0. Kenneth Iverson, APL and J

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. A. N. Whitehead, *An Invitation to Mathematics*.

Development of APL and J

APL had its origins in work begun by Kenneth Iverson - a Canadian who was born near Camrose, Alberta and who was a graduate of Queen's University, Kingston - while a graduate student at Harvard in the early 1950s. He became dissatisfied with the inadequacies of conventional mathematical notation and began to develop an alternative notation. After completing his doctorate, he continued this work while teaching in the newly established program in Automatic Data Processing at Harvard. The first applications of APL were for the description of algorithms arising in problems of sorting, searching and optimization.

After leaving Harvard in 1960, Ken joined the IBM Research Division in Yorktown Heights, New York where he continued this work. In 1962 he published a detailed account in the book *A Programming Language*, the title being the source of the name APL. The first experimental computer implementation was in 1965 and was used in a batch mode by submitting decks of punched cards containing programs and data and waiting to receive the output in printed form. In November 1966 the APL\360 system running on an IBM/360 Model 50 was providing service to users in the IBM Research Division in Yorktown Heights and could be used interactively from remote terminals. APL became publicly available in 1968.

The principles underlying the design of APL were simplicity, brevity and generality. The data objects in APL are, in addition to individual items, i.e., numbers or characters, lists of items arranged in rows, tables arranged in rows and columns, and in general rectangular arrays of arbitrary dimension. In addition to the usual elementary arithmetical functions of addition, subtraction, multiplication, division and raising to a power, there are a large number of additional functions which are defined for arrays as well as for individual numbers.

A simple analogy, due to Frederick Brooks who was a friend and colleague of Ken Iverson, may help illustrate an important difference between APL (and **J**) and "conventional" languages such as FORTRAN, BASIC, Pascal and Java. Suppose we have a box of apples from which we wish to select all of the good

apples. Not wishing to do the task ourselves, we write a list of instructions to be carried out by a helper. The instructions corresponding to a conventional language might be expressed something like the following:

> Select an apple from the box. If it is good, set it aside in some place reserved for the good apples; if it is not good, discard it. Select a second apple; if it is good put it in the reserved place, and if it is not good discard it. ... Continue in this manner examining each apple in turn until all of the apples have been examined and the good apples selected and put in the place reserved for them. In summary, then, examine the apples one at a time starting with the first one we pick up and finishing with the last one in the box so that we have selected and set aside apple-by-apple all of the good apples.

The instructions corresponding to an array language could be stated simply as follows:

> Select all of the good apples. (Of course, the apples would still have to be examined individually, but the details are left to the helper.)

Thus conventional languages may be considered apple-by-apple languages while the array languages APL and **J** - and Mathematica and MATLAB and a few others - may be considered all-the-apples-at-once languages. One characteristic evident on even a cursory examination of an APL program or only a line or so of APL code is the character set which was designed especially for APL and implemented initially on the "golfball" type element for the IBM 2741 Selectric Typewriter. In addition to conventional symbols such as +, −, ×, and ÷, there were arrows pointing left ←, right →, up ↑ and down ↓, small circles ° and large circles ○, a character resembling a badly made L ⌊ and its upside-down companion ⌈, and some strange characters with the appearance of having been made by overstriking two other characters as indeed they were. This gave rise to comments, many of them derisive, such as "APL? That's the language with all the funny symbols, isn't it?" More seriously, since many writers of APL programs seemed to delight in making each line of a program as long – and as incomprehensible – as possible, APL was often term a "write-only language", an obvious pun on the term "read-only memory".

APL soon gained an enthusiastic and growing following within IBM and in academia and private business. Ken Iverson was made an IBM Fellow in 1970 and was awarded the Association for Computing Machinery Turing Award in 1979. In 1980 he left IBM and returned to Canada to work for I. P. Sharp Associates, a firm with head offices in Toronto which was using APL to establish a time-sharing service that became widely used in Canada, the United States and Europe.

In 1987 Ken retired from I. P. Sharp, or to use his own words "retired from paid employment", and turned his attention to the development and promotion of a modern dialect of APL which was called simply **J**. His motivation for developing **J** was to provide a tool for writing about and teaching a variety of mathematical topics that was available either free or for a nominal charge, could be easily printed, was

implemented in a number of different computing environments, and maintained the simplicity and generality of APL. **J** was first implemented in 1990, and is now available on a wide range of computers and operating systems and utilizes the latest developments in software including graphical user interfaces such as MS Windows. One very obvious difference between APL and **J** is the use of the ASCII character set available on all keyboards. This removes the many difficulties associated with the APL character set, difficulties only exacerbated by the increased use of text-based email and the World Wide Web.

J was quickly accepted by many in the APL community and also by many who were new to array languages. The language has undergone a continued and orderly development since it was first released. Ken Iverson was actively involved in the extension, application and promotion of the language until his death in 2004 in his 84th year.

The following two sections give dialogues interspersed with explanatory notes and with data to be used in examples. **J** expressions entered at the keyboard are indented while responses begin at the left margin. Comments in the dialogue are preceded with `NB.` and are not executed. The first very short dialogue may be a sufficient initial introduction for the reader unfamiliar with **J**. A reading of the remaining sections – a second longer dialogue, explicit verbs, a summary of **J**, and the dice-throwing example - may be deferred for the present.

A very short dialogue with J

We shall introduce here only the operations, called *verbs* in **J**, of *plus* `+` and *times* `*`, *is* `=:` for assignment, and the derived verb `+/` which may be considered analogous to the conventional notation Σ for summation, and illustrate their use on individual numbers and on lists of numbers.

```
   3 + 5            NB. Plus
8
   3 * 5            NB. Times
15
   3 7 4 + 2 5 9
5 12 13
   3 7 4 * 2 5 9
6 35 36
   x=: 3 7 4
   y=: 2 5 9
   x + y
5 12 13
```

```
   x * y
6 35 36
   +/x
14
   +/x + y
30
   +/x * y
77
```

A small handmade clay pot used to hold spare change contains 4 nickels, 18 dimes, 13 quarters, 5 loonies and 7 toonies. We wish find the total amount of change in the pot by first finding the total for each denomination of coin and then adding these amounts.

```
   Denomination=: 0.05 0.10 0.25 1.00 2.00
   Number=: 4 18 13 5 7
   Denomination * Number
0.2 1.8 3.25 5 14
   +/Denomination * Number
24.25
   Total=: +/Denomination * Number
   Total
24.25
```

Now that we have a little money let us go to the grocery store and buy the following items: a box of crackers at $2.50, a package of whole wheat rolls at $1.99, a small pail of ice cream at $3.99, some grapes at $2.30, a box of strawberries at $2.50, some apples at $1.57, and one bunch of broccoli at $1.26. We wish to find how many different items we have purchased and their total cost. The following **J** dialogue first introduces the verbs *tally* `#` and *append* `,` and then shows the necessary calculations:

```
   #4 7 2 4              NB. Tally, the number of items in a list
4
   #0 1 2
3
   4 7 2 4, 12           NB. Append, join lists together
4 7 2 4 12
   a=: 4 7 2 4, 0 1  2
   a
4 7 2 4 0 1 2
```

```
   #a
7
   Price=: 2.50 1.99 3.99 2.30 2.50 1.57 1.26
   #Price                NB. Number of items purchased
7
   +/Price               NB. Total cost
16.11
   (#,+/)Price           NB. Shopping summary, i.e., 7 items purchased
7 16.11                  NB.   at a total cost of $16.11
   summary=: #,+/        NB. Defined verb for summary
   summary Price
7 16.11
```

A longer dialogue with J

The primitive functions, called *verbs*, are represented by a single character or by a single character followed by a period or by a colon, and, for example, + is *plus*, +. is g*reatest common divisor*, and +: is *not-or*. The following examples are illustrative:

```
   3 + 5             NB. Plus
8
   3 - 5             NB. Minus (The negative sign is represented
_2                   NB.    by _ and is considered part of the number)

   3 * 5             NB. Times
15
   3 % 5             NB. Divided by
0.6
   15 +. 6           NB. Greatest common divisor
3
   5 > 10            NB. Larger Than
0                    NB. 0 denotes the falsity of the expression
   5 > 3
1                    NB. 1 denotes the truth of the expression
   5 <. 3            NB. Lesser Of
3
```

```
   5 >. 3          NB. Larger Of
5
   5 >: 5          NB. Larger Or Equal
1
   Price           NB. Groceries again
2.5 1.99 3.99 2.3 2.5 1.57 1.26
   +/Price         NB. Total cost of shopping trip
16.11
   >./Price        NB. Maximum price (ice cream)
3.99
   <./Price        NB. Minumum price (broccoli)
1.26
```

All of the above verbs are *dyadic* in that they have two arguments, one on the right and the other on the left. Verbs also have a *monadic* form with a single argument on the right.

```
   - 5             NB. Negate
_5
   - _5
5
   % 4             NB. Reciprocal
0.25
   >. 2.5          NB. Ceiling (Round up)
3
   >: 2.5          NB. Increment (Add 1)
3.5
```

Precedence amongst verbs is determined by parentheses, and in their absence the right argument is the entire expression on the right and the left argument is the variable or constant immediately on the left as shown by the following examples:

```
   3 * 5 + 4
27
   3 * (5 + 4)
27
   (3 * 5) + 4
19
```

```
   2 + 3 * 5 + 4
29
   4 % 5
0.8
   % 4 % 5
1.25
```

Grocery shopping again

The **J** defined verb

```
summary=: #,+/
```

for the shopping summary is an example of a program written in an all-the-apples-at-once language in which one does not have to be concerned with the details of handling each price in sequence. As an example of a program for the same problem written in an apple-by-apple language, the following is a BASIC program for the same problem:

```
100 REM BASIC PROGRAM FOR SHOPPING EXAMPLE
110 DATA 2.50,1.79,3.99,2.30,2.50,1.57,1.26,0
120 N = 0
130 TOTAL = 0
140 READ PRICE
150 WHILE PRICE > 0
160    N = N + 1
170    TOTAL = TOTAL + PRICE
180    READ PRICE
190 WEND
200 PRINT N, TOTAL
210 STOP
220 END
```

It should be easy to see that the prices are considered one at a time and the tally increased by 1 and the total cost incremented by the price that has just been read by the READ statement until a price of 0 is encountered which indicates that all prices have been considered.

The same repetitive procedure may be expressed in **J** although for this problem it is certainly not necessary and indeed should be avoided. This program requires the primitive monadic verbs *head* `{.` and *behead* `}.` which take and remove, respectively, the first item from their list arguments as shown in the following dialogue:

```
   x=: 6 8 2 0 5
   {. x            NB. Head, first item of list
6
   }. x            NB. Behead, drop first item of list
8 2 0 5
   x=: }. x
   x
8 2 0 5
   z=: ,25         NB. One-item list
   #z
1
   z=: }. z        NB. Remove first item from one-item list
   z               NB. Empty list


   #z
0
```

The following is a **J** program for the grocery example in which the items are handled one at a time in an apple-by-apple manner, the items being removed from the list of prices until the list is empty:

```
   shopping =: 3 : 0
   price=: y.
   n=: 0
   total=: 0
   while. 0 < #price do.
   n=: >:n
   total=: total + {.price
   price=: }.price
   end.
   n, total
   )
```

The verb `shopping` is known as an *explicit* verb since its argument is stated explicitly as compared with a *functional* or *tacit* verb such as `summary` without explicit arguments or control structures. For future reference, and for completeness, we shall make a few remarks about the form of these explicit verbs using the simple verbs `f1`, `f2`, `f3a` and `f3b` as examples:

```
f1=: 3 : 0      f2=: 3 : 0      f3a=: 3 : 0       f3b=: 3 : 0
% y.            :               % y.              1 f3b y.
)               x. % y.         :                 :
                )               x. % y.           x. % y.
                                )                 )
```

The monadic verb `f1` gives the reciprocal so that, for example, `f1 2.5` is `0.4`, and the dyadic verb `f2` gives the quotient of its two arguments so that `15 f2 6` is `2.5`. The verbs `f3a` and `f3b` are ambivalent and each gives the reciprocal when used monadically and the quotient when used dyadically, i.e., `f3a 2.5` is `0.4` and `15 f3a 6` is `2.5`, with the same results for `f3b`. The first line in each definition gives the name and specifies that the definition is that of a verb. The last line of each definition is a right parenthesis. A colon **:** separates the monadic and dyadic definitions and is omitted for a monadic verb. Left and right arguments are represented by `x.` and `y.` respectively.

Summary

In this penultimate section we shall summarize those features of **J** discussed and illustrated in previous sections and add a few more remarks about the language:

- The ASCII character set is used.
- Primitives are represented by a single character or by a single character followed by a period or a colon.
- The terminology of English grammar is used rather than that of programming languages. Functions are referred to as *verbs*, and their arguments are called *nouns* and *pronouns* although the more conventional terms *constants* and *variables* are used here. Verbs may be modified by *adverbs,* and, for example, the verb +/ which gives the sum over an array is derived from the verb + *plus* by use of the adverb / *insert*. There are also c*onjunctions* and *gerunds* which will be introduced later as the need arises.
- Precedence amongst verbs is determined by parentheses, and in their absence the right argument is the entire expression on the right and the left argument is the constant or variable immediately on the left. Adverbs and conjunctions take precedence over verbs with the left argument being the entire verb phrase on the left.
- Negative numbers are indicated by a preceding underbar _ which is considered to be part of the number as is the decimal point. Also the decimal point is necessarily preceded by at least one digit so that, for example, two-fifths as a decimal fraction is represented as `0.4`.
- Most function symbols represent one function when used with one argument and another function when used with two arguments. For example, `%` represents the monadic verb *reciprocal* and the dyadic verb *divided by*, and `/` represents the monadic adverb *insert* and the dyadic adverb *table*.

- Constants may be single items or *atoms*, one-dimensional arrays or *lis*ts, two-dimensional arrays or *tables*, or arrays of higher dimension or *reports*. Thus the expression `a + b` is a valid sum as long as `a` and `b` are compatible arrays.
- Verbs may be defined in a *functional* or *tacit* manner without explicit arguments or control structures. However, *explicit* verbs may be defined where the arguments are specified and where the definition may extend over several lines and involve control structures similar to those in conventional programming languages.
- An uninterrupted sequence of three verbs is known as a *fork* and is a generalization of the notation of conventional mathematics where, for example, *(f+g)x* represents the sum *f(x)+g(x)*.

Rolling dice

The adverb `/` has a dyadic form *table* which may be used to give the familiar arithmetic tables learned in elementary school and also their generalizations with arbitrary dyadic verb arguments. The following shows its use to give the first three rows and five columns of an addition table, where, for example, the item in the second row and fourth column is the sum `2 + 4` or `6`:

```
   1 2 3 +/ 1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

The following example show similar portions of subtraction, multiplication and division tables:

```
   1 2 3 -/ 1 2 3 4 5
0 _1 _2 _3 _4
1  0 _1 _2 _3
2  1  0 _1 _2
   1 2 3 */ 1 2 3 4 5
1 2 3  4  5
2 4 6  8 10
3 6 9 12 15
   1 2 3 %/ 1 2 3 4 5
1 0.5 0.333333 0.25 0.2
2   1 0.666667  0.5 0.4
3 1.5        1 0.75 0.6
```

We shall use this adverb extensively in the next chapter on number systems and limit the present discussion to its use for tabulating the results of rolling an unbiased four-sided die as given in the following dialogue:

```
   Rolls = 3 1 2 1 2 3 3 4 4 3 1 3
   1 = Rolls
0 1 0 1 0 0 0 0 0 0 1 0   NB. 1 occurs on rolls 2, 4 and 11
   2 = Rolls
0 0 1 0 1 0 0 0 0 0 0 0   NB. 2 occurs rolls 3 and 5
   3 = Rolls
1 0 0 0 0 1 1 0 0 1 0 1   NB. 3 occurs on rolls 1, 6, 7, 10 and 12
   4 = Rolls
0 0 0 0 0 0 0 1 1 0 0 0   NB. 4 occurs on rolls 8 and 9
   Faces=: 1 2 3 4
   Faces =/ Rolls
0 1 0 1 0 0 0 0 0 0 1 0
0 0 1 0 1 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0 0 1 0 1
0 0 0 0 0 0 0 1 1 0 0 0
   Frequencies=: +/"1 (Faces =/  Rolls)   NB. +/"1 gives row sums
   Frequencies
3 2 5 2
   Faces ,: Frequencies   NB. Laminate ,: gives a 2-row table with
1 2 3 4                   NB.   faces in the first row and
3 2 5 2                   NB.   frequencies in the second
```