Arithmetic

Kenneth E. Iverson

Preface

Arithmetic is the basic topic of mathematics. According to the *American Heritage Dictionary* [1], it concerns “The mathematics of integers under addition, subtraction, multiplication, division, involution, and evolution.”

The present text differs from other treatments of arithmetic in several respects:

> The provision of simple but precise definitions of the counting numbers and other notions introduced.
>
> The use of simple but precise notation that is executable on a computer, allowing experimentation and providing a simple and meaningful introduction to computer programming.
>
> The introduction and significant use of fundamental mathematical notions (such as vectors, matrices, Heaviside operators, and duality) in simple contexts that make them easy to understand. This lays a firm foundation for a wealth of later use in mathematics.
>
> Emphasis is placed on the use of guesses by speculation and criticism in the spirit of Lakatos, as discussed in the treatment of proofs in Chapter 5.

The thrust of the book might best be appreciated by comparing it with Felix Klein’s *Elementary Mathematics from an Advanced Standpoint* [2]. However, I shun the corresponding title *Arithmetic from an Advanced Standpoint* because it would incorrectly suggest that the treatment is intended only for mature mathematicians; on the contrary, the use of simple, executable notation makes it accessible to any serious student possessing little more than a knowledge of the counting numbers.

Like Klein, I do not digress to discuss the importance of the topics treated, but leave that matter to the knowledge of the mature reader and to the faith of the neophyte.

Table of Contents

Chapter 1

Introduction

A. Counting Numbers

The list `1 2 3 4 5 6 7 8 9 10 11 12` shows the first dozen *counting numbers*, and any reader of this book could extend the list to tedious lengths. Although this definition by example captures the basic idea, it fails to address related questions such as:

1. Do counting numbers continue forever?
2. Are there other numbers that precede the first counting number?
3. Are there other numbers between the counting numbers or elsewhere?

These questions were addressed a century ago by Peano, who began by introducing the notion of a successor "operation" which, when applied to any counting number, produced its successor. For example, `successor 3` would produce `4`.

We will denote the successor operation by the two-character word `>:` . For example:

```
   >: 3
4

   >: 999
1000
```

The foregoing is an example of dialogue with the computer. Because the notation used here (and throughout the book) can be executed by a computer provided with the language **J** (available from website *jsoftware.com*), every expression used can be tested by executing it, as can related expressions that the reader may wish to experiment with. For example, one might apply the successor to *lists* of counting numbers as follows:

```
   >: 1 2 3 4 5 6 7 8 9 10 11 12
2 3 4 5 6 7 8 9 10 11 12 13

   >: 2 4 6 8 10
3 5 7 9 11
```

Is there a *last* or *largest* counting number? Peano answered this by asserting that every counting number has a distinct successor, thus introducing the idea of an unbounded or infinite list of counting numbers.

B. Integers

Since `7` is the successor of `6`, we may also say that `6` is the *predecessor* of `7`, and introduce a predecessor operation denoted by `<:` . For example:

```
   <:3 5 7 9 11
2 4 6 8 10

   >:2 4 6 8 10
3 5 7 9 11
```

It would be convenient if the predecessor (like the successor) applied to *all* counting numbers, but since `1` is the first counting number, its predecessor cannot be a counting number. We therefore introduce a wider class of numbers, in which every member has a predecessor as well as a successor. Thus:

```
   <: 1
0
   <: 0
_1
   <: _1
_2
```

This wider class of numbers is called the *integers*, and includes *zero* (`0`), as well as *negative* numbers (`_1 _2 _3` etc.).

It is helpful to form the habit of looking up any new technical term in a good dictionary; even if the term is already familiar, its etymology often provides useful insight. For example, in the *American Heritage Dictionary* (a dictionary to be recommended because of its method of treating etymology) the definition of *integer* refers to the Indo-European root *tag* that means "to touch; handle". This with the prefix *in-* (meaning *not*) implies that an integer is untouched, or whole; in contrast to one that is "fractured", like one of the *fractions* one-half, one-quarter, etc.

Similarly, the word *infinite* introduced in Section A will be found to mean *not* (in) finite, or without finish.

C. Inverses

The predecessor operation (`<:`) is said to be the *inverse* of the successor (`>:`) because it "undoes" its work. For example, `<:>: 8` yields `8`, and the same relation holds for any integer. Thus:

```
   >:1 2 3 4 5 6                 <:>:1 2 3 4 5 6
2 3 4 5 6 7                   1 2 3 4 5 6
```

In the original definition the successor applied only to the counting numbers. We now re-define it to apply to all integers by defining it as the inverse of predecessor. For example:

```
   >:<: _3 _2 _1 0 1 2
_3 _2 _1 0 1 2
```

D. Domains

The successor `>:` defined in Section A applied only to counting numbers, and they would be said to be its *domain* (over which it "ruled"). In defining the predecessor in Section B it became necessary to extend its domain to the *integers*, that also included zero and the negative numbers. By re-defining the successor as the inverse of the predecessor, we also extended its domain to the integers.

We will find that the introduction of further operations (such as the inverse of "doubling") will require further extensions of domains. However, to keep the development simple, we will restrict attention to simple domains as far as possible.

E. Nouns and Verbs

The successor operation `>:` can be said to "act upon" a counting number to produce a result, and is therefore analogous to an "action word" or *verb* in English. Similarly, the numbers to which the verb `>:` applies are analogous to nouns in English.

We will soon see that the terms *verb* and *noun* lead to further important analogies with adverbs, conjunctions, and other parts of speech in English. We will therefore adopt them, even though other terms (*function*, *operator*, and *variable*) are more commonly used in mathematics. However, *function* will sometimes be used as a synonym for *verb*.

F. Pronouns and Proverbs

Consider the following use of the pronoun `it` :

```
   it=: 1 2 3 4 5 6
   <: it
0 1 2 3 4 5

   >:<: it
1 2 3 4 5 6
```

The copula `=:` behaves like the copulas *is* and *are* in English, and the first sentence would be read aloud as "`it` is the list of counting numbers `1 2 3 4 5 6`" or as "`it` is `1 2 3 4 5 6`".

In English the names used for pronouns are restricted to a very few, such as *it*, *he*, and *she*; they are not so restricted here. For example:

```
   zero=: 0
   neg=: _1 _2 _3
   list6=: it
   list6,zero,neg
1 2 3 4 5 6 0 _1 _2 _3
```

A *pro*verb is used to stand for a verb, just as a *pro*noun is used to stand for a noun. (The word *proverb* in this sense is found only in larger dictionaries.) For example:

```
   increment=: >: decrement=: <:
   increment list6,zero,neg
2 3 4 5 6 7 1 0 _1 _2

   inc=: increment
   inc list6
2 3 4 5 6 7
```

G. Conjunctions

The phrase *Run and hide* expresses an action performed as a sequence of two simpler actions, and in it the word *and* is said to be a *copulative conjunction*. We will use the symbol `@` to denote an analogous conjunction. For example:

```
   add3=: >: @ >: @ >:
   add3 1 2 3 4 5 6
4 5 6 7 8 9

   identity=: <: @ >:
   identity 1 2 3 4 5 6
1 2 3 4 5 6
```

Although the verb **identity** defined above makes no change to its argument, it is an important verb, so important that it is given its own symbol. Thus:

```
   ] 1 2 3 4 5 6
1 2 3 4 5 6
```

Although a verb for the twelfth successor could be expressed by repeated use of `@`, it would be tedious, and we introduce a second conjunction illustrated below:

```
   list=: 1 2 3 4 5 6
   >:^:3 list
4 5 6 7 8 9

   >:^:12 list
13 14 15 16 17 18

   <:^:6 list
_5 _4 _3 _2 _1 0
```

The conjunction **^:** is called the *power* conjunction; it applies its left argument (the verb to its left) the number of times specified by its noun right argument.

H. Addition And Subtraction

The examples of the preceding section illustrate the fact that if **n** is any counting number, then the verb **>:^:n** *adds* **n** to its argument, and **<:^:n** *subtracts* **n**.

For example :

```
   n=: 5
   abc=: 10 11 12 13 14 15
   >:^:n abc
15 16 17 18 19 20

   <:^:n abc
5 6 7 8 9 10

   abc+n                        abc-n
15 16 17 18 19 20            5 6 7 8 9 10
```

The last two examples introduce the notation commonly used for addition and subtraction, and the whole set of examples essentially defines them in terms of the simpler successor and predecessor of Peano.

I. Verb Tables

Two lists can be added and subtracted as illustrated below:

```
   a=: 0 1 2 3 4 5
   b=: 2 3 5 7 11 13
   a+b                        a-b
2 4 7 10 15 18             _2 _2 _3 _4 _7 _8

   a+a
0 2 4 6 8 10
   a-a
0 0 0 0 0 0

   a +/ b
2 3  5  7 11 13
3 4  6  8 12 14
4 5  7  9 13 15
5 6  8 10 14 16
6 7  9 11 15 17
7 8 10 12 16 18

   a +/ a
0 1 2 3 4  5
1 2 3 4 5  6
2 3 4 5 6  7
3 4 5 6 7  8
4 5 6 7 8  9
5 6 7 8 9 10
```

The last two examples show *addition tables* that add *each* item of the first argument to *each* item of the second in a systematic manner. The verb `+/` is formed by applying the *adverb* `/` to the verb `+` , and is usually referred to as the verb "plus table". The adverb `/` applies uniformly to other verbs, and we can therefore produce subtraction tables as follows:

```
   a-/a                       b-/1 2
0 _1 _2 _3 _4 _5            1  0
1  0 _1 _2 _3 _4            2  1
2  1  0 _1 _2 _3            4  3
3  2  1  0 _1 _2            6  5
4  3  2  1  0 _1           10  9
5  4  3  2  1  0           12 11
```

To make clear the meaning of a verb table, draw a vertical line to its left and write the left argument vertically to the left of it; draw a horizontal line above the table, and enter the right argument horizontally above it. We can produce such an annotated display of a verb table by using the adverb **`table`** instead of `/`, as follows:

```
   a +table b
+-+---------------+
| |2 3  5  7 11 13|
+-+---------------+
|0|2 3  5  7 11 13|
|1|3 4  6  8 12 14|
|2|4 5  7  9 13 15|
|3|5 6  8 10 14 16|
|4|6 7  9 11 15 17|
|5|7 8 10 12 16 18|
+-+---------------+

   a-table a
+-+----------------+
| |0  1  2  3  4  5|
+-+----------------+
|0|0 _1 _2 _3 _4 _5|
|1|1  0 _1 _2 _3 _4|
|2|2  1  0 _1 _2 _3|
|3|3  2  1  0 _1 _2|
|4|4  3  2  1  0 _1|
|5|5  4  3  2  1  0|
+-+----------------+
```

J. Relations

Any two integers **`a`** and **`b`** are related in certain simple ways: **`a`** *precedes* (or *is less than*) **`b`**; **`a`** *equals* **`b`**; or **`a`** *follows* (or *is greater than*) **`b`**. We introduce the verbs `<` and `=` and `>` whose results show whether the particular relation holds between the arguments. For example:

```
   1<3              1=3                 1>3
1                0                  0

   a=: 1 2 3 4 5
   b=: 6-a
   b
```

```
5 4 3 2 1

   a<b
1 1 0 0 0

   a=b                            a>b
0 0 1 0 0                      0 0 0 1 1

   a</b
1 1 1 1 0
1 1 1 0 0
1 1 0 0 0
1 0 0 0 0
0 0 0 0 0

   a=/b                           a>/b
0 0 0 0 1                      0 0 0 0 0
0 0 0 1 0                      0 0 0 0 1
0 0 1 0 0                      0 0 0 1 1
0 1 0 0 0                      0 0 1 1 1
1 0 0 0 0                      0 1 1 1 1
```

A result of `1` indicates that the relation holds, and `0` indicates that it does not; it is reasonable to read the *ones* and *zeros* aloud as "true" and "false". The final example is a greater-than table.

K. Lesser-Of and Greater-Of

The *lesser* of (or *minimum* of) two arguments is the one that precedes (or perhaps equals) the other; the verb `<.` yields the lesser of its arguments. For example:

```
   a                 b
1 2 3 4 5         5 4 3 2 1

   a<.b              a>.b
1 2 3 2 1         5 4 3 4 5

   a<./b
1 1 1 1 1
2 2 2 2 1
3 3 3 2 1
4 4 3 2 1
5 4 3 2 1
```

L. List And Table Formation

Although any list can be specified by listing its members, certain lists can be specified more conveniently. The *integers* verb `i.` produces lists or tables of integers (beginning with zero) that are convenient in producing verb tables. For example :

```
   ] a=:i. 5
0 1 2 3 4

   a<./a
```

```
0 0 0 0 0
0 1 1 1 1
0 1 2 2 2
0 1 2 3 3
0 1 2 3 4

   4-a
4 3 2 1 0

   1+a
1 2 3 4 5
   i. _5
4 3 2 1 0
   i.3 4
0 1  2  3
4 5  6  7
8 9 10 11
```

The verb `#` *replicates* its right argument the number of times specified by the left:

```
   3#5
5 5 5

   5#3
3 3 3 3 3

   2 3 4 # 6 7 8
6 6 7 7 7 8 8 8 8

   b=: _2 + i. 5
   b
_2 _1 0 1 2

   c=:b>0
   c
0 0 0 1 1
   c#b
1 2
```

The verb `$` "shapes" its right argument, using cyclic repetition of its items as needed:

```
   8$2 3 5
2 3 5 2 3 5 2 3
```

```
   3 4$2 3 5
2 3 5 2
3 5 2 3
5 2 3 5
```

M. Punctuation

Although the two sentences:

The teacher said he was stupid

The teacher, said he, was stupid

differ only in punctuation, they differ greatly in meaning.

Arithmetic sentences may also be punctuated (by paired parentheses) as illustrated below:

```
   (8-3)+4
9
   8-(3+4)
1
   8-3+4
1
```

The last sentence illustrates the behaviour in the absence of parentheses: in effect, the sentence is evaluated from right to left or, equivalently, the right argument of each verb is the value of the entire phrase to its right.

Punctuation makes possible many useful expressions. For example:

```
   c=: 2 7 1 8 2 8
   (c=2)#c
2 2

   ((c=2)>.(c=8))#c
2 8 2 8

   (c<2)>.(c=2)
1 0 1 0 1 0
```

The last sentence can be read as "`c` is less than or equal to `2`". It is equivalent to the verb `<:` in the expression `c<:2`.

The beginner is advised to use fully-parenthesized sentences even though some of the parentheses are redundant. Thus, write `(c<2)>.(c=2)` even though `(c<2)>.c=2` is equivalent.

N. Insertion

```
   a=: 2 7 1 8 2
   2+7+1+8+2
20
   +/a
20
```

The foregoing sentences illustrate the fact that the adverb `/` produces a verb that "inserts" its verb left argument between the items of the argument of the resulting verb `+/` . Insert applies equally to other verbs. For example:

```
   >./a                        2>.7>.1>.8>.2
8                           8

   sum=:+/

   max=:>./
```

```
   min=:<./

   sum a
20

   spread=: (max a)-(min a)
    range=: (min a)+i. >:spread
   range
1 2 3 4 5 6 7 8
```

O. Multiplication

```
   m=:3
   n=:5
   n#m
3 3 3 3 3

   +/n#m
15
```

The final result above is clearly the *product* of **m** and **n**, and the sentences essentially define multiplication in terms of repeated addition. In mathematics the product verb is denoted in a variety of ways; we will use ***** as in:

```
   m*n
15

   dig=: 1+i. 6             odds=: 1+2*i. k=: 6
   dig                      odds
1 2 3 4 5 6              1 3 5 7 9 11

   */dig                    +/odds
720                      36
   !#dig                     k*k
720                      36
```

The last two sentences on the left illustrate the definition of a new verb, *factorial*, denoted by **!** .

P. Power

```
   m=: 3              n=: 5
   n#m                */n#m
3 3 3 3 3          243
```

The final result above is called the **n**th power of **m**, or **m** to the power **n**. Comparison with Section O will show that power is defined in terms of multiplication in the same way that multiplication is defined in terms of addition.

In most math texts there is no symbol for power, it being denoted by showing the second argument as a superscript. We will adopt the symbol **^** used by de Morgan [3] about a century ago. For example:

```
   m^n                  3^5
243                  243

   (3^5)*(3^2)          3^(5+2)
2187                 2187
```

As suggested by the equivalence of the last two sentences, **(a^b)*(a^c)** is equivalent to **a^(b+c)**. The reason for this can be seen by substituting the definition of power given above:

```
   (3^5)*(3^2)                (*/5#3)*(*/2#3)
2187                       2187

   (5+2)#3                    */(5+2)#3
3 3 3 3 3 3 3              2187
```

Q. Summary

The main results of this chapter may be summarized as follows:

1. The idea of the counting numbers is formalized and extended to infinity by introducing the notion that every counting number has a *successor*; it is extended to include *zero* and *negative numbers* by introducing the notion of *predecessor*, inverse to *successor*.

2. Symbols are introduced to denote successor and predecessor (**>:** and **<:**); because they specify *actions* they are called *verbs*, and the integers they act upon are called *nouns*.

3. The copula **=:** is introduced to assign a name (called a *pro*noun) to a noun or list of nouns and to assign a name (called a *pro*verb) to a verb.

4. Conjunctions (**@** and **^:**) are introduced to define verbs that are specified by a sequence of simpler verbs.

5. Addition is defined in terms of a sequence of successors; subtraction is defined in terms of predecessors.

6. Verb tables are introduced to display the behaviour of addition, subtraction, and other verbs that apply to two arguments, such as relations (**< = >**) and minimum and maximum (**<. >.**).

7. Parentheses are introduced as punctuation, that is, to specify the order in which phrases in a sentence are to be interpreted.

8 An adverb called *insert* (denoted by **/**) is introduced to insert a verb between items of a list argument, and **+/** is used with *replication* (**#**) to define multiplication in terms of repeated addition; power is defined in terms of repeated multiplication.

We will now summarize all of the notation used. This summary may be useful for reference, but because related symbols are used for related ideas, it should also be studied for mnemonic aids. Succeeding chapters conclude with similar summaries of notation, and all notation is available from the J Dictionary discussed in Book 1.

The table shows the verbs in three columns, each headed by the final character (dot or colon) of the verbs in that column: the first row shows *Less than* (`<`) in the first column, *Lesser of* (`<.`) in the second, and *Predecessor* (`<:`) in the third:

Verbs And Copula		**.**	**:**
`<`	Less than	Lesser of (Min)	Predecessor
`>`	Greater than	Greater of (Max)	Successor
`=`	Equals		Copula
`+`	Add		
`-`	Subtract		
`*`	Multiply		
`^`	Power		
`!`	Factorial		
`]`	Identity		
`#`	Replicate		
`$`	Shape		
`,`	Catenate		
`i`		Integers	

Adverbs

`/` Insert (when used with one noun argument, as in `+/b`)

Table (when used with two noun arguments, as in `a+/b`)

Conjunctions

`@` Atop (defines a verb by a sequence, as in `>:@>:@>:`)

`^:` Power (`>:^:3 is >:@>:@>:`)

In conventional math, the symbol `-` denotes *subtraction* when used with two arguments (`a-b`) and *negation* when used with one (`-b`). We will adopt this usage, defining `-b` by `0-b`.

The thoughtful reader may have noticed such usage in this chapter: the verbs produced by the adverb `/` (as shown above), and the `<:` used for *predecessor* throughout, but used dyadically (that is, with two arguments) for *Less or equal* in Section M. This ambivalent use of verbs is discussed fully in Chapter 2.

R. On Language

Notation, the term normally used to refer to the mode of expression in math, is defined (in the *AHD*) as “A system of figures or symbols used in specialized fields ... ”. An executable notation such as that used here is normally called a *programming language*; we will use the terms *notation* and *language* interchangeably.

Programming languages are commonly taught in specific courses, prerequisite to courses in topics that employ them. In mathematics, on the contrary, notation is not taught as such, but is introduced in passing as required by the subject. The same approach is adopted in this text.

Any reader interested in using the notation in topics other than those treated here should consult Section 9 L.

In a math course there is little reason for a student to be curious or concerned about notation that has not yet been used. In using a programming language the situation is somewhat different; a student who already knows something of the possibilities of computer programming may feel frustrated at not knowing what symbols to use for operations that she knows must be available in the language.

There are several avenues open to the student who may be more interested in the language than in the treatment of arithmetic:

1. Press key F1 in the top row to display the vocabulary of J. Then click the mouse on any desired entry in the vocabulary to display its definition. Press Esc to remove the display.

2. Use the computer to experiment with various facilities, and therefore to explore their definitions.

3. Range ahead to the *On Language* sections that conclude Chapters 2 and 9.

Exercises

In exercises first write (or at least sketch out) the result of each sentence without using the computer; then enter the sentence on the computer to check your answer.

In using the computer, it will be more efficient if you familiarize yourself with the available editing facilities. In particular, these allow you to revise entries being prepared, and to recall earlier entries for re-entry. Also learn to use expressions such as:

`names 0`	To display the names used for pronouns
`names 1`	To display the names used for adverbs
`names 2`	To display the names used for conjunctions
`names 3`	To display the names used for proverbs
`erase <'abc'`	To erase the name `abc`

Letters such as A and B in the labels below indicate the sections to which the associated experiments are relevant. Refer back to these sections for any needed help:

A1
```
>:12345

>:1 2 3 4 5

>:>:>:>:1 2 3 4 5
```

B1
```
<: _12345
<:_1 _2 _3 _4 _5
<:<:<:<:<:1 2 3 4 5
<:<:>:>:1 2 3 4 5
>:<:>:<:1 2 3 4 5
```

F1
```
a=:1 2 3
b=:4 5
>:a
a,b
>:a,b
```

F2
```
z=:0
n=:_5 _4 _3 _2 _1
n,z,a,b
b,a,z,n
```

F3
```
wax=: >:
wane=:<:
wax wax wane n,z,a,b
```

G1
```
list=:1 2 3 4 5
right=:>:@>:
left=:<:@<:
right list
left list
left right list
] list
```

G2
```
decade=:>:^:10
decade list
century=:decade^:10
century list
>:^:10^:10 list
```

G3 First review the discussion of inverses in Section C. Then enter the following sentences on the computer, observe their results, and try to state the effect of the power conjunction with negative right arguments:

```
>:^:_1 list

<:^:_1 list

>:^:_3 list

decade^:_1 list

decade^:2 decade^:_2 list
```

I1 Reproduce on the computer the last two tables of Section I.

J1 The verbs **over** and **by** used in the following sentences were defined and illustrated in Section I. As usual, first sketch the result of each sentence by hand before entering it on the computer:

```
d=: 0 1 2 3 4

d by d over d</d

d by d over d=/d

d by d over d+/d

d by d over d-/d
```

J2 Repeat Exercise J1 using the list `e=:_3 _2 _1 0 1 2 3` instead of the list `d`.

K1 Repeat Exercises J1 and J2 for the verbs `>.` and `<.`, that is, for tables of maximum and minimum.

M1 An integer such as `14` that can be written as the sum of some integer with itself is called an *even* number; a number such as `7` that cannot is called *odd*. Write an expression using the verb `i.` to produce the first twenty even numbers. Do not look at the answer below until you have tested your answer on the computer.

Answer: (i.20)+(i.20)

M2 Write an expression for the first 20 odds.

N1 Review Section M and note that the unparenthesized sentence `2-7-1-8-2` is equivalent to `2-(7-(1-(8-2)))` . Then evaluate the sentence and verify that your result agrees with `-/2 7 1 8 2`.

Evaluate and compare the results of the following sentences:

```
-/2 7 1 8 2

(+/2 1 2)-(+/7 8)
```

Then state in simple terms what the verb `-/` produces, and test your statement on other lists (including lists with both odd and even numbers of items).

Answer: `-/ list` produces the *alternating sum*, the sum of every other item of the list diminished by the sum of the remaining items.

O1 Construct the multiplication table produced by the sentence `(2+i.9)*/(2+i.9)` and observe that its largest item is `100`. Note that the table cannot contain *prime* numbers (which cannot be products of positive integers other than themselves and `1)`. Examine the table to determine all of the primes up to `9`.

P1
```
b=:i.7

b by b over b^/b

a=:b-3

a by b over a^/b
```

Chapter 2

Properties of Verbs

A. Valence, Ambivalence, And Bonds

In the phrases **`a-b`** and **`a<:b`** and **`a+/b`** the verbs "bond to" two arguments and (adopting an analogous term from chemistry) we say that in this context the verbs have *valence 2*; in the expressions **`-b`** and **`<:b`** and **`+/b`** the same verbs have valence 1.

From these examples it is clear that the verbs are *ambivalent*, the valence being determined by the context in which they are used. We also say that a verb used with valence 1 is used *monadically*, or *is a monad*; a verb used with valence 2 is a *dyad*.

In the phrase **`3&*`** the conjunction **`&`** *bonds* the noun **`3`** to the verb **`*`** to produce a monad. Thus:

```
   triple=: 3&*
   triple a=: 1 2 3 4
3 6 9 12
   square=: ^&2
   square a
1 4 9 16

   ^&3 a
1 8 27 64
```

Although **`a`** is the list **`1 2 3 4`**, it should be noted that the phrase **`^&3 1 2 3 4`** is *not* equivalent to **`^&3 a`**, because the sequence **`3 1 2 3 4`** is treated as a single list that is bonded to **`^`** to form a verb. However, **`^&3 (1 2 3 4)`** and **`^&3 a`** *are* equivalent.

The bond conjunction is extremely prolific because its use with any dyad **`d`** generates two families of monads, one using left bonding (**`n&d`**) and one using right bonding (**`d&n)`**. For example, with right bonding the verb **`^`** produces the square, cube, and higher powers; with left bonding it produces *exponential* verbs.

The conjunction **`@`** introduced in Section 1 G *composes* two verbs, as in **`i.@- 3`** to yield **`2 1 0`**; the verb **`i.@-`** also has a dyadic meaning, as in **`8 i.@- 3`** to yield **`0 1 2 3 4`**. In general, **`v1@v2 b`** is equivalent to **`v1 v2 b`**, and **`a v1@v2 b`** is equivalent to **`v1 (a v2 b)`**. In effect, the monad **`v1`** is applied "atop" the dyad **`v2,`** and the conjunction **`@`** (denoted by the commercial *at* symbol) is called *atop*.

B. Commutativity

The dyads `+` and `*` yield the same results if their arguments are interchanged or "commuted", and they are therefore said to be *commutative*. For example:

```
   3+5            5+3                  (3*5)=(5*3)
8              8                    1
```

The dyad produced by the *commute* or *cross* adverb `~` "crosses" the bonds of the verb to which it is applied. Moreover, the monad produced by `~` duplicates its single argument. For example:

```
   3-~5                 5-3
2                    2

  +~3                   ^~3
6                    27

   */~i.5
0 0 0  0  0
0 1 2  3  4
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16
```

C. Associativity

Compare the results of the following pairs of sentences, which differ only in the "associations" produced by different punctuations:

```
   (4+3)+(2+1)              4+((3+2)+1)
10                       10
   (4-3)-(2-1)              4-((3-2)-1)
0                        4
   (4>.3)>.(2>.1)           4>.((3>.2)>.1)
4                        4

   (4*3)*(2*1)              4*((3*2)*1)
24                       24

   (4^3)^(2^1)              4^((3^2)^1)
4096                     262144
```

Those verbs (`+ >.` and `*`) that yield the same results are examples of *associative* verbs; the others are *non-associative*.

D. Distributivity

The monad `>:` is said to *distribute over* the dyad `<.` because a sentence such as `(>:7) <. (>:4)` has the same result as the corresponding sentence `>:(7<.4)` in which the monad `>:` is "distributed over" the result of the dyad `<.` . Observe the further tests of distributivity:

```
   a=:7
   b=:4
   triple=: *&3
   (triple a) + (triple b)          triple (a+b)
33                               33

   (triple a) - (triple b)          triple (a-b)
9                                9

   (*&3 a) <. (*&3 b)               *&3 (a<.b)
12                               12

   (-&3 a) <. (-&3 b)               -&3 (a<.b)
1                                1

   (3&- a) <. (3&- b)               3&- (a<.b)
_4                               _1
```

In the last two pairs of sentences it appears that although the monad `-&3` (which subtracts `3` from its argument) distributes over minimum, the monad `3&-` (which subtracts its argument from `3`) does not.

This point is made to show the pitfall in a common practice in math, where it is stated that the *dyad* `*` distributes over addition, rather than stating (as we do here) that the family `*&n` of right bonds of `*` distributes over addition.

Because `*` is commutative, the left bond `c&*` is equivalent to the right bond `*&c`, and both distribute over addition. However, in the case of a non-commutative verb such as subtraction, it is possible that a right bond with a given dyad distributes while the corresponding left bond does not. In such a case it is clearly incorrect to say that the *dyad* distributes, and one is led to statements such as "`-` distributes *to the right* over minimum".

A *linear* verb (to be discussed further in Chapter 9) is one that distributes over addition.

E. Symmetry

If a dyad `d` (such as `+` or `*` or `>.`) is both associative and commutative, then the monad `d/` produced by insertion is said to be *symmetric*, because it produces the same result when the argument list to which it applies is re-ordered or *permuted*. For example:

```
   a=: 1 2 3 4 5
   b=: 3 1 5 2 4
   +/a              +/b
15               15

   */a              */b
120              120

   >./a             >./b
```

```
3                 3

  -/a                -/b
3                 9
```

F. Display of Proverbs

If a proverb is entered alone (that is, without arguments), its *representation* is displayed. For example, if the proverbs of Sections F and G of Chapter 1 are already defined, then:

```
   increment
>:

   add3
>:@>:@>:

   identity
<:@>:
```

G. Inverses

Review the discussion of inverses in Section C and Exercise G3 of Chapter 1. Then observe the results of the following uses of inversion:

```
   a=:0 1 2 3 4 5
   >:^:_1 a
_1 0 1 2 3 4

   >:^:_1
<
   +&3^:_1 a
_3 _2 _1 0 1 2

   +&3^:_1
-&3

   -&3^:_1 a
3 4 5 6 7 8

   3&-^:_1 a
3 2 1 0 _1 _2

   3&- 3&-^:3 a
0 1 2 3 4 5

   3&-^:_1
3&-
```

H. Partitions

The sum of a list (**+/list**) is equal to the sum of sums over parts of the list, and a similar relation holds for some other verbs such as ***/** and **>./** . For example:

```
   +/3 1 4 1 5 9          (+/3 1)+(+/4 1 5 9)
23                     23

   */3 1 4 1 5 9          (*/3 1)*(*/4 1 5 9)
540                    540

   >./3 1 4 1 5 9         (>./3 1)>.(>./4 1 5 9)
9                      9
```

These relations can be expressed more clearly in terms of the truncation verbs *take* (`{.`) and *drop* (`}.`). Thus:

```
   a=:3 1 4 1 5 9
   2{.a
3 1

   2}.a
4 1 5 9

   (+/2{.a)+(+/2}.a)        +/a
23                        23

   (*/2{.a)*(*/2}.a)        */a
540                       540

   (+/6{.a)+(+/6}.a)
23

   (*/6{.a)*(*/6}.a)
540
```

The last two examples are interesting because the list `6}.a` is empty, yet the results of `+/` and `*/` upon it are such as to maintain the identities seen for the other cases. Thus:

```
   +/6}.a   */6}.a
0  1
```

This matter is explored further in the succeeding section.

I. Identity Elements and Infinity

It is easy to verify that the monads `0&+` and `1&*` and `-&0` are *identity* verbs that produce no change in their arguments. A noun that bonds with a dyad to form an identity verb is said to be an *identity element* of that dyad. Thus, `1` is the identity element of `*`, and `0` is the identity element of `+` and of `-` .

Although `-&0` is an identity, `0&-` is not. We may therefore say more precisely that `0` is a *right* identity of `-` . The same is true for other non-commutative verbs. Thus, `1` is a right identity of `^` (power).

To ensure that identities of the form **(+/a)=(+/k{.a)+(+/k}.a)** remain true when one of the lists is empty, we define the result of **d/b** to be the identity element of **d** if the list **b** is empty.

Does the dyad **<.** (minimum) possess an identity element? If **h** were a huge number (such as **10^9**) then it would serve for all practical purposes as the identity element of minimum. However, since there is no largest number among the integers, we must again extend the domain by adding a new element, denoted by _ and called *infinity*. To provide an identity for maximum we also add a *negative infinity* denoted by __ . We will refer to the resulting domain as *integers+*. Thus:

```
   <./0#0                 >./i.0
_                      __
```

J. Experimentation

In experimenting with expressions on the computer you will find that many verbs, adverbs, and conjunctions have meanings that are more general than the definitions given in the text. For example:

```
   halve=: 2&*^:_1
   halve 2 4 6 8 10            halve 1 2 3 4 5
1 2 3 4 5                   0.5 1 1.5 2 2.5

   sqr=:*~
   sqrt=: sqr^:_1
   sqrt 1 4 9 16 25            sqrt 1 2 3 4 5
1 2 3 4 5                   1 1.41421 1.73205 2 2.23607

   sqrt - 1 2 3 4 5
0j1 0j1.41421 0j1.73205 0j2 0j2.23607
```

Some of the results of these experiments are fractions and complex numbers that lie outside the domain of integers treated thus far. There is no harm in experimenting further with any that interest you, but do not spend too much time on baffling matters that will be treated later in the text.

K. Summary of Notation

The notation introduced in this chapter comprises two nouns (_ and __) for the identity elements of minimum and maximum; two verbs *take* and *drop* (**{. }.**) for truncating a list; the *commute* adverb **~ ;** the conjunction **&** to bond nouns to dyads; and verbs produced by the *atop* conjunction **@** have dyadic as well as monadic cases.

L. On Language

Use the computer to test the following assertions:

1. The monad **|** yields the *magnitude* or *absolute value.*

2. The monad **|.** reverses its argument, and **3&|.** rotates it by three places.

3. The monad `-&|` is equivalent to `-@|`, but the dyad `-&|` applies the dyad `-` to the result of applying the monad `|` to each argument.

4. `%&4` is division by `4`, and is equivalent to `4&*^:_1` .

5. The monads `+:` and `-:` are *double* and *halve*.

6. The monads `*:` and `%:` are *square* and *square root*.

7. `'abcde'` is the list of the first five letters of the alphabet, and monads such as `|.` and `3&|.` and `3 4&$` apply to it.

Exercises

A1 Define a verb `sump` that sums the positive elements of a list.

Define `dsq` and `sqd` to double the square and square the double.

Answer: `sump=:+/@(0&>.)  dsq=:(2&*)@(^&2)  sqd=:^&2@(2&*)`

B1 Define the following verbs:

`from`	That subtracts its left argument from the right
`square`	Without using `^`
`double`	Without using `*`
`zero`	A monad that yields zero

Answer: `from=: -~` `square=:*~` `double=:+~` `zero=:-~`

C1 Test all the dyads defined thus far for associativity.

D1 Which of the monads defined in preceding exercises are linear?

E1 Use the arguments `a=: 1 2 3 4 5` and `b=: 3 1 5 2 4` to test all dyads (including `-~` and `^~`) for symmetry.

E2 The expression `?~ n` produces a random permutation of the integers `i. n`. Use it for further tests of symmetry.

G1 Experiment with inverses of the monads defined in preceding exercises.

H1 Test the dyad `<.` to see if `(<./k{.a)<.(<./k}.a)` agrees with `<./a` for various values of `k` and `a` .

H2 Repeat Exercise H1 for the dyads `-` and `^`

H3 Characterize those dyads that satisfy the test of Exercise H1.

Answer: They are associative

I1 Experiment with various dyads to determine their identity elements.

J1 Experiment with the dyad `%`

Chapter 3

Partitions and Selections

A. Partition Adverbs

The partition adverb **** (called *prefix*) applies to monads to produce many useful verbs. For example:

```
   a=: 1 2 3 4 5
   sum=: +/
   sum a
15

   sum\ a                    Subtotals or "running" sums
1 3 6 10 15

   (+/1),(+/1 2),(+/1 2 3),(+/1 2 3 4),(+/1 2 3 4 5)
1 3 6 10 15
   +/\a
1 3 6 10 15

   */\a                      Running products
1 2 6 24 120

   !a
1 2 6 24 120

   >./\ 3 1 4 1 5 9          Running maxima
3 3 4 4 5 9
```

The partition adverb **\\.** behaves similarly to produce a verb that applies to suffixes:

```
   sum \.a
15 14 12 9 5

   */\.a
120 120 60 20 5

   <./\.3 1 4 1 5 9
```

```
1 1 1 1 5 9

   (*/\.a)*(*/\a)
120 240 360 480 600

  (+/\.a)+(+/\a)
16 17 18 19 20

   (-/\.a)-(-/\a)
2 _1 2 1 2
```

The diagonal adverb `/.` applies to (forward sloping) diagonals of tables. It will later be seen to be useful in multiplying polynomials and integers expressed in decimal. It is also useful in treating correlations and convolutions:

```
   t=:1 2 1*/1 2 1
   t
1 2 1
2 4 2
1 2 1

   sum/. t
1 4 6 4 1

   (sum/. t)*(10^i.-5)
10000 4000 600 40 1

   +/(sum/. t)*(10^i.-5)
14641

  121*121
14641

   +//.1 2 1*/1 3 3 1
1 5 10 10 5 1

   +//.1 3 3 1*/1 4 6 4 1
1 7 21 35 35 21 7 1
```

B. Selection Verbs

The *take* and *drop* (`{.` and `}.`) used in Section 2 H are examples of selection verbs. A more general selection is provided by the verb `{` (called *from*). For example:

```
   primes=:2 3 5 7 11 13
   2{primes
5

   0 2 4{primes
2 5 11

   3{.primes
```

```
2 3 5
   (i.3){primes
2 3 5

   (i.-#primes){primes
13 11 7 5 3 2

   i.3 5
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14

   0 2{i.3 5
 0  1  2  3  4
10 11 12 13 14

   2 1 3 5 0 4{primes
5 3 7 13 2 11
```

The last sentence above is an example of a *permutation* that reorders the items of the list **primes**; a list such as **2 1 3 5 0 4** that produces a permutation is called a *permutation list*, or *permutation vector*, or simply a *permutation*.

If the items of a list **a** are distinct, then the selection **b=: i{a** has an inverse in the sense that for a given **b**, an index can be found that selects it. The dyad **i.** fulfills this purpose, and is called *indexing*. For example:

```
   a=:2 3 5 7 11 13
   ]b=:3{a
7

   a i. b
3

   a i. 11 2 5
4 0 2
```

More precisely, the monads **{&a** and **a&i.** are mutually inverse. For example:

```
   psel=: {&2 3 5 7 11 13
   pind=: 2 3 5 7 11 13&i.
   pind 7 2
3 0

   psel pind 7 2
7 2
```

A list such as **a** specifies a set of intervals, and an integer may be classified according to the interval in which it falls. More precisely, we will determine the index of the largest element in the list that equals or precedes it. Thus, **5** and **6** both lie in interval **2** of **a** because they are greater than or equal to **2{a** and less than **3{a**.

Indexing can be used to perform the classification as follows:

```
   a
2 3 5 7 11 13

   x=: 6
   x<a
0 0 0 1 1 1

   (x<a) i. 1
3

   ]i=: <:(x<a)i.1
2

   i{a
5
```

C. Grade and Sort

The monad `/:` grades its argument. For example:

```
   p=: 5 3 7 13 2 11
   /:p
4 1 0 2 5 3

   (/:p){p
2 3 5 7 11 13
```

More precisely, the monad `/:` produces a permutation vector that can be used to *sort* its argument to ascending order.

D. Residue

Just as the introduction of the predecessor as the inverse of the successor led to a new class of numbers outside the class of counting numbers, so an attempt to introduce an inverse to a multiplication such as `5&*` leads to new numbers when applied to an integer such as `17` that is not an integer multiple of `5`. In other words, `17` is not in the (integer) domain of the inverse `5&*^:_1` . Similar remarks apply to an arbitrary multiple `m&*`.

An approximate inverse in integers can be obtained by locating the argument in the intervals specified by the multiples `5*i.n` . For example:

```
   x=: 17
   m5=: 5*i.6
   m5
0 5 10 15 20 25

   d=: <:(x<m5)i. 1
   d                            5*d
3                            15

   r=: x-5*d
```

```
   r
2
   5|x
2
```

The result **r** is the difference between the original argument and the nearest multiple of **5** that does not exceed it; it is called the *residue of* **x** *modulo* **5**, or the **5**-*residue* of **x** .

The dyad **|** is called *residue*, and **x-m|x** is an integer multiple of **m**. Consequently it is in the domain of the inverse **m&*^:_1**. Thus:

```
   a=: i. 21
   a
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
   8|a
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4
   a-8|a
0 0 0 0 0 0 0 0 8 8 8 8 8 8 8 8 16 16 16 16 16

   8&*^:_1 a-8|a
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2

   10&*^:_1 a-10|a
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2
```

E. Characters

In English, the word Milk refers to a white liquid, whereas ‘Milk’ refers to the list of four literal characters ‘M’ and ‘i’ and ‘l’ and ‘k’. We will use quotes in a similar manner, as illustrated below:

```
   alph=: ' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
   9 0 9 9 0 9 9 9 0 9 22 0 22 0 22 9 0 22 9 9 { alph
I II III IV V VI VII

   t=: 4>*/~ 3 2 1 0 1 2 3
   t                          t { ' *'
0 0 1 1 1 0 0              ***
0 0 1 1 1 0 0              ***
1 1 1 1 1 1 1            *******
1 1 1 1 1 1 1            *******
1 1 1 1 1 1 1            *******
0 0 1 1 1 0 0              ***
0 0 1 1 1 0 0              ***

   sentence=: '1 2 3^4'
   reverse=: (i.-#sentence){sentence
   reverse
4^3 2 1
   do=:".
   do sentence
1 16 81
   do reverse
```

```
64 16 4

   ;: sentence
+-----+-+-+
|1 2 3|^|4|
+-----+-+-+
```

F. Box and Open

The *word-formation* verb `;:` can be applied to a character list that represents a sentence to break it into its individual words. Thus:

```
   letters=: 'abc=:i.3 4+2'
   words=: ;: letters
   words
+---+--+--+---+-+-+
|abc|=:|i.|3 4|+|2|
+---+--+--+---+-+-+

   #words
6
   (i.-#words){words
+-+-+---+--+--+---+
|2|+|3 4|i.|=:|abc|
+-+-+---+--+--+---+
```

As illustrated, the result of the word-formation is a list of six items, each of which is a boxed list representing the corresponding word.

A single box can also be formed by the *box* monad **<** as follows:

```
   <'abcd'
+----+
|abcd|
+----+

   <2 3 5
+-----+
|2 3 5|
+-----+

   (<(<'abcd'),<2 3 5),<2 3$(<'abcd'),<2 3 5
+------------+-------------------+
|            |+-----+-----+-----+|
|+----+-----+||abcd |2 3 5|abcd ||
||abcd|2 3 5||+-----+-----+-----+|
|+----+-----+||2 3 5|abcd |2 3 5||
|            |+-----+-----+-----+|
+------------+-------------------+
```

The box verb can also be very helpful in clarifying the behaviour of the partition adverbs. For example:

```
   <\a=:1 2 3 4 5
+-+---+-----+-------+---------+
|1|1 2|1 2 3|1 2 3 4|1 2 3 4 5|
+-+---+-----+-------+---------+

   <\.a
+---------+-------+-----+---+-+
```

```
|1 2 3 4 5|2 3 4 5|3 4 5|4 5|5|
+---------+-------+-----+---+-+

   i. 3 4
0 1  2  3
4 5  6  7
8 9 10 11
   </.i.3 4
+-+---+-----+-----+----+--+
|0|1 4|2 5 8|3 6 9|7 10|11|
+-+---+-----+-----+----+--+
```

The monad `>` is the inverse of box; where necessary it "pads" the result with appropriate zeros or spaces. For example:

```
   ]a=: ;: 'Gaily into Ruislip gardens'
+-----+----+-------+-------+
|Gaily|into|Ruislip|gardens|
+-----+----+-------+-------+
   >a
Gaily
into
Ruislip
gardens

   b=:</.i.3 4
   b
+-+---+-----+-----+----+--+
|0|1 4|2 5 8|3 6 9|7 10|11|
+-+---+-----+-----+----+--+

   >b
 0  0 0
 1  4 0
 2  5 8
 3  6 9
 7 10 0
11  0 0
```

G. Summary of Notation

The notation introduced in this chapter comprises three partition adverbs, *prefix, suffix, and oblique* (`\ \. /.`); the dyads *from* and *residue* (`{ |`); and the monads *box*, *open*, *grade*, and *word-formation* (`< > /: ;:`). Section E also introduced the use of quotes to distinguish literals and other characters.

H. On Language

Review Section R of Chapter 1, and pursue one or more of the options suggested.

Exercises

In exercises first write (or at least sketch out) the result of each sentence without using the computer; then enter the sentence on the computer to check your answer.

A1
```
q=:1 1&(*/)
q 1 2 1
```

```
r=:+//.@q
r 1 2 1
r 1
r r 1
r^:(5) 1
r^:(i.6 )
```

A2 Experiment with the dyad `!` for various cases, such as `3!5` and `4!5` and `(i.6)!5`.

A3 `(i.6)!5`   `!/~i.6`   `!~/~i.6`

`(!~/~i.6)=(r^:(i.6) 1)`

B1 `(2*i.3){2 3 5 7 11 13 17`

`0 2 3 1{i.4 4`

`2{0 2 3 1{i.4 4`

B2 `cl=:i.&1@<`

`6 cl 2 3 5 7 11 13`

`5 cl 2 3 5 7 11 13`

`4 cl 2 3 5 7 11 13`

B3 Experiment with negative left arguments to `{.` and `}.` and `{`

D1 `3|7`

`7|3`

`3|i.10`

`|/~i.7`

E1 `text=:'i sing of olaf glad and big'`

`/: text`

`(/:text){text`

`text{~/:text`

`text/:text`

F1 `<\'abcdefg'`

`<\.'abcdefg'`

`a=:3 4$'abcde'`

`<\a`   `<\.a`

Chapter 4

Representation of Integers

A. Introduction

Because we are so familiar with the decimal number system (which extends systematically to larger and larger numbers), the matter of distinct representations of successive counting numbers did not pose an obvious problem. However, in a system such as Roman numerals, the sequence I II III IV V VI VII has no clear pattern of continuation beyond a few thousand.

Although the decimal system is familiar, a careful examination of it is fruitful because it leads to simple procedures for determining the results of verbs such as addition, multiplication, and power. We begin by expressing the relationship of a single number (such as the number of days in a year) to the *list* of decimal digits that represent it:

```
   n=:365              d=:3 6 5       e=:2 1 0
   10^e
100 10 1
   d*10^e              +/d*10^e
300 60 5            365
```

The name **e** was chosen for the list **2 1 0** because the right argument of the power verb is often called an *exponent*. It could have been expressed using the verb **i.** as follows:

```
   i. -3
2 1 0
   +/d*10^i.-3
365
```

The foregoing expression is, of course, suitable only for a list **d** of three items. To write a more general expression for any list **d** it is necessary to use a verb that yields the number of items of its list argument. Thus:

```
   #d                      +/d*10^i.-#d
3                       365
   d=:1 7 7 6
   +/d*10^i.-#d
```

```
1776
```

The foregoing is an example of determining the *base-10 value* of a list of digits, and similar expressions apply for other number *bases* or *radices*. Thus:

```
   +/d*8^i.-#d
245

   b=:1 1 0 1
   +/b*2^i.-#b
13

   10#.d
365

   8#.d
245

   2#.b
13
```

The last three sentences show the use of the dyad `#.` (called *base-value*) for the same evaluations.

B. Addition

Two lists representing numbers in decimal may be added to produce a representation of their sum, as illustrated below:

```
   year=:3 6 5
   agnes=: 3 0 4
   base10=:10&#.
   year + agnes
6 6 9

   base10 (year + agnes)
669

   (base10 year) + (base10 agnes)
669

   year+year
6 12 10

   base10 (year+year)
730

   (base10 year)+(base10 year)
730
```

Although the sum `year+year` yields the correct sum when evaluated by `base10`, it is not in the usual *normal* form with each item in the list lying in the interval from `0` to `9`. It can be brought to normal form by subtracting `10` from each of the last two items and "carrying" *ones* to the preceding items to obtain the result `7 3 0` in normal form.

Since a zero can be appended to the beginning of a list without changing its decimal value, lists of different lengths can be added by appending leading zeros to the shorter. For example:

```
   dozen=:1 2
   base10 0,dozen
12

   year+0,dozen
3 7 7
```

C. Multiplication

A procedure for multiplication will first be stated, and its validity will then be examined:

```
   a1=:3 6 5
   b1=: 1 7 7 6
   (base10 a1)*(base10 b1)
648240

   over=: ({.;}.)@":@,
     by=: ' '&;@,.@[,.]
   a1 by b1 over a1*/b1
+-+----------+
| |1  7  7  6|
+-+----------+
|3|3 21 21 18|
|6|6 42 42 36|
|5|5 35 35 30|
+-+----------+

   a1*/b1
3 21 21 18
6 42 42 36
5 35 35 30

   ]p=:+//.a1*/b1
3 27 68 95 71 30

    base10 p
648240
```

Normalization of `p` by carries gives `6 4 8 2 4 0` and:

```
   base10 6 4 8 2 4 0
648240
```

The foregoing procedure for multiplication comprises three steps:

1. Form the multiplication table of the lists of digits.
2. Sum the diagonals of the table.
3. Normalize the sums.

The method is less error-prone than the one commonly taught, which distributes the normalization process through both the multiplication and summation phases. The validity of the process may be discerned from the following examples:

```
   a1=:3 6 5                    b1=:1 7 7 6
   a2=:10^2 1 0                 b2=:10^3 2 1 0
   a=:a1*a2                     b=:b1*b2
   a                            b
300 60 5                     1000 700 70 6

    (+/a)*(+/b)
648240

   a*/b
300000 210000 21000 1800
 60000  42000  4200  360
  5000   3500   350   30

   +/a*/b
365000 255500 25550 2190

   +/+/a*/b
648240
```

The fact that the product of the sums **+/a** and **+/b** can be expressed as the sum of products arises from two properties:

1. Multiplication distributes over addition.
2. Summation (**+/**) is symmetric.

In the expression **a*/b**, the arguments are themselves products and, because multiplication is both associative and commutative, **a*/b** can also be expressed as the product of two tables as follows:

```
   a1*/b1
3 21 21 18
6 42 42 36
5 35 35 30

   a2*/b2
100000 10000 1000 100
 10000  1000  100  10
  1000   100   10   1

   (a1*/b1)*(a2*/b2)                  a*/b
300000 210000 21000 1800           300000 210000 21000 1800
 60000  42000  4200  360            60000  42000  4200  360
  5000   3500   350   30             5000   3500   350   30
```

Each element of the table **a1*/b1** is multiplied by the corresponding element from the "powers of ten" table **a2*/b2**, and those elements of **a1*/b1** multiplied by the same power of ten can be first summed and then multiplied by it. Since equal powers lie on diagonals, the sums are made along these diagonals, as in the expression **p=:+//.a1*/b1** used in describing the multiplication procedure.

The reason that equal powers lie on diagonals can be made clear by noting that **a2** equals **10^e=:2 1 0**, that **b2** equals **10^f=:3 2 1 0**, and that **a2*/b2** equals **10^e+/f** :

```
   e+/f              10^e+/f
5 4 3 2          100000 10000 1000 100
4 3 2 1           10000  1000  100  10
3 2 1 0            1000   100   10   1
```

D. Normalization

The normalization process used in Section B can be expressed more formally. We first define the main verbs to be used, and illustrate their use:

```
   base10=:10&#.
   residue=: 10&|
   tithe=: 10&*^:_1
   n=: 98 45 19 24
   base10 n
102714

   remainder=: residue n
   remainder
8 5 9 4

   n-remainder
90 40 10 20

   carry=: tithe n-remainder
   carry
9 4 1 2

   carry ,: remainder     (,: laminates lists to form a table)
9 4 1 2
8 5 9 4

   +//. carry ,: remainder
9 12 6 11 4

   base10 +//. carry ,: remainder
102714
```

We begin by specifying a “temporary” name **t**, and repeatedly re-assign to it the result of the process illustrated above:

```
   t=: n
   t=:+//. (tithe t-residue t) ,: residue t

   t
9 12 6 11 4

   base10 t
102714
```

```
   t=:+//. (tithe t-residue t) ,: residue t
   t                             base10 t
0 10 2 7 1 4                  102714

   t=:+//. (tithe t-residue t) ,: residue t
   base10 t
102714
```

We will now use *trains* of isolated verbs (to be discussed below) to capture the foregoing process in a single verb, as follows:

```
   reduce=: +//.@ ((tithe @ (] - residue)) ,: residue)
   reduce n
9 12 6 11 4

   reduce ^:3 n
0 1 0 2 7 1 4

   reduce^:4 n
0 0 1 0 2 7 1 4
```

Because further repetitions of **reduce** continue to append leading zeros, we will instead use **trim@reduce**, where **trim** is defined to trim off a leading zero:

```
   trim=:0&=@(0&{) }. ]
   (trim @ reduce)^:3 n
1 0 2 7 1 4

   norm=: trim@reduce^:_
```

Three repetitions suffice for the argument **n**, but in general the number required is unknown. However, since the process **v^:k** stops when the successive results stop changing, it suffices to use a sufficiently large value of **k**, preferably infinity.

We now consider the trains used in the definitions of **reduce** and **trim**. The phrase **] - residue** occurring in the former has an obvious meaning, as illustrated below:

```
   ] - residue n
_8 _5 _9 _4
```

However, the same sequence of three verbs isolated by parentheses (as they are in the definition of **reduce**) is called a *train*, and has the meaning illustrated below:

```
   (] - residue) n
90 40 10 20
   (]n) - (residue n)
90 40 10 20

   (3&< <. 9&>) i. 15
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0
```

```
   (3&< i.15) <. (9&> i.15)
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0
```

Thus, the middle verb in a train of three applies dyadically to the results of the outer verbs. Such a train also has a dyadic meaning defined similarly. For example:

```
   3 (+*-) 7
_40

   (3+7)*(3-7)
_40

   3 (< >. =) 2 3 4 5
0 1 1 1

  3<:2 3 4 5
0 1 1 1
```

E. Mixed Bases

The base-value dyad `#.` used in Section A with the simple bases `10` and `8` and `2` can also be used with a *mixed* base defined by a list. For example:

```
   base=: 7 24 60 60
   base #. 0 1 2 3
3723                        # of seconds in 0 days, 1 hour, 2 minutes, 3 seconds

   a=:i. 2 4
   a
0 1 2 3
4 5 6 7
   base #. a
3723 363967

   base #: 3723
0 1 2 3

   base#: base #. a
0 1 2 3
4 5 6 7
```

The last results illustrate the fact that the dyad `#:` provides an inverse to the base value, and can be used to produce the list representations of integers in any base. For example:

```
   2 2 2 #: i. 8
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
```

```
1 1 0
1 1 1

   10 10 10 #: 24 60 365
0 2 4
0 6 0
3 6 5

   fbase=: 3-i. 3
   fbase
3 2 1
   fbase #: i.!3
0 0 0
0 1 0
1 0 0
1 1 0
2 0 0
2 1 0
```

The final example employs an unusual “factorial” base, that will be used in the discussion of permutations in Chapter 7.

F. Experimentation

The verb `mag=: ] >. -` yields the *magnitude* of its argument; for example, `mag 9 _9` yields `9 9`. However, the monad `|` does the same.

Although it is probably unwise to spend time memorizing bits of notation before they arise in context, it is worthwhile to experiment with the monadic cases of dyads already encountered (and conversely), and to adopt those that appear useful. The language summary at the back of the book can be used to suggest further experiments. It is also worthwhile to experiment with the use of tables and other higher-rank arrays such as the rank-3 array `i. 2 3 4` and the rank-4 array `i. 2 3 4 5`. Three matters merit attention:

1. Just as the insertion `+/` inserts the verb `+` between items of a list, so does it between *items* of a higher rank array: between the *rows* of a table, and between the *planes* of a rank-3 array. Consequently, `+/` applied to a table adds one row to another. For example:

```
   i. 3 4                  +/i. 3 4
0 1  2  3              12 15 18 21
4 5  6  7
8 9 10 11
```

2. Expressions such as `a */ b`, already used to form tables when applied to lists, also apply to higher-rank arrays. For example:

```
   2 3 5 */ i. 2 4
 0  2  4  6
 8 10 12 14

 0  3  6  9
12 15 18 21
```

```
 0  5 10 15
20 25 30 35
   1+i.2 3             */// (1+i.2 3)
1 2 3                 4  5  6
4 5 6                 8 10 12
                      12 15 18
```

3. The *rank* conjunction `"` determines the rank of the sub-array to which a verb applies. For example:

```
  sum=:+/
    ]a=:i. 2 3
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

   sum a                 sum"2 a          sum"1 a
12 14 16 18           12 15 18 21        6 22 38
20 22 24 26           48 51 54 57       54 70 86
28 30 32 34
```

G. Summary of Notation

Notation introduced in this chapter comprises isolated trains of verbs (as indicated in the diagrams at the right); one conjunction (*rank* `"`) ; and four verbs -- *base value* and its inverse, *laminate*, and *magnitude* (`#.  #:  ,:  |`).

```
   g          g
  / \        / \
 f   h      f   h
 |   |     / \ / \
 y   y     x y x y
```

Exercises

A1
```
base10=: 10&#.
base8=: 8&#.
base2=: 2&#.
a=:1 0 1 0 1
base2 a          base2 -a
base8 a          base8 -a
base10 a         base10 -a
```

C1 Compare the multiplication process described at the beginning of Section C with the commonly-taught process for multiplying `365` by `1776` by actually performing both.

C2 Repeat Exercise C1 for various arguments, and note particularly the relative difficulties of reviewing the work for suspected errors.

E1 What is the result of applying the verb `norm` to a single number such as `1776`?

E2 Enter `t=: ?4 2$10` to define a table `t` of decimal digits. Then define a verb `sum` such that `sum t` gives the list representation of the integers represented by the rows of `t`. Check your result by applying `base10` to it and `+/base10` to `t`.

Answer: `sum=: norm@(+/)`

E3 Write an expression that gives the list representation of the product of the integers represented by the rows of `t`.

Answer: `norm +//."2^:(<:#t) *//t`

F1 Enter `#: i. 8` and compare the result with the use of the dyad `#:` in Section E. Use further experiments to determine and state the definition of the monad `#:` .

Answer: `#:x` is equivalent to `(n#2)#:x` , where `n` is chosen just large enough to represent the largest element of `x`.

F2 Define `t=: ,"1~&0 , ,"1~&1` . Then enter `]b=:i.2 1` and `t b` and `t t b`, and so on, and compare the results with the results of `#:i.2^k` for various values of `k` .

Chapter 5

Proofs

A. Introduction

A proof is an exposition intended to convince a reader that a certain relation is true, and perhaps to provide some insight into *why* it is true. For example, Section O of Chapter 1 provided, in passing, an illustration that the sum of the first six odd numbers was equal to six times six, that is, the square of six. Thus:

```
   odds=:1+2*i. k=:6
   odds
1 3 5 7 9 11

   +/odds
36

   k*k
36

   *:k
36

   *:#odds
36
```

This relation for the case of six odds suggests that a similar relation might hold for any number, and the prefix scan (`\`) provides a convenient test:

```
   d=:1+i.15
   d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

   odds=:1+2*i.15
   odds
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29

   +/\odds
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
```

```
   *:d
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225
```

This result provides rather strong evidence that the sum `+/1+2*i.k` equals the square of `k` for any value of `k`, but it provides no insight into why this should be so.

The following numbered sequence of sentences begins and ends with the pair whose equivalence is to be established. The intermediate sentences differ in simple ways that can provide insight into *why* the relations would hold true for any value of `k`:

```
S1             odds=:1+2*i.k=:10
               odds
            1 3 5 7 9 11 13 15 17 19

S2             +/odds
            100

S3             |.odds
            19 17 15 13 11 9 7 5 3 1

S4             +/|.odds
            100

S5             -: (+/odds) + (+/|.odds)  (-: halves its argument)
            100

S6          -: +/ (odds+|.odds)
            100

S7             +/ -: (odds+|.odds)
            100

S8             odds+|.odds
            20 20 20 20 20 20 20 20 20 20

S9             -: odds+|.odds
            10 10 10 10 10 10 10 10 10 10

S10                k#k
            10 10 10 10 10 10 10 10 10 10

S11            +/k#k
            100

S12            k*k
            100

S13            *:k
            100
```

Sentences S2 and S4 to S7 show that the sum of the first ten odds can be written in several equivalent ways, but really demonstrate it *only* for the specific case of `k=:10`.

However, we may see reasons to believe that the relations between successive sentences should hold for other values of **`k`**.

For example, because **`+/`** is symmetric (as defined in Section 2 E), and because the monad **`|.`** permutes its argument, S2 and S4 agree for any list **`odds`** . Further, in S5, one-half of the sum of two equal things is equal to either one of them, and similarly simple arguments can establish the equality of the pairs S6, S7; S7, S11; S11, S12; and S12, S13. In particular, S12 agrees with S11 because their agreement expresses the *definition* of multiplication.

We will call a sequence such as S1-S13 an *informal* proof; it provides insight but leaves to the reader the task of providing precise reasons for the equivalence of certain pairs of sentences. A *formal* proof is one in which each sentence is annotated by a clear statement of the reasons for its equivalence with an earlier sentence.

An informal proof is satisfactory only if the relations between successive sentences are obvious to the reader. If so, why is it ever desirable to make formal a good informal proof? Firstly, what is obvious to one reader may not be to another. A second, more serious, reason is that *obvious* reasons for relations may, in fact, be wrong, or at least incomplete.

For example, does **`+/1+2*i.k`** equal **`k*k`** for the case **`k=:0`** ? The answer is *yes*, but this does not follow from the arguments given thus far, since they took no account of the definition of the summation of an empty list. A complete proof would require examination of the definition of identity elements in Section 2 I.

In the foregoing example the conclusion remained correct even though the reasons provided were incomplete, but unexamined proofs and definitions can also lead to errors or contradictions. For example, the *prime* numbers illustrated in Exercise O1 of Chapter 1 have the important property that any counting number greater than one can be expressed as a product of one or more primes, and that this *factorization* is unique. For example, using the first five elements of the list obtained in the cited exercise:

```
   pr=:2 3 5 7 11
   e=:2 0 2 1 0
   pr^e
4 1 25 7 1
   */pr^e
700
```

Thus, the exponents **`2 0 2 1 0`** specify the prime factorization of the integer **`700`**, and no other factorization in primes is possible.

We turn now to a definition of primes that is commonly used in high-school: A prime is an integer that is divisible only by itself and one. The integers in the list **`pr`** satisfy this condition, but so does the integer **`1`**. We now consider a list of "primes" that includes **`1`**, and see that the factorization of the integer **`700`** in terms of it is *not* unique:

```
   p=:pr,1
   p
2 3 5 7 11 1

   */p^2 0 2 1 0 0
700
   */p^2 0 2 1 0 3
```

```
700
```

The loss of unique factorization clearly lies in a definition of primes that admits **`1`** as a member. We turn to an informal development of primes that leads to a suitable definition:

```
   i=:>:i.8

   i
1 2 3 4 5 6 7 8

   rem=: i|/i
   rem
0 0 0 0 0 0 0 0
1 0 1 0 1 0 1 0
1 2 0 1 2 0 1 2
1 2 3 0 1 2 3 0
1 2 3 4 0 1 2 3
1 2 3 4 5 0 1 2
1 2 3 4 5 6 0 1
1 2 3 4 5 6 7 0

   div=: 0= i|/i
   div
1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1
0 0 1 0 0 1 0 0
0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

   +/div
1 2 2 3 2 4 2 4

   2=+/div
0 1 1 0 1 0 1 0

   (2=+/div)#i
2 3 5 7
```

The table **`rem`** is the table of remainders (or residues), and **`div`** is a divisibility table that identifies zero remainders. The sum **`+/div`** sums the columns of **`div`** to yield the number of divisors of each of the integers **`i`**, and the final sentence selects those integers that have exactly two distinct divisors. It furnishes a suitable definition: A prime is an integer that has exactly two distinct divisors.

We conclude this section with an example of an informal development designed to clarify some matters of elementary algebra.

The expression $\mathbf{a}^3$ is commonly used to denote what we denote here by **`a^3`**, and is defined by saying that it is the product of three factors **`a`** (which we would write as **`a*a*a`**) but also by continuing to define $\mathbf{a}^0$ as **`1`**. What is meant by a product of no factors, and why should its result be **`1`** ? Somewhat less mysteriously, what is a product of one factor ($\mathbf{a}^1$), and why should it yield **`a`** ?

The definitions of expressions such as **`a^n`** and **`!n`** are commonly extended to arguments that do not fall under the initial definition, by extending them so as to maintain certain significant “patterns” or “identities”. These patterns can often be made clear by applying functions to lists (such as **`i.n`**) that themselves maintain simple patterns. For example:

```
   a=:4

   e=:3 4 5

   a^e
```

```
64 256 1024
```

To evaluate the next in sequence (that is, `a^6`), one might perform the calculation `4*4*4*4*4*4` or, more efficiently, note that the result is simply `4` times the preceding case `a^5`. In other words, the pattern extends to the right by multiplication by `4`. Consequently, and more interestingly, it proceeds to the left by division by `4`. Thus, since `4^3` is `64`, it follows that `4^2` is `16`, that `4^1` is `4`, and that `4^0` is `1`.

These last two results provide some insight into why `a^1` and `a^0` are defined as `a` and `1` for any `a`, including the case where `a` itself is zero. It is worth noting that some college texts state that `0^0` is undefined, even though the result `1` is clearly needed to make it possible to evaluate the general form of the polynomial in `x` with coefficients `c`, namely, `+/c*x^i.#c`.

Going, for a moment, outside the domain of the integers, we may also note that the pattern continues through negative and fractional values. Thus:

```
   a=:4
   e=:3 4 5
   a^e
64 256 1024

   e=:3-~i.7
   e
_3 _2 _1 0 1 2 3

   4^e
0.015625 0.0625 0.25 1 4 16 64

   f=:-:i.6
   f
0 0.5 1 1.5 2 2.5

   4^f
1 2 4 8 16 32
```

In the final example, there are two steps rather than one between successive integers of the equally-spaced elements of the exponent `f`, and `4^f` must therefore exhibit a pattern of multiplication by a factor which applied twice produces multiplication by `4`; in other words, a factor that is the *square root* of `4`.

B. Formal and Informal Proofs

Although topics in mathematics are often presented *deductively*, as a sequence of formal proofs that appear to lead to collections of indisputable facts, we will continue to use an informal approach that emphasizes the use of expressions (such as the pair `+/\odds` and `*:d` of Section A) that suggest relations, and sequences of expressions (such as S1-S13) that outline a proof.

The reasons for adopting such an informal approach are rooted mainly in the view of mathematics expressed clearly and entertainingly in the dialogue in Lakatos' *Proofs and Refutations* [5] (discussed briefly in Section C), but also in the characteristics of the notation used here; characteristics that make it easy to express patterns in lists and tables, and to display them accurately and effortlessly by entering the expressions on a computer.

To appreciate these characteristics the reader should attempt to render various expressions in this text clearly and completely in more conventional notation. For example, `+/odds` may be expressed by using sigma notation, but `+/\odds` would probably be expressed as:

$$c_i = \sum_{j=1}^{i} odds_i$$

an expression that does not yield an entire list as does `+/\odds`, but specifies it indirectly by specifying each of the elements of some list denoted by `c`.

In a similar vein, it might be assumed that the sigma notation used for `+/odds` would also serve for `+/|.odds` as follows:

$$\sum_{i=1}^{n} odds_i \qquad\qquad \sum_{i=n}^{1} odds_i$$

However, the summation from `n` to `1` is normally taken to denote summation over an empty set, since no summation from `j` to `k` could otherwise denote the empty case.

It might also be noted that the symbol `n` commonly used in sigma notation has no clear connection to the number of elements in the argument, and cannot be expressed as a function of the argument without introducing some notation analogous to `#odds`.

C. Proofs and Refutations

Of his *Proofs and Refutations* [4], Lakatos says “Its modest aim is to elaborate the point that informal, quasi-empirical, mathematics does not grow through the monotonous increase of the number of indubitably established theorems but through the incessant improvement of guesses by speculation and criticism, by the logic of proofs and refutations.”

He goes on to say that there is a simple pattern of mathematical discovery - or of the growth of informal mathematical theories - that consists of the following stages (also quoted from [4]):

1. Primitive conjecture
2. Proof (a rough thought-experiment or argument, decomposing the primitive conjecture into sub-conjectures or lemmas).
3. ‘Global’ counterexamples (counterexamples to the primitive conjecture) emerge.
4. Proof re-examined: the ‘guilty lemma’ to which the global counter-example is a ‘local’ counterexample is spotted. This ‘guilty’ lemma may have previously remained ‘hidden’ or may have been misidentified. Now it is made explicit, and built into the primitive conjecture as a condition. The theorem - the improved conjecture - supersedes the primitive conjecture with the new proof-generated concept as its paramount new feature.

As a result, "Counterexamples are turned into new examples - new fields of inquiry open up."

Lakatos illustrates this process by following a simple conjecture through surprising twists and turns, citing positions held by dozens of eminent mathematicians. To quote from a review cited on the cover, "The whole book, as well as being a delightful read, is of immense value to anyone concerned with mathematical education at any level."

We will illustrate the process briefly. Having counted the number of vertices **`v`**, edges **`e`**, and faces **`f`** of various polyhedra (bounded by multiple flat *faces*, *surfaces*, or "seats" as suggested by the root *hedra*), a class arrives at the conjecture that the expression **`f+v-e`** yields **`2`** for any polyhedron. For example:

	f	**v**	**e**	**f+v-e**
Tetrahedron	4	4	6	2
Square-base pyramid	5	5	8	2
Cube	6	8	12	2

The teacher provides the following proof or "thought-experiment" to establish the validity of the relation for all polyhedra:

1. Triangulate each face by (repeatedly) drawing a line between some pair of vertices not already joined by an edge. [In the square-based pyramid this requires one diagonal on the base; in the cube it requires one diagonal on each face.] Since each line drawn adds one edge and one face (splitting one existing face into two), the triangulation does not change the result of **`f+v-e`**.

2. Remove one face, leaving a hole bounded by three edges.

3. Dismantle the body triangle-by-triangle until only one remains, removing at each step one edge and one face, or one vertex, two edges, and one face. Either action leaves **`f+v-e`** unchanged.

4. For the final triangle, **`f+v-e`** is **`1+3-3`** (that is, **`1`**), which, together with the face removed in step 2, gives a result of **`2`** for **`f+v-e`**.

The validity of each step of the process is challenged by students who enter the dialogue, and the validity of the conjecture itself is challenged by counterexamples, including one provided by a body formed by fitting together into a square "picture frame" four identical moldings (polyhedra) having the following end and side views:

```
   ___          ____________________________________
  /   \        /                                    \
 /_____\      /______________________________________\
```

A direct count gives **`16+16-32`** or **`0`**, contradicting the conjecture.

Attempts are first made to sharpen the definition of a polyhedron so as to save the conjecture by barring the picture frame from consideration (as a "monster"), and later to revise the conjecture so as to account for such a monster.

One such revision is based on the observation that the "well-behaved" polyhedra shared the property that (if constructed of elastic surfaces) they could be inflated to a sphere, but the picture frame could not. Moreover, a single cut through one limb of the frame (which would appear as a vertical line in the side view above) would form a body with two new faces, eight new vertices, and eight new edges, restoring the result of **`2`** for **`f+v-e`**, and producing a body that could be inflated to a sphere.

A revised conjecture taking into account the "connectedness" or "number of cuts needed to produce a 'spherical' body" can therefore be formulated; but it again is subject to further criticism and refinement.

We conclude this section with an extended quotation from Lakatos (page 73):

> TEACHER: No! Facts do not suggest conjectures and do not support them either!
>
> BETA: Then what suggested **`2`**=**`f+v-e`** to *me* if not the facts, listed in my table?
>
> TEACHER: I shall tell you. You yourself said you failed many times to fit them into a formula. Now what happened was this: you had three or four conjectures which in turn were quickly refuted. Your table was built up in the process of testing and refuting these conjectures. These dead and now forgotten conjectures suggested the facts, not the facts the conjectures. *Naive conjectures are not inductive conjectures: we arrive at them by trial and error, through conjectures and refutations.* But if you - wrongly - believe that you arrived at them inductively, from your tables, if you believe that the longer the table, the more conjectures it will suggest, and later support, you may waste your time compiling unnecessary data. Also, being indoctrinated that the path of discovery is from facts to conjecture, and from conjecture to proof (the myth of induction), you may completely forget about the heuristic alternative: deductive guessing.

D. Proofs

Throughout this text we will present examples intended to stimulate the formulation of conjectures, but will not develop proofs. However, the reader is encouraged to provide formal and informal proofs for any conjectures that suggest themselves. The present section will provide examples of proofs of identities that are familiar in elementary mathematics, but are often treated in more limited forms.

In this section we will use the name **`X`** to denote a single element (or *scalar*), and other names to denote lists (or *vectors*). We will write one sentence below another to indicate that they are equivalent. Thus:

Thm1: **`+/X*W`**

`X*+/W`

asserts that the sum over a scalar times a list is equivalent to the scalar times the sum over the list, and labels the identity as Thm1 (Theorem 1) for future reference.

A formal proof of a theorem is provided by annotating each sentence after the first with the reason that it is equivalent to the sentence preceding it. Thus:

Thm1: **`+/X*W`**

`X*+/W` **`X&*`** distributes over **`+`** (Section 2 D)

If values are assigned to the names used in a theorem, then each sentence may be entered and executed as a test for the case of the particular values assigned. Thus:

```
   X=: 3
   W=: 3 1 4 1
   +/X*W
27

   X*+/W
27
```

This executability is reassuring in developing an identity or proof, because a mis-statement will very likely produce a different result. For example:

Thm2: `V=: 2 4 6`

```
    +/V*/W
  36 12 48 12
```

`(+/V)*W`	Thm1 applied for each element of `W`
`36 12 48 12`	(since `+/V` is a scalar)

A sequence of equivalent sentences implies that the first sentence is equivalent to the last. Hence the following is a formal proof that the sum of the column sums of the multiplication table `V*/W` equals the product of the sums `+/V` and `+/W`:

Thm3: `+/+/V*/W`

`+/V*(+/W)`	Thm2 and commutativity of `*`
`(+/V)*(+/W)`	Thm1 (with `+/W` for `X` and `V` for `W`) and commutativity of `*`.

The following theorem can be proved formally by showing that the element of column `j` of row `i` of the first table is equal to the corresponding element of the second table:

Thm4: `(A*P)*/(B*Q)`
`(A*/B)*(P*/Q)`

It can be illustrated as follows:

```
   A=:2 3 5
   B=: 3 1 4 1
   P=: 4 3 2
   Q=: 2 7 1 8


   (A*P)*/(B*Q)
48 56 32 64
54 63 36 72
60 70 40 80

   (A*/B)*(P*/Q)
48 56 32 64
54 63 36 72
60 70 40 80
```

Since **x^n** is defined by ***/n#x**, it is easy to show that **(x^n)*(x^m)** is equivalent to **x^(m+n)**. This result can be used in the proof of the following theorem:

Thm5: **(X^A)*/(X^B)**

X^(A+/B)

The foregoing theorems will be used in an exercise in Section B of Chapter 9 to prove that the product of two polynomials with coefficients **C** and **D** is equivalent to a polynomial with coefficients **+//.C*/D**.

The fact that multiplication distributes over addition is commonly extended to a product of sums and expressed in conventional notation as:

LHS= (a+A)(b+B)

RHS= (ab)+(aB)+(Ab)+(AB)

the left-hand side LHS being equivalent to the right-hand side RHS.

This identity can be extended to a product over any number of sums as follows:

LHS=(a+A)(b+B)(c+C)

RHS=(abc)+(abC)+(aBc)+(aBC)+(Abc)+(AbC)+(ABc)+(ABC)

LHS=(a+A)(b+B) ... (z+Z)

The last expression above uses the informal three-dot notation to suggest continuation of the same form to arbitrary lengths. To appreciate the difficulties of such informal notation, the reader should attempt its use in a clear definition of the corresponding right-hand side.

The use of vectors (lists) makes the expression of the left-hand side simple: ***/v1+v2** , where (in the three-element case above), **v1=:a,b,c** and **v2=:A,B,C**.

To clarify the pattern of the right-hand side, we will use explicit values for **v1** and **v2**, thus allowing the direct evaluation of every expression. We will also use numbers less than ten in **v1**, and greater than ten in **v2** to make patterns easier to recognize. Thus:

```
   v1=:2 3 4               v2=:12 13 14             v1+v2
                                                 14 16 18
   ]LHS=: */v1+v2
4032

   ]RHS=:(2*3*4)+(2*3*14)+(2*13*4)+(2*13*14)+(12*3*4)+
              (12*3*14)+(12*13*4)+(12*13*14)
4032
```

The pattern in the expression for **RHS** can be better seen in the following table:

```
   M=:>2 3 4;2 3 14;2 13 4;2 13 14;12 3 4;12 3 14;
              12 13 4;12 13 14
```

```
   M
 2  3  4
 2  3 14
 2 13  4
 2 13 14
12  3  4
12  3 14
12 13  4
12 13 14

   */"1 M
24 84 104 364 144 504 624 2184

   +/*/"1 M
4032
```

Because the items of **v2** exceed **10**, the pattern in **M** can be displayed more clearly as booleans:

```
   ]b1=: M<10                ]b2=: M>10
1 1 1                     0 0 0
1 1 0                     0 0 1
1 0 1                     0 1 0
1 0 0                     0 1 1
0 1 1                     1 0 0
0 1 0                     1 0 1
0 0 1                     1 1 0
0 0 0                     1 1 1
```

The right-hand side can now be expressed in either of two ways:

```
   ]RHS=: +/(*/"1 v1^b1)*(*/"1 v2^b2)
4032

   ]RHS=: +/*/"1 (v1,v2)^(b1,.b2)
4032
```

The details of these expressions can be explored by displaying the partial results. For example, the rows of **v1^b1** contain the appropriate elements from **v1** with the elements from **v2** being replaced by *ones* (the identity element of *****), and the product over the rows multiplied by the product over the rows of **v2^b2** yields the products to be summed. Thus:

```
   v1^b1          v2^b2
2 3 4           1  1  1
2 3 1           1  1 14
2 1 4           1 13  1
2 1 1           1 13 14
1 3 4          12  1  1
1 3 1          12  1 14
1 1 4          12 13  1
```

```
1 1 1        12 13 14

   */"1 v1^b1
24 6 8 2 12 3 4 1
   */"1 v2^b2
1 14 13 182 12 168 156 2184

   (*/"1 v1^b1)*(*/"1 v2^b2)
24 84 104 364 144 504 624 2184

   +/(*/"1 v1^b1)*(*/"1 v2^b2)
4032
```

Comparison of `b2` with the result of `#:i.2^3` in Exercise F1 of Chapter 4 should make it clear that `#:i.2^n` is the table appropriate to any list `v` of `n` elements. Moreover, as illustrated in Exercise F2 of Chapter 4, the verb `t=:  ,"1~&0,  ,"1~&1` applied to `#:i.2^n` yields the table for a list of one more element.

The foregoing facts can be used to formalize the following proof of the equality of general functions for the results illustrated above for `LHS` and `RHS`. We first define the functions:

```
   lhs=:*/@(+"1)

   rhs=:+/@(f*g)

     g=:*/"1@(]^T)@]

     f=:*/"1@(]^0&=@T)@[

       T=: #:@i.@(2&^)@#
```

For lists `V` and `W` of one element each, the results of `V lhs W` and `V rhs W` can easily be shown to be equivalent. We now present an *inductive* proof in which we assume that `V lhs W` and `V rhs W` are equivalent for any lists of `n` elements, and then use that *induction hypothesis* to prove that they are equivalent for lists on `n+1` elements. Thus:

`(x,V) rhs (y,W)`	
`+/(x,V) (f*g) (y,W)`	Def of `rhs`
`+/(L=:(x,V)f(y,W))*(x,V)g(y,W)`	Def of fork
`+/L**/"1(y,W)^T (y,W)`	Def of `g`
`+/L**/"1(y,W)^(0,"1 U),(1,"1 U=:T W)`	Structure of `T`
`+/L*((y^0)*Q),(y^1)*Q=:*/"1 W^U`	
`+/L*Q,y*Q`	
`+/((x*P),P=:*/"1 V^0=U)*Q,y*Q`	Analogous
`+/(x*P*Q),y*P*Q`	treatment of `L`
`(x+y)*+/P*Q`	
`(x+y)*V lhs W`	Induction
`(x+y)**/V+W`	hypothesis
`*/(x,V)+(y,W)`	

```
(x,V) lhs (y,W)
```

Chapter 6

Logic

A. Domain and Range

As stated in Section 1 D, the domain of a verb is the collection of arguments to which it can apply. For example, the integer **4** is in the domain of **>:**, but the characters **'!'** and **'b'** and **'4'** are not.

Similarly, the *range* of a verb is the collection of results that it can produce. The verb **>:** can produce any integer, and so its range is the same as its domain. This agreement of range and domain also holds for verbs such as **+** and *****; but not for **%**, which can produce *fractions* or *rational numbers*, and so has a wider range as discussed in Chapter 9.

A verb may also have a range more limited than its domain. For example, the verb **4&|** applies to any integer, but its results all fall in the finite list **i.4**, that is,**0 1 2 3**.

It is sometimes useful to examine the properties of a verb when it is applied only to a restricted part of its domain, particularly if it is restricted to its range. For example, when restricted to the domain **i.4**, the verbs:

```
pm4=: 4&|@*              (Product modulo 4)
sm4=: 4&|@+              (Sum modulo 4)
```

have the following tables:

```
   pm4/~ i.4                 sm4/~ i.4
0 0 0 0                   0 1 2 3
0 1 2 3                   1 2 3 0
0 2 0 2                   2 3 0 1
0 3 2 1                   3 0 1 2
```

We will use the phrase “**v** on **d**” to refer to the verb resulting from restricting the verb **v** to the domain **d**. For example, “**4&|@*** on **i.4**” refers to the product mod **4** restricted to the domain **0 1 2 3**, and “***** on **i.2**” refers to the boolean *and*, to be discussed in Section C.

B. Propositions

A *proposition* or *truth-function* is any statement which can be judged to be either true or false, and is therefore a verb having a range of two elements. Following Boole (the father of symbolic logic), we will denote these elements by **`1`** (for true) and **`0`** (for false). For example:

```
   p=: <&5
   p 3
1

   p a=:i.8                (p a)#a
1 1 1 1 1 0 0 0         0 1 2 3 4

   2=+/0=|/~ a
0 0 1 1 0 1 0 1

   a#~2=+/0=|/~ a
2 3 5 7
```

C. Booleans

The nouns **`0`** and **`1`** (the range of propositions) are called *booleans*, and a verb whose domain and range are boolean is called a *boolean function*, or *boolean*. For example, **`*`** limited to booleans might be called **`and`**; its table would appear as follows:

```
   and=:*
   and/~ b=:0 1
0 0
0 1

   ]c=:i.8
0 1 2 3 4 5 6 7

   (>&2 c) and (<&5 c)
0 0 0 1 1 0 0 0

   (>&2 and <&5) c
0 0 0 1 1 0 0 0

   c #~ (>&2 and <&5) c
3 4
   (] #~ >&2 and <&5) c
3 4
```

The sentence **`(>&2 and <&5)`** is a "compound" proposition whose result is true if the proposition **`>&2`** is true *and* the proposition **`<&5`** is true.

A verb **`or`** may be defined similarly:

```
   or=: *@+
   or/~b
0 1
```

```
1 1

   (=&7 c) or (<&5 c)
1 1 1 1 1 0 0 1
```

Note that the dyad **+** may produce non-boolean results, from which the monad ***** (called *signum*) produces booleans. Thus:

```
   * _2 0 2          +/~ b               * +/~b
_1 0 1              0 1                 0 1
                    1 2                 1 1
```

The booleans **and** and **or** are exceedingly useful. For example:

```
   dof10=: 0&=@(|&10)
   dof10 c =: 1+i. 20

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

   c#~dof10 c
1 2 5 10                                    Divisors of ten

   dof15=: 0&=@(|&15)
   c#~dof15 c
1 3 5 15                                    Divisors of fifteen

   c#~ (dof10 and dof15) c
1 5                                         Common divisors of ten and fifteen
   >./c#~ (dof10 and dof15) c
5                                           GCD of 10 and 15

   10 15 |~/ c
0 0 1 2 0 4 3 2 1 0 10 10 10 10 10 10 10 10 10 10
0 1 0 3 0 3 1 7 6 5  4  3  2  1  0 15 15 15 15 15

   0=10 15 |~/ c
1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

   and/0=10 15 |~/ c
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

   c #~ and/0=10 15 |~/ c
1 5

   >./c #~ and/0=10 15 |~/ c
5                                           GCD of ten and fifteen
```

The dyad `+.` is defined to yield the greatest common divisor of its arguments:

```
   10 +. 15                 +./ 10 15
5                        5
```

The least common multiple is denoted by `*.` as illustrated below:

```
   10 *. 15                 (10*15) % 10+.15
30                       30
```

D. Primitives

Verbs (such as `*` and `+` and `*.` and `i.`) that are denoted by single words are called *primitives*, to distinguish them from *derived* verbs produced by phrases such as that (`*@+`) used to define the boolean `or` in Section C. Since primitives and derived verbs are treated identically, this distinction is of little consequence except to the designer of a language, who must choose what primitives to provide.

Should new primitives be added for such important cases as the boolean *and* and *or*? Not if primitives already exist that give the appropriate results when restricted to the boolean domain. The dyads `<.` and `>.` (min and max) might be tested for this purpose. Thus:

```
   and=: *
   or=: *@+
   b=: 0 1
   <./~b            >./~b
0 0              0 1
0 1              1 1
   and/~b           or/~b
0 0              0 1
0 1              1 1
```

But do min and max provide the appropriate identity elements? The identity element for `or` should be `0`, and for `and` should be `1`, as illustrated below:

```
   0 or b             1 and b
0 1                0 1
```

However, the identity elements of min and max are infinities. Thus:

```
   <./i.0             >./i.0

_                  __
```

Other candidates for *or* and *and* when restricted to booleans are the greatest common divisor (`+.`) and the least common multiple (`*.`) introduced in the preceding section. Thus:

```
   +./~b              *./~b
0 1                0 0
1 1                0 1
```

```
   +./i.0              *./i.0
0                   1
```

Hereafter, these primitives will be used for *or* and *and*. It may be noted that Boole also represented *or* and *and* by then-current symbols for *plus* and *times*, but without the appended dot used here to distinguish them from these verbs.

E. Boolean Dyads

Are there any other boolean dyads in addition to `*.` and `+.` (*and* and *or*)? If so, how many?

To answer these questions we first display the tables for `*.` and `+.`, together with the *ravel* of each produced by the monad `,` :

```
   *./~ b=:0 1              +./~ b=:0 1
0 0                      0 1
0 1                      1 1

   ,*./~b                   ,+./~b
0 0 0 1                  0 1 1 1
```

We then observe that each table must contain four elements, each of which must belong to the range `0 1`. Since each element may have either of two values, there are `2*2*2*2`, or `2^4`, or `16` possible tables which, when ravelled to form a four-element list, must agree with one of the columns in the following transposed table:

```
   |:T
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

For example, columns `1` and `7` represent `*.` and `+.` :

```
   1{"1 T                   7{"1 T
0 0 0 1                  0 1 1 1

   and=: 1 b.               or=: 7 b.
   and/~ 0 1                or/~ 0 1
0 0                      0 1
0 1                      1 1

   and/i. 0                 or/i. 0
1                        0
```

As illustrated in the foregoing, the adverb `b.` applies to any of the indices (`0` to `15`) of the table `T` to produce the corresponding boolean dyad. It may be noted that the base-2 value of any row yields its index; for example, `2#.7{T` is `7`.

F. Boolean Monads

A monad that negates a boolean argument is equivalent to subtraction from `1`; it is called *not*, and is denoted by `-.` . There are in all four boolean monads as illustrated below:

```
   b
0 1

   -. b
1 0

   ] b
0 1

   ~:~ b
0 0

   =~ b
1 1
```

G. Generators

In English, compound propositions are commonly expressed using only *or*, *and*, and *not*. For example, using `p`, `q`, and `r` to denote propositions, and using parentheses to express the punctuation clearly:

`p and q`	`(1 b.)`	
`not (p and q)`	`(14 b.)`	
`(p or q) and not (p and q)`	`(6 b.)`	Exclusive-or
`not p and (not q)`	`(13 b.)`	Implication
`(p or q) or (p or not q)`	`(15 b.)`	True
`(p and q) and (p and not q)`	`(0 b.)`	False

Each of the foregoing phrases can be restated as definitions of verbs. For example:

```
   exclor=: +. *. -.@*.
   exclor/~ 0 1
0 1
1 0
```

Can all of the sixteen booleans be expressed using only *or*, *and*, and *not* ? The answer is *yes*, and for this reason the collection of verbs `+. *. -.` is said to be a *set of generators* of the booleans. For example, the case `0 b.` (which yields `0` for every pair of arguments) can be expressed as `(p and q) and (p and not q)`, and `15 b.` as `not (p and q) and (p and not q)`.

Is `+. *. -.` a minimal set of generators, or could one of them be omitted? This is easily answered by showing that `*.` itself can be expressed in terms of `+.` and `-.` and can therefore be omitted:

and is *not (not p) or (not q)*

The foregoing relation is sometimes expressed as "*and* is the *dual* of *or* (with respect to negation)."

The use of *or* and *not* as the only generators can lead to cumbersome expressions for some of the booleans, but all can be expressed in terms of them.

Can a single boolean serve as generator? It can be shown that either `8 b.` (*not-or* or *nor*) or `14 b.` (*not-and* or *nand*) will serve. This matter is developed in exercises.

H. Boolean Primitives

The primitives `+.` and `*.` (gcd and lcm) when restricted to the boolean domain provide the important boolean verbs *or* and *and*. Others are provided by similarly restricting relations:

`<`	`4 b.`	
`<:`	`13 b.`	Implication
`=`	`9 b.`	Identity
`>:`	`11 b.`	
`>`	`2 b.`	
`~:`	`6 b.`	Exclusive-or

Finally, +: and `*:` denote *nor* and *nand*, that is, `8 b.` and `14 b.` .

I. Summary of Notation

The notation introduced in this chapter comprises one adverb *boolean* (`b.`); five dyads *or*, *and*, *nor*, *nand*, and *not-equal* (`+. *. +: *: ~:`); three monads *not*, *signum*, and *ravel* (`-. * ,`).

Exercises

A1 Predict and test the results of `n | (i. n) +/ (i. n)` and of `n | (i. n) */ (i. n)` for various values of `n` including `10`.

A2 Define monads `S` and `P` such that `S n` and `P n` yield the tables of Exercise A1.

Answer: `S=: ] | i. +/ i.` and `P=:]|i.*/i.`

B1 Predict and test the result of applying to an integer `n` the verb `PR=: i. #~ T@(+/)@(0&=)@(|/~)@i.` for the cases `T=:2&=` and `T=:2&<` and `T=:3&=` .

B2 Define and test a verb `IN` such that `a IN b` yields `1` if `a` lies in the interval between the smallest and largest elements of `b`.

Answer: `IN=: (<./@] < [)*.(>./@] > [)`

B3 Define a verb **`L`** such that **`a L b`** lists the elements of **`a`** that lie in the interval defined by **`b`**.

Answer: `L=: IN#[`

C1 Explain the equivalence of the dyads **`*.`** and **`*%+.`** and test it in expressions such as **`(?7#100) (*. = * % +.)/ (? 10#100)`** .

E1 The verbs **`1 b.`** and **`7 b.`** may be called *and* and *or*. Recall or invent suitable names for as many of the remaining fourteen boolean functions as you can.

G1 Using only **`NAND=: 14 b.`** define a monad called **`NOT`** that is equivalent to the monad **`-.`** on the boolean domain.

Answer: `NOT=: NAND~`

G2 Using only **`NAND=: 14 b.`** and **`NOT`** define dyads **`AND`** and **`OR`** that are equal to **`*.`** and **`+.`** on the boolean domain.

Answer: `AND=: NOT@NAND` `OR=:NOT@(NOT AND NOT)`

G3 Repeat Exercises G1, G2 using **`NOR=: 8 b.`** instead of **`NAND`**.

Chapter 7

Permutations

A. Introduction

Permute is a verb meaning “to change the order of”, and **|.** is an example of a permutation:

```
   |. 'abcdef'
fedcba

   |. i. 5
4 3 2 1 0
```

Indexing provides arbitrary permutations. For example:

```
   2 0 1 5 4 3 { 'abcdef'
cabfed
```

A list of indices to **{** that produces a permutation is called a *permutation vector*, or *permutation*, and one that contains **n** elements is called a permutation *of order* **n**. A permutation of order **n** is itself a permutation of the list **i. n**.

To enumerate all permutations of order **n**, it is best to list them in ascending order (ascending when considered as the digits representing an integer), as illustrated in the following tables:

```
   p3                 p2
0 1 2              0 1
0 2 1              1 0
1 0 2
1 2 0                 p1
2 0 1              0
2 1 0

   i=:i.!3
```

```
   i{p4          (6+i){p4       (12+i){p4       (18+i){p4
0 1 2 3       1 0 2 3        2 0 1 3         3 0 1 2
0 1 3 2       1 0 3 2        2 0 3 1         3 0 2 1
0 2 1 3       1 0 3 2        2 1 0 3         3 1 0 2
0 2 3 1       1 2 3 0        2 1 3 0         3 1 2 0
0 3 1 2       1 3 0 2        2 3 0 1         3 2 0 1
0 3 2 1       1 3 2 0        2 3 1 0         3 2 1 0
```

A row (or rows) of any one of these tables can be applied to index (and therefore to permute) a list of the appropriate number of items. For example:

```
   3{p4
0 2 3 1

   (3{p4){'abcd'
acdb

   (3 4{p4){'abcd'
acdb
adbc

   (3 4{p4){i.4
0 2 3 1
0 3 1 2

   p3{'abc'                p2{'ab'
abc                     ab
acb                     ba
bac
bca
cab
cba

   3 A. 'abcd'
acdb

   3 4 A. 'abcd'
acdb
adbc
```

The last examples illustrate the use of the dyad **`A.`** in which **`i A. y`** permutes **`y`** by a permutation of order **`#y`**, the permutation being row **`i`** of the corresponding table of all permutations of that order.

The index **`i`** in the phrase **`i A. y`** can be thought of as an *atomic* (that is, single-element) representation of the permutation vector it applies, thus providing a mnemonic for the word **`A.`** .

From these examples it should be clear that the phrase **`(i.!n)A.i.n`** will produce the complete table of **`!n`** permutations of order **`n`**. Thus:

```
   PT=: i.@! A. i.

   PT 3              PT 2                PT 1
0 1 2             0 1                 0
0 2 1             1 0
1 0 2
1 2 0
2 0 1
2 1 0
```

B. Arrangements

Any selection of **k** different items from a list is called an *arrangement*, or **k**-*arrangement*. For example, **0 1{a** and **1 0{a** and **3 1{a** are **2**-arrangements from the list **a=:'abcd'**.

Any **k** columns of a permutation table will contain all **k**-arrangements, each arrangement appearing **!k** times. For example:

```
   ALL=: (PT #a) { a=:'abcd'
   AR2=: 2 {."1 ALL
   CLAR2=: ~. AR2
   ALL                AR2                CLAR2
abcd              ab                 ab
abdc              ab                 ac
acbd              ac                 ad
acdb              ac                 ba
adbc              ad                 bc
adcb              ad                 bd
bacd              ba                 ca
badc              ba                 cb
bcad              bc                 cd
bcda              bc                 da
bdac              bd                 db
bdca              bd                 dc
cabd              ca
cadb              ca
cbad              cb
cbda              cb
cdab              cd
cdba              cd
dabc              da
dacb              da
dbac              db
dbca              db
dcab              dc
dcba              dc
```

The table **ALL** contains all permutations of the list **a**; the table **AR2** contains all **2**-arrangements, with each arrangement appearing twice; the table **CLAR2** is the “clean” table of arrangements with redundant items suppressed. The suppression of redundant items is performed by the monad **~.** (called *nub*).

C. Combinations

The arrangement **'ca'** that occurs in the table **CLAR2** is a permutation of the arrangement **'ac'**, and the two cases therefore represent the same *combination* of elements from the list **a=: 'abcd'**. We may obtain a table of all **2**-*combinations* of **a** by first sorting each row of **CLAR2**, and then taking the nub of the sorted table:

```
   /:~"1 CLAR2              ~./:~"1 CLAR2
ab                        ab
ac                        ac
ad                        ad
ab                        bc
bc                        bd
bd                        cd
ac
bc
cd
ad
bd
cd
```

The steps in the development of combinations can now be assembled to define a verb **C** such that **k C n** produces the table of all **k**-combinations of order **n**:

```
   nub=: ~.
   rtake=: {."1
   rsort=: /:~"1
   C=: nub@rsort@nub@([ rtake (PT@]))
   2 C 4             (2 C #a){a=: 'abcd'
0 1                 ab
0 2                 ac
0 3                 ad
1 2                 bc
1 3                 bd
2 3                 cd

   1 C 3              2 C 3              3 C 3
0                  0 1                0 1 2
1                  0 2
2                  1 2

   2 C 5              3 C 5
0 1                0 1 2
0 2                0 1 3
0 3                0 1 4
0 4                0 2 3
1 2                0 2 4
1 3                0 3 4
1 4                1 2 3
2 3                1 2 4
2 4                1 3 4
```

```
3 4                 2 3 4

   $ 2 C 5             $ 3 C 5
10 2                10 3

   (!5)%(!2)*(!5-2)                  (!5)%(!3)*(!5-3)
10                                10
```

The foregoing definition of **C** shows clearly the relation of combinations to the permutations of the corresponding order. However, it is highly inefficient in the sense that **k C n** generates and sorts a large table (of **r=:!n** rows and **n** columns) in order to select from it a smaller table (of **r%(!k)*(!n-k)** rows and **k** columns). A more efficient alternative is developed in Exercise J10 of Chapter 9.

As illustrated by the preceding examples, the number of **k**-combinations of order **n** is given by **(!n)%(!k)*(!n-k)**. The number of combinations is a commonly-useful result; so important that the corresponding verb is treated as a primitive. For example:

```
   2!5                    (i.6)!5
10                     1 5 10 10 5 1

   !/~i.6
1 1 1 1 1  1
0 1 2 3 4  5
0 0 1 3 6 10
0 0 0 1 4 10
0 0 0 0 1  5
0 0 0 0 0  1
```

The last result is called the table of *binomial coefficients*; when transposed and displayed without the relevant sub-diagonal zeros it is also called *Pascal's triangle*.

D. Products of Permutations

If **p** is a permutation vector, then the verb **p&{** is a permutation. For example:

```
   p=: 2 3 4 1 0 5
   P=:p&{
   P a=:'abcdef'                 P P a
cdebaf                        ebadcf

   P^:2 a
ebadcf

   P^:0 1 2 3 4 5 6 7 8 a        P^:(i.9) i.6
abcdef                        0 1 2 3 4 5
cdebaf                        2 3 4 1 0 5
ebadcf                        4 1 0 3 2 5
adcbef                        0 3 2 1 4 5
cbedaf                        2 1 4 3 0 5
edabcf                        4 3 0 1 2 5
```

```
abcdef                        0 1 2 3 4 5
cdebaf                        2 3 4 1 0 5
ebadcf                        4 1 0 3 2 5
```

In the foregoing it may be noted that the sixth power of the permutation `P` agrees with its original argument, and the pattern therefore repeats thereafter. The *period* of this particular permutation is therefore said to be `6`.

E. Cycles

Column `3` of the tables produced by the power of the permutation `P` of Section D shows that position `3` of successive powers is occupied by the elements `'d'`, and `'b'` (or `3 1`) in a repeating cycle of length two. Column `1` shows the same cycle displaced.

Similarly, column `4` shows the length-`3` cycle `4 0 2`, and columns `0` and `2` show the same cycle displaced; column `5` shows the `1`-cycle `5`.

The permutation `P` could therefore be represented unambiguously by its *cycles* as follows:

```
   c=: 3 1 ; 4 0 2 ; 5
   c
+---+-----+-+
|3 1|4 0 2|5|
+---+-----+-+
```

The dyad `C.` produces permutations specified in cycle form. Thus:

```
   c C. a=:'abcdef'
cdebaf

   p { a
cdebaf

   p C. a
cdebaf
```

As illustrated by the last example, the dyad `C.` also accepts permutation vectors as the left argument, and in that case is equivalent to the dyad `{` . Finally, the *monad* `C.` provides a self-inverse transformation between the cycle and permutation-vector representations of a permutation. Thus:

```
   C. c
2 3 4 1 0 5
   C. C. c
+---+-----+-+
|3 1|4 0 2|5|
+---+-----+-+
   PT=: i.@! A. i.
   (PT 3);(C. PT 3);(C. C. PT 3)
+-----+-------------+-----+
|     |+-----+---+-+|     |
|     ||  0  | 1 |2||     |
|     |+-----+---+-+|     |
|0 1 2||  0  |2 1| ||0 1 2|
|0 2 1|+-----+---+-+|0 2 1|
|1 0 2|| 1 0 | 2 | ||1 0 2|
|1 2 0|+-----+---+-+|1 2 0|
```

```
|2 0 1||2 0 1|   | ||2 0 1|
|2 1 0|+-----+---+-+|2 1 0|
|     ||2 1 0|   | ||     |
|     |+-----+---+-+|     |
|     ||  1  |2 0| ||     |
|     |+-----+---+-+|     |
+-----+-------------+-----+
```

From columns `0` and `1` of the table of Section D it may be seen that the return to an identity permutation can occur only when the two cycles (of lengths `2` and `3`) complete at the same time, in this case after `2*3` applications of the permutation. The period of the permutation is therefore `6`.

In general, the period of a permutation is the least common multiple of the lengths of its cycles. This will be illustrated further by a permutation of order `20` :

```
   p20=:17 4 9 7 12 14 18 13 0 6 15 1 16 10 2 8 3 19 5 11
   ]c20=:C. p20
+-------------+----------------------------------+
|18 5 14 2 9 6|19 11 1 4 12 16 3 7 13 10 15 8 0 17|
+-------------+----------------------------------+
   #@> c20                    */#@> c20
6 14                       42
   p20&{^:18 a=: 'abcdefghijklmnopqrst'
bdcphfgiljrqnaotkesm

   p20&{^:(i.19) 'abcdefghijklmnopqrst'
abcdefghijklmnopqrst
rejhmosnagpbqkcidtfl
tmgnqcfkrsiedpjahlob
lqskdjoptfamhigrnbce
bdfphgcilorqnastkejm
ehoinsjabctdkrflpmgq
mncakfgrejlhptobiqsd
qkjrpostmgbnilceadfh
dpgticflqsekabjmrhon
hislajobdfmpregqtnck
nafbrgcehoqitmsdlkjp
kroetsjmncdalqfhbpgi
ptcmlfgqkjhrbdoneisa
iljqbosdpgntehckmafr
abgdecfhisklmnjpqrot
reshmjonafpbqkgidtcl
tmfnqgckroiedpsahljb
lqokdsjptcamhifrnbge
bdcphfgiljrqnaotkesm
```

F. Reduced Representation

There are exactly `!n` permutations of order `n`, and the “factorial” base `n-i.n` introduced in Section 4 E can be seen to provide exactly `!n` distinct lists of `n` integers, each belonging to `i.n`:

```
   R=: (]-i.) #: i.@!
   R 3
0 0 0
0 1 0
```

```
1 0 0
1 1 0
2 0 0
2 1 0
```

These lists can be used to represent the permutations in what we will call a *reduced* representation, as distinguished from the “direct” representation used thus far:

```
   D=: i.@! A. i.
   D 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

We will now define a verb **RFD** to yield the reduced representation from the direct, and an inverse **DFR**:

```
   RFD=: +/@({.>}.)\."1
   DFR=: /:^:2@,/"1
```

For example:

```
   RFD D 3                 DFR R 3
0 0 0                   0 1 2
0 1 0                   0 2 1
1 0 0                   1 0 2
1 1 0                   1 2 0
2 0 0                   2 0 1
2 1 0                   2 1 0
```

The definitions of these verbs will be discussed in exercises.

G. Summary of Notation

The notation introduced in this chapter comprises five verbs: *atomic permutation, cycle, nub, number of combinations,* and *random* (**A. C. ~. ! ?**).

Exercises

A1 Using as argument a list of four items, test the assertion that the monad **|.** is a permutation, and determine the value of **k** such that **k&A.** is equivalent to **|.** .

A2 Repeat Exercise A1 for the cases of lists of two, three, and five items.

A3 Test the assertion that a rotation such as **r&|.** is a permutation, and repeat Exercises A1 and A2 using rotations instead of reversal.

A4 Apply the monad **A.** to various permutation vectors, and state its definition.

A5 Experiment with `k A. 'abcd'` for negative values of `k`.

B1 Write an expression for the number of `k`-arrangements of order `n`.

C1 Define a monad `BC` such that `BC n` gives the table of binomial coefficients up to order `n-1`.

Answer: `BC=: !/~@i.`

C2 Without using `!` or `BC` define a monad `CS` that gives the column sums of `BC n`.

Answer: `CS=: 2&^@i.`

D1 Determine the power of the permutation `p=: 4824 A. i. 7`.

Hint: Examine the table produced by `p&{^:(i.20) i.7`

D2 Determine the power of the random permutation `q=: 5?5`.

E1 Predict and test the results of `C. k A. i.n` for various values of `k` and `n`.

E2 Predict and test the result of `C. 1 3;2 0 4`.

E3 Repeat Exercise E2 for various boxed arguments of `C.` .

E4 Use various permutations `p` to test the assertion that the power of `p` is the least common multiple of the lengths of the cycles in its cycle representation.

E5 Define a monad `PER` to give the power of a permutation `p`.

Answer: `PER=: *./@(#@>@C.)`

E6 What is the maximum period of a permutation of order `n` ?

F1 Predict and test the results of `R 4` and `D 4` and `RFD D 4` and `DFR R 4` and `(RFD@D = R) 4`.

F2 Define `rfd` equivalent to `RFD` except that it will apply only to a single permutation and not to a table of permutations.

Answer: Omit `"1` from `RFD`.

F3 Analyze the definition of `rfd` of the preceding exercise by defining and individually applying two functions such that `f @ (g \.)` is equivalent to `rfd`.

Answer: `f=:+/` `g=: {.<}.`

F4 Analyze `DFR`.

Chapter 8

Classification and Sets

A. Introduction

It is often necessary to separate a collection of objects into several classes, and then perform some operation upon each of the classes. The operation performed is often trivial compared to the complexity of the classification procedure itself, and classification is therefore an important matter. Indeed, most computation involves some classification, even though the classification process may be implicit rather than explicit.

As an example of the use of classification, consider a set of transactions that are recorded as a list of account numbers and a corresponding list of credits to the accounts. Thus:

```
   an=: 1010 1040 1030 1030 1020 1010 1040 1040 1050
   cr=:  131  755  458  532  218   47  678  679  934
```

A summary should therefore post the sum **131+47** to account **1010** and **218** to account **1020**, and so on. If:

```
   all=: 1010 1020 1030 1040 1050
```

is the list of all account numbers, then **c=: all =/ an** is the classification table, and:

```
   c=: all =/ an
   c
1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 1 1 0 0 0 0 0
0 1 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 1
   c*cr
131   0   0   0   0 47   0   0   0
  0   0   0   0 218  0   0   0   0
  0   0 458 532   0  0   0   0   0
  0 755   0   0   0  0 678 679   0
  0   0   0   0   0  0   0   0 934

   +/"1 c*cr
```

```
178 218 990 2112 934
```

The classification represented by the table **c** is both *complete* (each element being assigned to some class) and *disjoint* (each element being assigned to no more than one class). Classifications that arise from the expression **a =/ b** are disjoint if the elements of **a** are all distinct, and are complete if every element of **b** belongs to **a**. A boolean table **B** represents a complete disjoint classification if and only if each of its column sums is equal to **1**; that is, if ***./1=+/B** .

Since a table provides such a convenient representation of a classification, we will henceforth speak (rather loosely) of the table itself as a classification, or as an **n**-*way* classification, where **n=:#B**.

Meaningful classifications need not be disjoint. For example, the letters of the alphabet may be classified phonetically by a **27**-column table as follows:

```
   a=:'abcdefghijklmnopqrstuvwxyz '
   PH
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 0

   (0{PH)#a                    Sibilants
sz
   a#~1{PH                     Fricatives
fv

   a#~2{PH                     Plosives
bdpt
   a#~3{PH                     Vowels
aeiouy

   a#~4{PH                     Consonants
bcdfghjklmnpqrstvwxz

   a#~ >/4 2{PH                Consonants that are not plosives
cfghjklmnqrsvwxz
```

Moreover, if **t** is any text, then **(a i. t){"1 PH** provides classifications of it:

```
   t=: 'i sing of olaf'
   a i. t
8 26 18 8 13 6 26 14 5 26 14 11 0 5

   (a i. t) {"1 PH
0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 1 0 0 1 0 1 0
0 0 1 0 1 1 0 0 1 0 0 1 0 1

   ((a i. t) {"1 PH) # t
s
ff
```

```
iiooa
sngflf
```

Incomplete classifications are also useful. For example, the classification provided by **PH** is incomplete because the space belongs to none of the classes. Indeed, every **n**-way classification **B** implicitly defines a further class (which might be called *other*) defined by the expression **-.+./B**; that is, *not* the *or* over the classes. Any classification table may therefore be completed by applying the verb **comp=:] , -.@(+./)** .

Related classifications can be obtained from a table. Thus:

```
   ]M=:>1 0 0 1 0;0 1 1 0 0
1 0 0 1 0
0 1 1 0 0
   M *."0 1 PH
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

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

   sovfop=: +./"2 M *."0 1 PH
   sovfop
1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 1 0
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0

   ((a i. t) {"1 sovfop) # t
isiooa
ff
```

The first row of the resulting classification table **sovfop** includes sibilants or vowels; the second includes fricatives or plosives.

For any classification table **B**, a corresponding disjoint classification can be obtained by suppressing from each column any **1** except the first. This is achieved by the expression **</\B**. For example:

```
   </\PH
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
0 0 1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 1 0 0 0 0 1 1 0 0 0
```

The last class of the resulting table represents "all consonants that do not fall in the earlier classes".

B. Sets

A *set* is a one-way classification, and is therefore defined by a proposition. For example:

```
   GT10=: >&10            VOW=: +./@('aeiouy'&(=/))
   L=: 2 3 5 7
   MEML=: +./@(L&(=/))    III=: (]=<.) *. >&8 *. <&75
   GT10 2 3 5 7 11 13 17
0 0 0 0 1 1 1

   VOW 'happy those early days'
0 1 0 0 1 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1 1 0

   MEML i.15
0 0 1 1 0 1 0 1 0 0 0 0 0 0 0

    III 6 7 +/ 2%~i.10
0 0 0 0 0 0 1 0 1 0
0 0 0 0 1 0 1 0 1 0
```

Thus, `VOW` defines "The set of all vowels", `MEML` defines "The set of all members of the list `L` (a parameter that may be changed) ", and `III` defines "The set of all integers in an interval".

The proposition that defines a set is often itself defined in terms of the list of elements that belong to the set, as was done directly in the proposition `VOW`, and indirectly in the proposition `MEML`.

Although we often speak loosely of the set as the list itself (as in "The set `'aeiouy'`", or "The set `L`"), it is important to remember that the definition of the set is the entire *proposition*, that the ordering of the elements of the list therefore imposes no ordering on the members of the set, and that the repetition of elements in the defining list does not affect the definition of the set.

A set is completely determined by the proposition that defines it, and we will sometimes speak loosely of "the set `P`" rather than "the set defined by `P`". The defining proposition is often compound, and these compound propositions are often given special names. Thus:

`PI=: P1 *. P2`	The *intersection* of `P1` and `P2`
`PU=: P1 +. P2`	The *union* of `P1` and `P2`
`PD=: P1 >  P2`	The *difference* of `P1` and `P2`
`PSD=: P1 ~: P2`	The *symmetric difference* of `P1` and `P2`

Although a proposition defining a set may have an infinite domain (such as all numbers), it is also useful to consider propositions restricted to a finite list of arguments. We will denote such lists by names beginning with `U` (for *universe of discourse*).

For example, some or all of the letters of the alphabet might be assigned to colours, as in Acquamarine, Blue, Cyan, Dun, ... Orange, Pink, Quercitron, Red, ... Yellow, and Zaffer. The universe is then defined by:

```
   U=:'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

and the sets of primary and secondary pigment colours might be defined by the propositions:

```
   P=: +./@(1 17 24&(=/)@(U&i.))
   S=: +./@(6 14 21&(=/)@(U&i.))
```

For example:

```
   (P U)#U                  U#~S U
BRY                       GOV

   cv=: P U
   cv
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0

   ]ml=: cv # U
BRY
```

The vectors **cv** and **ml** defined above are the *characteristic vector* and *member list* of the set defined by the proposition **P** on the universe **U**. The set **P** could alternatively be defined in terms of them:

```
   P1=: {&cv@(U&i.)
   P2=: +./@(ml&(=/))
   U#~P1 U        U#~P2 U
BRY             BRY
```

The table **B=: #: i. 2^# U** (whose rows are the base-**2** representations of successive integers) provides an *exhaustive* classification of the universe **U**, including the *empty* set (represented by a characteristic vector of *zeros*), and the complete set (represented by a characteristic vector of *ones*). For example:

```
   ]EC=: #: i. 2^# U=: 2 3 5
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

This exhaustive classification is very useful. For example, the sums and products over all subsets of **U** can be obtained as follows:

```
   +/"1 U*EC                  */"1 U^EC
0 5 3 8 2 7 5 10           1 5 3 15 2 10 6 30
```

Moreover, since **EC** is exhaustive, any collection of subsets can be obtained by selecting rows from it. For example:

```
   5 1 2{EC          (2=+/"1 EC)#EC
1 0 1             0 1 1
0 0 1             1 0 1
0 1 0             1 1 0
```

C. Nub Classification

The nub of an argument contains all of its distinct items. Thus:

```
   nub=: ~. text=: 'mississippi'
   nub        ]i=:nub i. text             i{nub
misp       0 1 2 2 1 2 2 1 3 3 1        mississippi
```

A classification of an argument in terms of its nub will be called a *nub* or *self* or *auto* classification. For example:

```
   nub =/ text                      = text
1 0 0 0 0 0 0 0 0 0 0            1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 0 1 0 0 1            0 1 0 0 1 0 0 1 0 0 1
0 0 1 1 0 1 1 0 0 0 0            0 0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0            0 0 0 0 0 0 0 0 1 1 0
   +/"1 = text
1 4 4 2
```

The table on the right shows the use of the *nub-classification* monad `=` ; the expression `+/"1 = text` gives the *distribution* of the items of its argument, that is, a frequency count of its distinct items.

D. Interval Classification

A list of integers `L` may be classified according to its *interval*, that is, the list of successive integers beginning with the largest element of `L` and continuing through the smallest. Thus:

```
   (INT=: >./ - i.@>:@(>./ - <./)) L=:8 3 0 _1 0 3 8
8 7 6 5 4 3 2 1 0 _1
   (INT L) =/ L           ' *' {~ (INT L) =/ L
1 0 0 0 0 0 1             *     *
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 1 0 0 0 1 0              *   *
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 1 0 1 0 0               * *
0 0 0 1 0 0 0                *
```

If the list `L` is the result of some function, then the foregoing classification is called a *graph* of the function. For example, if:

```
PARABOLA=: -&2 * -&4
```

then `PARABOLA i. 7` yields the list `L` used above. The foregoing results can be collected to define a graphing function as follows:

```
GRAPH=: ] =/~ >./ - i.@>:@(>./ - <./)
```

Moreover, the expression `+./\GRAPH L` produces a *barchart* of `L`. Conversely, (in the case of non-integer values of `L`) it may be better to define a barchart function directly by substituting the comparison `<:/` for the `=/` used in `GRAPH`:

```
BARCHART=: ] <:/~ >./ - i.@>:@(>./ - <./)
```

A graph may then be provided by the expression `</\ BARCHART L`. Finally, it may be remarked that a barchart is a *classification* of its argument, and that the phrase `</\` applied to it produces the corresponding *disjoint* classification used as a graph.

E. Membership Classification

The functions `VOW` and `MEML` of Section B provide examples of defining a classification according to membership in a list, using an *or* over *equality*, as in `MEML=: +./@(L&(=/))` . Membership in a list is important enough to be accorded a primitive, denoted in mathematics by the Greek letter *epsilon*, and here by `e.` . For example, the function `MEML` could be defined by `e.&L` .

Membership can be used to define a form of *plotting* that supplements the barcharts and graphs provided by the interval classification in Section D. If `B` is a boolean table, then `B{' *'` gives a plot of the points indicated by the *ones* in `B`:

```
   B                  B{' *'
1 1 1 0 0 0         ***
1 0 1 0 0 0         * *
1 0 1 0 0 0         * *
1 1 1 0 0 0         ***
```

Such a table can be specified by the coordinates of its *ones*; for example, the coordinates defining `B` are the columns of the table:

```
b=:0 1 2 0 2 0 2 0 1 2,:0 0 0 1 1 2 2 3 3 3
```

Laminate (`,:`) forms a table from list arguments:

```
   b
0 1 2 0 2 0 2 0 1 2
0 0 0 1 1 2 2 3 3 3
```

If `A` is a table of all coordinates of `B`, then `B` itself can be specified in terms of the index list `b` by using membership (`e.`) in the expression `A e. boxcol b`, where `boxcol` boxes the columns of its argument. We first define a function to generate all indices of a table, using the *catalogue* function **{** that forms boxed lists by choosing an element from each of the boxes in its argument:

```
   ]w=:'ABC';'abcd'
+---+----+
|ABC|abcd|
+---+----+

   {w
+--+--+--+--+
|Aa|Ab|Ac|Ad|
+--+--+--+--+
|Ba|Bb|Bc|Bd|
+--+--+--+--+
|Ca|Cb|Cc|Cd|
+--+--+--+--+

   (i.&.>"1)  4 6
+-------+-----------+
|0 1 2 3|0 1 2 3 4 5|
+-------+-----------+

   ALLIX=: {@(i.&.>"1)
   ALLIX 4 6
+---+---+---+---+---+---+
|0 0|0 1|0 2|0 3|0 4|0 5|
+---+---+---+---+---+---+
|1 0|1 1|1 2|1 3|1 4|1 5|
+---+---+---+---+---+---+
|2 0|2 1|2 2|2 3|2 4|2 5|
+---+---+---+---+---+---+
|3 0|3 1|3 2|3 3|3 4|3 5|
+---+---+---+---+---+---+
```

We now use **ALLIX** to form the lists of *coordinates* in the usual form; that is, with the x-coordinate first and increasing from left to right, and with the y-coordinate increasing from bottom to top:

```
   ALLCO=: |.&.>@:|.@:ALLIX@:>:
   ALLCO 4 6
+---+---+---+---+---+---+---+
|0 4|1 4|2 4|3 4|4 4|5 4|6 4|
+---+---+---+---+---+---+---+
|0 3|1 3|2 3|3 3|4 3|5 3|6 3|
+---+---+---+---+---+---+---+
|0 2|1 2|2 2|3 2|4 2|5 2|6 2|
+---+---+---+---+---+---+---+
|0 1|1 1|2 1|3 1|4 1|5 1|6 1|
+---+---+---+---+---+---+---+
|0 0|1 0|2 0|3 0|4 0|5 0|6 0|
+---+---+---+---+---+---+---+


   plot=: {&' *'@(ALLCO@[ e. boxcol@])

     boxcol=: <"1@|:


   4 6 plot b
```

```
***
* *
* *
***
```

A function equivalent to **plot** can also be defined by replacing all of its component functions by the expressions that define them:

```
  PLOT=:{&' *'@(|.&.>@|.@({@(i.&.>"1))@>:@[e.<"1@|:@])
```

If **f** and **g** are two functions, then a plot of the points with x-coordinate **f k{a** and y-coordinate **g k{a** will be called a plot of **f** *with* **g** or, alternatively, a plot of **g** *versus* **f**. Thus:

```
   f=: *:          g=: +:          a=:0 1 2 3
   (f ,: g) a
0 1 4 9
0 2 4 6

   7 10 PLOT (f ,: g) a

         *

    *

 *

*
```

F. Summary of Notation

The monads *self-classification* and *catalogue* (**=** and **{**), and the dyads *membership* and *laminate* (**e.** and **,:**) were introduced in Sections C and E.

Exercises

A1 Enter **b=: ?5 7$2** to produce a random boolean table, and **n=:(7#2) #. b** to produce the base-2 values of its rows. Then enter **(7#2)#: n** and compare the result with **b** .

A2 The base -2 value of the rows of the phonetic classification table **PH** is given by:

n=: 258 2097184 41945216 71569476 62648250

Use this fact to enter the table **PH** and then experiment with its use.

B1 Define two or three propositions, and experiment with their intersection, union, and differences.

B2 Predict and enter the complete classification table for four elements, and select from it the classification table for all subsets of two elements.

C1 Experiment with nub-classification on various arguments, including the boxed list **;:'A rose is a rose is a rose.'**

D1 Enter the verbs defined in Section D, and experiment with them.

E1 Predict and verify the result of **{'ht';'ao';'gtw'**

E2 Plot **-&2*-&4** versus **]** on **i.7**, and compare the result with the parabola in Section D.

E3 Plot **2&^** versus **^&2**

Chapter 9

Polynomials

A. Introduction

A polynomial is a weighted sum of non-negative integer powers of its argument. For example:

```
   x=:1 2 3 4 5
   e=: 0 1 2 3
   c=: 1 3 3 1
   x^/e                          c*x^/e
1 1  1   1                    1  3  3   1
1 2  4   8                    1  6 12   8
1 3  9  27                    1  9 27  27
1 4 16  64                    1 12 48  64
1 5 25 125                    1 15 75 125

   +/"1 c*x^/e
8 27 64 125 216
```

The final result is the value of a polynomial with *exponents* `e` and weights (or *coefficients*) `c` applied to an argument list `x`.

A zero coefficient effectively suppresses the effect of the corresponding exponent (e.g., `+/"1 (0 0 1 2)*x^/0 1 2 3` is equivalent to `+/"1 (1 2)*x^/2 3`); it is therefore convenient to express a polynomial only in terms of its coefficients `c`, and to assume that the corresponding exponents are `i.#c` :

```
   POL=: +/"1 @ ([ * ] ^/ i.@#@[)
   c POL x
8 27 64 125 216
```

The discussion in Sections A-D will be limited to polynomials with integer coefficients, but general polynomials admit real and complex numbers, as discussed in Section F. Because a general polynomial admits an arbitrary number of arbitrary coefficients, polynomials can be designed to approximate almost any function of practical interest.

Although its utility rests largely on its potential for approximation, the polynomial has other important characteristics that can be discussed in the restricted context of integers: the following four functions are themselves polynomials:

1. The sum or difference of polynomials.
2. The product of polynomials.
3. The derivative (or "rate of change") of a polynomial.
4. The integral of (or "area under") a polynomial.

Although the coefficients of the polynomials for cases 3 and 4 are trivial to compute (`}.c*i.#c` and `0,c%>:i.#c`), their treatment will be deferred to Section H.

B. Sums and Products

The cases of the sum and product may be illustrated as follows:

```
   x=: 0 1 2 3 4 5
   c=: 1 3 3 1                 d=: 1 2 1
   c POL x
1 8 27 64 125 216

   d POL x
1 4 9 16 25 36

   (c POL x) + (d POL x)
2 12 36 80 150 252

   (c+d,0) POL x
2 12 36 80 150 252

   (c POL x) * (d POL x)
1 32 243 1024 3125 7776

   TIMES=: +//. @ (*/)
   c TIMES d
1 5 10 10 5 1

   (c TIMES d) POL x
1 32 243 1024 3125 7776
```

It will be more illuminating to discuss the sum and product of polynomials in terms of a table of an arbitrary number of coefficients. For example:

```
   ]TC=: >1 3 3 1 ; 1 2 1 ; 1 1
1 3 3 1
1 2 1 0
1 1 0 0

   +/TC
3 6 4 1
   (+/TC) POL x
```

```
3 14 39 84 155 258

   TIMES/TC
1 6 15 20 15 6 1 0 0 0

   (TIMES/TC) POL x
1 64 729 4096 15625 46656

   TC POL"1 x                     */TC POL"1 x
1 8 27 64 125 216              1 64 729 4096 15625 46656
1 4  9 16  25  36
1 2  3  4   5   6
```

It should be noted that the final zeros appended to coefficients in forming the table **TC** do not change their effects as coefficients. However, it may be convenient to trim redundant trailing zeros from a result such as **TIMES/TC** above. Thus:

```
   trim=: +./\.@* # ]
   trim TIMES/TC                  (i.7)!6
1 6 15 20 15 6 1               1 6 15 20 15 6 1
```

C. Roots

If a function **f** applied to an argument **a** yields **0**, then **a** is said to be a *zero* or *root* of **f**. A function is sometimes defined in terms of its roots. For example:

```
   PIR=: */@(-~/)
   r=: 2 3 5
   x=: 0 1 2 3 4 5 6
   r PIR x                    (x-2)*(x-3)*(x-5)
_30 _8 0 0 _2 0 12         _30 _8 0 0 _2 0 12

   r&PIR x
_30 _8 0 0 _2 0 12
```

The monad **r&PIR** is also said to be a polynomial (or polynomial in terms of roots) because it can be shown to be equivalent to a polynomial **c&POL** for appropriate coefficients **c**. This is best demonstrated by defining a function **CFR** that produces the coefficients from the roots. Thus:

```
   AS=: #:@i.@(2&^)@#
   AS r                          Boolean table of all subsets of #r items.
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

   POAS=: */"1@(-^AS)
   POAS r                        Product over all subsets of -r.
1 _5 _3 15 _2 10 6 _30
```

```
   CLBN=: =@(+/"1@AS)
   CLBN r
1 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0
0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 1
```

Classification by number of elements in set.

```
   CFR=: +/"1@|.@(CLBN*POAS)
   CFR r
_30 31 _10 1

   (CFR r) POL x
_30 _8 0 0 _2 0 12
   r PIR x
_30 _8 0 0 _2 0 12
```

Coefficients from roots.

D. Expansion

If the polynomial **d&POL** is equivalent to **c&POL x+1**, then the coefficients **d** are said to be the *expansion* of the coefficients **c**. More formally, **d** is the expansion of **c** if **d&POL** and **c&POL@>:** are equivalent. For example:

```
   x=: i. 6             c=:3 1 4 2
   ]d=: +/ c * !~/~i.#c
10 15 10 2

   d POL x
10 37 96 199 358 585
   c POL x+1
10 37 96 199 358 585

   EXP=: +/@(] * !~/~@i.@#)
   EXP c
10 15 10 2

   EXP^:4 c
199 129 28 2

   (EXP^:4 c) POL x
199 358 585 892 1291 1794

   c POL x+4
199 358 585 892 1291 1794
```

The definition of the function **EXP** will be analyzed in exercises.

Although the function **EXP** and its non-negative powers can produce expansions for **c POL x+i** for any non-negative integer **i**, it must be modified to handle the general case for fractional values of **i** such as **0.1**. This matter will be addressed in Section F, after the introduction of real numbers.

E. Graphs And Plots

Graphs and barcharts of functions with non-integer results can be produced by the methods of Section 8 D.We first define a uniform grid of a specified number of intervals, and use it to classify the non-integer results. Thus:

```
   space=:(>./ - <./)@] % [
   grid=: <./@] + space * i.@>:@[
   graph=: {&' *'@ (</\@|.@ (grid </ ] + -:@space))
   10 graph %: i. 40
                                   ****
                            *******
                     *******
                *****
           *****
       ****
    ***
  **
**

*
```

The plots of Section 8 E may be extended similarly:

```
  GPLOT=: [ PLOT |.@([ classify"0 1 ])

  classify=: <:@(+/@(grid </ ] + -:@space))

  PLOT=:{&' *'@(|.&.>@|.@({@(i.&.>"1))@>:@[e.<"1@|:@])

  6 10 GPLOT (*:,:+:) i.5
          *


       *

     *
* *
```

F. Real And Complex Numbers

In order to discuss further uses of polynomials, it will be necessary to extend the domains of our primitives beyond the integers to which they have been restricted thus far.

Just as the inverse of the *successor* led to results outside of the counting numbers, so do inverses of certain functions on integers lead outside the domain of integers. For example:

```
   a=: 1 2 3 4
   *&2 ^:_1 a                    Rational numbers
0.5 1 1.5 2

   %&2 a
0.5 1 1.5 2
```

```
   %&2 -a
_0.5 _1 _1.5 _2
```

```
   ^&2 ^:_1 a
1 1.41421 1.73205 2
```

Irrational numbers

```
   %: a
1 1.41421 1.73205 2
```

```
   %: -a
0j1 0j1.41421 0j1.73205 0j2
```

Imaginary numbers

```
   a+%:-a
1j1 2j1.41421 3j1.73205 4j2
```

Complex numbers

The rationals include the integers and, together with the irrationals, they comprise the *real* numbers. The informal extension of primitives to the real domain is straightforward; they are extended so as to maintain the properties discussed in Chapter 2. The imaginary and complex numbers are treated similarly, but merit further discussion.

Since the square of any real number is non-negative, the square root of `_1` must be a new number outside the domain of reals. It will be denoted by `0j1.` The product of `0j1` with any real number shares the property that its square is a negative number. This follows from the normal properties of multiplication:

```
   b=: 1 2 3 4 5
   b*0j1
0j1 0j2 0j3 0j4 0j5

   (b*0j1) * (b*0j1)
_1 _4 _9 _16 _25

   b*b * 0j1*0j1
_1 _4 _9 _16 _25

   (b*b) * (0j1 * 0j1)
_1 _4 _9 _16 _25

   (b*b) * _1
_1 _4 _9 _16 _25
```

If `a` and `b` and `c` and `d` are real numbers, then `a+0j1*b` and `c+0j1*d` are complex numbers. Moreover, their sum can be derived from the familiar properties of addition and multiplication:

```
   a=: 1+b=: 1+c=: 1+d=: 1
   a,b,c,d
4 3 2 1
```

```
   (a+0j1*b) + (c+0j1*d)
6j4

   (a+c) + 0j1*(c+d)
6j3

   (a+c) + 0j1*(b+d)              6+0j1*4
6j4                            6j4
```

The product of complex numbers can be derived similarly:

```
   (a+0j1*b) * (c+0j1*d)
5j10

   ((a*c)+(0j1*0j1*b*d)) + (0j1*((a*d)+(b*c)))
5j10

   ((a*c)+(_1*b*d)) + (0j1*((a*d)+(b*c)))
5j10

   ((a*c)-(b*d)) + (0j1*((a*d)+(b*c)))
5j10
```

These processes can be described succinctly by representing each complex number by a two-element list, and using the primitive `j.` defined as follows:

```
     j. y is 0j1*y
   x j. y is x+j.y
   j. b               a j. b              j./a,b
0j3                4j3                 4j3
```

The "complex plus" and "complex times" functions on two-element lists can now be defined as follows:

```
   cplus=: +
   ctimes=: -/@:* , +/@([ * |.@])
   m=: 3 4                  n=: 5 2
   j./m                  j./n
3j4                      5j2

   ]sum=: m cplus n         ]prod=: m ctimes n
8 6                      7 26

   j./prod                  (j./m)*(j./n)
7j26                     7j26
```

Although a collection of complex numbers could be represented by the rows of a two-column table, it is more convenient to adopt an *atomic* representation, obtained by boxing each list. Thus:

```
   M=:<m
```

```
   N=:<n
   M,N
+---+---+
|3 4|5 2|
+---+---+
   < (>M) ctimes (>N)
+----+
|7 26|
+----+
```

As illustrated above, the verb `cplus` can be applied to these representations only by first applying `>` (*open*), and the corresponding atomic representation is obtained by applying the inverse `<` (*box*).

The whole can be achieved by the conjunction `&.` in which the verb `u &. v` first applies `v`, applies `u` to that, and finally applies `v^:_1.` The conjunction `&.` is called *under*, because `u` is applied "under" `v` in the sense that surgery is performed under anaesthetic, the patient being restored from its effects at the end of the operation:

```
   M ctimes&.> N
+----+
|7 26|
+----+
   M,N,M
+---+---+---+
|3 4|5 2|3 4|
+---+---+---+
   ctimes&.>/ M,N,M
+-------+
|_83 106|
+-------+

   CPLUS=: cplus&.>
   CTIMES=: ctimes&.>
   M CPLUS N CTIMES M
+-----+
|10 30|
+-----+
```

The monad *magnitude* (`|`) is extended to complex numbers to yield the square root of the sum of the squares of its imaginary parts:

```
   | _5
5

   | 3j4
5

   %:+/*:3 4
5
```

In other words, the magnitude is the distance of a point from the origin when the imaginary part is plotted against the real part.

G. General Expansion

The function **EXP** of Section D has the property that **(EXP c) POL x** is equivalent to **c POL x+1**. We will now define a more general expansion such that **(y GEXP c) POL x** is equivalent to **c POL x+y**:

```
   x=: i. 6
   y=: 0.1
   c=: 3 1 4 2
   GEXP=: +/@(] * !~/~@i.@#@] * [ ^ -/~@i.@#@])
   y GEXP c
3.142 1.86 4.6 2

   (y GEXP c) POL x
3.142 11.602 41.262 104.122 212.182 377.442
   c POL x+y
3.142 11.602 41.262 104.122 212.182 377.442
```

The definition of the expansion will be analyzed in exercises.

H. Slopes And Derivatives

If **s** is a small quantity, then the difference **(f x+s)-(f x)** gives an indication of the change in the result of the function **f** in the vicinity of the point **x**. Moreover, the ratio **s%~(f x+s)-(f x)** obtained by dividing the "step size" **s** into this difference gives an indication of the *rate* at which **f** is changing. Because on a graph of the function this ratio is the slope of the secant line joining the points with coordinates **x,f x** and **(x+s), f x+s**, it is called the *secant slope* of **f**. For example:

```
   f=: *:                                The square function
   x=: 4 [ s=: 2
   (f x+s)-f x           s%~(f x+s)-f x
20                    10

   ]s=: 10^-i.5
1 0.1 0.01 0.001 0.0001

   s%~(f x+s)-f x
9 8.1 8.01 8.001 8.0001
```

We now define a dyadic function **F** such that **s F x** gives the secant slope of **f** at **x** with step size **s**:

```
   F=: [ %~"0 1 f@([+/,@])-f@]
   2 F x=: 4 5 6 7
10 12 14 16

   s F x
     9      11      13      15
   8.1    10.1    12.1    14.1
  8.01   10.01   12.01   14.01
 8.001  10.001  12.001  14.001
8.0001 10.0001 12.0001 14.0001
```

For a small step size, the secant slope `s F x` is a close approximation to the slope of the tangent to the graph of `f` at the point `x`, a value called the *derivative* of `f` at the point `x`. For example:

```
   s=:10^_10
   s F x                   Approximate derivative of square
8 10 12 14

   2*x
8 10 12 14
   f=:^&3

   s F x                   Approximate derivative of cube
48 75 108 147

   3*x^2
48 75 108 147

   f=:^&4
   s F x                   Approximate derivative of fourth power
256 500 864 1372

   4*x^3
256 500 864 1372

   n=:5
   f=:^&n


   s F x
1280 3125 6480 12005

   n*x^n-1
1280 3125 6480 12005

   n&([ * ] ^ <:@[) x
1280 3125 6480 12005
```

The foregoing results suggest that the derivative of `^&n` is the function `n&([ * ] ^ <:@[)`. This relation will be explored by displaying the terms that must be summed to produce the results used in determining the slope, that is, `f x+s` and `f x` and `(f x+s)-f x` and `s%~(f x+s)-f x`.

For the power function `f=:^&n` and for the case `n=: 3`, the terms of `f x+s` are easily obtained from the direct expansion of the product `(x+s)*(x+s)*(x+s)` to the form :

```
   ((s^3)*(x^0)+(3*(s^2)*(x^1))+(3*(s^1)*(x^2))+((s^0)*(x^3))
```

`Thus for x=:2 and s=:0.1:`

```
  1 3 3 1 * (x^0 1 2 3) * (s^3 2 1 0)          Terms of ^&3 x+s
0.001 0.06 1.2 8

  0 0 0 1 * (x^0 1 2 3)                        Terms of ^&3 x
0 0 0 8
```

```
  1 3 3 0 * (x^0 1 2 3) * (s^3 2 1 0)          Terms of difference
0.001 0.06 1.2 0

  1 3 3   * (x^0 1 2  ) * (s^3 2 1  )          "
0.001 0.06 1.2
  1 3 3   * (x^0 1 2  ) * (s^2 1 0  )          Terms of slope
0.01 0.6 12

  1 3 3   * (x^0 1 2  ) * (0^2 1 0  )          Slope for s=:0
0 0 12

  1 3 3   * (x^0 1 2  ) *  0 0 1                "
0 0 12

  3*x^2                                         "
12
```

In the general case of `^&n`, the coefficients `1 3 3 1` and `0 0 0 1` become `EXP CP n` and `CP n`, and the difference becomes:

```
   CP=: #&0,1:
   EXP=: +/@(] * !~/~@i.@#)
   CP 4
0 0 0 0 1

   EXP CP 4
1 4 6 4 1
   (EXP CP 4)-CP 4
1 4 6 4 0

   <@(EXP@CP - CP)"0 i. 6
+-+---+-----+-------+---------+-------------+
|0|1 0|1 2 0|1 3 3 0|1 4 6 4 0|1 5 10 10 5 0|
+-+---+-----+-------+---------+-------------+

   <@(_2&{.)@(EXP@CP - CP)"0 i. 7
+---+---+---+---+---+---+---+
|0 0|1 0|2 0|3 0|4 0|5 0|6 0|
+---+---+---+---+---+---+---+
```

It appears that the last two elements of the binomial coefficients of order `n` are `n` and `1`. Since the binomial coefficients are the coefficients that represent the product `(x+1)^n`, insight can be gained by applying the product process of Section B to the corresponding coefficients `1 1`:

```
   1 1 */ 1 1
1 1
1 1
   </.1 1 */ 1 1
+-+---+-+
|1|1 1|1|
+-+---+-+
   ]b2=:+//. 1 1 */ 1 1
1 2 1
   1 1 */ b2
1 2 1
1 2 1
```

```
   </. 1 1 */ b2
+-+---+---+-+
|1|2 1|1 2|1|
+-+---+---+-+

   ]b3=:+//. 1 1 */ b2
1 3 3 1
```

I. Derivatives of Polynomials

From the definition of the secant slope it is clear that the slope of a multiple of a function (`m&*@f`) is the same multiple of its slope, and that the slope of the function `f+g` is the sum of the slopes of `f` and `g`. The same relations hold for derivatives.

The polynomial `c&POL` applied to an argument `x` is a sum of terms of the form `(i{c)*(x^i)` and (using the results of Section H) its derivative is `(i{c)*i*(x^i-1)`. The derivative of the polynomial `c&POL` is therefore a polynomial with coefficients `}.c*i.#c`. For example, using the functions `F` and `POL` of Sections H and A:

```
   x=:1 2 3 4 5            c=:3 1 4 2
   D=: }.@(] * i.@#)
   D c                     (D c) POL x
1 8 6                   15 41 79 129 191

   f=:c&POL
   (s=: 10^-10) F x
15 41 79 129 191
```

J. The Exponential Family

We will now examine coefficients of the form `%!i.n`, and their relation to the coefficients of the corresponding derivative polynomial:

```
   ]ce=: %!i.n=: 7
1 1 0.5 0.166667 0.0416667 0.00833333 0.00138889

   D ce
1 1 0.5 0.166667 0.0416667 0.00833333
```

Except for the final coefficient, the function `ce&POL` and its derivative `(D ce)&POL` agree, and the agreement improves as `n` increases.

The primitive monad `^` (called *exponential*) is the limiting value of this polynomial. It is therefore a “growth” function, whose rate of growth is equal to the function itself. For example:

```
   f=: ^
   f x
2.71828 7.38906 20.0855 54.5982 148.413

   s F x
```

```
2.71828 7.38906 20.0855 54.5982 148.413
```

Not only is the exponential important in its own right, but the odd and even parts of `^` and `^@j.` produce the *hyperbolic* functions (*sinh* and *cosh*, denoted by `5&o.` and `6&o.`) and the *circular* or *trigonometric* functions (*sine* and *cosine*, denoted by `1&o.` and `2&o.`).

A function `f` is said to be *symmetric* or *even* if it gives the same result for positive and negative arguments; that is, if `f` and `f@-` agree. In terms of its graph we may say that an even function is "reflected in the vertical axis". A function `f` is *skew-symmetric* or *odd* if `f` equals `-@f@-` or, equivalently, if `f` equals `f&.-` . Its graph is reflected in the origin.

The functions:

```
   e=: -:@(f+f@-)

   o=: -:@(f-f@-)
```

are, respectively, even and odd functions. Moreover, `e+o` equals `f`, and they are called the *even* and *odd parts* of `f`.

The adverbs `..-` and `.:-` yield the even and odd parts of their arguments. For example:

```
   cosh=: ^ ..-             space must precede ..
   sinh=: ^ .:-
   ]x=: 0.2*i.6
0 0.2 0.4 0.6 0.8 1

   cosh x
1 1.02007 1.08107 1.18547 1.33743 1.54308

   cosh -x
1 1.02007 1.08107 1.18547 1.33743 1.54308

   sinh x
0 0.201336 0.410752 0.636654 0.888106 1.1752

   sinh -x
0 _0.201336 _0.410752 _0.636654 _0.888106 _1.1752

   5 o. x
0 0.201336 0.410752 0.636654 0.888106 1.1752

   (sinh+cosh) x
1 1.2214 1.49182 1.82212 2.22554 2.71828

   ^ x
1 1.2214 1.49182 1.82212 2.22554 2.71828
```

The function `^@j.` and its odd and even parts yield further important functions. We first observe that the magnitude of any result of `^@j.` is `1`. Thus:

```
   2 3 $ ^@j. x
                 1 0.980067j0.198669 0.921061j0.389418
0.825336j0.564642 0.696707j0.717356 0.540302j0.841471
```

```
   |^@j. x
1 1 1 1 1 1
```

As remarked in Section F, this implies that a plot of the imaginary part against the real part of any result of `^@j.` lies on a circle whose radius has a length of `1`. Moreover, the even and odd parts of `^@j.` are its real and imaginary parts, and therefore the plot of one of the following functions against the other forms a circle:

```
   cos=: ^@j. .. -

   sin=: j^:_1@ (^@j. .:-)

   26 52 GPLOT (sin,:cos) 0.2*i.30
                    *    *    *
               *                   *
           *                            *

       *                                    *

    *                                          *

                                                  *
 *
                                                   *
*

                                                    *
*

 *


   *                                             *

     *                                        *

         *                                *
             *
                  *                   *
                       *    *    *
```

Moreover, `(cos,sin) 0` is `1 0`, and the length along the circle from this base point to the point with coordinates `(cos,sin) x` is `x`. Since the monad `o.` multiplies its argument by *pi*, the circumference of the circle with unit radius is `o. 2`, and the sin and cos applied to the points `o.4%~i.9` yield interesting results. Thus:

```
   o. 2
6.28319
   sin o. 2
_8.67362e_19

   clean=: **|
   clean sin o. 2
0

   ]p=:4%~i.9
0 0.25 0.5 0.75 1 1.25 1.5 1.75 2

   clean (cos,:sin) o. p
1 0.707107 0 _0.707107 _1 _0.707107  0  0.707107 1
0 0.707107 1  0.707107  0 _0.707107 _1 _0.707107 0
```

The monad `*` used in the definition of `clean` above is called *signum*: `*x` is `0` if `x` is near zero, `1` if it is greater than zero, and `_1` if it is less than zero.

K. Summary Of Notation

The notation introduced in this chapter comprises complex numbers (`3j4`) and the corresponding verb `j.` (as in `3 j. 4` and `j. 4`); three conjunctions *under*, *odd* and *even* (`&.  .:  ..`); and six monads: *sine, cosine, sinh, cosh*, *signum,* and *exponential,* (`1 2 5 6&o. * ^`).

L. On Language

In accord with the comments in the language section of Chapter 1, notation has been introduced sparingly, only as needed in the topics under discussion. As a consequence, many important language constructs have been ignored. This section presents a sampling of them, grouped according to contexts in which they commonly arise.

Programming. Computer programming concerns the definition and use of verbs in a language executable on a computer, and programming therefore runs through this entire text. Nevertheless, it might not be recognized as such by programmers familiar with other languages, primarily because it is *tacit* rather than *explicit.*

A *tacit* definition is one in which no explicit mention is made of the arguments to which the defined verb might apply. For example:

```
   IQ=: <.@%                 Integer quotient of arguments.
   317 IQ 10
31

   IQ 0.166                  Integer reciprocal of argument.
6
```

An *explicit* definition begins with an entry that includes the phrase `3 : 0`, and follows with sentences that use `x.` and `y.` to denote the arguments, uses a colon alone on a line to separate the definitions of the monadic and dyadic cases, and concludes with a right parenthesis alone on a line. For example:

```
   iq=: 3 : 0
if. y. < 0
  do. 0   else. %: y.
end.
:
<. x. % y.
)

   iq
\ 25
5

   iq _25
0
```

```
   317 iq 10
31
```

Tacit definitions facilitate the use of *structured* programming, in which complicated functions are defined in terms of a hierarchy of simpler functions, each of which is useful in its own right. The following example is from statistics:

```
   std=: sqrt@var              Standard deviation
     var=: mean@sqr@norm       Variance
       norm=: ] - mean         Normalization
         mean=: +/ % #         Mean
           sqrt=: %:
   sqr=: *:
a=:3 4 5                       std a            mean a
                            0.816497         4
   ]report=: ?3 4 5 $ 10
1 7 4 5 2
0 6 6 9 3
5 8 0 0 5
6 0 3 0 4

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

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


   mean report                 Mean over tables
5.33333 6.33333  6.66667 6.33333 2.33333
      2 6.66667        4 6.66667 6.33333
5.33333       4 0.666667 3.33333       3
5.33333       5        7       1       5

   mean"1 report               Mean over rows
3.8 4.8 3.6 2.6
6.6 5.2 3.8 5.4
5.8 5.4 2.4   6

   std"1 report
2.13542 3.05941 3.13688 2.33238
1.62481 3.05941 2.78568 2.57682
3.05941 2.15407  1.0198 3.52136
```

Adverbs And Conjunctions. Adverbs and conjunctions may be defined either tacitly or explicitly. The following illustrates the tacit definition of adverbs:

```
   ]a=: 1 2 3 4 5
1 2 3 4 5

   prsu=: \\.             A sequence of adverbs   (prefix and suffix)

   < prsu a
+-+---+-----+-------+---------+
|1|1 2|1 2 3|1 2 3 4|1 2 3 4 5|
```

```
+-+---+-----+-------+---------+
|2|2 3|2 3 4|2 3 4 5|         |
+-+---+-----+-------+---------+
|3|3 4|3 4 5|       |         |
+-+---+-----+-------+---------+
|4|4 5|     |       |         |
+-+---+-----+-------+---------+
|5|   |     |       |         |
+-+---+-----+-------+---------+

   +/ prsu a
1 3  6 10 15
2 5  9 14  0
3 7 12  0  0
4 9  0  0  0
5 0  0  0  0

   iprsu=: /\\.                  q=: /prsu
   * iprsu a                     *q a
1  2  6  24 120                1  2  6  24 120
2  6 24 120   0                2  6 24 120   0
3 12 60   0   0                3 12 60   0   0
4 20  0   0   0                4 20  0   0   0
5  0  0   0   0                5  0  0   0   0

   inverse=: ^:_1          A conjunction with one argument
   %: inverse a
1 4 9 16 25

   each=:&.>
   <\a
+-+---+-----+-------+---------+
|1|1 2|1 2 3|1 2 3 4|1 2 3 4 5|
+-+---+-----+-------+---------+

   |. each <\a
+-+---+-----+-------+---------+
|1|2 1|3 2 1|4 3 2 1|5 4 3 2 1|
+-+---+-----+-------+---------+

   slope=: 1 : '[%~ + -&x.f. ]'       Explicit definition of adverb
   0.000001 ^ slope i.5
1 2.71828 7.38906 20.0855 54.5982

   ^ i.5
1 2.71828 7.38906 20.0855 54.5982
```

The tacit definition of conjunctions will be illustrated first by using the case adverb-conjunction-adverb, whose result can be used to provide the ordinary matrix product:

```
   dot=: /@(("0 1)("1 _))
   m=:i.3 3
   m                        m + dot * m
0 1 2                    15 18  21
3 4 5                    42 54  66
6 7 8                    69 90 111
```

A second illustration produces a conjunction that applies one of its arguments to a prefix, and the other to a suffix:

```
   ps=: 2 : '(x.@{.)`,`(y.@}.)\'
   f=: *: ps %:
   3 f 2 3 4 5 6                     f"0 1~i. 5
4 9 16 2.23607 2.44949          0 1 1.41421 1.73205 2

   1 f 2 3 4 5 6                0 1 1.41421 1.73205 2
4 1.73205 2 2.23607 2.44949     0 1 1.41421 1.73205 2

   f 2 3 4 5 6                  0 1       4 1.73205 2
4 1.73205 2 2.23607 2.44949     0 1       4       9 2
```

Gerunds. The conjunction ` "ties" verbs together to form a *gerund*, a noun that (like the English word *cooking*) carries the force of a verb. Gerunds have a variety of uses, of which two are illustrated below:

```
   +`*/ 1 2 3 4 5                 Insertion of successive verbs
47
   1+2*3+4*5
47

   fac_or_sqr=: !`*: @. (>&5)     The conjunction @.(agenda)
   fac_or_sqr 8                   uses the index produced by
64                                its right argument to select a
   fac_or_sqr 5                   member of the gerund to
120                               produce the final result.

   fac_or_sqr"0 i. 10
1 1 2 6 24 120 36 49 64 81
```

Recursion. A function that is defined in terms of itself is said to be *recursively* defined. For example:

```
   fac=: 1:`(] * fac@<:)@.*
   fac 5                       fac"0 i.6
120                         1 1 2 6 24 120
```

The `1:` is the constant function that yields `1`, and the monad `*` (*signum*) yields `1` if its argument is greater than `0`.

Controlled Iteration. If `f` and `g` are functions and `h=: f ^: g`, then `x h y` "iterates" `f` by applying it repeatedly as long as the result of `g` is non-zero. For example, an iterative determination of the square root using Newton's method may be defined as follows:

```
   h=: (-:@(] + %))^:([ ~: *:@]) ^: _
   5 h 1
2.23607

   *: 5 h 1
```

```
5

   1 2 3 4 5 h"0 (1)
1 1.41421 1.73205 2 2.23607
```

Linear Functions. The expression `mp=:+/ . *` uses the *dot* conjunction to produce the *dot*, *inner*, or *matrix* product `mp`. For example:

```
   mp=: +/ . *
   v=: i.3                       m=: i. 3 3
   m                             m mp m
0 1 2                        15 18  21
3 4 5                        42 54  66
6 7 8                        69 90 111

   m mp v                        v mp m
5 14 23                      15 18 21
```

Moreover, `m&mp` is a linear function which (as stated in Section 2 D) distributes over addition. For example:

```
   LF=: m&mp
   a=: 2 3 4                     b=: 5 1 1
   LF (a+b)                      (LF a)+(LF b)
14 62 110                     14 62 110

   LF (m+2*m)                    (LF m)+(LF 2*m)
 45  54  63                   45  54  63
126 162 198                   126 162 198
207 270 333                   207 270 333
```

Any linear function `LF` can be represented in the form `M&mp` for a suitable matrix `M`. If `LF` applies to vectors of `n` elements, then `M` may be obtained by applying `LF` to the identity matrix `=i.n`. For example, if `p` is an arbitrary permutation vector, then the permutation function `p&{` is linear and:

```
   n=: 6                         ]p=: n?n
                              5 2 1 3 0 4
   LF=: p&{
   x=:2 3 5 7 11 13
   LF x
13 5 3 7 2 11

   M=: LF =i.n
   M&mp x
13 5 3 7 2 11

   M                             %. M
0 0 0 0 0 1                   0 0 0 0 1 0
0 0 1 0 0 0                   0 0 1 0 0 0
0 1 0 0 0 0                   0 1 0 0 0 0
```

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

   (%.M) mp 13 5 3 7 2 11
2 3 5 7 11 13

   M&mp^:_1 (13 5 3 7 2 11)
2 3 5 7 11 13
```

Exercises

A1 Experiment with the expression `c POL x` using `x=:i.7` and various coefficients `c`, including those from the columns of Pascal's triangle in Section 7 C.

A2 Using the value of `x` from Ex A1, evaluate `(x+1)^n` for various values of `n`, and compare the results with those of Exercise A1.

A3 Define a function `CP` such that `(CP n) POL x` equals `x^n`.

Answer: `CP=: #&0,1:`

B1 Evaluate `1 1&TIMES ^:n 1` for various values of `n`.

B2 Explore the definition of `TIMES` by evaluating the following:

```
c=: 3 1 4        d=: 2 0 3 5

c */d            </.c */ d      +//. c */ d
```

Also compare `TIMES` with multiplication of integers in Section 4 C.

B3 Use theorems 3-5 of Section 5 D to prove that the product of polynomials with coefficients `C` and `D` is equivalent to the polynomial with coefficients `+//.C*/D`.

C1 Predict and test the results of `CFR n#1` for various values of `n`. Repeat for `CFR n#_1`.

C2 Define a function `F` such that `n F r` gives the coefficients of a polynomial having `n` repeated roots `r`. Test it on expressions such as

```
5 F 1      5 F _1      5&F"0 -i. 6      F&_1"0>:i.6
```

Answer: `F=: CFR@#`

D1 Predict and test the results of `EXP&CP n` for various values of `n`, where `CP` is from Ex A3.

D2 Explore the definition of `EXP` by defining the functions:

```
A=: +/"1

B=: ] * C

C=: !/~@i.@#@]
```

and then evaluating expressions such as `C d=:3 1 4 2` and `B d` and `A B d`.

E1 Predict and test the results of the following expressions:

```
CTIMES/a=: 1 2;3 4;5 6

CTIMES/\a
```

```
a CPLUS CTIMES/a
```

G1 Experiment with **GEXP** for various arguments.

G2 Explore the definition of **GEXP** by defining the subtraction table function `ST=: -~/~@i.@#@]` and evaluating `ST c=: 3 1 4 2`.

G3 Evaluate `y^ST c` for various values of `y`, including `0`.

G4 Explain the equivalence of the expressions `(x+y)^n` and `(y GEXP CP n) POL x`, where `CP` is from Exercise A3.

H1 Extend the sequence that concluded Section H.

L1 Test the assertion that the scan `+/\` is linear.

L2 Predict and test the results of the following expressions:

```
c=: 3 1 4 2 6
+/\c
I=: =/~i.#c
M=: +/\ I
d=: M +/ . * c
(%.M) +/ . * d
(>:/~i.#c) +/ . * c
```

L3 Look through earlier chapters for other linear functions, and re-express them as inner products. In particular, identify the cases that can employ Pascal's triangle (`!/~i.n`) and Vandermonde's matrix `x^/i.#c`.

L4 Predict and test the results of applying the matrix inversion function `%.` to some of the matrices used in Exercises L2 and L3, and use them in defining linear functions.

L5 Examine the matrices `M` and `%.M` of Ex L2, and note that the former produces "aggregation" or "integration", and the latter produces "differencing".

L6 Review the discussion of combinations in Section 7 C, and enter and experiment with the following structured definition of a function for generating tables of combinations:

```
comb=: basis`recur@.test
  basis=:i.@(<:,[)
  recur=: (count#start),.(index@count{comb&.<:)
     count=:<:@[!<:@[+|.@start
     start=:i.@-.@-
     index=:;@:((i.-])&.>)
  test=: *@[*.<
```

[Try `3 comb 4`]

References

1. *American Heritage Dictionary of the English Language,* Houghton-mifflin (Any edition that includes the appendix of Indo-European roots).

2. Klein, Felix, *Elementary Mathematics from an Advanced Standpoint*, Dover Publications.

3. Cajori, F., *A History of Mathematical Notations*, Open Court Publishing Company, LaSalle, Illinois.

4. Lakatos, Imre, *Proofs and Refutations: the logic of mathematical discovery*, Cambridge University Press.

INDEX