*October 2005*

# My Life with Array Languages

**Keith Smillie**

## 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*. Another physicist and Nobel laureate, Ernest Rutherford, expressed the same belief somewhat more prosaically in his remark that "[Y]ou have not understood something unless you can explain it in terms that can be understood by an English barmaid."

Such remarks may be even more relevant today when universities emphasize the acquisition of new knowledge and its dissemination in scholarly publications. There appears to be little place for courses which consider "science as a humanity" to be studied for its own sake as part of a liberal education. Furthermore many of the introductory courses for students not specializing in the subject are practical courses presenting material which is intended to be immediately useful in the marketplace.

Also there is another reason why we should take the time to describe in non-technical language what we are doing. 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.

It is with these thoughts in mind that I shall 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 requiring extensive use of mathematical notation and 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 non-technical account of what I have been doing.

We shall begin with a look at some of the programming languages I have used and make the distinction between what I term "conventional" languages and "array" languages. We shall then have a look at the array languages I have used for most of my programming work and discuss a few applications which I have found interesting or useful. Then we compare the teaching of natural languages and the teaching of programming languages, and finally discuss the importance of keeping a historical prospective on one's work.

## Programming languages

The first computer I used was the National Cash Register 102A which, as were almost 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 102A 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 or base-2, the arithmetic performed in binary, and the 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. (The 102D, a later version of the 102A, 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 but still expressed as a list of numerical instructions which would be interpreted one instruction 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.

In the remainder of this section we shall give for the interested reader a very simple example of machine-language programming for a hypothetical computer. It may be skipped if desired since no subsequent material depends upon it.

A program for our hypothetical computer consists of a sequence of five-digit integer numbers with the first two digits specifying the operation to be performed and the remaining three digits the address of the number to be operated on. The memory consists of `1000` locations or “words” numbered sequentially `000`, `001`, `002`, ..., `999`, and an additional word of memory, termed the “Accumulator”, where arithmetic, i.e., addition, subtraction, multiplication and division, is performed. For example, the instruction `02025` means subtract the number in location `025` from the number in the Accumulator and store the result in the Accumulator, and the instruction `11000` means read a number from (hypothetical) paper tape and store it in the Accumulator. A program consists of a sequence of these numerical instructions stored sequentially in memory. It is executed starting with the first instruction, and execution proceeds sequentially

instruction-by-instruction unless the sequence is changed by a transfer-of-control instruction. Execution stops when a Halt instruction is encountered.

The following is a program for reading a list of non-negative numbers one at a time from paper tape and calculating and printing the maximum number:

```
05112 06113 11000 09110 06114 02113 08102 05114 06113 07102 05113 12000 00000
```

The program is designed to be stored in memory locations `100`, `101`, ..., `112`. Execution of the program begins with the instruction `05112` in location `100`.

Since a program in this form is difficult if not impossible to read, programs are written and annotated as shown below with the location of an instruction being given to its left and an explanatory note to its right:

```
100   05112    ) Set maximum no. Xmax to 0
101   06113    )
102   11000    Read no. X from tape and put in Accum.
103   09110    Is X = 0?
104   06114    Store X
105   02113    X - Xmax
106   08102    Is X – Xmax < 0?
107   05114    ) Replace X with Xmax
108   06113    )
109   07102    Next instruction from locn. 102
110   05113    Get X and place in Accum.
111   12000    Print Xmax
112   00000    Halt
113       0    Storage for Xmax
114       0    Storage for X
```

We note that the list of non-negative numbers is appended with a zero which serves as a flag to the program that all valid numbers have been processed.

Programs for real problems on real computers often consisted of hundreds or thousands of lines of such code, and took weeks or even months to write and to check, i.e., “debug”, on the computer. However, such machine-language programming or some simplified version which still was still numeric in form, was all that was widely available until Fortran became available in the late 1950s.

## Conventional programming

A breakthrough in programming came in the late 1950s with the development by the International Business Machines Corporation of Fortran, for Formula Translation, 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 many versions of Fortran were developed and the language 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.

In 1964 BASIC, for Beginner's All Purpose Instruction Code, was first released at Dartmouth College, a liberal arts college in New Hampshire. It was designed as a "pleasant and friendly" alternative to Fortran for undergraduate students, most of whom were in the social sciences and the humanities. The first BASIC program was run successfully at 4:00 a.m. on May 1, 1964, and performed the calculation $(7 \times 8) \div 3$. Since then there have been very many versions of BASIC and the language has become a lingua franca in computing.

Fashions in computing languages used in introductory courses change every few years. 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. BASIC was used in the 1980s in courses for students in Arts and Education. Pascal has now been replaced with C++ and Java in most introductory courses.

Although there are important differences between these languages, 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. This may be illustrated by the following short BASIC program for finding the maximum of a list of positive numbers:

```
REM Maximum of a list of numbers
DATA 19.11, 12.77, 21.31, 16.10, 12.19, 25.76, 17.49, 0
MaxTotal = 0
READ Total
WHILE Total > 0
   IF Total > MaxTotal THEN
      MaxTotal = Total
   END IF
READ Total
WEND
PRINT MaxTotal
STOP
END
```

The data shown in the program represent the total amount spent on groceries on each of seven trips to the supermarket. The procedure for finding the maximum is to examine each total in turn, and if it is larger than the current maximum it becomes the new maximum. This procedure is initialized by setting the current maximum to zero. As with the previous machine-language program a zero is appended to the list of shopping totals to act as a flag to the program that all valid prices have been handled.

### Selecting apples

Before discussing array languages we shall give a simple analogy which may help illuminate a fundamental difference between these languages and the conventional languages we have discussed in the previous section. 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 good apples have been selected. 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 until we have selected and set aside all of the good apples.

On the other hand the instructions corresponding to an array language could be stated simply as "Select all of the good apples from the box." Of course, the apples would still have to be examined individually, but the apple-by-apple details could be left to the helper.

Conventional programming languages may be considered then "one-apple-at-a-time" languages with machine languages being a very primitive form, whereas array languages may be considered "all-the-apples-at-once" languages. In two of the following three sections we shall consider the array languages APL and its "modern dialect" **J** with an intervening section giving a short discussion of the array language Nial which was influenced by APL.

### Array languages – APL

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 to develop his ideas while teaching in the newly established program in Automatic Data Processing at Harvard. Its first applications were 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. 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 have been simplicity, brevity 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 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.

As a very simple example suppose we have a list of unit prices and a list of quantities of each item purchased. A single multiplication of the list of prices by the list of quantities will give a list of the total amount spent for each item, and one summation of this list will give the total amount spent. These calculations may be expressed in APL as follows:

```
   PRICE ← 0.40 4.25 8.99 1.99 0.40 2.69
   QTY ← 3 2 1 2 1 1
   PRICE × QTY
1.2 8.5 8.99 3.98 0.4 2.69
   +/PRICE × QTY
25.76
   TOTAL ← +/PRICE × QTY
   TOTAL
25.76
```

As APL is an interactive language, these expressions are immediately executed and the results displayed when entered at the keyboard. Expressions entered by the user are indented while responses by the APL system are not.

Some of us in the Department were very early users of APL, beginning with the first experimental batch system and then with the APL\360 System at Yorktown Heights accessed by long-distance telephone. In 1968 we had APL running on the University's computer and accessed by one terminal in the hallway outside the computer room.

I used APL both for the development of statistical algorithms and for classroom teaching. I wrote a statistical package called STATPACK2 which gained considerable popularity and was widely used throughout the APL community. In those days before the Internet I distributed the programs on a 5¼-inch floppy disk and the documentation in a technical report with red covers and a black tape binding. The last work I did with STATPACK2 was to produce a version for the IBM Personal Computer in the mid 1980s. For some years I taught APL, along with Fortran and other conventional languages, in an introductory programming course for agriculture and science students. At first I had access to APL in the classroom through a terminal connected remotely to the University's computer which fortunately was in the same building as the classroom. From the mid 1970s until the early 1980s we had an IBM 5100 computer, the predecessor of the first IBM PC designated the IBM 5150, which we moved between lecture room and office on a small cart. I suppose we were doing pioneering work, but at the time it just seemed the natural thing to do.

Finally in this section we shall make a few more remarks about APL making use of the list of grocery prices introduced earlier in the BASIC example. Apart from the list of grocery prices the following material will not be referred to in later sections and thus may be skipped if desired.

The APL function +/ used in the simple APL example above may be considered analogous to the familiar sigma symbol Σ of conventional mathematical notation. Just as $\Sigma x$ or $\Sigma x_i$ represents the sum of the elements of the list $x$, so +/X represents in APL the sum of the items of the list X. The only extension of this concept in conventional notation is the symbol Π where $\Pi x$ represents the product of the items of $x$. However, in APL the operation represented by the symbol / may be extended to a large number of arithmetic and logical functions. For example, ×/X represents the product of the items of X, ⌈/X the largest item of X and ⌊/X the smallest item of X. Furthermore these operations are defined in a meaningful way for two-way tables and arrays of higher dimension.

The following APL examples will illustrate these ideas using the grocery prices:

```
   TOTALS ← 19.11 12.77 21.31 16.1 12.19 25.76 17.49
   +/TOTALS
124.73
   ⌊/TOTALS
12.19
   ⌈/TOTALS
25.76
```

As we may wish to know the number of times we were shopping for groceries, we introduce the function ρ giving the number of items in a list:

```
   ρTOTALS
7
```

We shall now introduce the function SUMMARY, the details of which need not concern us, which will allow us to perform all of these operations very simply, and we have that

```
   SUMMARY TOTALS
```

is the list

```
   7 12.19 25.76 124 .
```

Finally we may note that expression

```
   ⌈/19.11 12.77 21.31 16.1 12.19 25.76 17.49
```

is equivalent to the BASIC program of an earlier section.

### Array languages – Nial

Nial, Nested Interactive Array Language, combined concepts from APL and other languages within the framework of a mathematical model called array theory. It was developed during the 1970s and 1980s by Trenchard More of the IBM Cambridge Scientific Center in Cambridge, Massachusetts. The

name comes from the Old Norse Icelandic name *Njal*, a fact that was not overlooked in the promotion of the language, and most of the manuals had a sketch of a Viking ship on the cover. Nial, unlike APL, used the standard ASCII character set, and operations were given short meaningful names although the symbols +, -, * and / could also be used for the familiar arithmetic operations of addition,, subtraction, multiplication and division, respectively. Operations in Nial were defined over arrays of quite arbitrary dimension, structure and content.

I first heard of Nial when I met Trenchard at an APL conference in the late 1970s, and can still remember introducing myself to him as we were assembling for lunch one day. I was soon attracted to Nial both by the force of Trenchard's personality and by the beauty of the concepts in the language and their exposition in a series of carefully written reports. My only contribution to Nial was the preparation of *Nial Notes* which I described at the time as a "package of defined operations for statistical and selected other mathematical operations".

Now I look back with great pleasure on my associations with Trenchard and my trips to the Cambridge Scientific Center. I was working with Nial during the period when many of us in academia were struggling with the seemingly unsatiable demand for courses in "computer literacy", whatever that term may have meant. My work with Nial and the people associated with it added some badly needed intellectual respectability, and even sanity, to my professional life.

Nial had a small group of enthusiastic supporters in the 1980s and early 1990s, but now appears to be little used - if it is used at all. Indeed there may be some danger that all of the work that was done in the design, implementation and use of the language will go unrecorded and its considerable literature will be lost.

### Array languages – J

In 1980 Ken Iverson left IBM and returned to Canada to work for I. P. Sharp Associates, a firm wth 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 he retired from I. P. Sharp, or to use his words "When I retired from paid employment...", and turned his attention to the development and promotion of a modern dialect of APL called simply **J**.

Ken's 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 has undergone continuous development since then. It 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.

The most 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. As an illustration the following are the **J** equivalents of the three APL statements given earlier in the shopping example:


```
Price=: 0.40 4.25 8.99 1.99 0.40 2.69
Qty=: 3 2 1 2 1 1
Total=: +/Price * Qty
```

Although there are many differences between APL and **J** that make **J** a simpler and more satisfying language to use, we shall mention only one, and this is the use of terminology from natural language rather than from computing technology. For example, what are termed functions, and sometimes operators, in most languages are called verbs in **J**, and what are variables in almost all languages are termed nouns in **J**. (There are even gerunds in **J**!) This simple change gives a unity to the various elements in the structure of **J** and also suggests that there may be an affinity between programming languages and natural languages. We shall return to this topic in a later section.

As an example of the use of **J** we give here a Windows form for the grocery shopping example that allows the calculations to be done without any knowledge of the language. One simply enters the grocery totals in the window at the top of the form, and then clicks the OK button. The results appear in the Summary windows.

I first heard of **J** in 1991, a year before I retired. I was immediately attracted to the language and began experimenting with it in a number of different areas of application. Over the next dozen years or so I wrote almost two dozen papers which were published as technical reports, conference papers and journal papers. Many were made available on the Web. While most of the papers related to statistical applications and included the documentation of a **J** Statistical Package, other areas of application included machine simulation, calendar calculations, and the construction of weaving designs.

### An assessment of APL and J

After APL was released outside of IBM, it soon gained many enthusiastic users in business, industry and academia and was used for a wide range of applications. Soon there were several slightly differing versions of APL originating from different organizations, and successive releases contained both enhancements to the language resulting from experience with its use and also improvements which simplified its use on the computer. **J** had an enthusiastic reception both from users of APL and from new

users. However the design and implementation of **J** has been strictly controlled by Jsoftware Inc., a company of which Ken Iverson was one of the original founders.

Both APL and **J** have had, and continue to have, a large number of critics. Many objected to the unusual character set in APL and would refer derisively to APL as "that language with all the funny symbols". Many persons, with some justification, criticized the tendency of many APL users to write programs in as few statements as possible, the ideal being the "one-liner". Some critics, again with justification, termed APL a "write-only" language implying that APL programs could not be read intelligibly even by those who had written them. Similar criticisms could be, and undoubtedly have been, made about **J** and its supporters, except of course for the character set which is that used on all keyboards. Also we might add that some of the indifference, or even hostility, to APL and **J** may be due to the innate conservatism of many in the computing community, especially those in academia, who oppose the languages just because they are different.

I have always considered one of the great virtues of APL and **J** has been the suppression of the detail required in almost all other programming languages. The advantages of a suitable notation have been long known in mathematics, and have been admirably stated by the English mathematician A. N. Whitehead in *An Invitation to Mathematics* which first appeared in 1911 as follows:

> 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.

However I also believe that neither APL nor **J**, or any single programming language or system – and here I include application packages and spreadsheets – can be considered as the solution to all of one's programming problems. A person selects the language or system best suited for the application being considered although one justifiably has one's favourites. Just as a gardener has a variety of tools to serve his or her gardening needs so does a programmer use the most appropriate programming tool for the problem at hand. This is my view, and I speak from many years of experience as a programmer - and as a gardener.

The next three sections give a few examples I have used in my statistical packages and in classroom presentations. Although I have programmed almost all of them at various times in several different languages, the results given here have been obtained from programs written in **J**.

## Analyzing data

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 data arranged simply as a list of numbers. In two dimensions with the data arranged in rows and columns it is required

to find the row and column sums of the data. The calculation may be generalized to three or more dimensions where it is required to find some or all of the so-called marginal sums.

As an example consider some data representing 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 which may be represented in two-dimensions as

```
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 `2` 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 $2^3$ 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 $2^4$ 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.

We shall make only a very few remarks about the use of **J** to find marginal sums. If the four observations for the first variety-treatment combination are represented by `A1`, then the marginal sums giving the total of these observations is simply `+/A1`. If `A2` represents the three-by-four array for the three treatments and four replications for the first variety, then the treatment totals are the row sums `+/"1 A1`, the replication totals are the column sums `+/A1`, and the grand total is `+/+/A1`.

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 calculations for the marginal sums have been incorporated into general statistical programs written in all three array languages discussed here and have proven to be a considerable usefulness.

### Rolling dice and other forms of gambling

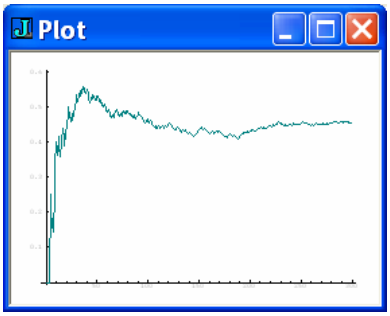
The rolling of dice, the tossing of coins, and the drawing of balls randomly from an urn have long been used to provide examples of statistical distributions and sampling procedures. Data could be generated by actually carrying out the experiments with real dice, coins, balls or marked slips of paper. Alternatively the experiments could be simulated using tables of random numbers, a common feature of statistical tables for many years. For example, a sequence of digits selected arbitrarily from a random number table could represent a sequence of coin tosses with an even digit representing a head and an odd digit a tail. There were even books of tables of random numbers, one being the RAND Corporation's *A Million Random Digits with 100 000 Normal Deviates* published in 1955 which was reviewed with some disbelief in *The New York Times*. In this section we will mention a couple of simulations often referred to in the statistical literature and give a few examples of our own.

Possibly 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 some of his statistical arguments. In one experiment he tabulated the results of rolling twelve dice 4096 times counting as a success the number of occurrences of a 4, 5 or 6 on each roll. Another set of dice data 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 fairly recent analysis of these latter results showed that the die was certainly biased. The author of this study remarked that it is reasonable to assume that a die of poor quality such as would have been manufactured in the nineteenth century would have developed a bias when rolled such a large number of times. He then remarked that dice now used in major gambling houses are machined to tolerances of

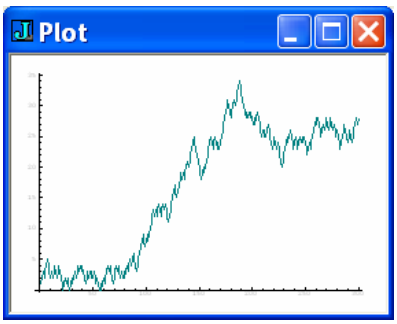
1/2000 of an inch, are made from a hard homogeneous material and are rolled only a few hundred times before being discarded.

Using **J** a simulation of Weldon's example took approximately 0.05 seconds and one of Wolf's took approximately 0.13 seconds.

It is well known that the ratio of the number of heads to the total number of tosses of an unbiased coin approaches 0.5 as the number of tosses increases. An example is shown in the first graph giving the




results of one simulation of 300 tosses. However, the cumulative excess in heads over tails tends to grow as the number of tosses increases. This is shown in the second graph representing the same simulation as that 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.




As another example consider the simulation of the very popular Lotto 6-49 in which a person purchases a ticket for one dollar and selects at random six integers between 1 and 49, inclusive. The payout in the weekly draw of six numbers is zero if the person has 2 or fewer matches with the numbers drawn, and 10, 75, 2500 and 100000 dollars for 3, 4, 5 or 6 matches, respectively. A little analysis will show that the chances of winning anything at all in a single draw are slightly less than 1 in 50. Furthermore we performed a simulation in which a person purchased two one-dollar tickets each week for 25 years. The results showed that the total investment of 2600 dollars gave a return of only 505 dollars.

A well-known example of a random process that is given in many statistics texts is Buffon's needle problem for estimating the value of $\pi$, the ratio of the circumference of a circle to its diameter. It was proposed by the French naturalist and biologist Comte de Buffon in 1760. Suppose we rule a series of parallel lines on a flat surface such as a table top and repeatedly drop at random on this surface a needle whose length is less than the distance between adjacent lines. By counting the number of times the needle crosses a line when it falls it is possible to estmate the value $\pi$. One trial of this procedure in which a needle was dropped 5000 times is reported to have given an approximation to $\pi$ of 3.15956.

A more intuitive example of randomly estimating $\pi$ is based on the observation that the ratio of the area of a quadrant of a circle inscribed in a square to the area of the square is $\pi/4$. The repeated random

generation of points within a square and determining if they lie within the inscribed quadrant is a simple programming exercise. Five simulations with 100000 random points in each simulation gave approximations to π of 3.1411, 3.1460, 3.1400, 3.1356 and 3.1398.

Random methods of estimating π must be considered only a small footnote in the long history of the endeavours - to some very interesting and to others completely useless - to calculate π to an ever-increasing number of digits. Two years ago a team of Japanese scientists used 400 hours of supercomputer time to compute π to 1.24 trillion places. If printed, this number which begins

```
3.14159265358979323846264338327950288419716939937510582097494 4...
```

would extend for almost 20 million miles.

Finally for the interested reader we give a few notes on generating random digits in **J**. The function *roll* `?` gives random non-negative integers so that, for example, the expression `?6` would give random integers between `0` and `5` and so could have the value `4` or `3` or `5` or ... , and the expression `?10$6` which is equivalent to

```
?6 6 6 6 6 6 6 6 6 6
```

would give `10` repetitions of sampling with replacement from the integers `0`, `1`, `2`, `3`, `4`, `5`, and could have the value `0 4 2 3 1 0 4 4 5 2`. The expression `>:?10$6` would represent sampling with replacement from the first `6` positive integers and could be used to simulate the rolling of a die so that the expression

```
+/"1 (>:i.6)=/>:?10000$6
```

gives a simulation of Wolf's die-throwing experiment cited earlier in this section.

### Collecting coupons

One of my favourite small statistical examples is one I met first as a graduate student and have used many times both as a classroom example and as a programming exercise. Indeed, about a dozen years ago I wrote a four-page brochure using it to introduce the **J** language. It is known as the coupon collector's problem, and has its application in the repeated purchase of some product such as breakfast cereal until a complete set of prizes, contained one in each box, is obtained. We are interested in knowing how many boxes of cereal must be purchased on the average until we have all of the prizes.

We may simplify the problem by eliminating the cereal boxes - and the cost of purchasing them and the bother of eating the cereal – by imagining that we have a box containing a number of slips of paper each with one of the integers 1, 2, ... written on it, one slip for each different prize. For example, if there are five prizes in the cereal boxes, then our simplified model would be a box with five slips bearing the integers 1, 2, 3, 4 and 5. Instead of purchasing the cereal, we would repeatedly draw a slip from the box, write down the number on it, replace the slip in the box, and continue until all of the five different

numbers appeared in our list. If there were six different prizes in the cereal boxes, we could use an even simpler model than drawing numbered slips by rolling an unbiased die repeatedly until all six different faces had appeared.

A statistician would call the above processes – whether the purchase of boxes of cereal or drawing numbered slips of paper – random sampling with replacement. He or she would then state the general problem as follows: If we sample with replacement from the first $n$ positive integers, what is the expected sample size required to obtain all $n$ integers in the sample? A little mathematics, which we will omit, shows that the expected sample size is simply

$n \times (1 + 1/2 + 1/3 + ... + 1/n)$ ,

or in words "$n$ times the sum of the reciprocals of the first $n$ positive integers. For example, if $n = 5$ corresponding to five different prizes, then the expected sample size is

$5 \times (1 + 1/2 + 1/3 + 1/4 + 1/5)$

which is equal to approximately 11.4. If $n = 6$ corresponding to the analogy of rolling a die, then the expected sample size is

$6 \times (1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6)$

or approximately 14.7. Finally if there were 100 different prizes, then the average number of purchases could be found to be approximately 518.7.

The general formula gives reasonable results for limiting small values of $n$. If there is only one prize, then $n = 1$ and the expected sample size is $1 \times 1$ or 1 which is reasonable since the single prize would be obtained on the first purchase. If there are no prizes, then $n = 0$ and the expected sample size is $0 \times 0$ or 0 and there would be no purchases to make.

The following table gives for a small range of values of number of prizes in the first row the expected sample sizes rounded to the nearest integer in the second row:

| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 29 | 50 | 72 | 95 | 120 | 145 | 171 | 198 | 225 | 281 | 338 | 397 | 457 | 519 |

So far we have considered only the average number of purchases required to collect a complete set of prizes. We might well ask what amount of variation about this average value may we expect. This question could be answered very simply by using a computer to simulate the sampling process a number of times and recording the size of each sample. As an example, 20 simulations for 5 different prizes gave sample sizes of

28 12 11 19 27 14 5 21 10 6 12 9 16 12 17 6 8 7 6 14

with a minimum of 5, a maximum of 28, and an average of 13. Another 20 simulations gave a minimum and a maximum of 7 and 26, respectively, and again an average of 13.

I thought I would see what examples of the coupon collector's problem could be found with today's cereals. Sadly the problem seems to have fallen out of fashion. Most brands of cereal offered no free

prizes although one offered a free watch and another a “Free Inside Outside Camera”, a phrase which required some parsing to make sense. Just as I was about to give up I found a box advertising on the front a “Free Inside Kidz Counter" described on the back ias follows: “Hey kids your mission is to walk an extra 2000 steps a day for 2 weeks, that’s about 20 minutes more activity a day, ...”. The “kidz counter” was a pedometer that meaured the number of steps only and came in three different colours. Thus if one wanted to collect all three colours the expected number of packages of cereal to be purchased could be found to be $3 \times (1 + 1/2 + 1/3)$ or approximately 5.5.

I have programmed the coupon collector’s problem in several languages. The simplest and most pleasing representation has been in **J**. As an example, the **J** expression for the expected sample size for `N` prizes is just `N*+/%>:i.N` which may be read as “`N` times (`*`) the sum (`+/`) of the reciprocals (`%`) of the first `N` positive integers (`>:i.N`)”.

## Teaching languages

For many years I had wondered what a non-Indo-European language was like, so shortly after I retired – or to repeat Ken Iverson’s delightful phrase “When I retired from paid employment...” - I started to study Japanese. I have doggedly persevered in this endeavour, first by taking formal classes and then by self-study, and now with considerable effort, consulting of dictionaries, and patience I can write a letter, keep a diary, or write stories about my cat who has the Japanese name “Torako” and who, incidentally, is sitting on the floor and meowing at me as I write this. In contrast my attempts to speak or to understand the spoken language have been dismal failures. My periods of despair with Japanese – and there have been many - might best be described by the following paraphrase of a well-known and now politically incorrect epigram of Samuel Johnson: “An elderly gentleman trying to learn Japanese is like a dog walking on its hind legs. It is not done well, but one is surprised that he learns anything at all.”

However there have been many unexpected pleasures resulting directly or indirectly from my study of Japanese. I have met many interesting people both in Canada and in Japan; I have had several delightful trips to Japan; I have eaten a very large number of most enjoyable Japanese meals; and I have gained just a little understanding of the Japanese people and their history. Also I think that I just may have a happier and fuller personal life. What I didn’t expect, though, was to become aware of the analogies between the learning and teaching of natural languages and programming languages. It is this topic which I wish to address in this section.

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 texts. 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 the 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. We follow Mr. Lloyd as he works with the company and meets some of the staff both at work and socially. Each of the twenty chapters has the same format: a summary of the story so far and another installment of the story; a list of new vocabulary; grammatical notes; exercises; a short reading exercise; and a one-page essay in English on some aspect of Japanese business. The Japanese hiragana and katakana syllabics are introduced at the beginning and the kanji (Chinese) characters a few at a time starting in Chapter 5, and blend well with the romaji (Roman) characters which are used.

Contrast this introduction to Japanese to the introduction to a programming language in most programming texts and followed in introductory courses. The texts (and courses) are really introductions to syntax with numerous examples and exercises intended to illustrate and reinforce grammatical principles. Furthermore, many of the exercises are artificial and even juvenile. An example in one recent text was a program to print either "ho-ho", "he-he" or "ha-ha" and then modified to print "yuk-yuk". As bad as is the pedagogy, the writing is even worse in some of the books. A colleague once remarked to me that most introductory programming courses were as interesting as a course in the conjugation of verbs.

In my opinion those involved in computing education have much to learn from how natural languages are taught. I firmly believe that the details of a language, **J** or Japanese or any other programming or natural language, should be introduced as needed in the exposition of the subject whether it be teaching multiplication tables in a Canadian classroom or discussing the introduction of the quality control methods of W. Edwards Deming to Japanese assembly lines.

### Remembering our 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 and 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. Very few professors have an appreciation of the development of their subject and are able to impart this understanding to their students. Moreover the historical development of a

scientific discipline is not considered to be a marketable skill, and has been displaced in computing courses at least, if it were ever in the curriculum, by topics based on current developments in hardware and software technology.

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. Bowden 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 a discussion of various applications of computers. Although much 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) when he was 79 years of age. 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 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. Finally a book I am reading as I write this paper is *Electronic Brains. Stories from the Dawn of the Computer Age* by Mike Hally (Granta Books, 2005). It is based on a BBC Radio Series which was described by one British newspaper as an "... offbeat, informative series [that has] captured the excitement of computing's early days." The book is proving to be just as entertaining.

And why should we be concerned with our history? An cogent answer has been given by the nineteenth-century Danish philosopher Søren Kierkegaard who said that "Life must be lived forward but understood backward." With computer technology developing at an increasingly frantic and exciting pace

we badly need guideposts for its intelligent and socially responsible application. Perhaps the history of our subject may provide some of the answers.

## Conclusion

All I have tried to accomplish in these pages is to give a very brief personal view of the development of programming languages and to show my enthusiasm for the three which I have enjoyed using the most, viz., APL, Nial and **J**. I do not know whether I have said anything that might appeal to Lord Rutherford's English barmaid, or at least to her twenty-first-century incarnation, but I hope I have.

Most of the material in this paper has appeared previously but has been rewritten so that it may appeal to a more general audience. Although appropriate citation of sources has been given in the original publications, a few acknowledgements are in order here. The data in the oat yield example is from *Principles and Procedures of Statistics* by R. G. D. Steel and J. H. Torrie (McGraw-Hill, 1960). The apple analogy originated with Frederick P. Brooks who worked closely with Ken Iverson in the early days of the development of APL. Some of the material on the evolution of APL and **J** comes from obituaries I have written recently in tribute to Ken who died on October 19, 2004 in his 84th year after having suffered a stroke three days previously.

This is not my first attempt to write a short apologia for my professional life. Each has produced something different although I trust all have been faithful to my perception at the time of what it is that I have been doing for so many years. I am reminded of one of Winnie-the-Pooh's many attempts at composing songs when he says

> "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."

---

*Keith Smillie is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2E8. His email address is smillie@cs.ualberta.ca.*