for your solution.

Find her birth weights on the specific weeks.

Complex Number Translation

In APL, complex numbers are written as xjy where x is the real part and y is the imaginary part; for example ; 2j3, -2j3, 2j-3, or -2j-3.

In mathematics, complex numbers are often written as $x+yi$.

The following Dfn converts a mathematical complex representation $x+yi$ into a real APL complex number xjy for $x, y \geq 0$.

```

Complex←{⍺ converts math complex numbers x+yi with x&y>0
[1]      ⍺ to APL complex numbers xjy
[2]      ⍺ω is char vec. 'x+yi'
[3]      w←ω~'i'
[4]      ((⊔'+ '=w)[]w)←'J'
[5]      ⊔w
[6]      }

Complex '3+4i'
3J4

```

Exercise 114

Modify **Complex** to also handle negative complex numbers in math notation such as “-2-4i” or “-2+4i” into APL notation.

Grade Point Average

Suppose there is a set of letter grades A, B, C, D, and F. If their numeric values are 4, 3, 2, 1, and 0 respectively, create a direct function to determine the grade point average of a set of letter grades.

For example,

```

3      gpa1 'CBBA'

3      gpa1 'C' 'B' 'B' 'A'

3      gpa1 'CBBA'

3      gpa1 'C' 'B' 'B' 'A'

[1]    gpa1←{⍵ ⍵ a vec of letter grades from set 'A,B,C,D,F'
[2]    ⍵IO←0
[3]    n←'FDCBA'⍵
[4]    (+/n)÷ ≠n

```

Check Balancing

Often when writing checks in your check book you will write the check, record it in the accompanying summary book, but do not subtract the value of the check from the running balance. Often this situation of check values and deposits continue for a while without updating the balance. Ultimately, the running balances need to be completed.

Create a *Dfn* that will accept the current balance of the checkbook and a vector of check values and deposits and then produce the corresponding running balances to record in the summary book.

Suggestion: Consider the check values as positive numbers and the deposits as negative numbers. It should simplify the solution.

```

bal←1000
chks←100 25 ^500 33 27.5 ^200
bal checkbal chks
1000 900 875 1375 1342 1314.5 1514.5

▽ checkbal←{Rα :current balance
[1]      R ω: numeric vector where
[2]      A deposits are - & withdrawals are +
[3]      A result a vec of corresponding balances
[4]      +\α,-ω
[5]      }
▽
```

A Series Expansion

Construct a *Dfn* for the following:

One plus the sum of the reciprocals of the factorials of the positive integers from *one* to *N*. This statement represents the series which approaches *e*. How large would *N* need to be to produce *e* to six decimal places?

Solution:

```

series←{⌈ω: positive integer
[1]      ⌈ result:ω term series expansion of e
[2]      ⌈io←1
[2]      1++/÷!⌈ω
[3]      }
```

```

series 8
2.71827877
```

```

series 9
2.718281526
```

```

*1
2.718281828
```

An inner-product solution is:

```

▽ Series←{⌈ω positive integer
[1]      ⌈ a series expansion of e
[2]      ⌈IO←1
[3]      1+1+÷!⌈ω
[4]      }
▽
```

```

Series 8
2.71827877
```

```

Series 9
2.718281526
```

```

*1
2.718281828
```

Adjacent Removal

Given a vector of data where there are adjacent duplicate items, create a function that would remove the adjacent duplicate items.

For example;

```
adjacent 'abbcccaaaccbb'
abcacb
```

Solution:

```

    ▽ adjacent←{A ω:vector with adjacent duplicate items
[1] A result: a vector without adjacent duplicates
[2] A e.g. for ω as 'abbcccaaaccbb'
[3] A result: abcacb
[4]      (1,2≠/ω)/ω
[5]    }
    ▽
```

Here we find an *n-wise reduction* on ω that creates a binary vector for compression on ω .

Why is a 1 catenated to the pairwise reduction?

Falling Factorials

A Falling Factorial is defined to be

$$X^n = (X-0)(X-1)(X-2) \dots (X-(n-1)) \quad (\text{in standard mathematical terms})$$

- a) Create a *Dfn* for a general Falling Factorial. The Combination of N things taken K at a time in terms of Falling Factorials is:

$$N^K \div K^K \quad (\text{in standard mathematical terms})$$

- b) Write the *Dfn* for determining the combination of N items taken K at a time, using the Falling Factorial definition.

The Combination of N things taken K at a time in terms of Factorials is

$$N! \div K!(N-K)! \quad (\text{in standard mathematical terms})$$

- c) Write the *Dfn* for determining the combination of N items taken K at a time using this factorial definition.

The complete definition of the Falling Factorial for an integer n is:

$$X^n = (X-0)(X-1)(X-2) \dots (X-(n-1)) \quad \text{for integers } n > 0$$

$$X^n = 1 \quad \text{for } n = 0$$

$$X^n = \frac{1}{(X+1)(X+2) \dots (X+(-n))} \quad \text{for } n < 0$$

- d) Modify the function of exercise a) to handle the complete definition of the Falling Factorial.

A Solution:

Here we have an example of nested calls. In this case a *Dfn*, **fallcomb**, calling another *Dfn*, **fallfac** (twice).

```

3 fallfac 10
720

```

a)

```

∇fallfac←{Aω: value raised to to the falling fac
[1]      A α: value of fall factor; integer≥0
[2]      □IO←0
[3]      ×/ω-1α
[4]      }
∇

3 fallcomb 10
120

```

b)

```

∇fallcomb←{Aω: total no. of items
[1]      Aα: no of item taken at a time
[2]      (α fallfac ω)÷α fallfac α
[3]      }
∇

3 comb 10
120

```

Note the nesting of *Dfns*; the *Dfn* **fallcomb** calls the *Dfn* **Fallfac** twice.

c)

```

∇comb←{Aω: total no. items
[1]      Aα: no taken at a time
[2]      Athe no. of combinations W item at α times
[3]      (!ω)÷(!α)×!ω-α
[4]      }
∇

```

d)

```
fallfacful←{⍵: value raised to to the falling fac
  ⍵ α: value of fall factor;integer
  α=0:1
  ⍵IO←0
  α>0:×/ω-ια
  ⍵IO←1
  ÷×/ω+ι-α
}
```

Inner Vowel Removal

One of the programming problems in the Dyalog 2014 contest was to create a *Dfn* to remove the inner vowels from the words of a character vector sentence.

For example,

```
(a solution) 'If you can read this, it worked!'
If you can rd ths, it wrkd!
```

```
(a solution) 'I am NOT a BEE'
I am NT a BE
```

- a) Create *Dfn* to do this.
In a similar problem, the requirement was that only the words of length greater than 3 were to have their inner vowels removed.
- b) Create a *Dfn* or modify the solution for problem 1. that will only remove inner vowels from words of length greater than *n*.

For example,

```
3 (a soln) 'If you can read this, it worked!'
If you can rd ths, it wrkd!
```


Solutions:

Problem 1

```
removein←{A ω: simple char. vector sentence
  A contains theDfn oneword
  A result: simple char vec without inner vowels
  ⌈m⌈←3
  onewd←{1≠ω:ω ⋄ (1↑ω),('aeiouAEIOU'~⌈1↓~1↓ω),~1↑ω}
  ε' ',~onewd"(ω≠' ')<ω
}
```

Problem 2

```
1) removeinN←{A ω: simple char. vector sentence
2) A α: positive integer leave words of length less than α alone
3) A result: simple char vec without inner vowels
4) ⌈m⌈←3
5) onewd←{((≠ω)≤α):ω ⋄ (1↑ω),('aeiouAEIOU'~⌈1↓~1↓ω),~1↑ω}
6) ε' ',~α onewd"(ω≠' ')<ω
}
```

```
3 removeinN 'If you can read this, it worked!'
If you can rd ths, it wrkd!
```

Discussion:

Consider the solution of problem 1. Line 5 contains the *Dfn oneword*. It was designed to do the deletion on a single word. Then in line 6 it was applied to *each* word of the sentence.

Back on line 5 you find essentially two expressions combined on one line by the *Diamond* separator, \diamond . This allows you to combine more than one statement on a line. The statements are evaluated in a left-to-right fashion.

The first statement in the *Dfn onewd* is a guard expression ending the *Dfn* if the word has only one character.

After the test for a single character word, the algorithm followed in *onewd* is as follows:

- Drop the first and last characters of the word.
- Remove the files from the inner portion of the word.
- Return the first and last character of the word.

In line 6, the sentence is partitioned into a vector of words to which the *Dfn oneword* is applied to each word. The blank characters in the sentence are the separating characters. Finally, a vector of words is returned to a simple character vector.

An Unusual Paragraph

Prologue: The following paragraph is offered as a puzzle:

“How quickly can you find out what is unusual about this paragraph? It looks so ordinary that you would think that nothing was wrong with it at all and in fact, nothing is. But it is unusual. Why? If you study it and think about it you may find out, but I am not going to assist you in any way. You must do it without coaching. No doubt, if you work at it for long, it will dawn on you. Who knows? Go to work and try your skill. Par is about half an hour.”

The puzzle is to determine what is unusual about this paragraph.

Problem:

Create a function that will determine the number of a given set of letters in a character vector.

For example,

```
sent<-'How quickly can you find out what is unusual about
this paragraph?'

letterct sent
a 7
e 0
i 4
o 4
u 7
```

Could the outer function help to solve the puzzle?

A Solution:

```

      ▽ letterct←{⍵: a char. sentence
[1]      ⍵ result: a count of the number of vowels
[2]      'aeiou',⍵+/'aeiou'∘.=⍵
[3]      ⍵ this function does not count Capitals
[4]      }
      ▽

```

Note the monadic function Table, “⍵”. It will take a vector and convert it into a one-column matrix. It has no effect on higher-rank arrays.

For example,

```

      ⍵ 'How'
H
o
w

      ρ ⍵ 'How now'
7 1

```

The combination ,⍵ is a good idiom to turn two vectors into a two-column matrix.

Finding Pythagorean Triples

Preface:

Recall that for a right triangle with sides a , b , and c where c is the longest side, the equation $a^2 + b^2 = c^2$ is true. When these numbers a , b , c are positive integers they are called *Pythagorean Triples*. Around 250 A.D., the mathematician Diophantus found all possible integer number solutions to the equation $a^2 + b^2 = c^2$.

His algorithm begins by taking any two whole numbers, and

- 1) Form the difference between their squares,
- 2) Take twice their product,
- 3) Take the sum of their squares.

This algorithm, starting with any pair of whole numbers, will lead to three numbers that form a Pythagorean triple.

Problems:

- 1) Given any three numbers, create a function that will determine if they are a Pythagorean triple.
- 2) Follow the Diophantus algorithm and create a function to produce a Pythagorean Triple.

Solutions:

Problem 1

```

PT←{Aω: 3 item vec
[1]   Aresult:1 if a Pythagorean triple
[2]   A 0 if not
[3]   ((⌈/ω)*2)=+/(ω≠⌈/ω)/ω)*2}

PT1←{Aω: 3 item vec
[1]   Aresult:1 if a Pythagorean triple
[2]   A 0 if not
[3]   c←⌈/ω
[4]   (c*2)=+/(ω≠c)/ω)*2}

PT2←{Aω: 3 item vec
[1]   Aresult:1 if a Pythagorean triple
[2]   A 0 if not
[3]   c2←(c←⌈/ω)*2
[4]   c2=+/(ω≠c)/ω)*2}

```

Problem 2

```

▽ PTrip←{Aω: 2 integers
[1]   Aresult: a Pythagorean Triple
[2]   (-/ω*2),(2××/ω),(+/ω*2)
[3]   }
▽

PTrip 8 5
39 80 89

PT PTrip 8 5
1

```

Data Smoothing with an Adjustable Weight Window

Often when getting data, smoothing the data provides a more accurate picture. Define a function, **SMOOTH**, which will accept a vector **v** of data and a smoothing weight vector **w** defining the weight assigned to the items involved in the smoothing.

The basic algorithm consists of repeatedly computing the product of **w** against (pw) adjacent items of **v**. Then each of these products is divided by the sum of the weight vector **w** to achieve an average for the combination.

First, vector **v** is padded with zeros on both ends. The number of zeros is no more than half of pw .

The averaging process continues with the weight vector **w** against all adjacent (pw) numbers of the extended vector **v**.

For example,

```

w ← 1 2 1
v ← ? 6 10
v
8 10 4 3 10 8

2 * w smooth v
8.67 10.67 7.00 6.67 10.33 8.67

```

Solution:

```

      w
1 2 1

      v
8 10 4 3 10 8

[0]  z←w smooth1 v;t;i;⊂ML
[1]  ♂v: Vector of measurements
[2]  ♂w: weight vector
[3]  ♂z: Vector of smooth data
[4]  ⊂ML←3
[5]  I←⌊0.5×ρ,w ♂ amt. of padding
[6]  t←(ρ,w),/(ip1↑v),v,(Ip¯1↑v) ♂ blding (ρw) tuples
[7]  z←+/⌈⌈(c w)×t
[8]  z←z÷+/w

      w smooth1 v
8.5 8 5.25 5 7.75 8.5

```

The FBI Checksum

In 2015, the FBI expanded its inventory of 8-character Universal Control Numbers (UCN) by adding 17 letters, A,C,D,E,F,H,J,K,L,M,N,P,R,T,V,W,X to the digits 0 to 9.

The UCN with its check character is now alphanumeric. Calculating the check character needs to convert the 17 additional characters to the integers 10 – 26. For example, the identifier D1368KHX in calculating its check digit translates into 12 1 3 6 8 17 15 26. These 8 digits, multiplied by the corresponding weights, 2 4 7 8 10 11 13 and then summed. The modulo 27 of this sum determines which character is chosen from the 27-character vector to be the checksum digit. For the above example, mod 27 of the sum 822 is 12, which matches the check character D and the UCN is D1368KHXD.

- 1) Create a *Dfn* to determine the checksum character for a given eight character Universal Control Number.
- 2) Create a *Dfn* that given the UCN including the checksum character, determine if the UCN is correct.

Problem 1

```

    ▽ FBIchsum←{Aω:an 8 character Univ.Ctrl No.
[1]      A result: the check sum digit
[2]      □IO←0
[3]      key←'0123456789ACDEFHJKLMNPRTVWX'
[4]      wt←2 4 5 7 8 10 11 13
[5]      key[27|+wt×keyιω]
[6]    }
    ▽

    FBIchsum 'D1368KHX'

D

```


Problem 2

```

        FBIchsum 'D1368KHX'
D
        FBIcheck 'D1368KHXD'
valid check sum digit

        FBIcheck 'D1368KHXA'
invalid check sum digit

▽ FBIcheck←{⍵: UCN with check digit
[1]  ⍵ returns 'invalid or valid check sum digit'
[2]  cdight←⌊1↑ω
[3]  ((cdight≠FBIchsum ⌊1↑ω)/'in'),'valid check sum digit'
[4]  }
▽

```

Generating Prime Numbers

The following function will generate prime integers less than a given integer R . The function will soon exceed the computer's memory for large R . However, it is illustrated here to commemorate the fact that the expression in the body of the *Dfn* was one of the very early illustrations showing the power of APL.

```

      primes←{Agenerates prime integers <ω
[1]      Aω: positive integer
[2]      (~t←t°.×t)/t←1↓t←ιω
[3]      }

      primes 25
2 3 5 7 11 13 17 19 23

```

Cardinal to Ordinal

Write a function that will accept a positive integer and return the character vector consisting of the integer followed by the appropriate ordinal representation. For example,

```
Ordinal" 10
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

Ordinal" 30+10
31st 32nd 33rd 34th 35th 36th 37th 38th 39th 40th

Ordinal" 1000 1001 1002 1003 1004
1000th 1001st 1002nd 1003rd 1004th
```

Solution:

```
Ordinal←{⌈ω:a    positive integer
[1]  ⌈ result: the number affixed with ordinal
[2]    ⌈ML←1
[3]    z←⌈ω
[4]    i←'123'⌈1↑z
[5]    t←i>('st' 'nd' 'rd' 'th')
[6]    t←(t'th')[1+'1'⇒2↑z]
[7]    z,εt
[8] }
```

The Totient/Phi function

The Euler *Totient/Phi Function* was introduced by Euler in the 1760's. For a given positive integer n , it determines the number of positive integers less than or equal to n that are relatively prime to n . An integer that is relatively prime to a given integer n is called a *totative* of n . In other words, the *Totient Function* can be defined to be the sum of *totatives* of n , the *Totient* of n . The mathematical standard notation for the *Totient function* is $\Phi(x)$, hence it is also called the *Phi function*. By convention $\Phi(0)=1$.

- 1) Create a totient function that given a positive integer n , will return the sum of the totatives of n .
- 2) Create a function that given a positive integer n , will return each of the totatives as well as the totient.

```

      Phi 24
8
      Phi 9
6
      Phi1 24
1 5 7 11 13 17 19 23 8

      Phi1 9
1 2 4 5 7 8 6

      ∇ Phi←{⌈ the Euler totient function
[1]      ⌈ω: positive integer
[2]      ω=0:1
[3]      +/1=ω∨~1ω
[4]      }
      ∇

      ⌈vr 'Phi1'

      ∇ Phi1←{⌈ the Euler totient function
[1]      ⌈ω: positive integer
[2]      ⌈result: the totatives followed by totient
[3]      ω=0:1,1
[4]      (ρz);~z←1=ω∨~1ω
[5]      }
      ∇
  
```

Comments:

Recall the extended use of the OR symbol, \vee , as *Greatest Common Divisor*.

Palindromes

A palindrome is a sequence of characters that read the same way forward and backward.

Create a *Dfn* to determine if a simple character vector is a palindrome, returning a 1 if it is or 0 if it is not. Thus,

```

1      palindrome 'now I won'

0      palindrome 'Now I Won'
```

Solution:

```

[1]  ∇ palindrome←{⌈ω :simple character vec.
[2]  ⌈result 1 if ω is a palindrome,else 0
[3]  1=ρω:1
[4]  0∈ω=ϕω:0
[5]  1
    }
```

∇

Pillow Problem #21 (07/04/1889)

Charles Dodgson (better known as Lewis Carroll) created a book entitled “*Pillow Problems Thought Out During Wakeful Hours*”. In it, his problem #21 states:

“Sum (1) to n terms and (2) to 100 terms the series $1 \times 3 \times 5 + 2 \times 4 \times 6 + 3 \times 5 \times 7 + \dots$ ”

Create a *Dfn* to solve the problem.

Solution:

```

      LC21  4
360
    
```

```

      LC21 100
27573000
    
```

```

      LC21←{A Lewis Carroll problem 21(7/4/1889)
[1]      □IO←0
[2]      +/*/'(1ω)+c1 3 5
[3]      }
    
```

The enclosure of `c1 3 5` produces the scalar *V fn S* relationship.

The system variable `□io` can be only 0 or 1.

In everyday usage, we start *counting* from 1 and start *measuring* from 0. So it is in computing, too.

Fibonacci Numbers

Preface:

The Fibonacci numbers are the numbers that appear in a sequence such that the numbers in the sequence are the sum of the two preceding numbers. That is,

$$f_0=0 \quad f_1=1 \quad \text{and} \quad f_n=f_{n-1} + f_{n-2} \quad (\text{for } n>1)$$

Thus 0 1 1 2 3 5 8 13 21 34 ... (note that the zero is often omitted.)

Fibonacci numbers are named after Leonardo of Pisa who later became known as Fibonacci after he introduced them to Europe in 1202.

Exercise 115

Create a Fibonacci *Dfn*, where ω is the number of terms desired and α is the starting pair of integers.

```

      1 1 fib1 7
1 1 2 3 5 8 13 21 34

```

Solutions:

There have been many different solutions.

1)

```

      z←x fib1 n
[1]      A generates n+2 fibonacci numbers
[2]      A x a starting pair of integers
[3]      z←x
[4]      :Repeat
[5]      n←n-1
[6]      z←z,+/-2↑z
[7]      :Until n=0

      1 1 fib1 7
1 1 2 3 5 8 13 21 34

```

2)

```

      fib2←{Aa Fibonacci sequence
[1]      Aα:starting pair of integers
[2]      Aω: no. of terms
[3]      z←α
[4]      z←z,+/-2↑z
[5]      ω=0:~1↓z
[6]      z ∇ ω-1}

      1 1 fib2 7
1 1 2 3 5 8 13 21 34

```

Note the ∇ in line [6] of *fib2*. It indicates that the function is to call itself again. The ∇ is the symbol for self-reference of the named function in which it is used.

3)

```

fib3←{A fibonacci no.
[1]   Aα:a pair of integers
[2]   Aω:the term of the series returned
[3]   α←0 1  A if there is no right arg
[4]   ω<0:θρα
[5]   (1↓α,+/α)▽ ω-1
[6]   }

1 1 fib3 7
21
fib3``ι8
1 2 3 5 8 13 21 34

```

Notice line [3]. When a *Dfn* is called without a left argument, the function may specify a default value in the body of the function.

```

fib4←{>{ω,+/-2↑ω}/φιω}
fib4 9
1 1 2 3 5 8 13 21 34

fib5←{A tail recursive
[1]   α←0 1
[2]   ω=0:θρα
[3]   (1↓α,+/α)▽ ω-1
[4]   }

fib5 9
34
fib5``ι9
1 1 2 3 5 8 13 21 34

fib6←{Astack recursive
[1]   ω∈0 1:ω
[2]   +/▽``ω-1 2
[3]   }

fib6 9
34

fib6``ι9
1 1 2 3 5 8 13 21 34

```


The Golden Ratio

Preface:

The Golden Ratio was first recorded in Euclid's Elements (325–265BC). It has fascinated mathematicians, architects, artists, and others ever since. The ratio of two quantities a and b is a Golden Ratio if the ratio of a/b is equal to the sum $a+b$ divided by the larger of a and b .

The value of the Golden Ratio is:

$$(1 + \sqrt{5}) \div 2 = 1.6180339887 \dots$$

Fibonacci (1170–1250 BC) mentioned that by taking the Fibonacci sequence:

2 1 3 4 7 11 18 29 ...

and create the following rational number sequence from it,

$$1/2 \quad 3/1 \quad 4/3 \quad 7/4 \quad 11/7 \quad 18/11 \quad 29/18 \quad \dots$$

that this sequence asymptotically approaches the GoldenRatio.

Exercise 116

- a) Create a *Dfn* to generate an arbitrary Fibonacci sequence. For example,


```
2 1 fib 7
2 1 3 4 7 11 18 ...
```
- b) Create a *Dfn* that will take a Fibonacci sequence (such as in problem a) and create a sequence of rational numbers in decimal form. For example, for the Fibonacci sequence:


```
2 1 3 4 7 11 18 29 ...
```

the rational number sequence would be created as

```
1/2 3/1 4/3 7/4 11/7 18/11 29/18 ...
```
- c) Determine how far along the rational number sequence one needs to go to determine the value of the Golden Ratio to 8 decimal places.

The United States Reapportionment Algorithm

Prologue:

Every tenth year after the Census is taken, the 435 sets of the House of Representatives are reapportioned based on each state's current population as determined by the Census.

The algorithm to determine the distribution of seats is an iterative one.

- 1) To begin, each state is assigned one representative.
Let NS represent the number of seats that state S has. (Note that this will initially be 1.) PS represents the population for state S as given the new Census count.
- 2) Calculate
 $CS=PS$ for each state.
- 3) Find the largest value of the CS 's for the 50 states.
- 4) Increase by 1 the value of the state that had this largest CS value.
- 5) Repeat steps 2, 3, and 4, 385 times. At the completion, the NS s should represent the new distribution of the seats in the House of Representatives

Exercise 117

Write a program to compute the new apportionment of seats in the House of Representatives given the 2010 Census data.

(For the purpose of this exercise, the workspace `ap1class` contains the two variables: `populations` and `states`. The variable `states` is a vector of postal code abbreviations. The variable `populations` is the vector of 2010 state populations.)

Here are the 14 states and their total populations from the first 1790 census:

VT	NH	ME	MA	RI	CT	NY	PA	DE	MD	VA	KY	NC	GA
85539	141885	96549	378787	68825	237946	340120	434373	59094	319728	747610	73677	393751	82528

Of course, one needs to know what the total number of seats there were to be in the House in 1790.

Solutions:

Here is one using the Power operator:

```

z←sn reapportion2 ps;n
[1]      A ps: vector of each states' populations
[2]      A      based on 2010 new census
[3]      A sn: states postal names
[4]      A z: a 50 by 2 matrix of seats for each state
[5]      A      with the states in alphabetic order
[6]      n←reapport1*385←1
[7]      z←sn,;n

```

with

```

reapport1←{
[1]      cs←ps÷(ω×ω+1)*0.5
[2]      ω+cs=⌈/cs
[3]      }

```

or combining them

```

z←sn reapportion1 ps;n
[1]      A ps: vector of each states' populations
[2]      A      based on 2010 new census
[3]      A sn: postal names both in states alphabetical order
[4]      A z: a 50 by 2 matrix of seats for each state
[5]      A      with the states in alphabetic order
[6]      n←{
[7]          cs←ps÷(ω×ω+1)*0.5
[8]          ω+cs=⌈/cs
[9]      }*385←1
[10]     z←sn,;n

```

```

z←sn reapportion ps;n;cs;ct;⌈IO
[1]  A ps: vector of each states' populations
[2]  A      based on 2010 new census in state alphabetical order
[3]  A sn: states postal names in alphabetical order
[4]  A z: a 50 by 2 matrix of seats for each state
[5]  A      with the states in alphabetic order
[6]  ⌈IO←1
[7]  ct←385 A no. of seats to be assigned
[8]  n←50p1 A each state gets 1 seat
[9]  i←1
[10] :Repeat
[11]     cs←ps÷(n×n+1)*0.5
[12]     n←n+cs=⌈/cs
[13]     i←i+1
[14] :Until I>ct
[15] z←sn.[1.1]n

```

[illegible]

Multiple Replacement

Replacing elements within an array is an action most applications need to be able to do. Initial material on replacement was covered in Selective Assignment on pages 71 through 72 especially in the exercises. A more challenging situation arises when there is a need to replace multiple elements in the array each with a different value. Formally stated,

Given an array and a set of elements within it to be replaced and an accompanying replacement elements, create a Dfn to achieve the replacements.

Discussion:

Two solutions follow. Each solution follows the same approach, namely, to determine a procedure to replace a single element in the array. Then expand the algorithm to include replacing multiple elements.

In one approach, the indices of the elements to be replaced were found and then used to insert the replacement elements.

In the second approach, binary arrays were used to find and replace the designated elements.

Part 1:

The replacement of a single element.

```

      ∇ replace0←{Aω:  simple numeric array
[1]      Aα: 2 item vec. 1↑α to be replaced
[2]      A ~1α replacement
[3]      z←εω
[4]      z[⊖z=1↑εα]←~1↑εα
[5]      z
[6]      Areturns a vec.
[7]      }
      ∇

```

```

(pr)ρ 2 200 replace0 r
1 1 6 200 1
200 3 5 200 6
5 3 3 200 6
8 200 5 4 5

```

```

      ∇ replace1←{Aω:  simple numeric matrix
[1]      Aα: 2 item vec. 1↑α to be replaced
[2]      A ~1α replacement
[3]      (ω×ω≠1↑α)+(~1↑α)×ω=1↑α
[4]      }
      ∇
2 200 replace1 r
1 1 6 200 1
200 3 5 200 6
5 3 3 200 6
8 200 5 4 5

```

Part 2:

v			
1 100	2 200	3 300	4 400

```

▽ REPLACE←{Rω: a simple numeric matrix or vec
[1]      Rα: a vector of r pairs a&b where
[2]      A a is a no. to be replaced &
[3]      A b is the replacement
[4]      A shape of result is the pω
[5]      z←replace0/α, cω
[6]      (pω)p∈z
[7]    }

▽
v REPLACE r
100 100    6 200 100
200 300    5 200    6
   5 300 300 200    6
   8 200    5 400    5

▽ REPLACE1←{Rω: AN ARRAY
[1]      Rα: vec. of 2 items
[2]      A 1↑α item to be replaced; 1↑α replacement item
[3]      replace1/α, cω
[4]    }

▽
v REPLACE1 r

```

100	100	6	200	100
200	300	5	200	6
5	300	300	200	6
8	200	5	400	5

Both `REPLACE` and `REPLACE1` use the Reduction operator. Since by its definition, its operand, in this case either `replace0` or `replace1`, is distributed between the elements of its argument and evaluated in the usual fashion, right to left. At each evaluation of the operand, one more replacement occurs.

Since there two solutions, how do they compare in performance?

```
)copy dfns.dws cmpx
C:\Program Files\Dyalog\Dyalog APL-64 18.0 Unicode\ws\dfns.dws
cmpx'v REPLACE larg' 'v REPLACE1 larg'
v REPLACE larg → 8.4E-6 | 0% ████████████████████████████████████████████████████████████
* v REPLACE1 larg → 6.9E-6 | -18% ████████████████████████████████████████████████████████████
```

Actually in several performance measurements, REPLACE1 outperformed REPLACE from 8% to 25%.

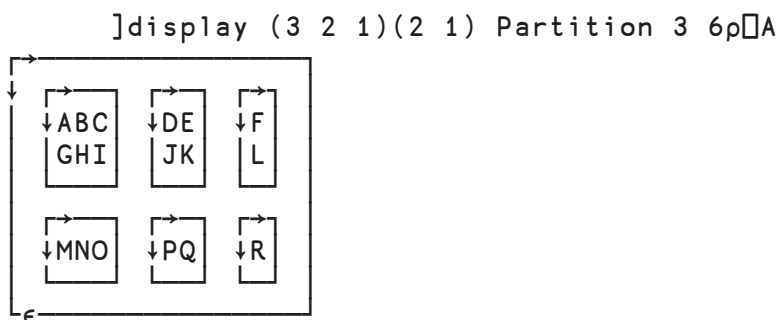
Working with binary data may improve performance.

Partitioned Matrices

In Linear Algebra one of the topics is Partitioned Matrices. Starting with a simple matrix, it is be subdivided into sets of sub matrices by indicating the rows and columns that will subdivide the original matrix. For example with

A
 ABCDEF
 GHIJKL
 MNOPQR

and indicating that A should be broken up after columns 3 and 5 and row 2, A becomes a 2×3 matrix of partitioned matrices. Or one also could have indicated the partitioning by indicating the number of consecutive rows and columns each partitioned matrix is to have. In the above example, it would have been for columns 3 2 1 and for rows 2 1.



Exercise 118

Create a *Dfn* to partition a matrix, given a simple matrix and an indication of where the columns and rows are to be separated.

Solution:

```
Partition←{Aα:2 two item integer vec.e.g.(3 2)(2 3)
A 1st vec for coln split, 2nd for row split
A ω: simple matrix
IO ML←1
z←Q↑(α[2])partitionr"cw←(εα[1])partitionmc ω
z
}
```

```
partitionmc←{Aω:simple array'
Aα: integer vec of column splits e.g. 3 2 4 1
Aresult nested array split by column
a←+\εα      A 3 5 9 10
b←a↑[2]"cω
c←0,-1↓a    A0 3 5 9
c↓[2]"b
}
```

```
partitionr←{Aω:simple arra
Aα: integer vec of row splits e.g. 3 2 4 1
Aresult nested array split by row
a←+\α      A 3 5 9 10
b←a↑"cω
c←0,-1↓a    A0 3 5 9
c↓"b
}
```

Exercise 119

- a) Modify the *Dfn Partition* so that it has a check to determine that the column and row splits match the shape of the array
- b) Modify the *Dfns Partitionmc* and *Partitionr* so that they have a check to determine that the column and row splits match the shape of the array

Postscript: Some Debugging Facilities

In programming, nothing runs correctly the first time. Hence, any programming language provides debugging facilities as part of the whole language package. Dyalog APL is no different. Let us examine some of the Dyalog debugging facilities.

The main facilities are the *Trace* and *Stop* controls. These controls can be set on individual lines of a function or operator. They enable one to examine intermediate calculations and watch the evolution of a function.

A *Stop* set on a line of a function produces a stop *before* the execution of the indicated line number.

A *Trace* set on a line produces an indication of the line's execution by displaying the last item computed on that line and execution continues on to the next line.

Both *Stop* and *Trace* may be set on the same line. These controls are set on lines of a function or operator while they are in edit mode. One may put a function into the editor by double-clicking on the function name anywhere the name appears in the session manager screen. However, initially you must create a new function. You can do that by executing

```
)Ed Foo
```

which will produce the editor window containing just the name **Foo**. At this point you can proceed to create the function header that you wish and enter line by line the body of the function. For example, enter:

```
▽ z←x Foo y
  z←x+y
  z←z,x-y
  z←z;y×x
  z←z x*y
▽
```

the *Escape* key will close the editor preserving what you have entered or changed. Without saving any changes use *Shift + Escape*.

Now, several choices exist. Clicking on the View item on the editor's toolbar produces a drop-down menu. In this View menu, check Trace, Stop, and Line Number. This action will produce line numbers as you define the function. Also a blank column delimited by two vertical lines appears before the function. It is in this area that Trace and Stop points are placed. By hovering on the right side of this blank column a red dot appears. Clicking on the right side of that column for a particular line will place a red dot on that line. This red dot indicates a Stop condition is placed on that line. By hovering on the left side of the column produces a yellow dot. Clicking on the left side of that column for a particular line will place a yellow dot on that line. This yellow dot indicates a Trace condition is placed on that line. By clicking on either dot again will remove it.

Close the function via the *Esc* key. Now execute the function **Foo** with, say, **3 Foo 7**. At this time the Debugger window appears at the bottom of the session manager window. It is divided into two parts; the right half indicates the contents of the pushed down execution stack. **)Reset** will clear this execution stack. The left half displays the function with the line where the execution failed is circled in red. In this case, for the **Foo** function it should be line 3.

While you may set Trace and Stop points in the debugger window, you need to go to the editor to correct the function. Double clicking on the name of the function either in the session window or the debugger window will change the debugger window from gray background to a white background, placing you in edit mode there. After completing any desired editing, a single depression of the *Esc* key will return the debugger window to a gray background. A second depression of the *Esc* key closes the debugger window and returns you to the session manager screen. While the debugger window is active, you may check the status of the local variables in the session manager window above.

You should find the Trace and the Stop facilities to be quite useful in ferreting out programming errors.

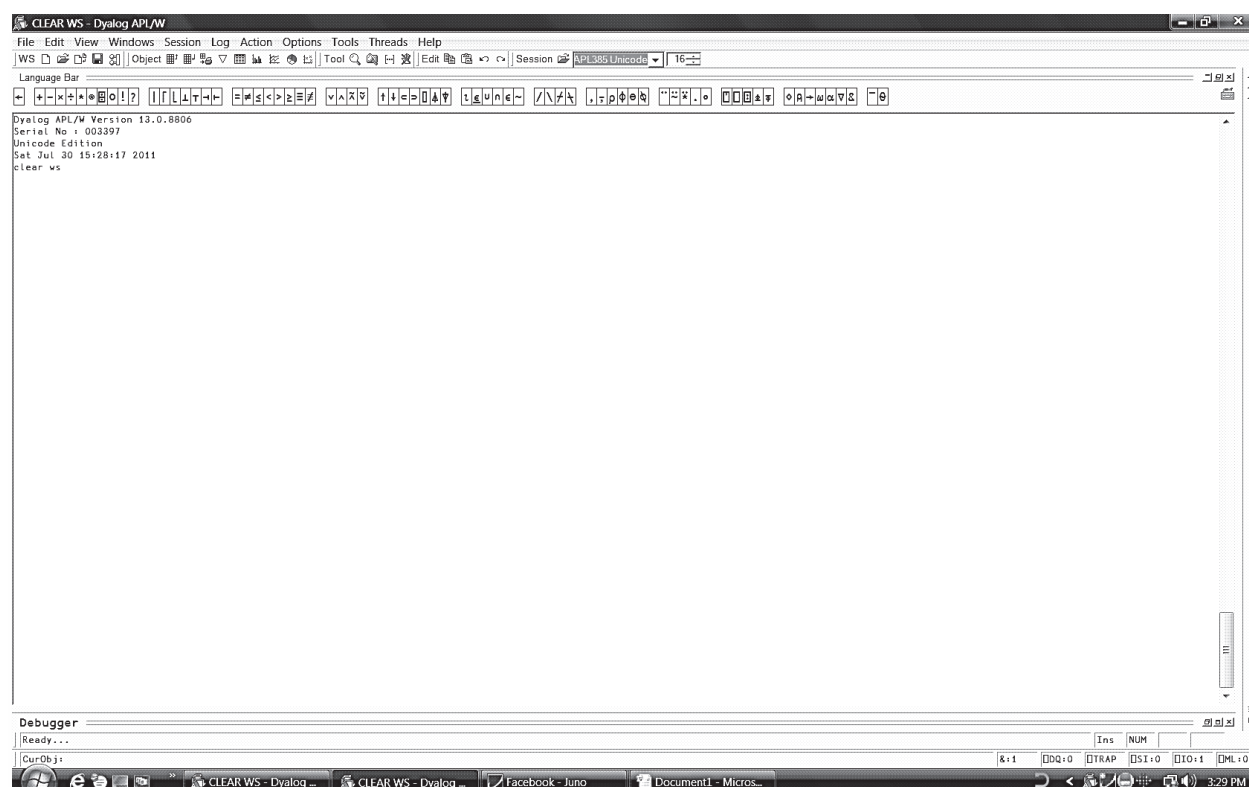
While trace and stop points may be set for procedural programs, currently only stop points may be set for *Dfns*.

Appendix A: The Dyalog Windows Working Environment

Surrounding any programming language is software that interfaces with the operating system and the computer hardware. Before launching off into the Dyalog APL language, it is useful to discuss some features of the APL system that are useful to one beginning to study the Dyalog APL language. It will be presented in the context of the Windows Operating System. We shall begin describing this APL environment assuming that Dyalog APL has been installed properly and a Dyalog APL icon exists on the desktop. When you click on this icon to launch APL, the *Dyalog Session Manager* Screen appears. It is best initially to maximize this screen.

The Session Manager

Upon entering APL, a window specially designed for Dyalog APL appears. This is the *Session Manager Screen*:



The major portion of the screen is blank awaiting activity by the APL user. This large area is the user's work area. This work area is officially called your *workspace*, the area for the entry and execution of APL material. It acts like an almost infinitely long sheet of paper. When you get to the bottom of the screen, it scrolls upward. You can scroll up to view what you had previously entered or how APL had responded. This scrolling upward could even possibly take you into earlier sessions.

Your workspace has a name; initially it was either **C**lear or **C**ontinue. However, you should create your own name for your workspace. In choosing a name for your workspace, do not begin with a number.

If you have just activated APL for the first time, create a name for the currently active workspace, then save and reload it. For example issue

```
)wsid APLclass ...then...
)save
)load APLclass
```

then, the next time you log on to APL again, you may issue

```
)load APLclass
```

and resume from where you left off when you last signed off.

It is important to note that unless you explicitly save your workspace the information in it is lost when you leave APL. Don't panic. There are facilities to save and retrieve your workspace and its content. This is through a set of commands called *System Commands*. They have a common characteristic in that they all begin with a right parenthesis. There is a large set of them. Here is a subset of them that you initially will need:

```
)Save name  saves your workspace with the given name.
)Load name  retrieves your named saved workspace.
)Wsid       returns the current workspace name.
)Wsid name  renames the active workspace "name".
)Continue   saves the active workspace and exits from APL
)Off        The proper way to leave APL.
```

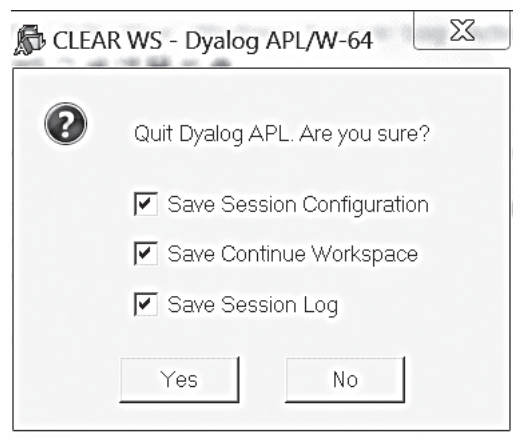
While `)Off` is the fastest way to log off of APL, another way is via the upper right corner "X". Upon clicking on it, a small window will appear.

You are asked if you want to quit APL. This gives you an opportunity to say no, since perhaps you forgot to save your workspace.

Before you say yes, there are three other choices you can make:

- *Save Session Configuration*, when checked will preserve system settings that you may have changed from their default settings. For example, you may have changed `⎕ml`.
- *Save Continue Workspace*, when checked, will save your current active workspace under the name *Continue*.
- *Save Session Log*, when checked, APL will save all of the activities you did during your session. This is useful if you wish to see what actions you took during your earlier previous sessions.

You may save your workspace anytime during your session. In fact, it may be a good idea to do that occasionally, in case you forget to save when you leave APL.



There are three bars across the top of the screen and three across the bottom. These bars either provide facilities that one may use or they offered status information related to the APL system.

The topmost bar contains familiar names such as File, Edit, and some related to the more advanced facilities such as Threads and GUI. Upon clicking any of these names produces a drop-down menu. While most of the items in the File drop-down menu are familiar and standard, most of the items in the other drop-down menus are APL specific and can be used as one becomes more familiar with APL.

Next bar down is the Tool bar. It contains icons representing specific APL system actions. It is subdivided and labeled as “Workspace,” “Object,” “Tools,” “Edit,” and “Session.” For example, the Session section offers you a choice of fonts and the ability to change the font size. Make certain that the font is set to “APL385 Unicode.”

Below the toolbar is the *Language bar*. It contains all the APL symbols. When you hover the cursor over one of the symbols, a drop-down window appears. It contains:

- the monadic and dyadic names of the symbol,
- the keystroke combination to generate the symbol, and
- examples of its use.

Whenever you click on the symbol on the language bar, it will appear in the session manger screen where the cursor currently is blinking.

Across the bottom are three more bars:

- The first is labeled “Debugger.” It is a minimized pair of Windows that come into play when one is debugging a program.
- The second bar is labeled “Ready.” It defines the mode of the session screen. The Ins and Num boxes defined settings of the keyboard.
- The last bar is labeled “CurObj.” It records the object name and type of the last item on which you clicked. On the right side of this bar are six labeled boxes. Of these, the two rightmost are important to the beginner. The \square IO (Index Origin) box defines the current origin of the session. The \square ML (Migration Level) box defines the current level of the Dyalog APL system. It always opens at level 0 in order to be compatible with applications written at an earlier time. The most recent level is level 3. The level we shall assume and in which we will work is one. In order to set it to that level, enter:

\square ML←1

if you change the level to anything else, the \square ML box will turn red, showing the value digit.

Clipboard work

There is one notable difference in accessing the clipboard. While in APL, the key combinations are for Cut, Copy, and Paste are:

- Cut: *Shift + Delete*
- Copy: *Control + Insert*
- Paste: *Shift + Insert*

(Note: these combinations were the initial ones that Microsoft assigned.)

While the above combinations will always work in Dyalog APL, the more universal combinations

Ctrl+X, Ctrl+C, and Ctrl+V

would be desirable too and the ones you would like to use. Now if the keyboard is “Dyalog APL IME(en-US)” then the above familiar key combinations will also work. Note, those three key combinations are in conflict with the use of the Control key in APL. Therefore to get the three APL symbols \triangleright , \cap , \cup , the Shift key needs to be used in conjunction with the Control key. Thus,

- *Control+Shift+x* produces \triangleright
- *Control+Shift+c* produces \cap
- *Control+Shift+v* produces \cup

A User Convenience

On several occasions, you may wish to execute a previously entered expression. Rather than re-keying the expression, it is possible to scroll forward or backward to such user entries. The repeated simultaneous depression of

- Control + Shift + Backspace

will cause the cursor to scroll backward to the previously entered expressions.

- Control + Shift + Enter

will cause the cursor to scroll forward to previously entered expressions. In either case, depressing the Enter key again will cause the expression to be executed. Repeated use of these keystroke combinations allows one to scroll forward or backward through successive entries. Regardless of the nature of the keyboard, these two key combinations will allow you to move to previous entered expressions without having to key in again.

In addition, you may move the cursor to the expression you wish to re-execute. Click on it and it will be re-executed.

Shifting Between Keyboards

With APL on your system, you have the option of using one of two keyboards: the normal text keyboard or the Dyalog APL IME (Input Method Editor) keyboard. Normally, when you are in Dyalog APL, the Dyalog APL keyboard is active. With the Dyalog APL keyboard, you are able to enter the APL symbols. Which keyboard is active is indicated as an icon on the right side of the taskbar: an orange stylized D indicates the Dyalog APL IME keyboard is active. A

small rectangular keyboard icon indicates that the standard text keyboard is active. Clicking on that icon produces a pop-up menu containing an entry for both keyboards. Clicking on one will activate it.

The Workspace

The term *workspace* refers to the area in which your work occurs. It reflects what you see on the session manager screen. The workspace concept isolates you from concerns about unnecessary host operating system details. The characteristics of the APL workspace are:

- Fixed size
- Only one active at a time
- Many inactive available
 - In libraries
- User named
- Accessed by
 - System Commands
 - System Functions
 - System Variables

Accessing a Workspace

When one initially enters APL, the workspace that appears in the session manager screen is the *Clear Workspace*.

It contains nothing and has default settings. As you develop programs and data, you wish to save them for later access. However, a *Clear* workspace *cannot* be saved. When you sign off whatever you did in the workspace disappears. Preserving your work and other interactions with the workspace is commonly done through system commands. A partial list follows at the end.

Let's start from the case of a clear workspace. The first step to saving its contents is to rename it, thus allowing it to be saved. The system command

```
)WSID      (Workspace ID)
```

does that. For example,

```
)WSID HSClass
```

would rename the workspace to **HSClass**. You can also specify a drive letter:

```
)WSID D:HSClass
```

This would enable the workspace to be saved onto the D drive. The actual saving is through the system command

```
)SAVE
```

However, while not mandatory, one should precede the **)SAVE**, with the system command

```
)RESET space
```

This system command does some housekeeping on the workspace.

A further listing of other system commands is found in Appendix A.

In saving a workspace, what are you saving? A saved workspace will contain only named items: variables and programs. Other keyboard activity is not saved. A record of that incidental activity is kept in the Session Log, where you may scroll back to view it.

At a later time you may wish to reactivate a previously-saved workspace. This is accomplished via the `)LOAD` system command. For example,

```
)LOAD HSClass
```

Finally, the proper way to log off of APL is via the system command

```
)OFF          (Remember to save first.)
```

Other system commands will come into play as the course continues.

Setting Default Settings

Once you have set your APL session window to suit you with, for example, opening as a full screen, and a desired font size, you can make these settings a default. You may do a sign off via the familiar “X” in the red box in the upper right corner. When you do this, a Dyalog APL pop-up window will appear. Place a check in the top box labeled “Save Session Configuration.” When you sign back on, your current settings are now the default ones.

In the window the third box should also be checked. It allows you to view and perhaps recover actions you did at a previous session.

System Commands

Command	Action
)Clear	Clear the Workspace
)Continue	Save into a Continue workspace and terminate APL
)Copy Y	Copy objects from another workspace
)Drop Y	Drop named workspace
)Ed Y	Edit object Y
)Erase Y	Erase the objects
)Fns {Y}	List user defined functions
)Lib {Y}	List workspaces in library Y
)Load Y	Load workspace Y
)Off	Terminate the session
)Ops {Y}	List user defined operators
)Pcopy Y	Protected copy
)Reset	Reset the state indicator
)Save	Save the active workspace
)Si	Show the state indicator
)Sinl	Show the state indicator with the Name List
)Vars {Y}	List global variables
)Wsid {Y}	Workspace identification
)Xload Y	Load a workspace without □LX executing

The behavior of each system command is conveyed by its name. Some need a bit more attention.

)Drop When you issue this command you discard the named workspace completely and irretrievably.

)Copy and **)Pcopy** Both will bring the named workspace contents into you currently active workspace. The **)Pcopy** will not bring in any named item that has the same name of a named item in the current active workspace. It will list such named items as Not Copied.

Both of these commands will allow you to copy in specific items by listing them following the name of the copied workspace.

)Off The command will take out of APL directly without any sort of questioning pause.

)Wsid This command by itself will return the name of the current active workspace. When this command is followed by another name and path if necessary renames the active with that name.

Appendix B: A Functional Summary

All APL data arrays are self describing.

*Associated with any APL array
is:*

Data derived from

- numbers (Real and Complex)
- characters
- heterogeneous (mixed)

Structure composed of

- *Shape:* the dimension of the array
- *Rank:* the number of dimensions
- *Depth:* the nature of nesting

APL is an array-oriented language that has a very rich set of functions and operators. Its set of functions can be subdivided loosely into different categories based on how they react to the associated attributes. Thus,

A Functional Classification

- Data Transformation
- Array Construction
- Array Inquiry
- Data Selectors and Selection
- Data Rearrangement
- Data Replacement

	Dyadic	Symbol	Monadic
Arithmetic	Add	+	Conjugate
	Subtract	−	Negate
	Multiply	×	Direction
	Divide	÷	Reciprocal
	Maximum	⌈	Ceiling
	Minimum	⌊	Floor
	Residue	⌊	Magnitude
	Power	*	Exponential
	Logarithm	⊗	Natural log
	Deal	?	Roll
	Binomial	!	Factorial
	Decode	⊥	
	Encode	⌈	
	Circle Functions	±	Pi Times
	Matrix Divide	⊞	Matrix Inverse
Type	Format	⌘	Format
	—	⌘	Execute
Logic	—	~	Not
	And / Lowest Common Multiple	^	—
	Or / Greatest Common Divisor	v	—
	Nand	~^	—
	Nor	~v	—

Circular Functions		
$(-X) \circ Y$	X	$X \circ Y$
$(1-Y*2)*.5$	0	$(1-Y*2)*.5$
Arcsin Y	1	Sine Y
Arccos Y	2	Cosine Y
Arctan Y	3	Tangent Y
$(-1+Y*2)*.5$	4	$(1+Y*2)*.5$
Arcsinh Y	5	Sinh Y
Arccosh Y	6	Cosh Y
Arctan Y	7	Tanh Y
$-8 \circ Y$	8	$(-1+Y*2)*0.5$
Y	9	Real part
+Y	10	Y
$Y \times 0J1$	11	Imaginary part
$*Y \times 0J1$	12	The phase of Y

	Dyadic	Symbol	Monadic
Array Construction	Reshape	ρ	
	Catenate	,	
	Direction	\rhd	Disclose
	Partition	\lhd	Enclose
Array Inquiry	Less Than	$<$	
	Less Than or Equal	\leq	
	Greater Than or Equal	\geq	
	Greater Than	$>$	
	Equal	$=$	
	Not Equal	\neq	
	Match	\equiv	Depth
		ρ	Shape of
		\neq	Tally
	Membership	\in	
	Find	$\underline{\in}$	

	Dyadic	Symbol	Monadic
Selector	Grade Up	\uparrow	Grade Up
	Grade Down	\downarrow	Grade Down
	Index of	ι	Interval
	Interval Index	$\underline{\iota}$	Where
Selection	Pick	\supset	
	Take	\uparrow	First
	Drop	\downarrow	
	Bracket Indexing	$[I; J; \dots]$	
	Indexing	$\boxed{}$	
	Without	\sim	
	Left	\lhd	Same
	Right	\rhd	Same
Data Rearrangement	Depth	$,$	Ravel
	Shape of	ϵ	Enlist
	Rotate	ϕ	Reverse
	Transpose	ϕ	Transpose
Data Replacement	<i>(Selective Expression) ← Data</i>		

Illustration: Data Replacement

```

Name1←('Polivka' 'Ray')('Polivka@APLclass.com')
Name1
Polivka Ray    Polivka@APLclass.com
1 2>Name1
Ray
(1 2>Name1)←'Raymond'
Name1
Polivka Raymond    Polivka@APLclass.com

```

n.b.: $V \triangleright A$ is Pick, where V is a path through a nested array to a single nested item of A .

System Control		
Dyadic	Symbol	Monadic
Namespace execute	$\underline{\text{⌵}}$	Execute

Operators	
Reduction	$f /$
Compression	$d /$
Scan	$f \backslash$
Outer Product	$\circ . f$
Inner Product	$f . g$
Each	$f ``$
Commute	$f \tilde{\sim} g$
Compose	$f \circ g$
Power	$f \dot{*} g$
Rank	$f \ddot{o} g$
At	$f @ g$
Key	$f \boxtimes$
Stencil	$f \boxminus g$
Variant	$f \boxplus g$
Spawn	$f \&$

Appendix C: The Computer System

Introduction

Before we address programming in APL, we can look at the nature of the computer system upon which programs will run. A computer system consists of two parts: the hardware and the software. The hardware portion is the physically tangible part of the system composed of electronics packaged into a visible object. The component parts of the computer hardware consists of *memory*, a *central processing unit*, *storage units* and *input/output devices*.

Memory exists at several levels distinguished by differences in data capacity and access speed. The term “memory” usually refers to the vast memory within the computer hardware. This is where data and active programs exist. It is the fastest but it is volatile, its contents disappearing when the power is turned off the computer.

The *central processing unit* interacts with the memory retrieving both data and programs in order to execute the program’s instructions. This portion of the computer hardware, in further detail, consists of

- an *arithmetic and logical unit (ALU)* that executes program instructions,
- *registers* to hold data,
- a *control unit* that translates instructions into zeros and ones and presents them to the ALU, and
- a *clock* that determines when various events in the hardware occur.

The *storage units* of a computer system are that part of the system that are capable of retaining information when the computer is no longer powered on. The data capacity can be quite large but access times will be slower.

The *input output units* serve as the interface to the users and outside world. For example, today there are keyboards, printers, flash drives, and monitors.

Today, this generic description of a computer system is built in a great variety of different forms.

The *software* of a computer system consists of coded instructions that are to be executed on the hardware of the computer system. This software runs on the computer hardware to produce desired meaningful results. This is where programming enters the picture.

Programming

What is computer programming? Computers are present to solve problems for us. You do this through the use of a programming language, a language which the user can learn, understand, and use. One might ask “What does a programming language look like?” Here we are about to discuss the *Dyalog APL* programming language. First of all, note it is a language. Languages are created to communicate. Our spoken languages are designed for communicating with other human beings. Programming languages, such as *Dyalog APL*, are designed to communicate with computers. Such languages are more precise than spoken languages. Nonetheless, they are languages each with their own grammar, terminology, and even idioms. The computer itself only manipulates zeros and ones. Thus, there exists a gap between the actions of a computer and the needs of a human being. This gap has many hierarchical layers. Fortunately, programming languages mask out the need to know the hierarchical levels below them.

Brian Hayes¹⁶ states the hierarchical layers nicely:

“Computer science has evolved a hierarchy of conceptual layers that hide the details of layers below them. At the bottom are physical entities such as transistors and electronic circuitry. Next come logic gates (And, Or, etc.), Which operate on symbols (true and false, or 0 and 1) rather than voltages and currents. The gates are assembled into registers, adders, and the like; an instruction set defines commands for manipulating data within these components. Finally, the details of the machine instruction set are hidden by the constructs of high-level programming languages: procedures, iterations, arrays, list, and so on.”

At the higher levels, software known as programs comes into play. From a user’s point of view, the purpose of a computer is to solve his or her problems. More specifically to aid in the process of solving them, particularly problems that are difficult because of large amounts of data or have complicated algorithms. In addition to such cases, programs can be created to solve the many relatively easy daily problems. Word processing is an excellent example. Even for such problems the software may be quite sophisticated. The actual process of creating a program is often call *coding*. Coding is done only after a procedure has been established to solve the current problem. More formally, an *algorithm* has been developed.

Within software, hierarchical layers also exists. Let us examine these layers in order to understand where the APL programming language fits in the hierarchy.

Machine Language

The first level of programming languages is the *machine language*. With it, one communicates directly with the hardware of the computer. The language, of course, is machine dependent. The original code was in binary. For example,

```
1111 0101 0000 0011
```

...but quickly software packages were created that allowed a coder to write instructions like

```
L5 03
```

Later on, one wrote and read instructions using a Hexadecimal number system.

¹⁶ Hayes, B. 2014 “*Programming Your Quantum Computer*,” American Scientist, Vol. pg. 22

Assembler Languages

These languages are low-level languages which permit the use of more meaningful mnemonics. For example,

Load Register3

Such languages are still computer dependent. Current assembler languages such as C and C++ have added features not directly related to hardware.

High-Level Languages

These languages are designed to be computer independent. Some of the earliest high-level languages are Fortran and Cobol. These languages are translated via a compiler or an interpreter into a lower level of computer dependent language which is then executed. There are literally hundreds of programming languages all with degrees of machine independence. For example there is,

APL, Java, Perl, Python, Visual Basic.

Among the newest languages evolving today are *Go, Scala, Rust, and Swift.*

Domain-Specific Languages

In addition to general purpose programming languages, some languages are specifically oriented to serve a specific area or enterprise. Among such languages are

SAS, R, Excel, Ruby.

Programming languages are further categorized by how they are executed. All higher-level languages need a means of translating their computer independent code into specific computer dependent machine code. The 2 major paths used are via a *compiler* or an *interpreter*.

A *compiler* accepts a collection of computer independent code (Source Code) and translates the collection directly into intermediate computer dependent code (Object Code). Then the object code is executed. *FORTRAN* and *COBOL* are examples of compiled code.

An *interpreter* accepts computer independent code translating and executing this source code essentially an instruction at a time. *APL, Java, and Python* are interpretive languages. The common thread of interpreted programming languages is the pattern *REPL* which stands for Read, Evaluate, Print, and Loop. Thus interpretive languages allow you to enter a statement and get an immediate response.

In summary, programming languages continue to evolve to meet user's needs and the more sophisticated hardware being developed. Regardless, any program instruction will either

- manipulate or transform data,
- determine the execution sequence, conditionally or unconditionally, of the next program instruction, or
- retrieve and preserve data.

Such set of instructions, when collected together, can be given a name and is called a *program*. This allows one to execute the set of instructions again and again with different data through just the use of the function name and data supplied to it..

Operating Systems

There is another bit of software that sits between the hardware and the user writing in a programming language. It is the *operating system* which acts as an interface environment between programming languages and the computer hardware. The main purpose of an operating system is to enable the computer resources to be efficiently shared. In the process it must manage and enforce the allocation of the computer's resources, retain users' data, protect the users from each other and perform accounting of the system resources. Operating systems are large programs. They have their own domain specific programming languages too. The operating system language must be computer specific.

Like the multitude of programming languages, there exist a wide variety of operating systems languages such as the *Windows* family, *Linux*, and *iOS*. In any case, the operating system is the software environment in which users' programming languages are executed.

History of APL

As you have just read, APL is also an interpretative language. While it is not a new language, even today it is an evolving one. The creator of the language was Dr. Kenneth E. Iverson, at the time a Harvard professor. Originally, he had no interest in creating a programming language, he was interested in providing a more consistent mathematical notation. In 1962 he published a book in which he expounded his proposed language. The title of the book is "*A Programming Language*."

At the suggestion of Adin Falkoff, this book title led to the name APL for this programming language. After he had joined IBM Research, with the help of four Stanford graduate students and the arrival of IBM System 360 computers, APL came out as an IBM Program Product in June 1966. It was the first commercially-successful *interpretative* programming language.

In the preface of his book he wrote:

"It is the central thesis of this book that the descriptive and analytic power of an adequate programming language amply repays the considerable effort required for its mastery."

Vita

Dr. Raymond Polivka has been associated with APL since 1968. Before retiring from a 35-year career with IBM, he was responsible for its company-wide APL education. He has personally taught a range of APL classes worldwide.

In addition to publishing many papers on APL, he has co-authored three APL textbooks:

- *APL: the Language and Its Usage* (1975)
- *APL2 At A Glance* (1988)
- *APL2 In Depth* (1995)

He has served as the chairman of SIGAPL, the Special Interest Group for APL, and the Executive Editor of SIGAPL's Journal, *Quote Quad*. In 1990, he received SIGAPL's Kenneth E. Iverson Award "for Outstanding Contribution to the Development and Application of APL."

Dr. Polivka holds an AB from North Central College, and a PhD from the University of Illinois.

This DRAFT is being distributed
for the purpose of getting feedback
prior to publication.

Please send your comments to
Polivka@APLclass.com

Use a subject line: Book
Thank you!

© 2022, Polivka Associates. All rights reserved.

No part of this material may be reproduced, transmitted, transcribed, stored on a retrieval system, or translated into any language or digital format, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Polivka Associates, 60 Timberline Drive, Poughkeepsie, New York, 12603–5546 USA