A Lecture on Array Languages

*—by **Keith Smillie***
Edmonton, Alberta, Canada

PROGRAMMING LANGUAGES are introduced briefly and a distinction is made between conventional and array languages. The language J is given as a modern exemplar of array languages and is illustrated with a few simple examples. Some comments are given on the teaching of languages and on the history of computing.

A much longer discussion of J from which most of the material in this paper has been taken may be found in Mostly J, *which is available at*
www.cs.ualberta.ca/~smillie/Jpage.htm

Introduction

"If you cannot—in the long run—tell everyone what you have been doing, your doing has been worthless."

So said the physicist and Nobel laureate Erwin Schrödinger in a series of public lectures published in the early 1950s as *Science and Humanism*. His remarks may be even more relevant today when universities emphasize the acquisition of new knowledge and its dissemination in scholarly publications, at times so it seems to the exclusion of the interests of those wishing some understanding of the sciences as part of their general education.

It is with thoughts like this in mind that I attempt to explain exactly what it is I have been doing for so many years. I have spent almost all of my professional life—very agreeably most of the time—programming, teaching programming, and writing about programming. Although much of my work has been of a mathematical or statistical nature and often has involved the use of mathematical notation and mathematical and statistical techniques, much of it is in principle rather simple and within the grasp of any literate person. I have found my work to be intensely interesting, and it is just conceivable that there are those who may find some interest in a brief description that is not burdened with either technical detail or mathematical notation.

Also I believe we have an obligation to explain to people how we spend our professional lives. We owe this to our family and friends, whom we on occasion may neglect when we become preoccupied with our work. We also have an obligation to others who support us directly and indirectly and who may often wonder what they are getting in return.

We shall begin with a look at some of the programming languages I have used and make the distinction between what I term "conventional" and "array" languages and emphasize the differences between them by a simple analogy of counting the number of good apples in a box of apples. We shall then introduce the array language J by means of a simple example of analyzing a short list of book prices. The use of J will be further illustrated by a discussion of a few problems with which I have been concerned during my career. This discussion of J concludes with a summary of the J operations we have used in the previous examples. We then return to programming languages in general with some remarks on both the teaching of programming languages and natural languages and also the importance of keeping an appropriate historical prospective in one's work. We conclude with a few remarks on the place of array languages in today's programming environment.

Persons wishing to learn more about J than is given in this paper are encouraged to visit the Iverson Software Inc. Web site at "www.jsoftware.com" for futher information, tutorials and links to related sites.

Programming Languages

Conventional languages

The first computer I worked with was the National Cash Register 102A which, as were all computers in the 1950s, programmed in machine-language. A program consisted of a sequence of instructions written in numerical form, each specifying the operation and the addresses (locations) in memory of the numbers to be operated on and the address of the result. For example, the instruction

35 2001 1025 1050

meant "add the number in location 2001 to the number in location 1025 and put the sum in location 1050." Data entered into the computer had to be converted to binary, the arithmetic performed in binary, and the final results converted to decimal before being printed. Moreover, the programmer had to explicitly keep track of the size of all numbers throughout a computation to ensure that they, or at least their binary equivalents, always remained less than unity in absolute value. (A later version of the 102A, the 102D, used decimal arithmetic which removed the problems of conversion between decimal and binary.) Thus programs for what we would consider today very simple or even trivial calculations could be quite lengthy. For example, one program for calculating the average of a list of numbers required about four dozen instructions.

To simplify the programming task, most computers were soon provided with programs which allowed the programmer to use a somewhat simpler language than machine-language which would be interpreted one command at a time as sequences of machine-language instructions. An example was Simple Code for the Stantec Zebra, a computer installed at the Suffield Experimental Station in Alberta and which I used for the Canada Department of Agriculture in Lethbridge, Alberta. Programming in Simple Code was still a most demanding task but it was very much easier than programming the Stantec Zebra in machine-

language with its repertoire of, theoretically at least, several million different instructions.

A breakthrough in programming came in the late 1950s with the development by the International Business Machines Corporation of FORTRAN, for Formula Translating System, which allowed the programmer to write programs in an algebraic-like language. The program would be first translated in its entirety to machine-language and then executed. Since its release in 1957 FORTRAN has been almost continuously developed—the latest version is FORTRAN90—and has had a profound effect on the the development of programming languages and their teaching.

The appearance of FORTRAN made the use of computers feasible for people who did not wish to become full-time programmers at the expense of their chosen professions. Courses in FORTRAN were soon established at universities and colleges for students in science and engineering. In the early 1970s at the University of Alberta FORTRAN was replaced briefly with ALGOLW and then with Pascal as the first language for computing science students. Pascal has just been replaced with Java in most introductory courses. Although there are important differences between these languages, they are sufficiently similar that students experience the same problems when learning to use them.

BASIC, Beginner's All-Purpose Symbolic Instruction Code, was developed at Dartmouth College, a small liberal arts college in New Hampshire, as a simple alternative to FORTRAN for undergraduate students, most of whom were in the social sciences and the humanities. The first BASIC program was run on May 1, 1964 at four o'clock in the morning. It was equivalent to the evaluation of the arithmetic expression `(7 + 8) ÷ 3`. The language was an immediate success and has become probably the most popular and widely used language in the world. It received only limited use at the University of Alberta until the mid-1980s when it became the first language for students in Arts and Education. Again, in spite of important differences with the languages already mentioned, students encountered the same problems in learning it as with the other languages.

I believe that FORTRAN, ALGOLW, Pascal and BASIC—and indeed many other languages—are sufficiently similar to be grouped in a class which I call "conventional" languages. In spite of their many differences, they all have the characteristic that the basic unit is, for numerical work at least, the individual number, and any computation must be broken down into sequences of operations on these units. For example, the procedure to finding the sum of a list of numbers would be analogous to that used manually with a pocket calculator: an accumulating register is first cleared, and then the numbers are added one at a time into this register until all of the numbers have been treated. If the sum were required for a table of numbers representing, say, expenditures in each of several categories for each month of the year, then the numbers would have to be added row-by-row and column-by-column. Thus programs for computations which we can either visualize or express verbally in very simple terms can result in very complicated programs which are tedious to construct and prone to error.

On the other hand, in array languages arithmetic and logical operations may be applied not only to individual numbers but to lists, tables, and structures of arbitrary dimension. Thus procedures which may become quite complicated with conventional languages can be expressed very simply with array languages. We turn to a brief overview of some of these languages in the next section.

Array languages

The first array langauge I used was APL, for A Programming Language, originally conceived by Kenneth Iverson while a graduate student at Harvard University in the early 1950s. It was intended as an alternative to conventional mathematical notation for the description of algorithms arising in problems of sorting, searching and optimization. After leaving Harvard, Iverson joined IBM where APL was first implemented on a computer in 1966. Since then there have been several major implementations, the current one being designated APL2.

The principles underlying the design of APL have been simplicity, brevity, uniformity and generality. While the conventions of mathematical notation have been respected, these principles have always been given precedence. The data objects in APL are one-dimensional lists, two-dimensional tables, and in general rectangular arrays of arbitrary dimension. In addition to the usual elementary arithmetical operations of addition, subtraction, multiplication, division and raising to a power, there is a large number of additional operations which are defined for arrays as well as for individual numbers. For example, if we have a list of unit prices and a list of quantities of each item purchased, then a single multiplication will give a list of the total amount spent for each item, one addition will give the total amount spent, and if there is a sales tax one more multiplication will give the total cost including tax.

Nial, Nested Interactive Array Language, combines concepts from APL and other languages within the framework of a mathematical model called array theory. It was developed over many years by Trenchard More of the IBM Cambridge Scientific Center. The name comes from the Old Norse Icelandic name *Njal*. The data objects in Nial are arrays whose items may be arrays, whose items in turn may be arrays, and so on. The basic items of arrays may be any of several types such as integers, decimal numbers, literal characters, etc., and different types may occur in the same array. As with APL, there is a large number of primitive array operations available and additional operators and programs may be defined.

Upon retirement about ten years ago, Kenneth Iverson began work on a "modern dialect" of APL that would provide the simplicity and generality of APL while at the same time be readily and inexpensively available on a variety of computers and

capable of being printed on standard printers. The language was given the name J by Roger Hui—who along with Ken, his son Eric, and Chris Burke have been its principal developers—because, he said, "the letter 'J' is easy to type." (We might note that the name J precedes the naming of Microsoft's Java development tool as Visual J++.)

There appear to be no data readily available on either the use of array languages relative to other programming languages or the relative use of APL, Nial and J. APL has been used extensively in a wide range of scientific and commercial applications and has had a group of devoted advocates in many universities. It continues to be used extensively and is supported by IBM and other major companies. Nial had a small but enthusiastic group of supporters, mostly academics, in the 1980s, but appears to be little used now. Interest in J has grown in the last few years, and should continue to grow as the language develops.

Counting apples

An excellent example of the difference between conventional languages and array languages has been given by Frederick Brooks of the University of North Carolina. In his example, which we have modified somewhat, we are to give a list of instructions for counting the number of good apples in a box of apples. We shall give the instructions first in the style they would be given in a conventional language and then in the style of an array language.

The first instruction in the conventional programming style would be to get a blank piece of paper for keeping a tally of the good apples as we come across them in the box. The next instruction would be to pick an apple from the box, and examine it for goodness according to some criterion. If it is a good apple, then put a mark on the tally paper. Then pick a second apple, examine it for goodness, and if it is a good apple put another tally mark on the paper. Then pick another apple, ... Continue this procedure of examining the apples one by one until all of the apples in the box have been examined and the occurrence of the good apples recorded. The number of good apples will be given by adding the number of marks on the tally sheet.

On the other hand, the instructions given in the style of an array language would consist of a simple statement such as "Mark all of the good apples in the box and then add up the marks." It would be assumed that the person to whom the instructions were given would be able to work out the details of marking the good apples one at a time and then determining how many apples had been marked.

A conventional programming language, then, is an apple-at-a-time language in which all of the details must be carefully specified in the program and tested to ensure that the program does what it is designed to do. An array language is an all-the-apples-at-once language in which most of the details are taken care of by the language itself and are not visible to the programmer. Of course, a program written in any language must be carefully tested but the testing is usually much simpler with array languages.

An example

We shall now consider how a very simple problem is programmed in a conventional language and in J. We shall discuss the conventional program briefly and defer a discussion of the J program until the next section. The present discussion will emphasize the difference between the apple-at-a-time approach of conventional languages and the all-the-apples-at-once approach of array languages.

Suppose, then, that we go to a bookstore and buy some books with the following prices: $20.95, $29.50, $22.50, $13.95 and $19.50. We are interested in the number of books we purchase, the total cost, and the prices of the cheapest book and of the most expensive book. We may very simply keep a cumulative record of the information we want by counting the books as we select them, adding the cost of the current one to the total cost, and when necessary updating the record of the cheapest and most expensive books. When we are finished we will have the information we desire.

The following is a BASIC program that will calculate the desired summary statistics given the prices of the books:

```
REM Summary of book prices
DATA 20.95, 29.50, 22.50, 13.95, 19.50, 0
N = 0
Total = 0
MinPrice = 999
MaxPrice = 0
READ Price
WHILE Price > 0
   N = N + 1
   Total = Total + Price
   IF Price < MinPrice THEN
       MinPrice = Price
   ELSEIF Price > MaxPrice THEN
       MaxPrice = Price
   END IF
   READ Price
WEND
PRINT N, Total, MinPrice, MaxPrice
STOP
END
```

A person with little or even no knowledge of BASIC should be able to see the similarity between the steps in the program and the informal method of the last paragraph. We note that the repetitive operations beginning with the `WHILE` statement continue until a book price of zero is encountered at the end of the list of valid prices. Furthermore we note previous to the `WHILE` statement the variables in the program are set to their initial values—zero for each of the tally and total cost, and unrealistically large and small minimum and maximum prices respectively.

The following is the J program for this problem:

```
num=: #
total=: +/
min=: <./
max=: >./
summary=: num, total, min, max
```

Without any knowledge of J at all, we may see the lack of explicit repetition in these statements, and the similarity of the program to the all-the-apples-at-once procedure of the last section. If we define the book prices by the statement

```
P=: 20.95 29.50 22.50 13.95 19.50 ,
```

then the program may be executed by the statement

```
summary P
```

to give the results

```
5 106.4 13.95 29.5
```

for the number of books, total price, and minimum and maximum prices, respectively.

In the next part of this paper we shall discuss a few simple applications of J, beginning with the book calculations of the last paragraph.

A taste of J

Buying books

We shall begin our brief look at J by developing the very simple programs required for the summary of book prices. We shall introduce just sufficient J as we proceed to solve the problem. First of all we shall define the list

```
P=: 20.95 29.50 22.50 13.95 9.50
```

of book prices. (We might have named the list `Prices` or `BookPrices`, or some other descriptive name, but have chosen the name `P` for brevity.) The list `P` could contain any number of prices, and would be defined in the same manner except that very long lists would be defined as a number of shorter lists which would be combined to give the final list.

We mentioned earlier that the familiar operations of addition, subtraction, multiplication and division have been supplemented in J with a large number of additional primitive operations usually referred to as "verbs" by analogy with English where the term refers to the action words in the language. We shall introduce some of these verbs in our solution of the present example.

First of all, the number of books purchased is equal to the number of items in the list `P` . This is given by the primitive verb *tally* represented by #, so that `#P` has the value `5`, the number of items in `P`. To avoid remembering the symbol #, we may define the simple verb

```
num=: #
```

so that `num P` is `5`.

Before proceeding further we might remark that J has been implemented on the computer as an interactive language so that expressions may be evaluated, and operations, i.e., verbs, defined immediately on input. Expressions entered by a user are indented three spaces and any system responses start at the left margin. For example, the operations performed so far in this example would appear as follows if input:

```
   P=: 20.95 29.50 22.50 13.95 19.50
   #P
5
   num=: #
   num P
5
```

In addition to being used in this manner programs may be entered and stored for subsequent execution.

The total amount spent on books is given by the sum of the items in the list `P`, or

```
20.95+29.50+22.50+13.95+19.50
```

which is `106.40`. For convenience, the familiar verb *plus* + may be applied to all of the items of a list by deriving from it the verb `+/` so that the expression `+/P` is equivalent to the previous expression for the sum of the items of `P`. As with the primitive verb # of the previous paragraph, we may define the verb

```
total=: +/
```

so that we may conveniently write `total P` which will have the value `106.40`.

The J verb `+/` may be considered analogous to the familiar sigma symbol Σ of conventional mathematical notation where Σx or Σx_i represents the sum of the items of the vector or list x. The sigma notation for a sum may be extended in conventional mathematical notation only to the product so that Πx or Πx_i represents the product of the items of the list x. In J, however, verbs defined for pairs of numbers may be simply extended to lists. We shall introduce two of these in our analysis of book prices.

The verb *lesser of* `<.` gives the smaller of two numbers, and, for example, `13.95 <. 19.50` is `13.95`, and by extension `<./` gives the smallest item of a list so that `<./P` is equal to `13.95`, the smallest book price. Similarly, the expression `>./`, where `>.` is the verb *larger of*, is equal to `29.50`, the price of the most expensive book. For convenience, we may define the verbs

```
min=: <./
```

and

```
max=: >./
```

so that `min P` is `13.95` and `max P` is `29.50`.

We may summarize the calculations for the book prices with the verb

```
summary=: num, total, min, max
```

so that

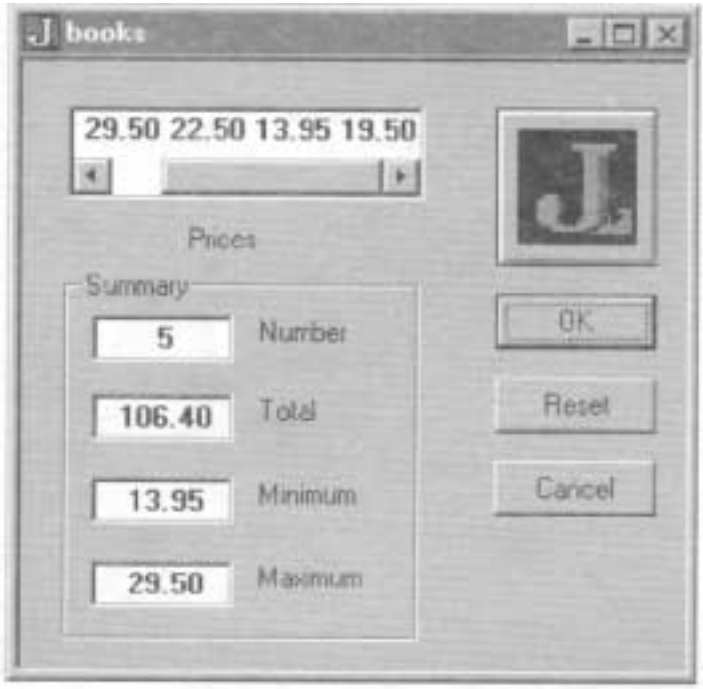
```
summary P
```

gives the four-item list

```
5 106.40 13.95 29.50
```

representing the number of books, total price, minimum price and maximum price, respectively.

The following Windows form written in J allows the calculations to be done without any knowledge of the J verbs developed above or indeed without knowing that J, rather than some other language, has been used. The user simply enters the prices in the Prices box and clicks the OK button.



Summing rows and columns

In this section we shall consider briefly a type of calculation that is fundamental to a large class of statistical applications. In a single dimension the calculation involves finding the sum of a list of numbers such as the book prices of the last section. In two dimensions it is simple to state, visualize and solve: find the row and column sums, the so-called marginal sums, in a table of data. It is the generalization to the various marginal sums in data arranged in an array of arbitrary dimension which is the difficult and important problem.

As an example we shall use some data taken from *Principles and Procedures of Statistics* by R. G. D. Steel and J. H. Torrie (McGraw-Hill, 1960) and even then we shall use only part of the data. The data are the yield in bushels per acre for each of two varieties of oats and each of three different seed treatments with four replications of each variety-treatment combination.

To begin, consider the four observations

```
62.3 58.5 44.6 50.3
```

for the first variety-treatment combination. There are the observations themselves and also their sum `215.7` which is a measure of the effectiveness of this particular combination.

Now consider the three treatments for the first variety which are given by the two-dimensional array:

```
62.3  58.5  44.6  50.3
63.4  50.4  45.0  46.7
64.5  46.1  62.6  50.3
```

with the rows representing treatments and the columns representing replications. There are now four different quantities to calculate: the observations themselves, the row sums

```
215.7 205.5 223.5
```

which give a measure of the effectiveness of each of the three treatments, the column sums

```
190.2 155 152.2 147.3
```

which measure the variability between the replications, and the overall sum `644.7` which gives a measure of the yield of the first variety of oats.

If we now consider the second variety of oats, we have the data arranged in the three-dimensional array

```
62.3  58.5  44.6  50.3
63.4  50.4  45.0  46.7
64.5  46.1  62.6  50.3

75.4  65.6  54.0  52.7
70.3  67.3  57.6  58.5
68.8  65.3  45.6  51.0
```

which has `two` levels each with `3` rows and `4` columns with the levels representing the varieties and the rows and columns representing treatments and replications, respectively. If we count the array itself and the total over all of the data, there will be `2 * 2 * 2` or `8` different marginal sums to compute. For example, the sum over the levels

```
137.7 124.1  98.6 103.0
133.7 117.7 102.6 105.2
133.3 111.4 108.2 101.3
```

measures the yields of varieties for both treatments and replications, and the sum over both levels and replications,

```
463.4 459.2 454.2  ,
```

measures the treatments.

If this experiment were repeated for two or more methods of cultivation, then the data would be represented as a four-dimensional array and there would be a total of `2 * 2 * 2 *-2` or `16` different sums that could be computed. The inclusion of a fifth factor, for example, the repetition of the experiment at a different location where the soil was different, would give `32` marginal sums.

Once the marginal sums, or at least an appropriate selection of them depending on the experimental design, have been computed, it is relatively simple—and we must emphasize the word "relatively"—to find the necessary components of the total variation to test whatever statistical hypotheses are of interest. The construction of appropriate verbs in J for the calculation of any or all marginal sums in arrays of arbitrary size and dimension is a problem to which a very considerable amount of attention has been devoted. The resulting verbs have formed the basis of statistical packages of considerable utility.

We shall make only a few remarks about the use of J to find marginal sums. The four observations for the first variety-treatment combination may be found very simply as

```
+/62.3 58.5 44.6 50.3
```

which is equal to `215.7`. If `V1` represents the three-by-four array given above for the three treatments and four replications for the first variety, then the treatment totals are the row marginal sums `+/"1 V1` or

```
215.7 205.5 223.5 ,
```

the replication totals are the column marginal sums `+/V1` or

```
190.2 155 152.2 147.3 ,
```

and the grand total is the sum over either of these margins which may be expressed as either `+/+/"1 V1` or `+/+/V1`. For arrays of more than two dimensions we shall say simply that any marginal may be found by appropriate rearrangement of the coordinate axes followed by application of the verb `+/` but we shall not bother the reader with the details.

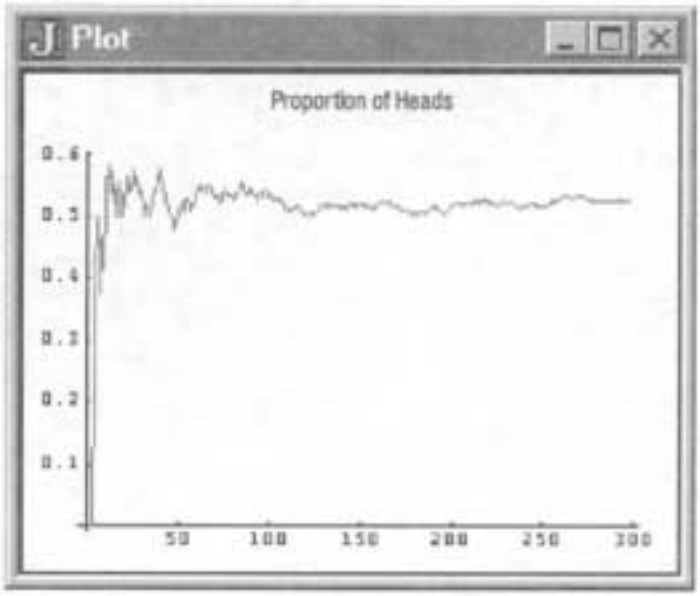
Tossing coins

The tossing of coins and the rolling of dice, and also the drawing balls from urns, have long been used to provide examples of statistical processes. Amongst the best-known examples in the statistical literature are the experiments of the English biologist W. F. R. Weldon (1860–1906) which he carried out to illustrate his statistical arguments. In one he tabulated the results of rolling twelve dice 4096 times counting as a success the occurrence of 4, 5 or 6. Another set of dice data which is apparently not as well known as Weldon's was generated by a Swiss scientist Rudolf Wolf who tabulated the results of 100 000 rolls of a die.

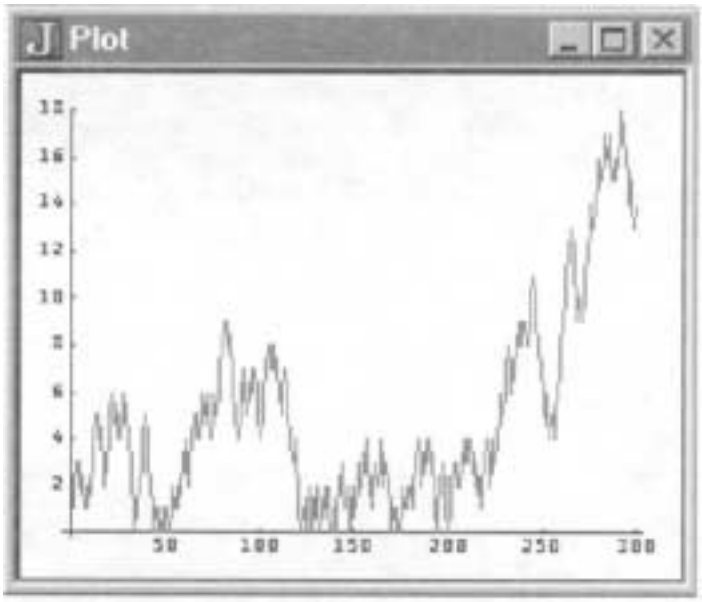
A more recent source of coin-tossing and dice-throwing examples is Warren Weaver's *Lady Luck: The Theory of Probability* (Doubleday, 1963). The author who was in his late 60s when the book was published had had a varied and distinguished career first as a mathematician in academia and later as a scientific administrator in a number of organizations. (Amongst his many publications is *Alice in Many Tongues* (The University of Wisconsin Press, 1964), a delightful discussion of the life of Lewis Carroll, the writing of the Alice books, and their translation into over forty languages.) *Lady Luck* gives an elementary and very readable introduction to the origins and development of probability theory and the statistics of chance. Weaver does not avoid mathematical notation but introduces and uses it in a manner which enhances rather than detracts from the exposition. *Lady Luck* is a gem of a book which I have used with great pleasure ever since I acquired my first copy. It is still in print, and deserves to remain so for a long, long time.

We might note that Weaver performed his random experiments, not from the actual tossing of coins and the rolling of dice, but from simulations performed using tables of random numbers, a common feature of statistical tables at the time he was writing. For example, a sequence of digits selected arbitrarily from the table could represent a sequence of coin tosses if an even digit were considered a head and an odd digit a tail. He mentions the RAND Corporation's *A Million Random Digits with 100 000 Normal Deviates* published in 1955 which was reviewed, I believe, with some disbelief in *The New York Times*.

In this section we shall discuss only the occurrence of the number of heads in the repeated tossing of an unbiased coin and discuss its simulation using J. It is well-known that the ratio of the number of heads to the total number of tosses approaches 0.5 as the number of tosses increases. The following graph shows the results of one simulation of 300 tosses:



However, Weaver points out that the difference between the numbers of heads and tails tends to grow as the number of tosses increases. This is shown in the second graph representing the same simulation as that shown in the first one:



Thus if one were to bet on the outcome of heads on each toss it is not altogether certain that one would necessarily break even in a long series of tosses since one's capital, if modest, could be wiped out by a long run of tails. This is especially true if one were using a martingale system of betting where one doubles one's bet on each loss and reverts only to the original bet on a win.

The above results may be taken as a warning, if any warning is necessary, against even the most innocent forms of gambling, except possibly as an occasional entertainment undertaken with a knowledge of the odds involved and with a firm resolve to keep one's expenditures within reasonable limits. However, a discussion of the use of J in the above simulation can give a further

understanding of the J language, and we give it here for the interested reader.

Random numbers may be generated very simply in J with the verb *roll* `?`. For example, the expression `?2` gives a random digit `0` or `1` which could be considered to represent the results of one toss of an unbiased coin where `0` represents tails and `1` heads. As another example, `?6` gives an integer picked at random from the first six non-negative integers, and could be used to simulate the rolling of a die with the faces numbered `0, 1, 2 , ..., 5`. If one prefers the more conventional numbering of faces, one could introduce the verb `>:` *increment* which adds `1` to its argument, and use the expression `>:?6` to simulate one roll of a die.

The expression

```
? 2 2 2 2 2 2 2 2 2 2 ,
```

which could have the value

```
0 1 0 1 0 0 1 1 1 0 ,
```

could represent `10` tosses of an unbiased coin with a head occurring on the second, fourth, seventh, eighth and ninth tosses. Therefore, the expression

```
+/? 2 2 2 2 2 2 2 2 2 2
```

`which gives the sum +/` over the expression to the right gives the number of heads occurring in `10` tosses of an unbiased coin. Finally, as a convenience we introduce the verb *shape* `$` to give a list of arbitrary length all of whose items are equal. Therefore, the expression `10$2` is the list of ten `2`s given at the beginning of the paragraph, and the expression `+/?10$2` gives the number of heads in `10` tosses.

We may now return to the coin-tossing example introduced earlier in the section. Let the number of tosses be `N` so that `N` has the value `300` in the above simulation. For convenience in this discussion we shall give `N` the value of `10` by the expression

```
N=: 10 .
```

Therefore, `10` coin tosses are given by `?N$2` which could have the value

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

`so that the cumulative number of heads is`

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

In J this is given conveniently by the expression

```
Heads=: +/\?N$2 ,
```

where the verb `+/\` gives the cumulative sum rather than the final sum given by `+/`.

To give the toss number we introduce the verb *integer* `i.` which gives a list of non-negative integers, and, for example, `i. 5` is the list `0 1 2 3 4`. Thus the toss number is given by

```
TossNum=: >: i. N,
```

where `>:` is the verb *increment* which has already been introduced, and has the value

```
1 2 3 4 5 6 7 8 9 10 .
```

The ratio of the number of heads to the number of tosses is

```
Ratio=: Heads % TossNum
```

whose rounded value is

```
0 0.5 0.333 0.5 0.4 0.333 0.429 0.5
0.556 0.5 .
```

The excess in heads over tails, or vice versa, on any toss is simply the number of that toss minus the difference between the number of heads and the number of tails obtained so far. A little thought will show that this is given by the expression

```
TossNum-2*Heads
```

`which has the value`

```
1 0 1 0 1 2 1 0 _1 0 ,
```

where `*` is the verb *times*. Finally, we introduce the verb *magnitude* `|` to make the differences non-negative, and we have the expression

```
Diff=: |TossNum-2*Heads
```

which has the value

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

Christmas cards

For some years now I have made my own Christmas cards and most of the other cards I use during the year. There are several reasons why I do this: I enjoy making them; I can send my friends cards which are different; and I save money. All of these cards have themes relating to some non-professional topic, and contain a picture, usually my cat or a Japanese scene, taken by me or by a friend—all of my cards, that is, except one. I would like to describe this card briefly now.

To begin with we recall from high-school mathematics the factorial function which gives the product of the successive positive integers. For example, *factorial 3* is 6 which is the product of the integers 1, 2 and 3, and *factorial 5* is 120 which is the product of the first five positive integers. Another example is *factorial 52*, the number of arrangements of the 52 cards in a deck, which is equal to approximately 8 followed by 67 zeros, a very large number indeed. (I had this statement in an earlier paper except that I gave the value as 8.1 followed by 67 zeros. One reader pointed out quite correctly that 8.1 followed by any number of zeros was still 8.1 and was not a large number at all!)

In J the factorial function is given by the *factorial* verb `!`, and, for example, `!3` is `6`, `!5` is `120`, and `!i. 9` is

```
! 1 2 3 4 5 6 7 8
```

which has the value

```
1 2 6 24 120 720 5040 40320 .
```

The factorial verb can easily produce very large numbers, and, for example, `!10` is `3.6288e6`, `!20` is `2.4329e18`, and `!52` is `8.06582e67` as we have seen. The factorial function may be computed exactly in J by suffixing the argument with `x`, and `!20x` is

```
2432902008176640000 ,
```

and `!52x` is

```
80658175170943878571660636856403766975289505440883277824000000000000 ,
```

an integer with `68` digits.

Some years ago Martin Gardner, that indefatigable writer of fascinating articles and books on a variety of mathematical and scientific topics, published the article "Factorial Oddities" in *Scientific American* which was later republished in his *Mathe-*

matical Magic Show (Vintage Books, 1978). In this article he introduced tree factorials as those factorials whose digits may be arranged in a triangle with one digit in the first row, three in the second, five in the third, etc. and could with a little imagination be considered as Christmas trees.

The factorials of the first three positive integers are trivially tree factorials of a single digit. The first non-trivial tree factorial is `!7` or `5040` which may be displayed as

```
 5
040 .
```

Gardner's article gives the twenty tree factorials between seven and one thousand, and displays in triangular form the factorials for 105 which has 169 digits and for 508 which has 1156 digits. The second last of these factorials may be computed exactly in J by the expression `!105x` which, with a little reaarangement the details of which we shall omit, may be displayed as

```
                    1
                   081
                  39675
                 8240290
                900504101
               30580032964
              9720646107774
             902579144176636
            57322653190990515
           3326984536526808240
          339776398934872029657
        9938729078134368160972 8
      000000000000000000000000
```

J so far (and a little more)

In this section we shall give examples of the J verbs we have introduced so far together with a few others. The following represents a dialogue with the computer where the J expressions entered by the user are indented automatically three spaces, and the responses by the computer begin at the left margin. The comments which follow the expressions and which begin with `NB.` are for the reader and are ignored during evaluation.

```
   3 + 5                NB. Plus
8
   2 * 3                NB. Times
6
   3 - 5                NB. Minus
_2
   15 % 6               NB. Divided by
2.5
   2 + 3 * 4            NB. Precedence
14
   2 * 3 + 4
14
   (2 * 3) + 4
10

   4 + 2 * 3
10
   % 8                  NB. Reciprocal
0.125
   *: 2.5               NB. Square
6.25
   %: 125               NB. Square root
11.1803
   6.5 <. 3             NB. Lesser of
3
   4 >. 10              NB. Larger of
10
   <: 8                 NB. Decrement
7
   >: 3.14              NB. Increment
4.14
   2.3 + 5 + 3.5 + 6    NB. Sum
16.8
   +/ 2.3 5 3.5 6
16.8
   +/\2.3 5 3.5 6       NB. Cum. sum
2.3 7.3 10.8 16.8
   w=: 2.3 5 3.5 6
   #w                   NB. Tally
4
   +/w
16.8
   +/\w
2.3 7.3 10.8 16.8
   <./w                 NB. Minimum
2.3
   >./w                 NB. Maximum
6
   10$6                 NB. Shape
6 6 6 6 6 6 6 6 6 6
   ?10$6                NB. Roll
0 4 2 3 1 0 4 4 5 2
   >:?10$6
4 5 1 1 4 5 1 3 1 3
   |_3.5 0 4            NB. Magnitude
3.5 0 4
   i. 10                NB. Integer
0 1 2 3 4 5 6 7 8 9
   a=: i. 3 4
   a
0 1  2  3
4 5  6  7
8 9 10 11
   +/a                  NB. Column sums
12 15 18 21
   +/"1 a               NB. Row sums
6 22 38
   !6                   NB. Factorial
720
   */1 2 3 4 5 6
720
   !25
1.55112e25
   !25x
155112100433309859840000 00
```

Additional topics

Teaching languages

Shortly after I retired I started to study Japanese. There are several reasons but the main one was probably my interest in languages. I'm not sure what I expected to learn, but now I can read and write a little, speak very little, and understand spoken Japanese almost not at all. What I didn't expect, though, was to become interested in the teaching of Japanese. My enjoyment in seeing how the language is presented helps compensate for my rather spectacular lack of progress in it.

Most of my Japanese texts teach the language by the telling of some continuing story which although fictional is intended to be

realistic. Let me mention very briefly one of my favourite books. It is *Business Japanese* by Michael Jenkins and Lynne Strugnell (NTC Publishing Group, 1993) and is in the well-known English "Teach Yourself Books" series. The story revolves around Wajima Trading Company in Tokyo and the British company Dando Sports which wants to market its sporting equipment and clothing in Japan through Wajima. We are introduced to various members of the staff at Wajima and learn about the company's organization and how business operates in Japan. In one of the later chapters we have a lecture on quality control. One of the main characters is Mr. Lloyd, marketing manager for Dando, who visits Japan on two occasions to draw up a contract. We follow Mr. Lloyd as he works with the company and meets some of the staff socially.

The Japanese hiragana and katakana syllabics are introduced in Chapter 1 and the kanji (Chinese) characters are introduced a few at a time very shortly thereafter. The three classes of characters soon take precendence over the romaji (Roman) characters used for transliteration. There are twenty chapters and the story begins in Chapter 2 with an assistant manager of Wajima checking out of a hotel before returning to Tokyo. Each chapter has the same format: a summary of the story so far (in Japanese beginning in Chapter 12), another installment of the story, in both Japanese characters and in romaji; new vocabulary; grammatical notes; exercises; a short reading exercise; and a one-page essay in English on some aspect of Japanese business. There are several Appendices with grammar summaries, and English-Japanese and Japanese-English glossaries. From the very beginning I had the feeling of meeting real Japanese people working in Japan and living as Japanese people live.

Contrast this introduction to Japanese to the introduction to a programming language in most programming texts and adopted in most introductory courses. (There are a few texts which are exceptions but they do not appear to be very popular.) The language does not seem to matter. It can be BASIC, Pascal, C/C++ or Java. The texts and courses are really introductions to syntax with numerous examples and exercises intended to illustrate and reinforce grammatical principles. (A colleague once remarked to me that most introductory programming courses were as interesting as a course in the conjugation of verbs.) Furthermore, many of the exercises are artificial and even juvenile. For example, the first example in Chapter 4 of one of the Java texts was a program to print either "ho-ho", "he-he" or "ha-ha"; it was then modified to print "yuk-yuk". As bad as is the pedagogy, the writing is even worse in some of the books. Good scientific and technical writing does exist, of course, but little is to be found in computing texts.

Of course, expositions of array languages can be poorly organized and presented too. However, one of the advantages of these languages is that they may be used almost immediately to do something useful without first introducing the amount of detail required with conventional languages. (Compare for example, the BASIC and J programs for the book-buying example given earlier in this paper.) Put another way, array languages may be easily used in the exposition of some subject—statistics, logic, some branch of arithmetic or algebra, say—without having details of the language intrude on the subject matter. Much of my later work has been directed toward this end.

My attitudes towards the teaching and use of programming languages have been influenced by Kenneth Iverson who has been giving his views in lectures, technical reports and books for almost forty years. Writing on the 25th anniversary of APL in 1991 in a paper giving one of the first published accounts of J he wrote "...Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects." Implicit in this statement is the conviction that the details of a language, whether it be J or Japanese, should be introduced as needed in the exposition of the subject whether it be teaching multiplication tables in a Canadian classroom or introducing the quality control methods of the American W. Edwards Deming to Japanese assembly lines.

Remembering the past

When I was an undergraduate I took a required course in the history of mathematics. I enjoyed the course but I have the feeling now that I probably wished then that I had been spending my time on something more practical such as another course in calculus or one in actuarial mathematics. However looking back now I realize that this course was one of the most important courses I took because it awakened my interest in the history of science, an interest which has never left me but has only increased over the years.

Unfortunately now there appears to be little opportunity for students to become acquainted with the history of their discipline. A few professors have an appreciation of the development of their subject and are able to impart this understanding to their students. However the historical development of a scientific discipline is not considered to be a marketable skill and has been displaced, if it were ever in the curriculum to begin with, by topics created by the many interesting and exciting developments in modern technology. In this section I can only mention a few of the historical references which I have enjoyed and indicate some of the newer sources of historical material.

The first book on computers that I bought was *Faster than Thought* which was edited by B. V. Bowden (Pitman, 1953), and was subtitled "A Symposium on Digital Computing Machines." It is a collection of twenty-four papers written by persons who were working in the new field of digital computation, some of whom are now considered to be amongst the great pioneers of computing. The book was reprinted seven times in the first fifteen years after its publication and still makes enjoyable reading. The editor contributed a Preface and four chapters, the most noteworthy in my opinion being the first, "A Brief History of Computing", which may be read for the pleasure of its literary

style alone.

A little book which I enjoyed reading and which I used in my teaching and research was *Electronic Computers* by S. H. Hollingdale and G. C. Tootill (Penguin, 1970). It was published first in 1965 and revised in 1970 and 1975. This book contains an excellent account of the history of computing, a discussion of the design of both analogue and digital computers, a treatment of computer programming, and finally a discussion of various applications of computers. Although very dated now, this book gives an excellent picture of computers and their use in the 1960s and early 1970s. The two chapters on the history of computing still make an excellent but brief intoduction to the subject. It is a pity that a modern version of this admirable little book is not available today.

We might note that Professor Hollingdale published *Makers of Mathematics* (Penguin, 1989) at the age of 79. In the Preface he remarks that he felt no need to include scholarly footnotes and that the references were "limited, with a few exceptions, to sources from my own library which I consulted while writing this book." A more pleasant way to spend part of one's retirement is difficult to imagine!

A more scholarly but very readable account of the history of computing is *A History of Computing Technology* by Michael Williams of the University of Calgary (Prentice-Hall, 1985; Second Edition, 1997) which describes the development of arithmetic and calculation tools from ancient Egypt to the IBM/360. This is an excellent introduction to the subject for the more serious reader.

Two books have recently come to my attention which provide an interesting contrast to the books just discussed. They are *Frontiers of Complexity* by Peter Covey and Roger Highfield (Fawcett Columbine, 1995) and *Darwin Among the Machines* by George B. Dyson (Perseus Books, 1997). Both books are written in an engaging style, take a strong historical approach to their subject, and have many references. The first is an introduction to the relatively new subject of complex systems and its applications in mathematics, physics, chemistry, biology and the social sciences; the second gives the author's idiosyncratic view of the evolution of computers. Indeed, Covey and Highfield's book with its over sixty pages of endnotes, ten-page glossary, and eleven pages of references may be recommended as a superb introduction to computers for the general reader.

Even this very brief discussion of a few of my favourite computing books would be incomplete without some mention of *Alan Turing, The Enigma of Intelligence* by Andrew Hodges, which was originally published by Burnett Books in 1983 and has been published since then in at least two paperback editions. Apparently a revised edition has just been published. One reviewer described it as "one of the finest pieces of scholarship to appear in the history of computing." This long biography—almost 600 pages in the original edition—gives an incisive account of the life and work of one the pioneers of computing who was the originator of the eponymous Turing machine, a theoretical device for studying the limits of computability. It also sheds light on the English class and education systems, code-breaking during the Second World War in which Turing played a decisive role, and events leading to the reform of the law regarding homosexuality in Britain. Incidentally, Turing figures briefly in *Enigma* by Robert Harris (Hutchinson, 1995), a fictional account of code-breaking which as well as being a most satisfying thriller gives references which were not available when Hodges's book first appeared.

About the only published accounts of array languages, apart from conference proceedings, are in the issue of the *IBM Systems Journal* (Vol. 30, No. 4, 1991) which marked the twenty-fifth anniversary of APL. Of particular note are Donald McIntyre's encyclopaedic "Language As an Intellectual Tool: From Hieroglyphics to APL" which reads almost as a hymn to Kenneth Iverson and his work with APL, and what was then the new dialect J, and Iverson's "A Personal View of APL" which gives one of the first published accounts of the evolution of APL into J. Unfortunately there appears to be nothing published on the development of Nial and its relation to APL and J.

Persons wishing to learn about the history of computing may consult the World Wide Web where there is an overwhelming amount of material available. Here we shall mention only three sites which because of the many links to other sites will provide a wealth of information. The Virtual Museum of Computing at

www.museums.reading.ac.uk/vmoc/

includes an "eclectic collection of World Wide Web (WWW) hyperlinks connected with the history of computing and on-line computer-based exhibits available both locally and around the world." The Alan Turing Home Page at

www.turing.org.uk/turing/

is maintained by Alan Hodges and was recently judged "one of the world's top 100 websites." A very interesting source of information on vintage and modern calculators is given at

www.dotpoint.com/xnumber/.

Conclusion

The question of the best programming language is similar to the question of what is the best natural language. English? or Japanese? or German? or ...? The answer depends on who we are, where we and our families have come from, what we are now doing, and for whom we are doing it. Regardless of how many languages we know, there will always be one or two which we find most useful and with which we are the most comfortable.

I feel the same way about programming languages. I have used many different ones and have liked them all. However, it is the array languages—first APL, then Nial and now J— that I have found the most appealing intellectually, the most pleasant to use, and the best able to satisfy most of my computational needs.

All I have tried to accomplish in this paper is to present a personal view of the development of programming languages and to show my enthusiasm for the three which I have chosen to call array languages and to emphasize the importance of keeping the historical development of one's speciality clearly in mind.

As this is my third attempt to explain what I have done with my professional life, the following quotation, which I have used before, from one of my favourite fictional characters, takes on an added significance:

> *"So there it is", said Pooh, when he had sung this to himself three times. "It's come different from what I thought it would, but it's come. Now I must go and sing it to Piglet."* ■

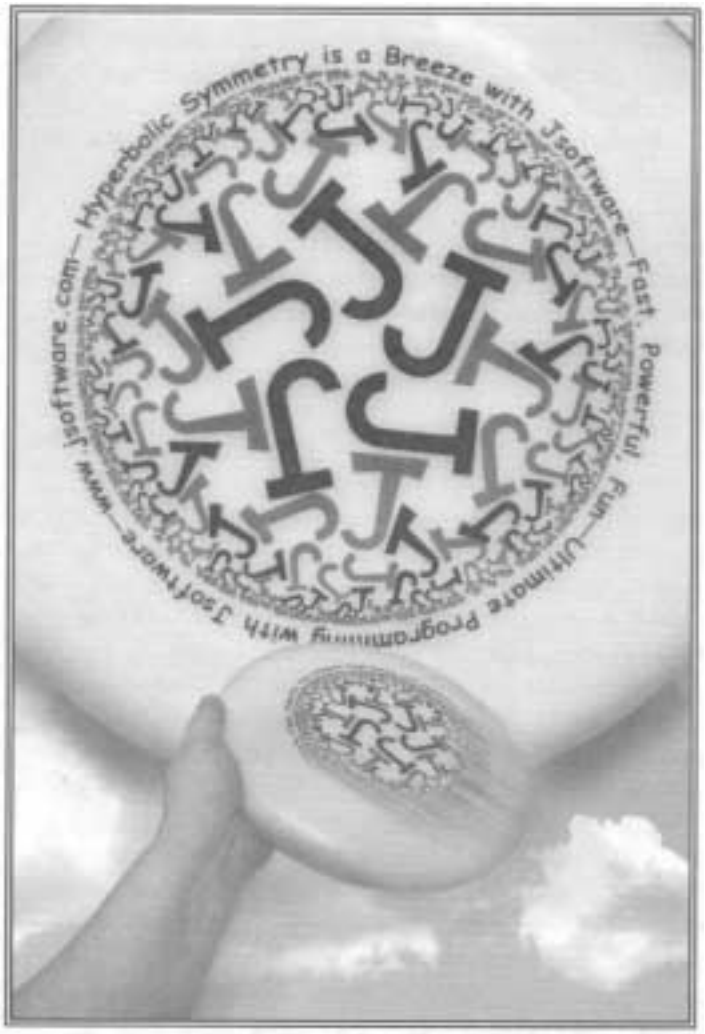
Acknowledgements

I wish to thank the following persons for their very helpful comments on earlier versions of this paper: Kenneth Iverson, Howard Peelle, Clifford Reiter, and Alison Smillie.

***Keith Smillie** is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2H1. His e-mail address is "smillie@cs.ualberta.ca".*

Non-Sequitur

The J Frisbee —*Courtesy of* ***Cliff Reiter***



APL In the New Millennium

—by ***Kenneth E. Iverson***
Toronto, Ontario

SINCE IBM's APL\360 BECAME AVAILABLE IN 1966, many dialects have been developed, and competition has led to emphasis on their differences, an emphasis reflected in their distinctive names: APL\1130, APL\360, APLSV, APL2, SHARP APL, Nial, Dyalog APL, A, APL2000, J, K, and others.

Although natural to healthy competition, the emphasis on differences has discouraged the sharing of ideas, and still tends to blind programmers to the ease of moving between dialects, an ease not shared by programmers unschooled in the core ideas of APL.

As emphasized in [1], these core ideas were:

- The adoption from Tensor Analysis of a systematic treatment of arrays, in which every entity is an array, and different *ranks* lead to *scalars*, *vectors* (or *lists*), *matrices* (or *tables*), and *higher-dimensional arrays* (or *reports*).
- Operators (in the sense introduced by Heaviside [2]), which apply to functions to produce related functions.

In this paper I will review developments in the APL dialects, emphasizing similarities and the ways in which competing ideas have been, and could be, shared and adapted to competing systems. My hope is to encourage the relatively small APL family to mute their differences, and present a more united face to the programming world.

Alphabets

Although the particular alphabet, or even the font used, is a most superficial aspect of a language, it can make a dramatic assault on a beginning reader—as anyone who first met German in the Gothic font can testify. First encounters with the unfamiliar alphabet of the earliest APL has certainly discouraged many, in spite of its highly-mnemonic character.

At the time of its design there was no adopted standard, and it seemed reasonable to exploit the newly available IBM Selectric typewriter (with its easily-changed *typeball*) to design our own alphabet, and to use the backspace ability of the typewriter to produce composite (*overstruck*) characters.

The APL community was too small to influence the design of the now widely-used ASCII alphabet, and our use of special characters led to a series of unforeseen difficulties that have significantly inhibited the use of APL:

- When the "glass terminal" provided by the cathode ray tube supplanted the typewriter, it was incapable of backspacing to provide the composite characters of APL.
- APL characters were not provided by early printers, and there was a considerable delay before specialized alphabets could be downloaded to them.