VECTOR

Remembering Ken Iverson

BCS®

The Journal of the British APL Association

A specialist Group of the British Computer Society

ISSN 0955-1433
www.vector.org.uk
Vol.22 No.3 August 2006

Contributions

All contributions to VECTOR may be sent to the Journal Editor at the address on the inside back cover. Letters and articles are welcome on any topic of interest to the APL community. These do not need to be limited to APL themes, nor must they be supportive of the language. Articles should be accompanied by as much visual material as possible (b/w or colour prints welcome). Unless otherwise specified, each item will be considered for publication as a personal statement by the author. The Editor accepts no responsibility for the contents of sustaining members' news, or advertising.

Please supply as much material as possible in machine-readable form, ideally as a simple ASCII text file or an HTML document. APL code can be accepted in workspaces from I-APL, APL+Win, IBM APL2 or Dyalog APL/W, or in documents from Windows Write (use the APL2741 TrueType font, available free from the website), and MS Word (any version, ideally using the Vector template which is also available from the website).

Except where indicated, items in VECTOR may be freely reprinted with appropriate acknowledgement. Please inform the Editor of your intention to re-use material from VECTOR.

Membership Rates 2005–2006

Category	Fee	Vectors	Passes
UK Private	£20	1	1
Overseas Private	£22	1	1
(Supplement for Airmail, not needed for Europe)	£4		
UK Corporate Membership	£100	5	5
Overseas Corporate	£110	5	
Sustaining	£500	10	5
Non-voting Member (Student, OAP, unemployed)	£10	1	1

The membership year normally runs from 1st May to 30th April. Applications for membership should be made to the Administrator using the form on the inside back page of VECTOR. Passes are required for entry to some association events, and for voting at the Annual General Meeting. Applications for student membership will be accepted on a recommendation from the course supervisor. Overseas membership rates cover VECTOR surface mail, and may be paid in sterling, or by Visa, Mastercard or JCB, at the prevailing exchange rate.

Corporate membership is offered to organisations where APL is in professional use. Corporate members receive 10 copies of VECTOR, and are offered group attendance at association meetings. A contact person must be identified for all communications.

Sustaining membership is offered to companies trading in APL products; this is seen as a method of promoting the growth of APL interest and activity. As well as receiving public acknowledgement for their sponsorship, sustaining members receive bulk copies of VECTOR, and are offered news listings in each issue.

Advertising

Advertisements in VECTOR should be submitted in typeset camera-ready format (A4 or A5) with a 20mm blank border after reduction. Illustrations should be photographs (b/w or colour prints) or line drawings. Rates (excl VAT) are £250 per full page, £125 for half-page or less (there is a £75 surcharge per page if spot colour is required).

Deadlines for bookings and copy are given under the Quick Reference Diary. Advertisements should be booked with, and sent to Gill Smith, Vector Production, Brook House, Gilling East, YORK YO62 4JJ. Tel: 01439-788385.

Email: gill@apl385.com.

Contents

Editorial: Remembering Kenneth Iverson

by Stephen Taylor

For the best memorial, die while your colleagues have energy to write. With this issue, Vector at last commemorates the life and work of Dr Kenneth E. Iverson. We had hoped to publish a substantial review, by colleagues and collaborators, of his life and work. But Ken Iverson's influence and long working life covered so many decades, countries and people that this larger project must be left to posterity.

Ken Iverson started developing his notation at Harvard in the 1950s as a mathematical code for describing computations. It blossomed through APL into a family of languages. Many hands have nourished these languages; but Ken Iverson was father to them all.

Iverson Notation was at the centre of the young computer industry. Ken was lured from Harvard to become an IBM Fellow. At IBM his notation was used for the first-ever formal description of an operating system. His Harvard teaching assistant, Fred Brooks, managed the development of this operating system, OS/360, the heart of IBM's breakthrough mainframe series. Brooks' experiences managing the project formed the basis of his classic The *Mythical Man-Month: Essays in Software Engineering*. Brooks remembers a mentor here.

The adage that the optimum size of a programming team is "fewer than six or more than a hundred" nicely distinguishes two ways of developing software. Industrial methods require large teams, disciplined administration and Big Design Upfront; craft programmers work flexibly, alone or in small groups. To industrial programmers, writing software looks like engineering; to the craftsmen, like developing a movie. Craft programmers find the use of powerful notations repays the effort of mastering them; managers of industrial programmers shy from high

skill levels, and rely on development methods to which coding agility and productivity have little to contribute.

There is not much to wonder at in any of this; it replicates a division found in many fields. In computing, the APLs remain an esoteric tradition, the province of solo programmers and small teams; there is no reason to expect this to change.

The more reason to heed Dr Iverson's appeal in his article "APL in the New Millenium", published here for the first time. He writes: My hope is to encourage the relatively small APL family to mute their differences, and present a more united face to the programming world.

Keith Smillie weighs in with a survey of his life as an array programmer. Professor Smillie teaches at the University of Alberta, Roger Hui's *alma mater*, in the home province of both Ken Iverson and Arthur Whitney. (Perhaps the acronym should denote the Albertan Prairie Languages.)

Love of music and mathematics frequently coincide. Ken Iverson shared them with Norman Thomson, who here uses J to reach new clarity in an exposition of the mathematics of tempering. Exposition is the theme too of Paul Berry 's memoir of Expository Programming at IBM in the 1960s, a subject that is resurfacing these days under the less elegant name of Embedded Domain-Specific Languages.

For his tribute, Eugene McDonnell measures a quarter-century of progress by comparing the number of tokens used in solutions written in APL\360 and in J.

Adrian Smith's work has him frequently switching languages. Writing C# is something you do to earn a living; fooling around in an APL session is something you can do of an evening, with Bach on the radio, and a nice log fire at your feet. (That music again.) He celebrates, by relaxing with a New Scientist puzzle and an APL interpreter.

Peter McDonnell's caricatures of APL luminaries ornament this edition. We print memories of Ken from Linda Alvord, Bob Bernecky, Jim Brown, Eugene McDonnell and Joey Tuttle. Larry Breed recounts the implementation of APL\1130. Gitte Christensen, Michel Dumontier and Richard Hill remember Ken Iverson in Denmark, France and Australia. (English represents arrays too.) Roger Hui, Donald McIntyre, and Keith Smillie write of their decades of collaboration with KEI. And we finish with Roger Hui's collection of quotations and 'Kenecdotes'.

Stephen Taylor

Quick Reference Diary

Date	Venue	Event
Oct 16-20 2006	Denmark	Dyalog 2006 Conference.

The 4th Annual Dyalog User Group Conference will take place on 16-20 October 2006 at LO-Skolen in Helsingør, Denmark - the same venue as last year. Due to popular demand, this year there are two training days and a full day is dedicated to a general "Array Language Conference" - attended by several other vendors of APL programming products and other Array Languages.

Nov 6-9 2006	Naples, Florida	APL2000 User's meeting

The Eleventh Annual APL User Conference will be held November 6 - 9, 2006 at the Naples Beach Hotel & Golf Club in Naples, Florida. The registration fee will cover attendance at all four days. Monday and Tuesday will focus on the APL product for .NET. Wednesday and Thursday will focus on the APL product for the Win32 environment.

Dates for Future Issues of VECTOR

	Vol.22 No.4	Vol.23 No.1	Vol.23 No.2
Copy date	in press	10th Sept	2nd Dec
Ad booking	in press	17th Sept	9th Dec
Ad Copy	in press	24th Sept	16th Dec
Distribution	Sept 2006	Nov 2006	Jan 2007

ARTICLES

APL in the New Millennium

by Kenneth E. Iverson

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.
- Such difficulties have led some dialects (such as Nial) to adopt ordinary names as *reserved words*, an approach that the special characters had allowed us to avoid.
- Use of the internet also poses problems, because APL characters are not generally handled by browsers.

J and K use only the ASCII alphabet, but yet avoid the use of reserved words. K uses mainly single-character non-alphabetics, and J supplements these by a scheme that uses a suffixed period or colon. For example, < denotes *less-than*, <. denotes *lesser-of* (minimum), and =: denotes assignment.

It would, of course, be impractical for any APL system to switch to a rival alphabet, and this discussion is meant only to suggest supplements based on rival ideas. For example:

1. Many ASCII symbols (such as @, &, ^, and %) go unused in most APL systems, and could be used in various ways:

   One or two might be used as suffixes, as in <@ or nub@ as names for primitives.

   The * adopted for power in APL\360 has since become universally recognized as the symbol for multiplication, and might better be replaced by the ^, as first proposed by De Morgan in mathematics, and recently adopted in some non-APL languages.

2. The percent symbol (which signifies "divided by 100") has been adopted for division by some non-APL systems, and could be more widely adopted in APL. After all, the APL symbol ÷ is no longer familiar as a symbol for division.

Number Representation

APL\360 introduced the useful representation of a negative number distinct from the negation function (-) that might produce it. This has been continued in all APLs in forms that vary from the special overbar symbol used in APL\360. Constrained to ASCII symbols, J uses the underbar, and K uses juxtaposition: `-3` for a negative number, as contrasted with `- 3` for negation.

APL systems use the exponential notation adopted from Fortran, some using the uppercase `E`, and some the lowercase. The notion has been extended to other kinds of numbers, as in `2d45` for a complex number in polar representation (APL2); `3j4` for a complex number (SHARP APL); and `2r3` for a rational number (J).

Grammar

The grammar (parsing rules) of APL\360 were simple and relatively uniform, with no precedence among functions, but with operators given precedence over functions.

In particular, the relative positions of functions and arguments were fixed; for example, factorial n was written as `! n`, *not* `n !`. There are, however a few characteristics that merit comment.

Name assignment

A left arrow was used to assign names to arrays of numbers and characters, but a quite different mechanism was used for assigning names to functions, and there was no provision for defining operators.

In APL2, operator definition was introduced by extending the system for function definition to allow further parameters. Most systems have adopted this scheme, but J uses a single copula (the `=:` mentioned earlier) for all three cases, and Dyalog APL uses it for two.

Valence

APL\360 systematically adopted the double use of symbols from the scheme suggested by subtraction (`3-2`) and negation (`-2`) in mathematics, calling the two cases *dyadic* and *monadic*. For example, `! n` denotes the factorial, and `m ! n` denotes the related binomial coefficients.

Most systems (with the exception of Nial) continued this *ambivalent* use of primitives, but did not extend it to the *derived* functions produced by operators. In

J, *all* functions are ambivalent. For example, `+/ y` denotes *sum over* `y`, and `x +/ y` denotes the addition table; that is, the plus-outer-product denoted by `x ∘.+ y` in APL\360 and APL2.

Indexing

Because of the need for multiple index arguments for a multi-dimensional array, APL\360 adapted from Fortran the notation `A[I;J;K]`, departing from the normal form for a dyadic function. In particular, it was not possible to assign a name to the complete index argument.

The introduction of *nested arrays* in APL2 made possible the treatment of a set of indices such as `I;J;K` as a single entity. However, with the exception of J, APL systems did not fully exploit this to rationalize indexing.

APL\360 introduced a further anomaly in indexing by providing either 1-origin or 0-origin indexing, set originally by a *system command* of the form `)IO`, and later by a *system variable* `⎕IO`.

This choice of index origin has been maintained in most systems, but J is restricted to `0`-origin. Since indexing in J is a normal dyadic function, an operator can be used to modify it, as well as the related index generator analogous to the iota of APL\360.

The restriction to `0`-origin has simplified the introduction of *negative indexing*, with indices for `n` items running from negative `n` to `n-1`.

The *indexed assignment* `A[I;J]←M` is a further convenient (though grammatically anomalous) scheme introduced in APL\360. It is maintained in most systems, although the three essential arguments can be handled by an *amend* operator, such as the `}` used in J in the form: `M ind } A`.

Terminology

Coming from a common background in math, we naturally adopted mathematical terminology in APL\360, in spite of the facts that:

- The term *operator* (used in the sense of Heaviside) conflicts with its common use in elementary mathematics as a synonym for function.
- The term *variable* used for a name assigned to a quantity suggests that its meaning might *vary* due to possible re-assignment. But the same possibility exists for defined functions, and the terms *variable* and *function* do not adequately reflect the possible cases.

Terms from English grammar can provide the necessary distinctions, using the close analogy between *function* and *verb* as denoting *actions*, together with the *nouns* and (variable) *pronouns* on which they act.

Moreover, *adverbs* are analogous to *operators* (such as `/`) that modify a single verb, and *conjunctions* (such as the copulative conjunction *and* in the phrase "run and hide") are analogous to operators (such as the inner-product) that apply to two verbs.

Finally, the familiar English terms *list*, *table*, and *report* are more commonly understood (and are fully as accurate as) the terms *vector*, *matrix*, and *higher-dimensional array*.

Opportunities

Most APL systems share unexploited cases that may be introduced without conflict. We will illustrate these opportunities by three classes.

Complex arguments

Although complex numbers serve primarily in mathematical expressions, their two parts (real and imaginary) can be convenient in functions that require the specification of two independent parameters.

For example, a format function `F` might be defined so that `12j3 F x` formats `x` with a width of `12` spaces and with `3` digits following the decimal point. A list of such complex arguments could be used to specify each column of the format of a table `x`.

Vector arguments

In APL\360, the expression `⍳5` produced a list of five successive integers, but the domain of `⍳` did not include a vector argument. A useful non-conflicting extension could be defined to apply to a list of n integers to produce an array of successive integers of rank n (as in J) or a nested array of indices of an array of the same rank (as in Dyalog APL).

Trains

In calculus, the expression `f+g` is sometimes used to define a function equivalent to the sum of the functions `f` and `g`, and `f*g` is used for their product. Moreover, such a train of three functions is treated as an error in most APL systems, and could therefore be introduced without conflict.

Any three functions may be used. For example +/ divided-by $ (the number of elements) defines the *mean* function, and f,g defines a function that catenates results, as in +/,*/.

More generally, a train of any odd number of functions defines a function, the last three defining a function as stated above, and this function entering into a similar definition with the remaining train.

For example, the identity function followed by - (subtraction) followed by the preceding three-element train for the mean defines the function "centre on the mean".

Extensions

Functions and operators new to any system can of course be adapted from other systems without conflict. In early systems the choice of symbols posed a problem, soon solved in a general way by the adoption of a class of "⎕names"; alphabetic names preceded by the quad character. This solution appears to continue in systems that adhere to the special APL character set.

We will now discuss a few of the many functions and operators that are candidates for adoption.

GCD and LCM

The logical *or* and *and* could be construed as special cases of *maximum* and *minimum* (when restricted to the Boolean domain 0 and 1), or as special cases of *greatest common divisor* (GCD) and *least common multiple* (LCM).

E.E. McDonnell noted that only the latter functions possessed the same identity elements as *or* and *and*, and he ensured that the logical functions were extended to GCD and LCM in the SHARP APL system. The same extension could be made without conflict in any APL system.

Repeatable random numbers

In debugging a program it is often useful to generate "random" arguments in a repeatable manner. This can be done by resetting the random seed. It is more convenient to provide a random number generator (denoted, perhaps, by ?.) that resets the seed on each use.

Variants

APLSV used the *system variable* ⎕CT to specify the comparison tolerance to be used in relations such as < and =. Such control can be made more convenient by a variant operator, as in = VAR 0 for exact comparison.

A more interesting opportunity for variants occurs in the case of the *rising* and *falling* factorial functions, defined by `×/x+s×⍳n`, for `s` assigned the values `1` and `¯1`, respectively. Moreover, a zero value for `s` gives the function `x` to the power `n`, and these functions (including the useful case of non-integer values for `s`) can all be treated as variants of the power function.

Ties

The sum `a+b+c+d+e` can be written as the reduction `+/a,b,c,d,e`. Moreover, the continued fraction `a+b÷c+d÷e` might lend itself to a similar reduction, except that it requires the *two* functions of addition and division.

Such a pair of functions (or any number) can be provided by a `TIE` operator, to be used in the form `+TIE÷ / a,b,c,d,e`.

The result of a tie can be used in other ways, as with a *case* operator, in which `f TIE g TIE h CASE i` uses the index produced by the function `i` to select one of the functions for execution. In particular, the tie of two functions can be used to make a recursive definition.

Universal sorting

A strict ranking can be imposed on *all* arrays (including characters, real and complex numbers, and open and nested arrays of any rank) so that sorting can be applied to *anything*. An example of such ranking is provided in J, and could be adapted to any system.

The Workspace

When first proposed by Adin Falkoff for APL\360, the (32K) workspace provided a convenient and efficient organization of memory. However, the fixed size, and other characteristics of the workspace, have come to inhibit the use of APL.

In particular, the workspace organization did not facilitate the exploitation of the memory management offered by later machines and operating systems. Arthur Whitney made the first step in employing these facilities in his A system, and used text (*script*) files rather than workspaces.

The advantages of text files are beginning to be recognized. In an item on page 59 of the April issue of *Vector* (Vol. 16 No. 4), Anssi Seppälä is quoted as follows: "the more I can work with text files, the easier it is – I am no longer a fan of the APL workspace".

Acknowledgment

I am indebted to Chris Burke for many helpful comments, particularly for his suggestion to discuss the matter of the workspace versus script files.

References

[1] Iverson, K.E., *A Personal View of APL*, IBM Systems Journal, Vol 30, No. 4, 1991.

[2] Heaviside, Oliver, See the 1971 Chelsea Edition of *Heaviside's Electromagnetic Theory* and the article by P. Nahin in the June 1990 issue of *Scientific American*.

Discovering Array Languages

by Keith Smillie

APL? That's the language with all the funny symbols, isn't it?" "J? What a funny name for a language!" "What's so special about APL [or J]? Why not use BASIC [or C++ or Java or ...]? We've all heard remarks such as these, spoken with various degrees of doubt, sarcasm or even derision. In this paper we shall try to provide answers to these questions in a simple manner in a way which might encourage the reader to seek further information about APL or J.

In addition to answering the sceptics, there is another reason why we should take the time to describe in simple 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.

When we attempt to explain our work in simple non-technical language, we are in excellent company. The physicist and Nobel laureate Erwin Schrödinger in a series of public lectures which were published in the early 1950s as *Science and Humanism* said that "If you cannot – in the long run – tell everyone what you have been doing, your doing has been worthless."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."

We shall begin with a simple metaphor for classifying programming languages and follow this with examples of programming a simple example in both a hypothetical, but we hope representative, machine language and in BASIC which we will consider representative of a conventional programming language. We shall then discuss the origins of APL and J, illustrate their use with the same example as previously used, discuss the acceptance of these languages in the programming community, and discuss a few applications which may be of general interest and even of some utility. Then we compare the teaching of natural languages and the teaching of programming languages, and finally discuss the importance of keeping a historical perspective on one's work. Two Appendices give the J programs for the machine-language simulator and the applications.

Selecting Apples – a Metaphor

Frederick Brooks, a colleague of Kenneth Iverson, whom we shall soon introduce as the originator of APL and J, has given a nice analogy of the difference between a conventional language such as BASIC and an array language such as APL or J. 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 would be something like the following:

> Reserve a place for the good apples. Then select an apple from the box, and if it is good put it in a reserved place. Select a second apple, and if it is good put it too in the reserved place. ... Continue in this manner examining each apple in turn until all of the good apples have been selected.

On the other hand the instructions corresponding to an array language would be simply

> 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 would be left to the helper.

Based on the above analogy we may classify programming languages as follows:

- One-apple-at-a-time languages
  Fortran, BASIC, Pascal, C++, ..., Java, ...
- All-the-apples-at-once languages .
  ..., APL, ..., J, ...
- Other languages
  Spreadsheets, application packages, ...

Conventional Languages

A program for any of the early computers had to be written in so-called machine language – different for each model of computer – and consisting 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 addresses of the result. For example, for the first computer I used, the National Cash Register 102A, 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." As another example, the instruction 33 0345 0656 0273 meant "If the number in location 0345 is greater than the number in location 0656, take the next instruction

from location 0273; otherwise take the next instruction in sequence." A program consisted of a sequence of these numerical instructions, and was executed starting with the first instruction and proceeding sequentially instruction-by-instruction unless the sequence was changed by a transfer-of-control instruction or a Stop instruction was encountered. Programs for real problems often consisted of hundreds or thousands of lines of such code, and might take weeks or even months to write and debug.

To illustrate a machine-language program we shall consider a hypothetical computer almost identical to that given in *Electronic Computers, Revised Edition* by S. H. Hollingdale and G. C. Tootill (Penguin, 1971). The memory consists of an arbitrary number less than or equal to 999 of locations or "words" numbered sequentially 000, 001, 002, . . ., and an additional word of memory, called the "Accumulator" and represented by A, where arithmetic and logical operations are performed. An instruction is a five-digit integer with the first two digits giving the operation and the last three the address. 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 09176 means take the next instruction from location 176 if the number in the Accumulator is negative. Data are considered input one number at a time from paper tape, and output is printed one number at a time.

The complete order code is as follows, where (xxx) and (A) represent the contents of location xxx and A, respectively:

```
01xxxA ← A + (xxx)   06xxxPrint (xxx)
02xxxA ← A - (xxx)   07xxxxxx ← (A)
03xxxA ← A × (xxx)   08xxx          Read no. from tape into xxx
04xxxA ← A % (xxx)   09xxx          Next instr. from xxx if (A) < 0
05xxxA ← (xxx)       10000          Stop
                     11xxx          Next instr. from xxx
```

The last instruction which transfers control unconditionally to the specified address has been added to the instruction set given in Hollingdale and Tootill.

As an example 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:

```
5114 2114 7114 8115 5115 9112 7115 2114 9103 5115 7114 11103 6114 10000
```

The program is stored in memory locations 100, 101, . . ., 112. Execution of the program begins with the instruction 05112 in location 100. The input consists of

a list of non-negative numbers appended with an arbitrary negative number which serves as a flag to the program indicating that all valid numbers have been processed.

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    05114      ) Set maximum no. Xmax to 0
101    02114      )
102    07114      )
103    08115      Read no. X from tape
104    05115      Copy X to Accum.
105    09112      Is X < 0?
106    07115      ) X - Xmax
107    02114      )
108    09103      Is X - Xmax <0?
109    05115      ) Replace X with Xmax
110    07114      )
111    11103      Next insruction from locn. 103
112    06114      Print Xmax
113    10000      Stop
```

This program is a "one-apple-at-a-time" program since each number has to be compared in turn with the current maximum, set initially to zero, which it replaces if it is larger.

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. Since its release in 1957 many versions of Fortran were developed and the language 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.

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. The following is a BASIC

program for the example problem of 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, -1
Max = 0
READ Number
WHILE Number >= 0
   IF Number > Max THEN
      Max = Number
   END IF
READ Number
WEND
PRINT Max
STOP
END
```

This program, like the machine-language program given earlier in the section, is also a one-apple-at-a-time program since each number must be examined in turn and the current maximum updated if necessary. However the BASIC program is much easier to write and can probably be understood by a person with little or no knowledge of the language.

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 for the description of algorithms arising in problems of sorting, searching and optimisation.

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 his 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 first example consider the data given in the previous BASIC program which represent the totals of several grocery shopping trips. These amounts may be defined by the simple APL statement

```
TOTALS ← 19.11 12.77 21.31 16.1 12.19 25.76 17.49
```

The sum of these numbers is given by `+/TOTALS` which has the value `124.73`. The APL function `+/` may be considered analogous to the familiar sigma symbol Σ of conventional mathematical notation. The only extension of this concept in conventional notation is the symbol Π for product. However, in APL the operator `/` may be applied to a large number of arithmetic and logical functions. For example, `⌈/TOTALS` or `25.76` is the largest item of `TOTALS`, `⌊/TOTALS` or `12.77` is the smallest item, and `⍴TOTALS` or `7` is the number of items. We note that the expression

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

is equivalent to the machine-language and BASIC programs of the last section. (The product is given by `×/TOTALS` although it is difficult to think of a meaningful interpretation of this value for the present data.)

As a further example of array operations in APL we shall consider the prices and number of units purchased for each of the items for the sixth shopping trip. 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. If the prices and units purchased are represented by *PRICE* and *QTY* respectively, then 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, where expressions entered by the user are indented while responses by the APL system are not.

As an example of how APL functions and operations may be extended to arrays of higher dimensions, consider finding the frequencies of occurrence of each of the six faces when a die is rolled a number of times. Suppose

```
ROLLS ← 4 5 2 1 4 2 4 4 3 2 3 3 3 3 3
```

is a list of the results of rolling a die 15 times and `FACES` is the list of the first 6 positive integers. Then the expression `FACES ∘.= ROLLS` gives the table

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

where the second row for example shows that a 2 occurred on the third, sixth and tenth rolls. and `+/FACES ∘.= ROLLS` gives the row sums `1 3 6 4 1 0` of the required frequencies. These expressions make use of the "outer product" `∘.` , a generalization of the familiar addition table of elementary arithmetic, so that for example if `I` is a list of the first twelve positive integers then `I∘.+I` is a twelve-by-twelve addition table.

Array Languages – J

In 1980 Ken Iverson left IBM and returned to Canada to work for I. P. Sharp Associates, a firm with head offices in Toronto which was using APL to establish a time-sharing service that became widely used in Canada, the United States and Europe. In 1987 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
```

The J equivalent of the APL expression in the dice-throwing example is `+/"1 Faces =/ Rolls` where / is the adverb "table".

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 in most languages are called verbs in J, operators are called adverbs since they modify verbs, and the variables of 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.

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 as we noted earlier 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 English-language keyboards.

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." This was a favourite quotation of Ken Iverson's, one which he used in introducing his Turing Award lecture "Notation as a tool of thought". Another even broader injunction for brevity is given in the apocryphal Ecclesiasticus (c. xxxii, v. 8) where we are instructed to "Let thy speech be short, signifying much in a few words."

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.

The next three sections give a few examples I have used in my statistical work 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.

Analysing Data

In this section we shall consider briefly a type of calculation that is fundamental to a large class of statistical applications. The data have been taken from *Principles and Procedures of Statistics* by R. G. D. Steel and J. H. Torrie (McGraw-Hill, 1960). 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 23 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 24 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.

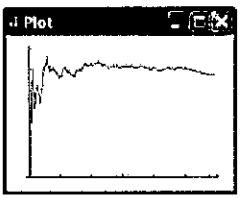
We shall make only a very few remarks about the use of J to find marginal sums. If the four observations for the first variety-treatment combination are represented by `A1`, then the marginal sums giving the total of these observations is simply `+/A1`. If A2represents the three-by-four array for the 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 very useful.

Rolling Dice and other Simulations

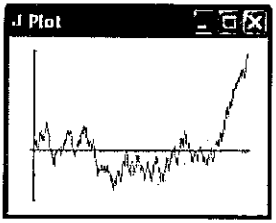
The rolling of dice, the tossing of coins, and the drawing of balls randomly from an urn have long been used to provide examples of statistical distributions and sampling procedures. Data could be generated by carrying out the experiments with real dice, coins, balls or marked slips of paper. Alternatively 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 one of the best-known examples in the statistical literature is the work 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 Wolf's 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.



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 π, 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 estimate the value π. One trial of this procedure in which a needle was dropped 5000 times is reported to have given an approximation to p of 3.15956. A discussion of the simulation of Buffon's problem is given in *Computer Approaches to Mathematical Problems* Jurg Nievergelt, J. Craig Farrar and Edward M. Reingold (Prentice-Hall, 1974). Five repetitions of this simulation for 5000 trials each gave estimates for π of 3.13185, 3.11526, 3.13381, 3.11042 and 3.11042.

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 p 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.

Collecting Coupons

One of my favourite 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 shall 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 would 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 1×1 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 0×0 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 as 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 measured 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.

Teaching Languages

Most texts used in introductory computing courses give either an introduction to whatever language is currently in fashion or an overview of computing science in which programming is treated somewhat briefly. The programming examples and exercises are carefully and often meticulously presented with the purpose of illustrating the syntax of the language. While many of the problems may have some intrinsic interest, when viewed together they present no continuous narrative which the student can see developing. Unfortunately in some texts many of the examples are artificial or even juvenile. An example in one C++ text was a program that gave a prompt for the user to input a "favourite" number and gave the response `I think that [whatever number was input] is a nice number`. One Java text has as an example a program to print either `ho-ho`, `he-he` or `ha-ha` which was then modified to print `yuk-yuk`. One programming assignment I have seen required the student to prepare a table of n^7 and 7^n for a

range of values of n, an exercise of little interest and of doubtful application. Examples such as these undoubtedly prompted a colleague to remark that most introductory programming courses were as interesting as courses in the conjugation of verbs.

It is of interest to compare such approaches to teaching programming languages with the teaching of natural languages. We shall give two examples, one in teaching children to understand and read English, and the second in teaching a foreign language to native speakers of English.

One delightful book intended to introduce English to young children is *Richard Scarry's Storybook Dictionary* (Paul Hamlyn, London, 1967), a book I purchased years ago for my young daughter. I now have the pleasure of reading it to her children, much to their own delight as well as to mine. This large format book introduces the child to 2500 words by means of 1000 pictures through the adventures of such colourful characters as Ali Cat, Dingo Dog, Gogo Goat, Hannibal Elephant and Andy Anteater. In the Introduction we are told that "He [presumably girls are included too] will *not* be given rules. Rather, he will be shown by examples in contexts which completely catch his interest and hold his attention." If we would only teach programming in the same way!

The second example is related to the teaching of Japanese, a language which I took up shortly after retirement and which I have pursued doggedly for several years. My periods of despair with Japanese – and there have been many – might best be described by the following paraphrase of the well-known epigram of Samuel Johnson: "An elderly gentleman trying to learn Japanese is like a dog walking on its hind legs. He does not learn 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.

Most of my Japanese texts teach the language by the telling of some continuing story which although fictional is intended to be realistic. In my first text, *Japanese for Busy People I* (Association for Japanese Language Teaching, Kodansha International, 1984) a prominent figure in most of the reading exercises which begin each chapter is Mr Smith ("Sumisu-san" in Japanese), a lawyer working in Tokyo, and we see him as he meets Japanese colleagues and visits some of them in their homes. In another little book, *Conversational Japanese in 7 Days* (Etsuko Tsujita and Colin Lloyd, Passport Books, 1991) – the title is not to be believed – we are

introduced to the Japanese language and culture as we accompany Dave and Kate Williams as they spend a week as tourists in Japan.

My favourite text 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 features 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. One of the main characters is a 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 instalment 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 *rōmaji* (Roman) characters which are also used.

Ken Iverson was motivated, as was mentioned earlier in this article, to develop APL and J because of his concern for the inadequacy of conventional mathematical notation for teaching many of the topics arisng in computing. He would return to this theme frequently in his writings, one example being given in *A personal view of APL* where he writes that "As stated at the outset, the initial motive for developing APL, was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects."

Because of their terseness APL and J are ideal tools for exposition, since much of the detail required in conventional computing languages may be omitted. Of course some introduction, however brief, must be given to either language and its interactive use before it may be used. However, such an introduction need not be much more than given in this article with possibly a few remarks on programs (which are "functions" in APL and "verbs" in J). With this simple introduction one can begin an exposition of the desired topic introducing additional features of APL or J as required.

Ken Iverson wrote and lectured unceasingly on the use of first APL and then J for the exposition of a variety of topics. One of his earliest works was *APL in Exposition* (IBM Philadelphia Scientific Center Technical Report No. 320-3010,

1972), a very small technical report beginning with a short summary of the entire language and followed by short accounts of the use of APL in the teaching of various topics such as elementary algebra, coordinate geometry, finite differences, logic, sets, electrical circuits, and the computer. The last section in just 15 pages presents the logic of a simple computer for executing algebraic expressions, the parsing and compilation of compound expressions, and a program for this computer to generate a sequence of Fibonacci numbers. A contemporary publication was his *ALGEBRA: An Algorithmic Treatment* (APL Press, 1972) giving a lengthy treatment of variety of topics in algebra. Many publications followed using first APL and then J in the exposition of various mathematical topics. He also published in print and in releases of J and on the J Website at *www.jsoftware.com* a number of "J companions" intended to supplement well-known texts. A typical one published initially in print form was *Concrete Math Companion* intended to be read with *Concrete Mathematics: A Foundation for Computer Science* by R.L. Graham *et al.* (Addison-Wesley, 1988). One of his projects at the time of his death (on October 19, 2004) was a companion to the encyclopaedic *Handbook of Mathematical Functions* by M. Abramowitz and I. A. Stegun.

Another very early publication on the use of APL as a notation rather than a programming language was "The Architectural Elegance of Crystals Made Clear by APL" by Donald McIntyre, then Professor of Geology at Pomona College in Claremont, California (*Proceedings of an APL Users Conference*, I. P. Sharp Associates, Toronto, Ontario, pp. 233 – 250, 1978) which examined the geometry of the atomic structure of crystals using APL. In the Introduction the author states that "... I introduce notation only as needed for the work in hand, minimizing the computer and machine characteristics. Indeed, I do not mention APL to start with, treating the primary functions as natural extensions and revisions of algebra, and I use no APL text."

Finally we should mention the extensive use of APL and J in the exposition of a variety of topics in probability and statistics and in the development of statistical packages in these languages. The conciseness of either language and its interactive implementation make it ideal for the exposition of statistical concepts in the classroom, further practice in the laboratory, and analyses arising in research. Furthermore statistical packages which may be used with very little knowledge of APL or J may be developed with relatively little effort from the material prepared for classroom and laboratory use.

Remembering our Past

There appears to be little opportunity now for our students to become acquainted with the history of computing. Few faculty appear to have much knowledge of, or interest in, the development of the subject. Also the historical consideration of a scientific discipline is probably not popular because it is not considered to be marketable. Introductory computing texts usually have a small amount of historical material often presented in sidebars so as to not interfere with the presentation of topics considered to be more relevant.

And why should we be concerned with our history? A 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. In this section I would like to mention just a few of the books dealing with the history of computing which I have enjoyed.

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 Hollingdale and Tootill's *Electronic Computers* already mentioned in the discussion of machine-language programming. 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 I might mention a very recent book, *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 just as entertaining.

Further Reading

The Web pages maintained by Jsoftware Inc. at *www.jsoftware.com* are an excellent source of the current state of J listing several introductions to J which may be downloaded at no charge and information on others available through booksellers. Of the many papers dealing with the development of APL and J the following three, all of which have extensive bibliographies, are to be recommended:

Kenneth E. Iverson, "Notation as a tool of thought", *Communications of the ACM*, vol. 23, no. 8, 1980, pp. 444 – 465.

K. E. Iverson, "A personal view of APL," *IBM Systems Journal*, vol. 30, no. 4, 1991, pp. 582 – 593.

D. B. McIntyre, "Language as an intellectual tool: From hieroglyphics to APL", *IBM Systems Journal*, vol. 30, no. 4, 1991, pp. 554 – 581.

An earlier version of this paper with the title *My Life with Array Languages* is available on the Web at *www.cs.ualberta.ca/~smillie/Jpage/Jpage.html* .

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.

Appendix 1.
Machine-language Simulator

```
InstrNames=: add';'sub';'mpy';'div';'get';'out';'str';'inp';'trn';'hlt';'jmp'

ht=: 3 : 0
whilst. -. HALT do.
   IR=:CR{M                              NB. Get instruction
   CR=: >:CR                             NB. Increment counter
   'OP ADDR'=: 100 1000 #: IR            NB. Decode instruction
   select. > (<: OP) { InstrNames        NB. Execute instruction
      case. 'add' do. A=: A + ADDR } M
      case. 'sub' do. A=: A - ADDR } M
      case. 'mpy' do. A=: A * ADDR } M
      case. 'div' do. A=: A % ADDR } M
      case. 'get' do. A=: ADDR } M
      case. 'out' do. PRINTER=: PRINTER, ADDR } M
      case. 'str' do. M=: A ADDR } M
      case. 'inp' do. TAPE=: }. TAPE [ M=: ({. TAPE) ADDR } M
      case. 'trn' do. if. A < 0 do. CR=: ADDR end.
      case. 'hlt' do. HALT=: 1
      case. 'jmp' do. CR=: ADDR
   end.
end.
empty ''
)

Clear=: 3 : 0
empty M=: 1000$0
)

Load=: 3 : 0
:
empty M=: y. (x.+i.#y.) } M
)

Run=: 4 : 0
CR=: x.                              NB. Address of first instruction
TAPE=: y.                            NB. Data tape
PRINTER=: i. A=: IR=: HALT=: 0       NB. Reset registers, etc.
ht ''
)

NB. Maximum
Ex1=: 5114 2114 7114 8115 5115 9112 7115 2114 9103 5115 7114 11103 6114 10000
Tape1=: 19.11 12.77 21.31 16.1 12.19 25.76 17.49 _1
NB.      Clear ''            NB. Reset memory
NB.      100 Load Ex1        NB. Load program in 100, 101, ...
NB.      100 Run Tape1       NB. Start execution in locn. 100
NB.      PRINTER             NB. Display maximum
```

Appendix 2. J Verbs for Applications

```
NB.  r=: x name y, where
NB.  y is right argument,
NB.  x is left argument if any, and
NB.  r is result.
NB.  Suggested example calculations given in parentheses
NB. Marginal sums
msum=: 4 : '+/^:(+/-.x.)(/:x.)|:y.'
NB.  x: dimensions to be summed over
NB.  y: array
NB.  r: Marginal sum
NB. (A=: >:i. 12, and 1 1 msum A, 0 1 msum A, 1 0 msum A, 0 0 msum A)
NB. Rolling dice
each=: &.>
EACH=: &>
fr=: +/"1 @ (=/)
PolyDice=:  [: <"1 [: |: [: >: [: ? ] $ [
NB. Weldon's example
NB. (i. 13) fr +/ EACH (e.&4 5 6 each) 6 PolyDice 12 4096
NB. Wolf's example
NB. +/"1 (>:i.6)=/>:?10000$6
NB. Tossing coins
NB. N=: 300
NB. TossNum=: >: i. N
NB. Heads=: +/\?N$2
NB. Ratio=: Heads % TossNum
NB. Diff=: TossNum - 2*Heads
NB. load 'plot'
NB. plot TossNum;Ratio
NB. TossNum;Diff
NB. Buffon's problem
rand=: (? % ]) @ ($&1e9)
Pi=: (2:*]) % [: +/ [: (</"1) rand ,. [: 1&o. 1.5708&*@rand
NB.   y: Number of repetitions (Pi 5000)
NB. Coupon collector's problem
cc=: * +/ @: % @: >: @ i.
NB.   y: Number of coupons
NB.   r: Expected sample size (cc 5)
ccsim=: 3 : 0
n=. y.
r=. i. 0
while. n > # ~. r do.
   r=. r, ?n
end.
>:r
)
NB.   y: Number of coupons
NB.   r: One simulation     (cc 5)
ccs=: (#@ccsim)"0 @ #
NB.   x: Number of simulations
NB.   y: Number of Coupons
NB.   r: Sample sizes        (10 cc 5)
NB. Buffon's problem
rand=: (? % ]) @ ($&1e9)
Pi=: (2:*)) % [: +/ [: (</"1) rand ,. [: 1&o. 1.5708&*@rand
NB.   y: Number of simulations    (Pi 5000)
```

J-ottings 45: Musical J-ers

by Norman Thomson

In the course of the last year or two Ken and I had occasional internet chats on the subject of temperament in music, based on a book called "Temperament – the idea that solved Music's Greatest Riddle" by Stuart Isacoff. (The word "temperament" was first used in this context round about 1500, and means, according to Chambers, "a system of compromise in tuning"). This is an excellent example of a technical subject, whose understanding and exposition is greatly facilitated by J. Putting this to the test, I scribbled down my thoughts from time to time with the idea of their eventually maturing into a J-ottings article, of which I think Ken would have taken note. Whether he would have liked it or not, alas I shall never know. No doubt he would have done it much better, but I hope that my attempts here may count as a small tribute to Ken.

In the Beginning ...

Pythagoras, he of the hypotenuse, also had strong ideas about music. He realised that a vibrating string when stopped at its middle point produces a note melodically identical to the original, only, in modern terminology, an octave higher. Call the melodic value of both these notes the tonic, so that advancing an octave is a way of "listening" to the fraction 1/2. If the string is now stopped at its 2/3 point the result is another note called the dominant which, when sounded at the same time as the tonic, produces a pleasant sound combination. From this starting point two separate experiments proceed.

Simple Fractions Sound Nice

The first experiment progresses by stopping a string at other fractional points with small integer numerators and denominators. Since the octave represents a full melodic circle which is repeated at 1/4 then 1/8 and so on, there is little point in considering stops other than those which lie between 1/2 and 1. The next "interesting" stop is thus at 3/4, followed by others at 3/5, 4/5 and 5/6. At this point all fractions with components of 6 or less have been exhausted, and in all cases pleasing sound combinations with the tonic are obtained. This experiment has thus provided a means of "hearing" the following range of fractions : 1/2 2/3 3/4 4/5 5/6 which incidentally are defined by the hook

```
   (%>:)1 2 3 4 5     NB. octave, fifth, fourth, third, minor third
0.5 0.667 0.75 0.8 0.833
```

The interval names in the comment are descriptions of *distances* from the tonic. The difference between intervals and notes is analogous that between the gaps in the fence as opposed to the fence posts themselves. The piano tuner twists strings, and the organ tuner tweaks pipes to produce *notes*, but the listener primarily hears *intervals*, that is relatives rather than absolutes. Also, unlike violinists, say, for whom stopping, and thus tuning, is dynamic in performance, keyboard players must in the nature of things complete each performance on an instrument tuned in advance.

Advancing to higher integers in the above sequence, any fraction involving a 7, that is 4/7, 5/7, 6/7 and 7/8 produces unpleasant sound combinations. As for 8s, there is just one ratio which has not been investigated, namely 5/8 which sounds nice, and corresponds to the interval called a minor sixth. All of the intervals 1/2 2/3 3/4 3/5 4/5 5/6 5/8 are "pure" in the sense that they can be related to pleasant sounds which arise from the *physical* properties of a bowed string or a resonating pipe.

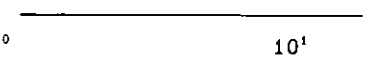
Calculating a Circle of Fifths

The second experiment concerns how *several* strings should be tuned in order to produce pleasing combinations of sounds, and so it is about intervals rather than notes. Since local stopping on a single string produces a range of notes it makes sense to think in terms of a larger interval as a basic unit for between-string tuning when introducing new strings. The dominant interval is a natural choice for a second string, as Pythagoras was aware, then the dominant of the dominant for a third string and so on. The stopping position for the dominant of the dominant is clearly at 2/3 of 2/3 = 4/9, which is outside the range 1/2 to 1. However, the first experiment showed that doubling the fraction lowers the note by an octave but makes no difference to its melodic quality of the note, in which terms it is equivalent to a stopping position of 8/9. Next repeat the experiment to find the dominant of the dominant of the dominant at 2/3 of 8/9 = 16/27 which does not need doubling since it is already in the range 1/2 to 1. To continue this process, use J to develop a compound verb "multiply-by-2/3-and-double-if-outside-1/2-to-1". To do this, observe first that compressing an infinite set of numbers into a finite range is already a familiar matter to anyone acquainted with scientific notation. Using logarithms, the fine detail of a real number is compressed into the range 1 (inclusive) to 10 (non-inclusive), while the exponent defines the wider territory within which the number lies. Thus, if a number is expressed as v e x then

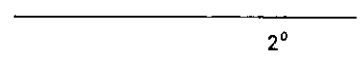
```
   x=.<.@(10&^.)  NB. exponent
   x 2999
3

   v=.%10&^@x     NB. value
   v 2999
2.999
```

In pictorial terms v compresses numbers into the space



Now return to the musical experiment with its "special" multiplication in which e.g. $(2/3)^2 = 8/9$, so that the result always remains in the range 1/2 to 1. The "compression region" can be drawn as



Call the process "musical arithmetic", which helps write the analogous verbs

```
   x2=.>.@(2&^.)  NB. 2-exponent
   x2 2r3
0

   v2=.%2&^@x2    NB. value of fraction converted to range (0.5,1]
   v2 1r8 2r3 4r9
1 0.667 0.889
```

In musical arithmetic the "nice" sounds as defined above, (or as Pythagoras would have more grandiosely called them "celestial harmonies") are inverses according to the following plan:

2/3 :	fifth	4/5 :	major third	3/5 :	major sixth
1/(2/3) = 3/4	fourth	1/(4/5) = 5/8	minor sixth	1/(3/5) = 5/6	minor third

Now apply the power adverb to the "musical multiplication" verb to continue the progression of fifths, and at the same time recording the notes which are reached form a starting point of C :

```
   v2 (2r3)^>:i.12   NB. progression of fifths
0.667 0.889 0.593 0.79 0.527 0.702 0.936 0.624 0.832 0.555 0.74 0.987
G     D     A     E    B     F#    C#    G#    D#    A#    F    C
```

After twelve iterations, a value 0.987 is obtained which is not too far from 1 representing the tonic.

Given that the numerators are powers of 2 and the denominators are powers of 3, there can never be any question of solving $(2/3)^k = 1/2$ exactly, and so it is reasonably satisfying to get as close as 0.987/2 in twelve steps. The musical interpretation is that progressing by intervals of a fifth twelve times take us through a cycle of notes which can then be made to repeat itself after a small adjustment to make the octave pure. This much was probably known to the Greeks of Pythagoras' time, and the tuning system based on it is known as "Pythagorean tuning". A natural place to make the adjustment necessary for a pure octave is at the final step, but it could be made at any intermediate step or indeed spread across several steps.

The next question concerns what would happen if the above exercise was repeated for 3/4 rather than 2/3. The primary interval in this case is the fourth, and the result is

```
   v2 (3r4)^>:i.12    NB. progression of fourths
0.75 0.563 0.844 0.633 0.949 0.712 0.534 0.801 0.601 0.901 0.676 0.507
F    Bb    Eb    Ab    Db    F#    B     E     A     D     G     C
```

that is, the same notes only in reverse order, some with different names, and finishing round about the high octave, value 1/2, rather than the low octave at 1.

The great riddle referred to in the opening sentence comes about because, for example, if the stopped positions on a string are obtained by Pythagorean tuning, by the time F comes round its value will not be quite the "pure" value of 3/4 as dictated by the physics of a single string. Likewise the second list shows that under tuning by successive fourths, G would be a shade impure. Similar considerations apply to the other notes, which in turn means that the intervals will differ from the intervals identified in the first experiment. More importantly, the adjustment noted above which is needed to make the octave pure at the twelfth and final step is called a "comma", or more specifically a "Pythagorean comma".

The Problem of D

So far six of the eight notes of the diatonic scale (that is the white notes on the piano in the scale of C) have been given places in the "simple vulgar fractions" scheme of things, the two remaining being D and B. Since these are symmetrically placed at either end of the octave, a discussion of one is automatically a discussion of the other, so focus on D. D is not consonant with C, so there is no physical "right" fraction for it, rather there are two candidates. The first comes from

considering the fact that D is one whole tone away from C, and there is already a whole tone represented, namely F – G, whose ratio is (2/3) / (3/4) = 8/9. The second candidate arises from the fact that in order to make D – A a pure fifth D must be set by solving (3/5)/x = 2/3, that is, at (3/5) / (2/3) = 9/10. (A harpsichord with two such D keys was in fact built in Holland in 1639, but did not prove particularly popular for what would seem to be obvious reasons!) The diagram below shows D set to 8/9, in which case solve (1/2)/x = 8/9 to obtain 9/16 as the symmetrically consistent choice for B.

1/2	9/16	3/5	5/8	2/3	3/4	4/5	5/6	8/9	1
C	B	A	G#	G	F	E	Eb	D	C
octave	7th	6th	dim 6th	5th	4th	3rd	dim 3rd	2nd	

Symmetry and the Note in the Middle

A full octave in the chromatic scale (that is including two tonics) has 13 notes, and thus 12 intervals, and thus has a middle note, namely F#, analogous to the "six o'clock" position on a clock face. Where does it appear in the above table? The answer is that it doesn't, because "half-way" on a multiplicative scale means the 1•2 position, so playing the interval C – F# on the piano is a way of "hearing" the square root of 2! Musicians call this the tritone, and it has also been called "the devil in music" on account of the difficulty for singers of pitching this interval. Further, the two progression series above show that under musical multiplication both $(2/3)^6$ and $(3/4)^6$ are approximations to $1\sqrt{2}$, one being about 0.005 above and the other the same amount below. On either side of "six o'clock", the interval of a fifth, C – G, consists of seven semi-tones, whereas a fourth, C – F, consists of five semitones, in which respect fifths and fourths are mirror images of each other. This also explains why the two progression series above are, in note terms, each the reverse of the other.

Adjusting the Scale

The following is another copy of the 1/2 to 1 region in which fractions are labelled with interval names rather than notes (dim stands for "diminished").

1/2	9/16	3/5	5/8	2/3	3/4	4/5	5/6	8/9	1
C	B	A	G#	G	F	E	Eb	D	C
octave	7th	6th	dim 6th	5th	4th	3rd	dim 3rd	2nd	

Under this scheme the whole tones D – E and G – A have values (4/5) / (8/9) and (3/5) / (2/3), both of which are equal to 9/10, which was the alternative candidate for D. Also A – B has the value 15/16. This means that there are different types of whole tone in this scale, with the result that, for example, the first three notes of "Three Blind Mice" become a melodic progression having unequal steps, e.g. 9/10, then 8/9. Also the ratios for the main consonant intervals, obtained in each case by dividing the value of the second note by that of the first in musical arithmetic, are

perfect fifths				major sixths		major third
F – C	G – D	D – A	A – E	F – D	G – E	F – A
2/3	2/3	27/40	2/3	16/27	3/5	4/5

The values of 2/3, 4/5 and 3/5 are consistent with those for the tonic C, but D would need to be 9/10 to make D - A a pure fifth in which case G –D would be 27/40. Similarly in the key of G# the major third is G# – C, ratio 1 / (5/8) = 4/5 and the fifth is G# – Eb = (5/6) / (5/8) = 2/3 both of which are pure. However, in the key of E the major third E – G# has the ratio (5/8) / (4/5) = 25/32 or 0.781 which is just a touch impure. And so one could go on. Once a set of strings, say, is tuned for pure concordances in key C, compromises must be made, not only for melodies and harmonies in the key of C itself, but even more importantly for melodies played in other keys. How best to make such compromises has engaged the minds of musicians since medieval times, which is the subject of the book referenced at the head of this article. The history of the debates on temperament is complex, but broadly speaking, the D problem gave rise to two solution streams, one called **just intonation** which tolerated differences in whole tone values as a price worth paying for purity of most major thirds, the other called **mean tone temperament** which is based on making whole tones uniform. "Just" in this context should be thought of as being a derivative of "adjustment". The notion of making adjustments to organ pipes or strings on keyboards may well date as early as the late 14th. century.

Equal Temperament

In the mid 16th century the concept of the equal-tempered scale emerged which has dominated Western music to this day. In equal temperament each of the twelve semi-tone intervals are equal on a multiplicative scale. Such tunings first found favour among lute players for whom other forms of tuning necessitated the undesirable feature of having frets at unequal distances for different strings. Again, J can clarify and quantify what musicians and musical historians mean when they discuss this topic. In an equal-tempered system (not to be confused with the well-tempered system, which is yet another ingenious tuning scheme

developed in the 17th century with similar objectives), the common ratio of the series of semi-tone values is, as a consequence of the definition, the reciprocal of the twelfth root of 2:

```
   ]e12=.%12%:2  NB. e12 is 1/12th root of 2, could also be 2^-%12
0.944
```

and so the stopping ratios required to go up the scale are given by

```
   e12^i.13       NB. well-tempered stop positions
1 0.944 0.891 0.841 0.794 0.749 0.707 0.667 0.63 0.595 0.561
0.53 0.5
```

The following series is the corresponding ordered version of the ratios under Pythagorean tuning:

```
   \:~ v2 (2r3)^i.13  NB. Pythagorean stopping positions
1 0.987 0.936 0.889 0.832 0.79 0.74 0.702 0.667 0.624 0.593 0.555 0.527
```

in which `0.987` is the approximation to the pure octave value of `0.5` in the equal-temperament system.

For ready comparison of tuning systems it is a near necessity to convert from a multiplicative scale ranging from 1 to 1/2 to an additive one from, say, 0 to 1200 in which each semitone interval is represented by 100 in an equal-temperament system. This converts

the multiplicative scale $\{1/2, \quad 2^{-(1/2)}, \; 2^{-(1/12)}, \quad 1\}$

to the additive scale $\{1200, \quad 600 \; , \quad 100, \quad 0\}$

Musicians call the unit which divides an octave into 1200 parts a "cent", and J readily provides the means of conversion

```
   cent=.1200&*@(2&^.)@%     NB. convert stop positions to cents
   cent %12 4 3 2 1.5%:2     NB. well tempered C# Eb F F# G#
100 300 400 600 800
   cent e12^.13
0 100 200 300 400 500 600 700 800 900 1000 1100 1200
```

Pragmatically, only people with exceptionally sensitive hearing would be able to detect a difference of 12 cents or less (that is about an eighth of a semi-tone), but most people would find differences of 20 cents or thereabouts harmonically unpleasing, although melodically acceptable, that is fine when notes are heard in sequence but not when they are played together.

The inverse adverb in J makes the above process readily reversible, that is, given a cent value, the stop ratio is immediately available:

```
   (cent^:_1)cent e12^i.13
1 0.944 0.891 0.841 0.794 0.749 0.707 0.667 0.63 0.595 0.561
0.53 0.5
```

The cent values of pure fifths, major thirds and major sixths are important numbers to keep in mind, and are calculated respectively as:

```
   cent v2 2r3 4r5 3r5 NB. cents for 5th. maj 3rd, and maj 6th
702 386.3 884.4
```

Thus the equal-tempered scale fails to achieve purity for fifths, major thirds and major sixths by margins of approximately 2, 14 and 16 cents respectively. By symmetry the values of the three complementary intervals in the diatonic scale (fourths, minor sixths and minor thirds) are simply 1200 minus the above values, as confirmed by

```
   cent v2 3r4 5r8 5r6 NB. cents for 4ths, min 3rds. 6ths.
498 813.7 315.6
```

Next establish the cent values of all twelve points on the Pythagorean chromatic scale, that is tuning by successive fifths, and then arranging the notes in ascending scale order (n.b. I have taken a few minor liberties in editing the J output in the interests of greater clarity):

```
   /:~cent v2 (2r3)^i.13   NB. conversion of Pyth tuning to cents
0 23.5 114 204 318 408 522 612 702 816 906 1020 1110
C1  C2   C#  D  Eb   E   F   F#  G   G#  A   Bb    B
```

All of this can be embodied in a single verb `ptune` which takes as left input a fraction in the range (0.5 - 1) which is used to define a fifth, and as right input the powers to which this is to be raised:

```
   ptune=.cent@v2@^    NB. x. defines a 5th; y. is list of powers
   ]pscale=.}./:~2r3 ptune i.13
23.5 114 204 318 408 522 612 702 816 906 1020 1110
oct             3rd  4th      5th      6th
```

In order to avoid too much clutter, it is best to focus on just the major thirds, fourths, sixths and octave:

```
   2r3 ptune&>< 1 4 3 12 NB. Pythag'n tuning for various iterations
702 407.8 905.9 23.46    NB. cents for 5ths(input),3rds,6ths,comma
```

Increasing the input value of 2r3 means reducing the cent value of around 702 cents, or in musical terms, flattening the fifth. In particular flattening it to around 697 cents (an imperceptible difference to most listeners) gives nearly pure thirds and sixths at the cost of a severely impure octave :

```
   (cent^:(_1) 697) ptune&>< 1 4 3 12
697 388 891 1164
```

The value of 23.5 in `pscale` (in musical terms about a quarter of a semitone) represents the Pythagorean comma by which tuning by repeated pure fifths "misses" the octave whose purity in *any* tuning system is sacrosanct. "Comma" in music means essentially *discrepancy*, and when used unqualified it means the Pythagorean comma. Since F is the second last note to be tuned in a Pythagorean system, this is roughly the interval by which the fourth C – F is impure within a equal-tempered system. When a piano tuner tunes a set of twelve strings, his choice of techniques is analogous to finding ways of fitting a set of bricks lengthwise into a closed gap whose length is not quite equal to the sum of the lengths of the bricks. The amount of excess or shortfall is the comma associated with the technique.

In a Pythagorean scale the excess at the octave is 702-23.5 = 678.5 cents. Two notes at this interval produce a discordant sound known since mediaeval times as the "wolf fifth", presumably because of its supposed likeness to a wolf braying. Of course it is possible, in principle at least, to spread the corresponding comma adjustment throughout the scale to provide a "smoothed" scale.

An octave can also be considered as the sum of three major thirds and of four minor thirds, views giving rise to discrepancies of 1200 – (3×386.3) = 41.1 below and (4×315.6) - 1200 = 62.4 above respectively, which are also considered as commas. Yet another type of comma is motivated by the fact that the major third is in some sense a more "beautiful" consonance than the "fifth" (think of songs in which soprano and alto voices proceed in a blend in parallel thirds). In Pythagorean tuning the note E required for the interval of a third is encountered after four steps (C – G – D – A – E), so in the same way that `%12%:2` "equalises" the octave into twelve semitones, so `%4%:5` (the reciprocal of the fourth root of five) equalises the third into four equal fifths. The value of this quantity is 0.66874, or in cent terms

```
   cent e4=.%4%:5
696.578
```

Flattening the fifth to this value produces increasing discrepancies from Pythagorean tuning as follows :

```
    (2r3 ptune i.5)- e4 ptune i.5
0 5.38 10.8 16.1 21.5
```

By the time E is reached at the fourth step there is a shortfall of 21.5 cents, which musicians call the **syntonic comma**, or sometimes the "comma of Didymus" (*didymos* is the Greek word for a twin, presumably because notes a third apart are in some sense like twins). Just as the Pythagorean comma is the adjustment needed to equalise the octave, so the syntonic comma is the adjustment needed to equalise the third. The effect of continuing this tuning procedure until the octave is reached is a shortfall of 41 cents:

```
    e4 ptune 12
1159
```

Another way of looking at `pscale` is to calculate intervals (differences) rather than note values. An appropriate adjustment is made to accommodate an embracing pure octave:

```
     2-~/\0,(}.pscale),1200  NB. Pythagorean scale intervals
 114 90 114 90 114 90 90 114 90 114 90 90
C       D       E   F       G       A       B   C
```

Every semitone here is worth one of two values, namely 90 cents or 114 cents. If now the two semitones in the diatonic scale (that is the C scale without any black notes) are equalised at 90 by interchanging the 114 and 90 between E and F#, then the five whole tones in the diatonic scale are also equalised at 204 cents. (5×204) + (2×90) = 1200, which confirms the purity of the octave. The difference of 24 cents between some semitones and others is not generally objectionable in a melody, and harmonically speaking all semitones are discordant anyway!

As an aside, the result of tuning by successive fourths is that each note is a comma smaller than its counterpart in fifths tuning, so that there is a shortfall of 23.5 cents rather than an excess when the octave is reached:

```
    cent \:~v2 3r4^i.13
0 90.2 180 294 384 498 588 678 792 882 996 1086.3 1177.5
```

Just Intonation

Just intonation systems are developments of the ideas of the preceding section. Some of these ideas were known in the time of Ptolemy in the second century A.D., hence the occasional use of the term "Ptolemaic tuning" as a synonym. They gave rise to vigorous debate amongst musical theorists as early as the fifteenth century.

To illustrate, consider a scheme of intervals which accepts C – D as 204 cents as above, corresponding to a harmonic value of 8/9, but make the next tone D – E equal to 386 - 204 = 182 cents, a reduction of a comma. Reducing the interval G – A by the same amount simultaneously adjusts both the F – A and G – B thirds to the pure value of 386. This leaves the two diatonic semitones to take up the slack of 44 so each becomes 90 + 22 = 112. Now consider the chromatic notes. D – F# is currently 182 + 112 + 114 = 408, in excess by a comma. Switching the semi-tone values between F and G, and changing them slightly from 114/90 to 112/92 makes D – F# pure, as are also F# - A and G# - B. This little bit of ingenuity leads to the scheme

```
   ]just=.92 112;90 92;112;92 112;92 90;112 92;112
+------+-----+---+------+-----+------+---+
|92 112|90 92|112|92 112|92 90|112 92|112|
+------+-----+---+------+-----+------+---+
C      D     E   F      G     A      B   C
```

in which there are two types of whole tone, some 8/9 and others 9/10. It is now possible to use J to observe the effects of this particular just tuning on the principal intervals based on all the different possible starting notes:

```
   2+/\13$;just   NB. whole tones
204 202 182 204 204 204 204 182 202 204 204 204
C   C#  D   Eb  E   F   F#  G   G#  A   Bb    B
   7+/\18$;just   NB. fifths
702 702 680 702 702 702 702 702 700 702 702 702
   4+/\15$;just   NB. major thirds
386 406 386 408 408 386 406 386 406 408 408 406
   9+/\20$;just   NB. sixths
884 904 884 906 906 906 904 884 904 906 906 906
```

All but one of the fifths remains pure, but most of the major thirds and sixths are sharp by 22 cents, that is, within rounding, a syntonic comma. As with Pythagorean tuning there are considerable differences in size between some semi-tones and others.

Other theorists had different ways of getting around the "pure thirds" problem, and because of the *ad hoc* nature of such systems, just systems are sometimes referred to as irregular temperaments. (It is no accident that the word "temperament" shares the same Latin root as "tamper"). It is doubtful whether such systems were much applied in practice to harmonised music after the sixteenth century.

Mean-tone Temperaments

Mean-tone systems were designed primarily for keyboard instruments, and overcome the objection of different cent values for the two whole tones in a major third by replacing them with their average value, hence the name "mean-tone". As already noted, the pure third measures 386.3 cents, so that the two tones which comprise it have a *mean* value of 193.15 cents (cf. 204 in the just system). This mean is the size of all the whole tones in this system. Carrying out the sort of accountancy in the previous paragraph means that the values of the semi-tones intervals must be half of 1200 - (5×193.15) = 117 cents.

This in turn means that those semi-tones which are not part of the diatonic scale must have the value 193 - 177 = 76 (cf. 92 under the just tuning described above) leading to a cent scale

```
+------+------+---+------+-----+-------+---+
|76 117|117 76|117|76 117|76 117|117 76|117|
+------+------+---+------+-----+-------+---+
C       D       E   F       G      A       B   C
```

in which there is nearly a quarter-tone difference in values between some semi-tones and others. (n.b. the numbers in the above list total 1199 due to a small rounding effect.)

But we can go further than this. From the above

```
   /:~2r3 ptune i.13
23.5 114 204 318 408 522 612 702 816 906 1020 1110 1200
oct              3rd  4th      5th      6th
```

shows that the purity of the fifth is obtained at a quite considerable over-valuation of 408 for the third. The effect of flattening the fifth by lowering its cent value has already been investigated. Another way to measure the effects of such flattening is to express it as a fraction of the syntonic comma. A natural fraction to use is 1/4 since that spreads the effect equally over the four Pythagorean steps needed to reach a third.

```
   /:~(cent^:(_1)702-21.5%4)ptune i.13
0 76.36 193.2 269.6 386.5 462.8 579.7 696.6 773 889.9 966.2 1083 1159
```

This time the octave is undershot at 1159, so that although the third is nearly pure at 386.5, the observed comma at the octave is 41. A whole range of possible compromises can be tested by, for example

```
   ((cent^:_1)&><702-1r3 2r7 1r4 2r9 1r5 1r6 3r14*21.5)ptune&><1 4 3 12
                    NB. fraction of observed comma
                    NB. syntonic comma,  at octave

694.8 379.3 884.5 1138      NB.   1r3              62
695.9 383.4 887.6 1150      NB.   2r7              50
696.6 386.5 889.9 1159      NB.   1r4              41
697.2 388.9 891.7 1167      NB.   2r9              33
697.7 390.8 893.1 1172      NB.   1r5              28
698.4 393.7 895.2 1181      NB.   1r6              19
697.4 389.6 892.2 1169      NB.   3r14             31
```

The first three columns above represent fifths, major thirds and sixths which should be compared as before with the pure values of 702, 386.3 and 884.4 respectively. The final column is subtracted from 1200 to give the *observed* comma at the octave. The third row confirms the purest thirds at 1/4 comma, whereas the first row gives almost pure sixths at the expense of a big comma at the octave.

Frequencies

In terms of frequencies life is even simpler, since for Pythagorean tuning the relative frequencies of notes in the scale are now compressed into

$$\frac{2^0}{\qquad\qquad 2^1}$$

```
fx=.<.@(2&^.)        NB. exponent
fv=.%2&^@fx          NB. value
```

Using the verb `fv` of course requires some correction due to the comma effect and the practical requirement that successive octaves should have values 1 and 2 :

```
   /:~fv 1.5^i.13
1 1.014 1.068 1.125 1.201 1.266 1.352 1.424 1.5 1.602 1.688 1.802 1.898
C   C'    C#    D     Eb    E     F     F#    G   G#    A     Bb    B
  octave dim 2nd 2nd dim 3rd 3rd 4th dim 5th 5th dim 6th 6th dim 7th 7th
```

Under equal temperament the corresponding frequencies are

```
   /:~(%e12)^i.12
1  1.059 1.122 1.189 1.26 1.335 1.414 1.498 1.587 1.682 1.782 1.888
```

Comparison of Systems

The following table summarises in cents the values of the notes of the chromatic scale of C in some of the systems of tuning considered in detail above, prior to comma adjustment to make the octave pure :

	C	C#	D	Eb	E	F	F#	G	G#	A	Bb	B	C
eq-temp:	0	100	200	300	400	500	600	700	800	900	1000	1100	1200
Pyth :	0	114	204	318	408	522	612	702	816	906	1020	1110	1223
just :	0	92	204	294	386	498	590	702	794	884	996	1088	1200
1r4 mean	0	76	193	270	386	463	580	697	773	890	966	1083	1159

There is nothing in the above which cannot be found in, say, *Encyclopedia Brittanica*, or *Grove's Dictionary of Music and Musicians*. However, the accounts there are not altogether easy to understand, and exposition in J would have helped me greatly. Perhaps like so many others who have followed in Ken's footsteps, I am just temperamentally inclined towards J...

At Play with J: Token Counting

by Eugene McDonnell

One measure of the difference between APL\360 and J is the number of tokens needed in a function to do the same work. I'll discuss two examples, comparing well-written, but rather old, APL solutions, to well-written J solutions.

Direct from Atomic

The first APL example is a marvelous function written by Luther Woodrum, that appears on page B.11 of the *APL\360 User's Manual* of 1968. Luther is best known to me by his design and implementation of the original *upgrade* and *downgrade*. His function *PERM*, below, is given a left argument *A*, the length of the permutation to be constructed, and a right argument *B*, the anagram index of the permutation to be constructed. It uses an algorithm first discussed, I believe, by D. H. Lehmer of the University of California in Berkeley. Here it is:

```
     ∇ Z←A PERM B;I;Y
[1]    I←⍴Z←1+(⍳A)⊤B-1
[2]    →0×⍳0=I←I-1
[3]    Z[Y]←Z[Y]+Z[I]≤Z[Y←I+⍳A-1]
[4]    →2
     ∇
```

Disregarding the header, footer and line numbers, this has 55 tokens. What does it do? Let's suppose that *A* is `9` and *N* is `288918`. Then the result *Z* is

```
7 1 3 2 6 4 0 5 8
```

and this is the 288918th permutation of order 9.

Here I should tell you that much of the material that follows is taken from my *At Play With J* article in *Vector 12.1* (July 1995). This is not surprising, since it deals with the subject of Luther's function.

The first step is to make a function that gives the factorial digits of permutations of length *A*. Luther's function uses origin 1, and that obfuscates things, so I'll use the more suitable 0-origin indigenous to J. To give you some idea of what the factorial digits number system is like, here are the six factorial digits in the system for three:

```
   fdb =: >: @ i. @ -

   (fdb 3) #: i. ! 3
0 0 0
0 1 0
1 0 0
1 1 0
2 0 0
2 1 0
```

You can see the regularity in the rows. Notice also that every row ends in zero. This is true for all factorial digits systems.

In *PERM*, line 1 gives *Z* the factorial digits value for *B*.

```
   fdb 9   NB. fdb n yields the radix digits of order 9
9 8 7 6 5 4 3 2 1
   (fdb 9) #: 288918
7 1 2 1 3 1 0 0 0
```

Lines 2 and 4 control the executions of line 3, so that line 3 is executed only as long as *I* is positive. Line 3 can be defined as function g :

```
g =:  [ , ] + ] <: [    NB. left , right + right >: left
```

Here I'll have to pause, and to point out that what I'm doing with g is taking the clutter out of *PERM* line 3. What we've done so far reduces line 3 from 26 tokens to 7, yet it does precisely what line 3 does. Perhaps if I show the successive uses of g you'll get the idea:

```
                    0
                  0 1
                0 1 2
              1 0 2 3
            3 1 0 2 4
          1 4 2 0 3 5
        2 1 5 3 0 4 6
      1 3 2 6 4 0 5 7
  7 1 3 2 6 4 0 5 8
```

Successive lines are formed this way: for example, given line

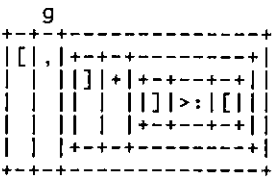
```
1 4 2 0 3 5
```

the next line is formed by beginning with the corresponding factorial digit, in this case 2, and following with the previous line in which each item greater than or equal to 2 has 1 added to it.

```
2 1 5 3 0 4 6
```

Notice that each line is a permutation.

The function g is made of three forks:



The three forks are:

```
fz =: ] >: [
fy =: ] + fz
fx =: [ , fy
```

Here's a J function, a functional duplicate of *PERM*:

```
   sr                              NB. standard form from reduced
/:@/:@,/
   ra                              NB. reduced form from atomic
([: fdb [) #: ]
   sra =: sr@ra f.                 NB. standard from atomic
   sra
/:@/:@,/@(([: >:@i.@- [) #: ])

   ;: sra                          NB. tokens of sra
+--+---+-+-+-+-+-+-+--+--+---+-+-+-+-+--+-+-+
|/:|@|/:|@|,|/|@|(|([:|>:|@|i.|@|-|[|)|#:|]|)|
+--+---+-+-+-+-+-+-+--+--+---+-+-+-+-+--+-+-+

   # ;: 5!:5 < sra                 NB. count of tokens in sra
20
```

The function sr uses an identity I found March 9, 1970, when I was looking at Luther's *PERM* once more:

```
   N =: 7
   P =: 1 3 2 6 4 0 5 7
   (/:/:N,P) -: (N,P+N<:P)  NB. double upgrade matches addition
1
   N,P+N<:P
7 1 3 2 6 4 0 5 8
   /:/:N,P
7 1 3 2 6 4 0 5 8
```

So double-upgrade can take the place of Luther's line 3, and `sra` squeezes *PERM* down from 55 to 20 tokens.

Pyramigram

In *APL Quote Quad* 11.1, September, 1980, I asked for solutions to a problem posed by Linda Alvord, of Scotch Plains Fanwood High School in Scotch Plains, New Jersey. Here it is:

> Write an APL function *PG* that takes a scalar integer argument from 1 to 26 and produces a rectangular character matrix containing a pattern like this:
>
> ```
>     PG 5
>      Q
>     W Q
>    Q E W
>   R Q W E
> Q E R T W
> ```
>
> In each row *r* there are *r* randomly selected and randomly ordered letters, separated by single spaces, arranged to form an equilateral triangle. The (*n*-1) letters in row *n*-1 are selected from the *n* letters in row *n*.

This was one of the most popular problems I'd ever given, and there were a wide variety of solutions, including ones by some fairly gifted programmers, but one stood out from the rest, from Roger Hui. It took me several hundred words to describe what his function did. Roger recently told me that he had written his function without having access to an APL implementation. His function `PG` was not written in the conventional way that a function was defined in APL\360. Instead, it uses the alpha-omega form introduced by Ken Iverson, in which the left and right arguments are denoted by ⍺ and ⍵. Here is a J function which uses the same algorithm as his from 1980:

```
PG =: ([:i.[:-])(|."0 1)1j1"_ #"0 1(([:/:(([:-/\]-
~[:|.[:i.[:+:])#([:i.[:+:])
*[:*:])+[:?~[:*:]){[:,(],]){.(('ABCDEFGHIJKLMNOPQRSTUVWXYZ'"_{~]?26"_)
{.~[:-.[:+:])$~],[:+:])$~],]
```

I won't dwell on this version other than to say that (1) it has 103 tokens and (2) it is in tacit form. The details are discussed in the cited issue of *APL Quote Quad*. It is a truth universally acknowledged that a good programming language, worked over and pondered over for a sufficiently long time by the same people who had produced the original, may very well show advantages over the original. Thus I sent an email to the J Forum list, and messages to key people in the APL community, for new solutions to the problem. I made it clear that the degree of improvement in expressiveness, as measured by token count, would be the criterion used. I received new solutions from the J and the APL communities. The shortest token count among the numerous APL solutions was 30, and there were several that used up to 60 tokens. The shortest J solution, by Roger Hui, was 20 tokens long, so I'll discuss that one only. This is it:

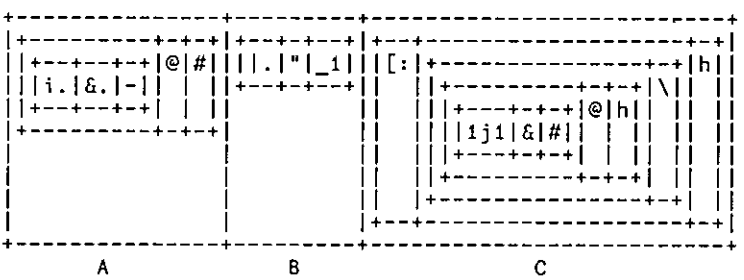
```
h   =:  /: # ? #
pyr =:  i. &. - @ # |. " _1 [: 1j1 & # @ h \ h
```

His `h` has 4 tokens, and `pyr` has 16, totalling 20.

The subfunction `h` is a hook. It randomizes its argument. Its first function is `/:` and its second function is the fork `# ? #` .

```
h 'qwert'
ertqw
```

The function `pyr`'s structure may best be seen using *box* display:



The three outer boxes, A, B, and C, tell us that we have a fork. Box C yields the object to be rotated, and is the most interesting part. It produces a right-triangle version of the equilateral result required.

Its copy function # is a dyad. By bonding (`&`) it on the left with `1j1` we create a monad that makes one copy of its argument item and follows this with one fill item. A further function is made by using atop (`@`) between the aforesaid monad

and h. One further step is to apply the prefix adverb (\ ) to this combined function to yield `1j1 & # @ h \`. One last step supplies cap (`[:`) to the left of this and h to the right, and we have a function that randomizes the argument `'qwert'`, then randomizes each prefix, and provides a space after each item of the prefix, like this:

```
   ([: 1j1 & # @ h \ h)  qwert
r
r t
t q r
e r t q
w q e r t
```

Rotation (box B) uses reversal, item rank (`|.  _1`) so that scalar items from the left argument (box A) rotate list arguments from the right argument (box C).

Box A creates the left argument. It makes good use of negative arguments to `i.` and of *dual*. Ordinarily `i.  y` yields a descending list of positive integers, but we want to do right rotations, which require a negative value. That's the reason for using *dual minus*. (`&. -`).

```
   i. _5
4 3 2 1 0
   (i. &. -) 5
_4 _3 _2 _1 0
```

Thus, the whole result is given by 20 tokens:

```
   (i. &. - @ # |. " _1 [: 1j1 & # @ h \ h)'qwert'
    e
   r e
  r e t
 e q r t
r w q t e
```

From 103 tokens in 1980 to 20 in 2005, and by the same author, is a huge reduction.

Enigma 1368 from New Scientist

attacked by Adrian Smith (adrian@apl385.com)

Background

The *New Scientist* magazine runs a regular series of little combinatorial puzzles, most of which are clearly designed with computer solutions in mind. Sometimes you just look at them and think "That looks like a one-liner" and don't actually bother to code them up, and sometimes you get sufficiently interested to flip open your laptop for some gentle fireside programming. This being the Ken Iverson tribute edition of Vector, I thought I could make a small contribution by logging my attack on one of these puzzles in APL. Why do I think this is appropriate? Basically because without Ken's work, I would never have used the computer to have fun. Writing C# is something you do to earn a living; fooling around in an APL session is something you can do of an evening, with Bach on the radio and a nice log fire at your feet. So much of today's software leaves you feeling frustrated, angry and stressed; APL and its many descendants are doing their little bit to redress the balance, and Ken deserves all our thanks for that.

The Puzzle

EEN, VIER and NEGEN are Dutch for 1,4,9. Replacing letters by numbers we get three perfect squares (no number starts with zero). What is the value of the square root of `( EEN × VIER × NEGEN )`?

The Attack

This one is interesting, in being largely just a logic puzzle rather than a huge combinatoric. It is a nice example of where a 'desk calculator' APL can be really helpful in expressing the tests you need to reduce the lists of numbers to a manageable size. No programming was required at all here, just typing.

The first stage is obvious – how many perfect squares are there with the pattern EEN. I immediately thought of 225, but let's list them to be sure:

```
      (10↓⍳31)*2
121 144 169 196 225 256 289 324 361 400 441 484 529 576 625 676 729
    784 841 900 961
```

Damn, forgot about 441. However there are only two of these, so we have only two choices for E and N. Let's go with E=2, N=5 and see how we fare. Here are the candidates for NEGEN, created by copy/paste in the session:

```
    2⍕ 52025 52125 52225 52325 52425 52525 52625 52725 52825 52925 * 0.5
228.09 228.31 228.53 228.75 228.97 229.18 229.40 229.62 229.84 230.05
```

Nope, none of those look very integer. Let's try E=4, N=1 and play the same game:

```
    2⍕ 14041 14141 14241 14341 14441 14541 14641 14741 14841 14941 * 0.5
118.49 118.92 119.34 119.75 120.17 120.59 121.00 121.41 121.82 122.23

    121*2
14641
```

Got it! Now we just need a 4-digit perfect square with 4 as the third digit. There are only 9000 or so candidates, but at this point a bit of computer assistance is quite welcome!

```
      qq← (4=( 10 10 10 10⊤ (⍳9999))[3;])/⍳9999

      ⍴qq
1000

      qq/⍨←qq>1000
      ⍴qq
900

      qq/⍨ {⍵=⌊⍵} qq*0.5
1444 1849 3249 3844 5041 6241 7744 8649
```

Visual inspection (and the backspace key) can do the rest. Anything with repeated digits is out of it, for starters. Then we can remove anything with 1 or 6 as these are already allocated other letters. So the answer is ...

```
 ( 3249 × 441 × 14641 )*0.5
144837
```

No, I didn't win the £15 book token, but at least I could paste my entry directly from the APL session to save a bit of typing in Outlook. It really was rather a simple one, and they must have had plenty of correct solutions.

REMEMBERING KEN IVERSON

Obituary: Kenneth E. Iverson

from the Toronto Globe and Mail 22 October 2004

Noted Computer Scientist

Kenneth E. Iverson, a pioneer in the field of computer science, died on Tuesday, October 19th in Toronto, Canada. He was 83. He is survived by his wife of 58 years, Jean (née Nicholson); three sons, Eric (Suzann), Paul and Keith (Marcella); daughter Janet Cramer (Kevin); foster-daughters Robin Dick and Sherry Matusky; and five grandchildren.

Born on a small farm in Camrose, Alberta in 1920, he served in the Canadian military during World War II. Dr. Iverson earned a B.A. in Mathematics and Physics from Queen's University and M.A. in Mathematics and Ph.D. in Applied Mathematics from Harvard University. While on the faculty of Harvard, Dr. Iverson helped establish the first graduate course in computer science and also developed a concise mathematical notation that formed the foundation for APL (A Programming Language). He then joined IBM in 1960.

While at IBM, Dr. Iverson made an historic contribution to computer science by developing APL into an interactive programming language that was used widely in academic and commercial applications. An original thinker and noted scholar, he was named an IBM Fellow in 1971. For his efforts in mathematics and computer science, Dr. Iverson received in 1979 the A.M. Turing Award given by the Association for Computing Machinery, the most prestigious award in computer science. He was awarded the Harry M. Goode Memorial Award in 1975 in recognition for his conception and development of APL and named by the IEEE Computer Society in 1981 as a Computer Pioneer Charter Recipient for his efforts in the creation and continued vitality of the computer industry. In 1998, he received from York University an honorary degree of Doctor of Science.

During his career Dr. Iverson worked in various IBM research facilities in the northeastern United States before moving to Toronto, Canada in 1980 to join I.P. Sharp Associates, a timesharing computer system provider. In recent years Dr. Iverson was involved in the development and implementation of the "J" programming language with Jsoftware Inc. Dr. Iverson's love of language and teaching were significant factors in his lifetime work of trying to impose a grammar and discipline on the language of mathematics.

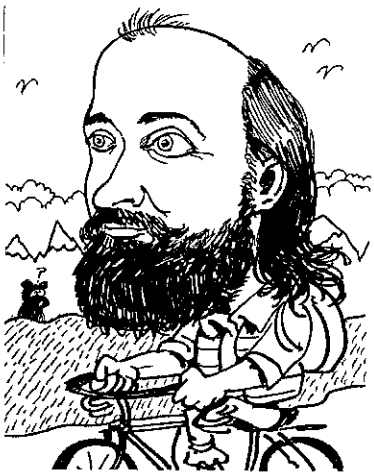
Memories of Ken

From Linda Alvord

Meeting Ken Iverson could cause mental transformations. Your mental shards of ideas might experience changes much as your laundry does in a washing machine. First all your notions about a topic slosh around in your head, back and forth in a washing cycle that confronts inconsistencies and redundancies. Then at some moment, a spin cycle rearranges everything. Conflicting concepts and incongruities confront each other in an endless swirl. A moment of absolute silence changes everything. Excess ideas are gone. And a rinse cycle reveals unifying concepts and unseen connections with amazing clarity. And thus, your 'Tools of Thought' are clean, refreshed, and ready for use.

Thankfully I had the opportunity to spend time with Ken and experience these kinds of changes. I also have his legacy of APL and J and a vast array of books, articles and labs to lead my thinking in new directions. Mostly, I treasure the memories of his presence.

From Bob Bernecky



I am going to talk about a teacher.

In early 1971, I told a headhunter in Toronto that I was looking for a job where I could work my own hours, work on compilers, wear jeans, and I didn't care about the pay. He told me: "There's only one place in town like that." I ended up talking with Ian Sharp, who told me that I'd "have to talk with Roger." I found Roger Moore in the middle of staring at a complex piece of code and, rather than conducting an interview, he and I ended up talking for an hour or so about the code, at which point I had to go to another interview. I asked him about the job interview, at which time he said that I was hired.

Shortly thereafter, I took a 20% pay cut to join the

development group at I.P. Sharp Associates, working on ways to break the IPSCOBOL compiler that was nearing completion. I used APL, a language I had seen before, but never used in practice, to generate and evaluate test suites for the large number of cases required by COBOL's numerous data types – Packed Roman Numerals to floating-point Aztec conversion and the like.

I soon noticed that my evaluations were taking inordinate amounts of computer time, and eventually narrowed the problem down to poor performance of the "indexof" primitive, aka "dyadic iota", in the APL interpreter. When I pointed this out to Roger, he said "Fix it", at which time I became an APL implementor.

Sometime in next year or so, I had the honor and great luck to meet Ken Iverson. At the time, I had no idea how greatly Ken's ideas would influence the course of my thoughts, modest future achievements, and career.

Ken and I talked on many occasions, but it was only around 1980. when Ken left IBM and joined my APL Design Group at Sharp, that we began to work closely with one another on language design issues. We had daily blackboard [There are those who call these things chalkboards, but we know they are not made from chalk.] discussions, whose lively nature was dampened only by less enthusiastic colleagues asking us to stop shouting.

Terminology was a fundamental focus in our work, for didactic reasons as well as for the great power that we, like the ancient sorcerers and sorceresses of yore, knew arises from knowing a thing's true name. Often, merely changing the name of a verb, adverb, or conjunction, as when "reduce" became "insert", would give us insight into simpler, more powerful concepts.

Our quest for, and delight in finding, the right word meant that we always had a dictionary or thesaurus at hand. We shared this delight with many others in the community, including Larry Breed, Eugene McDonnell, Bob Smith, and Donald McIntyre. Yet, Ken was adamant, in the sense that McIntyre would approve of, about more than terminology: he viewed mathematical rigor as a vital force in programming language design. However, that view was always tacit – I don't recall him ever saying it, yet, it was an *eminence grise*, omnipresent and powerful in our design talks, but unmentioned.

Around 1999, I tried to evoke that eminence, and wrote, in a paper called "Ergonomics and Language Design," these three design rules for human-centered programming languages:

Few rules

Simple rules

Consistent rules

These rules, first embodied in APL, then generalized and simplified in J, comprise the palette from which Ken created his art. As with all great art, simplicity produces beauty and insight.

As time passed, I began to comprehend that beauty and insight, and to recognize the power and brilliance of Ken's understanding of the need to design programming languages for people, rather than for computers. His understanding produced concepts that, like a carefully tended garden, have blossomed with time, creating unexpected delights for people all over the earth.

Ken's mathematics will play a key role in taking programming from being a craft to being an engineering discipline, by helping us to create, at long last, reliable computer programs. The power of array languages like APL and J have given us the concepts we need to construct tools to analyze programs and make assertions about their behavior, much as mechanical engineers analyze structures, for the same reason.

Ken Iverson's Tools of Thought comprise a wonderous gift that he, like so many other great teachers and mathematicians, bequeathed to all those who come after him.

Ken taught us much and gave us much; I believe the world still has much more to learn from his teachings.

This is a slightly cleaned-up version of what I said at the Kenneth E. Iverson Memorial at OISE Auditorium, Toronto, 2004-11-18

From Eugene McDonnell

Ken at Harvard

Just today I tried to remember the economist that Ken worked for at Harvard – and a Google search led me to the right man: Wassily Leontief – the originator of the input-output matrix for the economy of nations. Leontief was on Ken's PhD committee – and it must have stimulated his interest in matrices. Leontief won the 1973 Nobel Economics medal for this work.

In his (French) obituary Sylvain Baron mentioned Ken's PhD thesis (1954) "Machine Solutions of Differential Equations", which topic I imagine was suggested to him by Howard Aiken – the developer of three or four Harvard relay computers during WWII. I once made an attempt to read this – to no avail. I had studied differential equations as the last portion of Integral Calculus while I was in the Army (1945) – and could solve differential equations with my army-issued K&E loglog duplex decitrig polyphase slide rule – and the final test was three differential equations. I held up my hand and asked Dr Downing if we could use our slide rules, and he said yes – so I handed in my twice-checked test paper in ten minutes, while the rest of my fellow soldiers sweated for two hours.

I remember being told that http://www.cs.unc.edu/%7Ebrooks/brooks_cv.htmlFred Brooks was Ken's teaching assistant, and this led to their partnership in writing "Automatic Data Processing", the first edition of which used the IBM 650 as the model machine – and I think they had a formal description of the machine in the book.

And I think it was Brooks who, after he had left Harvard to work at IBM, recruited Ken to come work there too – after Ken didn't get tenure at Harvard. In his introduction to *A Programming Language* Ken mentions the large debts he owed to Aiken and Brooks.

While at Harvard Ken wrote a report to Bell Labs which was the earliest publication using his notational schemes. I remember that the specification arrow pointed to the right!

While still at Harvard, in 1960, Ken wrote "The Description of Finite Sequential Machines", which is the first place in which the notation for floor and ceiling appears. This was read by Mike Montalbano, then at Stanford, who had worked with George Dantzig, the developer of the simplex method for linear programming at the National Bureau of Standards, in Washington, D.C. He writes

> The page had an illustration that, in a few short lines, described George Dantzig's simplex algorithm simply and precisely. That was the overwhelming, crucial experience. In the previous thirteen years, I had participated in so many murky discussions of what was here presented with crystal clarity that I knew that what I was reading was of enormous significance to the future of computing.

I had a fellowship at Harvard 1949-1951 and left before Ken began to study there. My major study there was Dante's Divine Comedy – in Italian.

From Arthur Whitney

The acts of great men deserve acts, not words, in their honour. So I'll keep this short.

In the fall of 1950, at Harvard, a math professor came up to my dad, and said, with typical American enthusiasm and shaky geography, "There's another student here from Saskatchewan!"

Well, they were both from Alberta. And they became very good friends. It's well known that a lot of programming languages come from Alberta. Probably less well known is the fact that good languages, like APL, come from northern Alberta. Java is from Calgary.

Ken and my dad had a lot in common. They were both born in 1920 in small towns in northern Alberta, served in the Royal Canadian Air Force during the war, won scholarships to Harvard Graduate School. And they were both gifted mathematicians.

It was very lucky for me that they were friends because when my father died in the Sixties I adopted Ken even though he lived three thousand miles away. For the last 35 years, every year or two we've visited, and I've really enjoyed his friendship.

Ken was extremely gracious, dignified and intelligent. And he had a sense of humour. One time I was complaining about an impertinent customer who was making suggestions, and he said, "You know, Arthur when I worked at IBM, I

learned that it's important to listen to your customers; and it's even more important to disregard what they say, and do the right thing."

Another time when I was concerned about getting credit for some idea I had, he said, "Arthur, if it's a really good idea, you're going to have to shove it down people's throats. Don't be worried about getting credit." Or when I mentioned "computer science", he said, "Any field of study that has the word science in it probably isn't." This is how mathematicians make friends.

Ken was a wonderful combination of pragmatism and wit, and he had the conversational style of Socrates. An evening with Ken was very exciting, and could be a little stressful. Declarative statements were questioned and had to be justified. "Compared to what?" was a common refrain. But some statements are easy; they require no justification. APL and J are elegant languages. And Ken was a great man, and I will miss him.

Expository Programming

from Paul Berry

Well, quite simply, meeting Ken Iverson changed my life. I'd been a psychologist, but after I met Ken and Adin Falkoff in 1966, I knew what I'd rather do, and I've worked in computing ever since, though I think I've retained a psychologist's sympathy for the human user.

When I joined IBM, the company was enthusiastic about the prospects for using computers in primary and secondary schools. Iverson too believed that the language he was devising could play a huge role in education; but the vision Ken had was quite different from what was imagined by most of IBM and by most teachers and administrators in the ed biz. Iverson firmly believed that his contribution, and the contribution of his APL language, was notation; a way of writing clearly about mathematics and about mathematical models. Ken conceived of APL first – first – as a medium for humans to read, write and understand; a language for papers or blackboards, or the backs of envelopes. And then, in 1966, Iverson's language became directly executable by machine. Thanks, Larry.

With it, students would be able to read, write or discuss an algorithm and then, with minimal effort, type it at a machine, see its output, trace its execution, experiment with alternative arguments, try out alternative definitions.There was an office of internal education at Yorktown Heights, which routinely asked instructors to describe what they proposed to teach by referring to a list of identified topics and skills. But when I was asked to teach there, they didn't seem to have anything that corresponded to Ken's vision of programming. So we added to the list a description of "didactic programming". It was probably not the best name, but that's what we called it. Didactic programming is the art of writing a program that is both executable by machine, and makes clear to a human reader what it does and how it works.*

Now this was altogether different from the thrust of the then burgeoning movement of computer-assisted instruction. CAI, of which there was a large

group at IBM then, sought to develop programs that would run autonomously, and that by their behaviour, would give the students experiences that would cause them to learn. The student was not supposed to run the program, and especially not supposed to look inside to see what it did or how it did it. To describe the quite different way he thought computers should be used, Ken coined the phrase 'open use' of the computer. One of the papers we wrote at that time was called "Using the computer to compute", which you'd think was a silly title, except that it was not what everybody in CAI was doing.

When IBM Italy sent us three researchers to Yorktown Heights, to study CAI, Ken successfully hijacked their mission and they became our collaborators not in CAI, but in what they came to call *l'uso aperto del elaboratore*, or something like that. (Forgive my Italian.)

Ken made a great many appearances at schools and universities, to explain how he thought students could and should make use of computers, and in particular, how they could start right away, using APL. At some of these guest experiences, I got to serve as his assistant, and I felt a bit like Robin getting to carry Batman's bag. But listening to Ken give those demonstrations, I marvelled at the spontaneous and natural way he seemed to talk. Especially I loved the way he deftly handled questions from the audience. He seemed endlessly able to respond to their queries with neat examples and appropriate jokes. Only after I had listened to these presentations many times, did I begin to catch on to how he did this. He carried around in his head a huge fund of examples, jokes, aphorisms, stories, but he kept them in reserve. He didn't offer them. He waited until a listener's question happened to set up one of them, and then he sprang it forth. It sounded just like something he'd thought of that instant in response to what the questioner had asked.

Well, people weren't going to believe it was possible to write a program whose listing is itself a readable explanation until there were some examples of the genre. So, when Iverson was made an IBM Fellow, IBM provided him funding to work on projects of his own choosing. Instead of building a staff of his own, he used his budget to bring in visiting teachers and professors from a range of disciplines. The idea was to get them started writing APL programs that would serve as explanations in their respective fields. He hoped they'd keep going when they returned to their home institutions after they'd spent two or three months with him. And indeed, a great number of them did.

My job became looking for lower-level introductory examples of that sort of thing. Ken encouraged me to collaborate with John Thorstensen, a graduate student in astronomy, on a program called STARMAP. The idea was to explain the

movement of the stars and planets across the sky and to do it by writing programs that clearly stated the simple formulas that described their motion. Well, we did this, it ran, we printed it, but people largely didn't get the point that we were trying to make. The program got wide distribution, but not as a way to show students the formulas of planetary motion; instead, IBM gave it lots of publicity as a way for journalists to tell people where in the sky to look for the comet Kohoutec, which didn't actually ever show up, and which we had not even planned, when we were writing this program, to include. But there was worse. When we asked for approval to publish this example of didactic programming we got back from the publications review committee a message that said, "Permission denied. This paper discloses proprietary algorithms that are the intellectual property of IBM." I sent in an appeal saying, as diplomatically as I could, Don't be silly. This is an APL restatement of Kepler's function from about 1610. But then we got a new rejection. This time it said, "Permission denied. This is not original work."

Ken believed passionately that brevity is essential to clarity. "Be concise!" he'd say. For me, this made for endless tension. In my role as populariser, or explainer, I never got over my belief that, in English, readability is mostly redundancy. Ken would look at what I'd drafted and say, "Long-winded. You can cut out two thirds."

He didn't just admire reduction to a few words; he also wanted those few words in a small space. He was very proud of his APL *vade mecum*, an entire APL reference no bigger than a credit card.

Well, as you all know, very early in the development of APL, its enthusiasts found it was so powerful that very often they could express an entire process in a single statement. There were soon contests to see who could find a one-line version of an algorithm previously thought to require many. Sometimes this improved clarity. Sometimes it had the opposite effect, and brevity trumped readability. By readable, I mean that you feel comfortable reading it, not just that you are able eventually to trace out what it says. Whether a very concise statement is readable depends very much on your background and experience. The same expression that delights the old-timer can mystify, appal and repel a novice. Sadly, I have to say that enthusiasts of APL, and of its later reincarnation as J, have continued to storm magnificent new heights but they no longer try very much to address novices. They have left behind the masses that we once sought to address.

So, if I try to sum up, I'd celebrate my personal delight at having been able to be part of this marvellous project and at the same time my pain that there's still a lot left that wasn't achieved, a lot left to do.

A few years ago I went to a computer convention at Moscone Center in San Francisco. From the head of the escalator I looked down into the eager, busy crowds. People of all ages were excitedly swarming to examine the newest products and hear the latest words of Steve Jobs. I was stabbed with a sad realisation that *this was how I thought it was going to be for APL.* I had been fired up by Ken Iverson's dream that he would produce something to revolutionise computing not just for a minority of devotees and experts but for all the rest of us as well. That part hasn't really happened yet.

*A subsequent check in the old files shows that we actually called it 'expository programming' rather than 'didactic programming.'

How we got to APL\1130

from Larry Breed

APL\1130 was implemented overnight in the autumn of 1967, but it took years of effort to make that possible. In 1965, Ken Iverson's group in Yorktown was wrestling with the transition from Iverson Notation, a notation suited for blackboards and printed pages, to a machine-executable programming environment. Iverson had already developed a Selectric typeball and was writing a high-school math textbook. By autumn 1965, Larry Breed and Stanford grad student Philip Abrams had written the first implementation in 7090 Fortran (with batch execution). Eugene McDonnell was looking for applications to run on TSM, his project's experimental time-sharing system on a virtual-memory 7090. With his help, Breed ported the Fortran code, added I/O routines for the 1050 terminals, called the result IVSYS; and Iverson Notation went interactive.

Iverson, Breed and other associates now faced issues of input and output, function definition, error handling, suspended execution, and workspaces. They also struggled with both extending the notation to multidimensional arrays and new

primitives, and limiting it to what could reasonably be implemented. Whatever it was, it wasn't "APL"; that name came later.

Iverson was also collaborating with John Lawrence of Science Research Associates (SRA), an IBM subsidiary in Chicago aimed at educational markets. Lawrence had been editor of the IBM Systems Journal when it published the Iversonian *tour de force* "A Formal Description of System/360." At SRA he was leading a project to make Iverson's notation the heart of a line of instructional materials, both books and interactive computer applications.

SRA hoped to exploit a single-user workstation being developed at IBM's Los Gatos Laboratory. There, veteran 1401 designer Fran Underwood and his colleague Ans Schellenberg were building a laboratory prototype of a computer workstation that they called the PCT, or IBM 1570, or "Elsie," for Little Computer. Elsie comprised a processor, 512 bytes of core memory, a console typewriter, a card reader (cards fed by hand, one at a time), and a disk. The disk, an early version of what would become the floppy disk, held 32k 16-bit words. The processor had the architecture of a stripped-down 7090: one data register ("Accumulator/MQ") and three index registers, all simulated by low-memory locations; a 16-bit data size and instruction size; and 64 opcodes.

SRA consultant Bill Worley (who, decades later, directed Hewlett Packard's computer architecture effort) considered the Elsie architecture outmoded. He, Abrams and Breed proposed a redesign that looked like a stripped-down System/360: 16 general registers (actually low-memory locations), 16- and 32-bit instruction lengths, immediate operands, and just 8 basic opcodes. The arithmetic/logical capabilities were, in full, Add and Nor (each with the option to complement one operand), and Rotate. Sample programs turned out to be substantially shorter and faster, and Underwood was persuaded to adopt the new architecture and to expand the memory to two kilobytes.

With the Fortran implementation behind them, Breed and Abrams turned to assembly-coding a limited implementation for Elsie. How limited? Identifiers were a single alphabetic character; 52 in all. Arrays were of rank two or less, with dimension limited to 255. That particular limit didn't come into play, since total data space was 350 words, or 175 data values; the only data type was 2-word floating point. (Even for character data.) Worley produced a 7090 cross-assembler and emulator, and arranged for Hirondo Kuki, mastermind of System/360's Fortran function library, to develop floating-point arithmetic and math functions.

By spring 1966 Abrams, working at Stanford with occasional forays to Los Gatos, produced an Elsie interpreter that worked but lacked many features. In Yorktown, Breed wrote a specification that for the first time documented APL syntax,

operators (not yet called "primitive functions") and user interaction. The document bore the name "APL", which Adin Falkoff had recently acronymized from Iverson's pioneering book, "A Programming Language." (The pronunciation was not yet settled. "Use Apple," Falkoff said one day, "the language that has appeal.")

Then the Elsie effort slowed. Circuit boards and memory were scarce and expensive, due to System/360 production demands. Only four prototypes were ever built. Breed and Roger Moore of I. P. Sharp Associates Ltd. (IPSA) started designing APL\360. They began in July 1966 and it went into service in November, gaining immediate popularity and heavy use within IBM.

Seeking to counter the popularity of Dartmouth Basic (running on teletypes and a remote GE computer) at the prestigious Hotchkiss School in Connecticut, IBM V.P. Arthur K. Watson, who was a Hotchkiss trustee, had APL terminals and access to the Yorktown APL\360 system installed for Hotchkiss students. The students loved it, and carefully steered newbies to the Basic teletypes so as to maximize their own APL time.

This last may be irrelevant, or it may be why IBM's Hartford branch office knew about APL and its appeal. In late 1967, the Hartford office asked Iverson's group to implement an APL for the 1130, a machine for which sales had apparently been flagging. Breed was at first reluctant to divert resources into this, but then thought of the Elsie implementation. The Yorktown 7090 was long gone, though, and with it Worley's cross-assembler and emulator.

Over a few days, Breed built an Elsie assembler on APL\360 while Reve Carberry, an 1130 programmer provided by the Hartford office, coded up an Elsie emulator. Abrams' assembler code, with some tinkering, was processed through; someone typed "2+2", and APL\1130 printed out

```
4
```

It was not fast. Of Elsie's 2k memory, about half – 1k bytes, 500 words – was allocated to code space. Every operation required multiple overlays from the floppy disk. "2+2" took about five seconds; "2*3" produced about forty-five seconds of disk activity before printing

```
8.00001
```

Improvement was possible, though. Elsie's 2k memory limit could be expanded. The emulator accepted one new I/O operation, "Escape to 1130." One by one,

high-usage routines were coded as native 1130 instruction sequences, and APL\1130 got faster and faster.

There were other problems. The console keyboard was adapted from a Model 029 keypunch, with only one alphabetic case; about thirty APL characters couldn't be entered. Charles Brenner found homes for them by designing and implementing support for the notorious triple-shift keyboard. Brenner gradually took over the APL\1130 development effort, with help from summer student Alan Nemeth. They improved execution times and implemented the rest of the primitive functions.

APL\1130 was released to the public late in 1968 on a self-loading card deck about three inches thick. The Hartford office helped with this, providing Steve Raucher, who created the documentation and jumped through the hoops necessary to register APL\1130 as "Type III" software: available without charge, liability, or support.

Demand was immediate, and led to purchases of several 1130s. It wasn't hard to persuade IBM Marketing to fund development of Version 2, which added multi-character identifiers and most of the other APL\360 features. Paul Berry adapted his informative and appealing APL\360 Primer to APL\1130. David Oldacre and Eric Iverson of IPSA delivered Version 2 in 1969. This is the APL\1130 you remember and may even still be using. It became the most popular free software in IBM history.

Acknowledgements

Thanks to Eugene McDonnell, Philip Abrams, Charles Brenner, Paul Berry, Roger Moore, Eric Iverson, and Brian Knittel for preserving and recalling essential elements of this account.

DEDICATION

Mathematical Computing in J (Volumes 1 and 2) are dedicated to Dr. Kenneth E. Iverson in recognition of his lifetime of profound teachings.

Professor Howard A. Peelle
Mathematics and Computer Science Education
University of Massachusetts
Amherst, MA 01003 USA

The Language, the Mind and the Man

from Fred Brooks

Introduction

I was Ken's first Ph.D. student, because although Aiken was officially (and actively) my thesis supervisor, Ken had more *de facto* influence. I arrived at Harvard for Fall term '53-'54; Ken was finishing his dissertation that year. I'll remark on the dissertation later, because it was an important piece of work in its own right.

Every afternoon at five o'clock if Aiken was in town, everybody in the Comp Lab gathered in the machine room for coffee and conversation with and around the Boss. Towards spring, the Boss turned to Ken, who would start a faculty appointment in the fall: "I want you to put together a year-long course on the business applications of computers."

Well, nobody had ever taught a course like that, anywhere in the world! Immediately I said to Ken, "May I be your teaching assistant?", because I had been especially interested in all kinds of primitive business machines before I ever encountered computers. He said, "Yes." The Comp Lab was a small place; Ken and I were housed in the same office starting that summer and for the next two years. Consequently most of my professional development was guided by Ken.

So I want to talk tonight about the language, the mind, and the man.

The APL Language

APL came about in this way. We had taught the course the first year. When we were getting ready to start the second year, we felt that we really had it under control and started writing course material which later became the book *Automatic Data Processing*. Ken began trying to analyze sorts. (This was long before Don Knuth's really great work on sorts appeared.) In the process Ken, being a mathematician, said he was going to use a mathematical notation for analyzing sorting algorithms. He came to the conclusion that there were many useful different notations, but they were inconsistent. Ken set about then to unify the mathematical notations.

He quickly encountered a problem. Whereas in ordinary algebra the variable names are letters and the values are numbers, in business data processing, the

values are often character strings; and, in machines with numeric addresses, the variable names are numbers. All of this had to be generalized and solidified. That's how APL started: strictly, as Paul says, as a thinking language, a way of getting your mind around these applications and mathematizing what had heretofore been done with words.

The *Automatic Data Processing* manuscript got fatter and fatter, and fatter and fatter. We sent it off to John Wiley. They sent it out to reviewers, one of whom was Bob Ashenhurst. Two of the reviewers, including Bob, came back recommending to cut it into two books. So we did that; it became the *A Programming Language*[1] book and the *Automatic Data Processing*[2] book.

Two incidents in the development of APL are especially memorable. When Aiken finished with coffee and went home, the graduate students and the junior faculty would regularly resume real work, and we went to supper considerably later. One night Ken and I were wrestling with the question of, "Which does one really want: a floor function or a ceiling function?" Each mathematical notation we found had only one of these; some had one and some had the other. We kept encountering cases where we needed first one and then the other in the same treatment. As the night wore on we said, "Let's ask John Wiley to take the square brackets and cut off one serif at the top and one serif at the bottom to make us four new characters. Originally we used them dyadically, ⌈ ⌉ and ⌊ ⌋, to both bracket and round. Ken recognized that this was inconsistent and switched to using them monadically.

The second vivid memory was how APL got its name. One night we wrestled with names. For probably two hours we wrestled with names. Finally we were tired, we were hungry, and I don't know which one of us said, "Why don't we just call it 'A Programming Language'?" So that's how APL came to be known as "A Programming Language". It was in desperation after all the other name tries had failed.

Some years later I asked Ken, probably at the 1978 ACM History of Programming Languages Conference, "What is the touchstone for making an elegant programming language?" He said, "The secret is it has to do what you expect it to do." If you stop and think about APL, if you think about J, and if you think about Ken's work generally, it all has that high degree of consistency—the product of an exceptionally clear mind.

It is also the product of a fierce determination not to invent any new constructs until one has to. I'm sure many of you engaged in one of the arguments when someone wanted to put so-and-so into the language. Ken would say, "Yes that's an interesting idea," and a little while later he would say, "You can do it this way, using the tools and components already at hand." Again and again that was the

solution: it's already in the language, it's just a question of being a little more ingenious in one's approach.

My final story about the language is that Ken didn't get tenure at Harvard. He did his five years as an assistant professor, and the faculty decided not to put him up for promotion. I asked him what went wrong. He said, "The Dean called me in and he said, 'The trouble is you haven't published anything but one little book.'" The "one little book" later won the Turing Award. I think that says more about academic process and Harvard than it does about Ken.

Ken's Mind

I never saw anybody who thought as clearly. Again and again, he would get down to fundamentals. Again and again he would teach and explain, or even wrestle verbally, until clarity emerged from his very articulation.

His dissertation was the development and application of a program for solving the input-output model of Harvard economics professor Wassily Leontief. It used floating-vectors, each of which had one exponent and then a whole set of mantissas. This worked well on the Mark IV. It depended on the mathematical fact that one is going to pivot one's matrices anyway. Leontief's matrices were big: fifteen hundred by a thousand. Doing these on the Mark IV, which had 230 words of core memory, 4000 words of drum data storage, and drum storage for 10,000 instructions, was an accomplishment indeed. Leontief later received one of the early Nobel Prizes in Economics for this work, for which Ken provided computational tools. The concepts were Leontief's, of course.

The clarity and the consistency of Ken's thinking were very impressive. For me personally, his biggest contribution was that Ken taught me to write. He wanted things very concise: "When in doubt, leave it out. Do not put runways for your prose to take off at the start or to land at the end of a chapter. When you're finished, quit."

As a matter of fact, the first chapter of *Automatic Data Processing* is far and away the densest, because we edited it, and we edited it, and we edited it.

Ken taught me some very useful productions. "If it's a clause, turn it into a phrase. If it's a phrase, turn it into an adjective or an adverb. If it's an adjective or an adverb, omit it. Apply these recursively."

One of the things that hampered the adoption of *Automatic Data Processing* is that the first chapter got over-concisified. Students found it very difficult to get through the mathematical background, which is expressed with great clarity but

also great conciseness in that first chapter; and they never got to the easy chapters.

Let me share with you one page of pure Iversonian writing as an example of his clarity and economy. This is from Chapter 8 on programming systems. The chapter has already dealt with interpreters, assemblers, linkers and loaders, generators, and now we come to Section 8.6 on compilers:

> The compiler is the most general type of translator. It differs from the assembler in possessing the following facilities to make the argument language easier to use.
>
> 1. Each elementary operation of the argument language may correspond to a multi-statement program (a subroutine) in the function language rather than to a single elementary statement. This permits the definition and use of a virtually arbitrary set of operations in the argument language. Functions such as $\sin \chi$ and $\log_{10} \chi$, for example, can be incorporated as the elementary operations SIN and LOG.
>
> 2. The individual statements of the argument program may be compound in the elementary operations of the argument language. The programmer may use directly a compound algebraic statement such as $z \leftarrow u(v+w*\mathrm{SIN}(x+y))$. The burden of analyzing this into its elementary components is assumed by the compiler.
>
> 3. The representation used for the operands is not constrained to a single pattern but may vary widely. Consequently, the argument language must include declarative statements or conventions to specify the representations of the variables. Each imperative statement refers to each operand by a single name—the particulars of the representations are automatically considered in the analysis of the statement. Furthermore, separation of the representation from the processing descriptions permits easier modification of each.
>
> 4. The imperative statements of the argument programs are not constrained to be independent or independently translatable. The type of interdependence is usually limited: certain *control statements* specify the sequence of execution of other statements, the values of certain of their parameters, or the formats of their operands. The scope of a control statement (i.e., the set of statements controlled) may include other control statements. This facility simplifies the writing of loops.
>
> 5. Since argument languages become more complex as they become more powerful, compilers typically include elaborate *diagnostics* which bring real or apparent syntactic and semantic mistakes to the programmer's attention.

Now, just for sheer clarity, elegance, and generality, think about each of those statements and all the cases that it covers of the distinctions between a compiler and an assembler. It shows such a clean mind.

The Man

I found him exceptionally unselfishly helpful and giving of himself. If a box needed to be toted, he was ready to take one end of it. If a car needed to be started, "Let's run out and start it."

He taught me to take a nap every day after lunch. We needed to put our heads down on the desk and zonk out. I've been doing that ever since graduate school. It's one of the most useful things anybody ever taught me. I just stretch out on the rug in my office now. My secretary's instructed to say "He's out.", which is accurate.

Ken's manner was always honest and straightforward. He also was very brave. In World War II he left the Home Guard and transferred to the Royal Canadian Air Force because he wanted to be nearer to the war, and the Home Guard could not be assigned overseas.

Ken regularly stood up to Howard Aiken. Aiken stood six-foot-three and had Spockean ears and pointed tufts of hair: When he r'ared up and looked down at you, you thought you were looking at the Devil. And when he growled at you, you thought that too. If you stood up to Aiken, he respected that; and if you didn't, he just trod you underfoot. Ken stood up to him. Ken's manner was always argumentative, but argumentative with an effective expression: "Perhaps one ought to think about it this way..." This was in contrast with Aiken's "Goddamit it's gotta be so-and-so, all right?" Ken was firm in his views but he was reasonable. He satisfied the old Latin motto, *Numquam incertus, semper apertus*—never uncertain, always open.

These characteristics made him a very warm friend, a person one admired as a mentor, a person from whom one learned so much, and the person for whom we named our eldest son. A very great mind, and we miss his conversations; a very great man, and we miss his character.

References

[1]Iverson, K. E., *A Programming Language*, New York: Wiley, 1962, 286 pp.

[2]Brooks, Jr., F. P., K. E. Iverson. *Automatic Data Processing*, New York: Wiley, 1963, 494 pp.; System/360 Edition, 1969, 466 pp.

Ken at Yorktown Heights

from Jim Brown

Without question, the most influential person in my professional life was Ken Iverson. You all know his many accomplishments and I will not restate them here. Rather, I'd like to give a more personal history of my involvement with Ken. Please forgive any factual errors in my recollections.

I first heard of Ken Iverson shortly after I started work with IBM in 1965. A colleague was talking in the hall about this person who "reduced the design of the System 360 to a single symbol". I thought this was a bit of an outrageous claim (and it probably was) but I decided to check up on it. I had been interested in symbolic notations for a while and when in college read much of Alfred North Whitehead, Bertrand Russell's *Principia Mathematica* working out all the proofs. I went out and got Ken's book *Applied Data Processing* co-authored with Fred Brooks. Then I got *A formal description of SYSTEM/360* by Falkoff, Iverson, and Sussenguth - an amazing piece of work which later became known as the grey manual. I thoroughly enjoyed time spent on this document and came to appreciate that symbolic notation really can be used for describing real as well as formal systems. I didn't find any fault with this document but I hope Larry Breed correctly describes how he found an error and told Ken about it at lunch at Stanford being careful to bring it up when Ken had cake in this mouth.

Ken's name next came to my attention when a friend and I took our wives to a dinner meeting of the local ACM chapter. I always felt there was no way to impress a woman like an ACM meeting. In fact, we did go for a social evening and didn't even make note of the speaker who was someone I never heard of anyway. It was Adin Falkoff with a portable (by some extended meaning of that word) 2741 terminal which he dialed into Yorktown and showed us APL. He carefully and patiently explained to the telephone operator that the phone line would be used to communicate with a computer and the noises on the line were normal. Nonetheless, the session was interrupted several times when the connection was terminated by the operator.

I was absolutely blown away by the whole presentation. Adin typed in "2 space 3 backspace backspace plus" and pressed return and it typed "5". He called it visual fidelity. I recall our wives were not so impressed by this as they were apparently able to work out this result in their heads without use of a terminal and phone line.

There was APL itself which I just found amazing but I was just as impressed by the fact that Adin gave credit to the APL team – Larry Breed, Dick Lathwell, Roger Moore, and (again that name) Ken Iverson. I had never in my career (now over 20 months in duration) heard an IBM presentation that gave credit to real people.

IBM Owego had a 2741 terminal with a phone and I was able to sign on to the Yorktown APL service from work. My manager would not permit this kind of playing around so I could not continue to use it. When I heard that Syracuse University needed someone to manage a new proposed APL installation, I left IBM and went on Campus to do that.

Six months later when the money ran out, I applied to Yorktown Heights for summer work but was turned down. I also applied directly to the APL group and was invited for an interview.

Karen and I showed up at Yorktown Heights at the appointed time only to find that the interview had been canceled. Because of the death of President Eisenhower, the lab had been closed the previous day and a scheduled "APL Machine" seminar had been moved one day later. Thus, March 1969 is the first time I saw Ken – standing outside the IBM Research Auditorium, during a break in the seminar, surrounded by people engaged in animated conversation. It turns out they had tried to reach me to cancel the interview but we had taken our daughter to our parents to watch while we came to Yorktown. I don't know why they didn't try my cell phone or send mail.

I came back the next day for the interview and met Ken, Adin, Larry, Dick, and the team for the first time. I got the job and spent the summer at the lab with the APL team. I know now that Ken and Adin sort of took turns managing the group but the focus was always on the work not the management. I've told this story many times before but I had assumed that Adin was my manager. It turns out that Larry Breed was actually my manager but I didn't find out until the exit interview at the end of the summer. Because of an administrative error, my summer job was not terminated and I continued for three years working remotely from Syracuse. Not many people had the ability to work on computers remotely in those days. Each summer I would go back to Yorktown for a few months.

Ken and the team we already thinking about extensions to APL and I left one of those summers with the idea that there was no reason that arrays needed to be entirely character or entirely numeric and that they did not necessarily need to be strictly rectangular – why not ragged! I changed my PhD thesis topic at Syracuse to deal with extensions to APL.

I had lots of ideas for extending APL in those early days and many of them were really bad. One thing I remember about Ken was his willingness to listen to ideas. No matter how bad the idea, he would listen, make some comment like "sounds interesting" and then begin to carefully analyze the various aspects and find flaws or suggest improvements. I think Ken was always the teacher and in this activity he taught me a lot about how to approach proposals – how to analyze and take thoughts to their natural conclusions.

I don't think of Ken as a person who told a lot of jokes but when he did, there was usually something more than just a funny story – there was something to learn or something to make you think. One story he told stands out in my memory and I've repeated it many times because in addition to being a good story, it also describes the central fallacy of communism. What's wrong with "From everyone according to their ability and to everyone according to their need"? And the answer is that you're dealing with people! Here's the story:

> A farmer is being interviewed and he's asked: "If you had two acres of land and a friend had none, would you give him an acre of land?"
>
> The farmer replied: "If I had two acres of land and a friend had none, then for the greater good of the state, I would give him an acre."
>
> "If you had two horses and your friend had none, would you give him one of your horses?"
>
> The farmer replied: "If I had two horses and a friend had none, then for the greater good of the state, I would give him a horse."
>
> The interviewer continued: "If you had two cows and your friend had none, would you give him one of your cows?"
>
> "No!" said the farmer.
>
> "I don't understand, you would give him an acre and a horse, why not a cow?"
>
> "Well," the farmer explained, "I *have* two cows."

After graduation from Syracuse, I joined the APL group at the Philadelphia Scientific Center. My PhD Thesis "A generalization of APL" contained my ideas of how to extend APL. My ideas did not line up exactly with the general thinking at

the Center or with Trenchard More's ideas. Trenchard was able to use my theory to prove that '1' equals '2' – an important result I thought.

Over the next years, there were many private and spirited discussions on the right way to extend APL between Ken and me and with others. Even though Ken and I had some pretty fundamental differences, these discussions, though heated, never became unfriendly. (This was not always true in discussions with others.) My ideas and my designs for a new APL were shaped by these discussions even though I never came to completely embrace Ken's position. At all times, Ken was the gentleman scholar. My most pleasurable times at IBM were these stimulating technical discussions.

The world will remember the work of this great man.

Perhaps the one thing of which I am most proud is that I could call this great man "friend".

Written for the celebration of Kenneth Iverson's life and achievements, held at the Computer History Museum, Mountain View, California, November 30, 2004.

Ken Iverson in Denmark

from Gitte Christensen (APL90 Chairman)

The Danish APL community is glad to be able to contribute to this memorial for Ken, who has meant so much to us during our lives.

When the sad news of Ken's passing reached us, a few of us APLers in Denmark started to collect material for an obituary and fortunately some of the APLers from the very early days are still around and were able to share their memories.

Per Gjerløv, who is a lifetime APLer, and who many of you may know, had a lot to contribute. He was present when APL came to Europe in the late sixties, and I would like to share his memories with you.

In July 1967 Hans Helms arranged a summer school with the title *Programming Languages*. It was a huge success with many prominent participants. Amongst languages presented were Snobol, Lisp and Algol 98.

IBM sent a young man called Ken Iverson, from the research labs in Yorktown Heights. Iverson had developed a mathematical notation which was now executable on the system 360. Along with Ken Iverson came Dick Lathwell with a magnetic tape containing the system and the intention was to do a live demonstration on a typewriter terminal 2741. The dialog was to be shown on TV cameras to the audience.

Now that was a tall order – typewriter terminal sessions were not very common then – but Dick Lathwell and a young engineer, Henrik Nyegaard, went to the University of Bergen (500 miles away, in Norway!) where computer time could be rented and installed the tape.

After a while, many members of the audience were starting to suffer from fatigue. Several complicated languages had been talked about and demonstrated before

Dr. Iverson appeared. A short way into his presentation Iverson performed a number of keystrokes including keying the number 2 and the plus sign explaining that he would now create the expression 2+2 – and magically the number 4 was returned by the machine. Asked about declarations, Iverson explained that the computer was smart enough to figure out that 2 was an integer all by itself. The audience was suddenly wide awake!

Now IBM Sales heard about this system. ØK-data had just announced a timesharing service based on a system from General Electric using the language BASIC. IBM did not have anything in their portfolio to match this until APL arrived, and in November 1967 it was decided to go with APL and after a couple of hectic months IBM was able to offer an APL timesharing system on system 360 model 40. An impressive system capable of supporting 30 users simultaneously with good response.

Hans Helms wanted the system to be shown in November at the yearly SEAS meeting in Scheveningen, Holland – SEAS being the European equivalent of the American SHARE – user group meetings for large IBM customers. Again Henrik Nyegaard was charged with the task of providing a line to the APL system running in Bergen. He managed this for the second time, this time further challenged by the task of getting the connection safely through the manual extension board at the hotel where the conference took place. Unfortunately this bright young man was quickly lost to the APL community; he embarked on a management career and ended up running IBM Denmark.

The demonstration went well and was a great success and many large corporations in Europe started using APL, among them VOLVO who to this day are still using APL for some of their planning applications. The centre in Denmark was the first in Europe but soon many more came too.

Ken Iverson visited Denmark on several occasions in the early days – specially when the municipalities data centre started to offer the service to primary and secondary schools. On one occasion Ken almost lost the attention of the audience after the break. He had filled out the left side of the blackboard before the break writing with his right hand. After the break he continued filling the right side – now writing with his left hand not to obstruct the view of the audience. A heated debate broke out amongst the participants whether he had been right handed before the break or not.

As most of you will know Ken was left handed but was ambidextrous with respect to writing.

In 1980 around the time of what I would call the second wave of APL – where many customers got the in-house APL systems on mainframes – Ken retired from IBM and came to work for I. P. Sharp Associates in Toronto. It was at about that time that I was recruited to the Danish IP Sharp office.

Ken visited us now and again and I remember once I was struggling with translating his *An Introduction to APL* into Danish. I told Ken that I was completely unable to come up with a good translation for the APL function *ravel*, which had previously suffered the prosaic name "make list of" in Danish. "You should use the word you use when you have knitted something and you then undo the knitting – when you are removing the structure and are left with the thread". Suddenly the term *ravel* made sense to me in a completely different way.

Now the mainframe environment became very political and strategic to many companies, and the idea of end users having access to the machine on their own accord became deprecated.

The saving grace was the personal computer. It brought along the third wave of APL and allowed many of us to move off the mainframe and to continue to build marvellous applications on the PC. Many commercially viable applications were born using APL and many remain there to this day.

Just last week Morten Kromberg and I were in Florida where the two main vendors of APL systems for Windows had their yearly user conferences back-to-back and I can assure you that at least half of the conference audience did not have grey hair.

Many a tribute was paid to Ken there – and surprisingly or maybe not – a good part of them contained a thank you for giving us the opportunity to have so much joy and fun in our professional lives.

John Scholes, the main language designer at Dyalog Limited – a person perfectly capable of discussing the foundations of APL with Ken as an equal – remembered his last encounter with Ken like this: "In Scranton in 1999 during one of the sessions I was sitting next to Ken – and he leaned over and said to me – in his impish way – John, what is an array? Now I knew better than to rush into an answer to Ken. I guess I'm still working on my answers to that question."

When Reuters took over I.P. Sharp in 1987, Ken retired from I.P. Sharp. But did he rest? – Of course not!

In 1990 I was the Chairman of the annual APL conference – in Copenhagen.

Not hampered by an installed user base and desiring to get rid of the character set problem, Ken had invented J – and submitted papers with the announcement to the APL90 conference.

Well – nothing Ken ever did left people cold – but the resulting discussion in the program committee was probably the hottest I have ever experienced. Finally I had to draw a line seeing that the rest of the programme would not come into existence if this debate was not stopped.

"We will not decline a paper from Ken Iverson – and it will go into the main conference stream".

Not that a refusal to accept Ken's paper at APL90 would have done anything to slow him down, of course...

Last year Morten and I had an opportunity to for the first time in our professional lives to catch our breath a bit. We went to see Jsoftware to catch up on the progress of J and we spent a wonderful week with Ken, Eric, Roger Hui and Kirk Iverson, for which we will be eternally grateful.

One of the last things Ken said to us was that, although he could understand why commercial software developers had been slower to adopt J than the educational world, he could not understand why APL vendors had not implemented some of the elegant concepts introduced in J.

And Ken – I hope you are listening in now – at the closing session of Dyalog APL's conference last week John Scholes announced that he would implement the rank operator in Dyalog APL – so the torch you handed us is still being carried forward. It might not sparkle as much and it might not shine as bright as when you carried it – but we will do our very best.

Gitte Christensen, November 2005.

Hommage à Ken Iverson

by Michel Dumontier (mjdumontier@free.fr)

An earlier version of this article appeared in Les Nouvelles d'APL, N°41; this appears by kind permission of the AFAPL [Ed.]

An Introduction to APL – and APLers

Although Ken Iverson and I both attended APL71 in Paris (so far the only international APL conference in France) we did not meet until much later.

Kenneth E. Iverson at an AFCET APL conference in Paris, 1983 on "Teaching, Computing and APL", a subject always close to his heart.

I started with APL in 1969 at the Centre de Calcul Scientifique de l'Armement (CCSA) in Arcueil, at the Fort de Montrouge. There I met Gérard Lacourly and got his book *Informatique par téléphone* (Computing by phone), written with Philip S. Abrams.

After this meeting, at the annual SICOB show in La Défense, I was awe-struck by the APL demonstrations, particularly of linear algebra. I practised on the APL service from Maryland University in digraph mode, and later on a Sperry Univac service with APL characters, firstly on a Tektronix terminal and then on a timesharing APL terminal.

At that time, Gilles Martin worked at the Centre Interarmées de Recherche Opérationnelle (CIRO) quite close to the CCSA. We were both linear algebra specialists and swapped Fortran programs in this field. We lost touch with each other, then in 1974, I met Paul Braffort at the Centre de Réflexions sur le Futur in Arc et Senans during a symposium. Paul told me that Gilles Martin was also using APL...

Later I worked as expert on the international standard at the beginning in 1978 at AFNOR, the French standards association. Raymond Tisserand was then president of the APL group of AFNOR. There I met Gérard Langlet, Jean-Jacques Girardot and other members of the group, and found Gilles Martin again. I had already met Raymond when he worked with Guy Varin and G. Fustier at the National Research Institute in Computing and Automation (INRIA).

From 1982, I had the benefit of the IP Sharp Associates service, and was able to exchange emails with the standards group there. One connected by typing `)LOAD 666 BOX` from a connection kindly provided by the Ecole Nationale Supérieure des Mines at Saint Etienne, thanks to Jean-Jacques Girardot.

For a long time I wanted to attend APL conferences but could find no sponsor, even though recommended by Raymond Tisserand at AFNOR in 1984. (Given the popularity of APL, no one will be astonished!) So I went at my own expense to all 14 of the conferences from 1985 to 2000.

Ken was not in Seattle at APL85. I had brought the alpha-omega notation workspace I had written specially for the Ampère WS1. This was a portable computer with an LCD screen. I had spent my holidays working on this at the SOFREMI society. The machine was dedicated to APL – it did nothing else. (Japanese delegates also presented their own machine.)

Meeting Ken Iverson

I did not meet Ken Iverson until APL86 in Manchester. He came to our first ISO APL meeting on extensions. Until then, we had only corresponded. In 1985, The APL standard was published, Raymond Tisserand passed the torch, and Leroy Dickey became the new convenor of the APL standards group. We had decided to continue the work of standardisation on extensions to the language.

Ken was an affable man. We agreed to speak each in our own language, and that worked well: we wanted to be sure to express our ideas clearly.

He was always cordial with me; there are many examples of that. This one is close to my heart, on a subject on which we shared common ground, the J language.

Ken and Eugene McDonnell came to Princeton for a meeting on extending the APL standard. I spoke of the functionality I'd like to find in APL; without it I was hard put to express myself in mathematics. In APL then there were only 5 operators, which took primitive functions as arguments, from which to derive new functions. I wished we could define our own operators, and that they would modify defined as well as primitive functions. Complete freedom... This would entitle APL to one of its numerous acronyms, devised at Manchester in one of those famous conference games: the 'Art to Program in Liberty'! (This may be true compared to Pascal, one of the most restrictive languages I knew at that time – but it's all relative...)

Eight months later, in answer to one of my letters, Ken wrote to me of his plans since Princeton. He had not only met my demands, but gone much further, starting from his previous work "A Dictionary of APL". He was very attached to teaching APL, as the letter on the following page shows ...

Apt. 405
70 Erskine Avenue
Toronto, Ontario
Canada M4P 1Y2

April 20, 1990

Dear Michel:

Thanks for your letter of March 1. Your news about the state of APL in France was depressing but not surprising, since the same seems to be happening here. In particular, I.P. Sharp Associates (since being bought by Reuters) appears to be moving away from the use of APL as quickly as possible, and most of the experienced APL people (such as Bob Bernecky and my son Eric) have already left the company. Furthermore, the situation of APL in education is about as bad here as you say it is in France.

However, I am not discouraged. Since retiring from Sharp, I have concentrated on defining a simpler and more powerful version of the language, and developing an implementation in C that can be ported to a wide variety of machines and be distributed as Freeware. Moreover, the system uses a simple spelling scheme that uses only ASCII characters, and avoids all of the character-set problems that you pointed out in your letter. By this means, I hope to make APL easily available to schools and, perhaps more importantly, to get it into the hands of young people working outside of the school system.

I enclose a copy of a paper on these matters that will be presented at APL90 in Copenhagen. I would be pleased to receive any comments that you have, and hope to see you at the conference.

As to any help or advice on coming to Canada to work, I do not believe that I can offer you anything useful. Since retiring from Sharp, I have no affiliations of* contacts that could be useful. Moreover, as I said, the state of APL here seems to be no better than in France. However, if you can afford the time to work on treatments of mathematical topics in APL for use in schools or in independent study, I would be pleased to collaborate in any way possible.

I was sorry to hear the news of Gilles Martin. I do hope that I will see you at APL90.

Yours truly,

Ken

**sic* Ed.

I need hardly say how gratified I was to receive this letter, and my reply could not but show my pleasure at his asking me for comments. Of course, I went to APL90. The paper he had written of is nothing but "APL\ ?": the extension of APL to anything you want, no limits. That is, of course, the J language.

Ken Iverson and Eugene McDonnell at the APL ISO meeting at Princeton University in 1989

After the conference, I asked him why he named it J, and proposed: is it because the J letter is placed between I and K (Ken Iverson) and the language K already exists? Moreover, the letters H, I, J and K are in order – H for Roger Hui, who did the development in C he mentioned. He answered that it was Roger who chose J as the key we have the best chance of finding on the keyboard with closed eyes! Moreover, it is has an embossed mark for the blind.

Searching systematically in the various SHARP APL libraries, I found a library of Ken's workspaces containing MODEL and further MODEL7051... The main function was simply: `APL`.

I told him by email in January 1986 that I had discovered this workspace and used it. I had read in APL Quote Quad (16.2, December 1985, page 35) of his visit to Finland in May 1985 and his two conferences, one on his paper "A Dictionary of the APL Language", the other on teaching programmers how to take advantage of APL in applied mathematics. I asked him if he would give me the texts of those conferences.

In his new terminology each constant is a 'noun', each variable is a 'pronoun', each function is a 'verb', and each operator is an 'adverb'. Many of the ideas of Dr Iverson looked very promising as tending to clarity, simplicity, and consistency.

Comparing his new syntax and APL2 he made some very critical remarks about allowing so many exceptions and vague concepts in APL2.

Ken Iverson at APL97, Toronto

I should contrast this review with my memory, some years later, of the extension of the APL standard heading "dangerously" towards APL2. I recall a lively discussion on this topic at Hellerup during an APL ISO meeting just after APL90. E. McDonnell and I were of the same mind... as would Ken have been had he been there!

Ken left me only good memories. He always helped me with information I wanted and always encouraged me. To quote a few examples, before it was sold, he gave me all the J versions I needed and also the SHARP APL version and documentation; he encouraged me to give the J tutorial at APL94 in Antwerp, where he gave me the latest J version for Windows. (I had bought the previous non-updated version). He kindly accepted the translations into French I made of Tangible Math and Programming in J. He even wished me a retirement as happy as his! (I shall try!)

There have been many tales of his ready and surprising wit on intellectual points. One evening in a restaurant (at Princeton, if I remember correctly) he was kind enough to join us at what had become the French table. He was facing me, Gérard Langlet was on my left and Jean-Jacques Girardot was opposite Gérard. During the discussion, we introduced a poser an examiner had set a candidate for the Polytechnic School: to calculate the product (x-a)(x-b) ... (x-z).

Those without presence of mind begin the development at once. We were astounded (and surprised!) by the immediate and absolutely exact answer he wrote on a paper napkin: `X←x ⋄ A←a,b,c,...,z` the product is `Y←×/X-A`.

In his turn, he was surprised when we told him that the result was 0 since the series contains the factor (x-x). But bravo for the answer anyway – exact and fast!

Creating pictures in Cliff Reiter's J workshop at APL97.
The author is in the foreground.

Ken Iverson and Roger Hui in Toronto at the J2000 conference, by kind permission of Ellis Cave.

I last saw Ken Iverson in Toronto at APL97.

Ken Iverson at the End of the World

from Richard Hill

I am unaware of Iverson visiting Kerguelen, or even Perth, WA, so we can consider his visit to Melbourne, Australia in November 1994 the furthest he ever got from his origin in Alberta.

Melbourne is the capital of Victoria. The State of Victoria is the inverse of the Province of Alberta (regally) which should appeal to a J-ish sense of humour. Iverson came to Australia to run a workshop in J, sponsored by Walter Spunde of the University of Southern Queensland. After the workshop he came down to Melbourne to give a lecture at Melbourne University, sponsored by Moshe Sneidovich. I remember the lecture well. There were around 200 people present and quite a vigorous exchange of views on tacit and functional programming. After the lecture Iverson stayed overnight at our place in Black Rock.

As a fairly simple engineer, I felt on quite a different plane to Ken Iverson and the only specific saying of his that I remember is "Engineers are distinguished, not so much by their ignorance, as by their pride in it". I asked him a question, "If you compare J and APL, over a range of problems, what can you say about the number of tokens needed on average to program the solution?" He said that he thought that J needed about half as many tokens as APL for the same problem.

This leads on to a humble suggestion for a practical memorial to Ken Iverson. A new APL primitive to be called Nub.

My own work is supporting APL programs written by various people, and involves a lot of "data rummaging". After meeting Iverson, I tried to read the J dictionary and wrote a couple of small things in J, and I came across J's primitive "Nub". Nub is trivially achieved in APL, being ((V{iota}V)={iota}{rho}V)/V.

But, I wrote a little APL utility called Nub, which also works with arrays, I found it so handy that I realised what Iverson meant by "Notation as a Tool of Thought". It is really true. Being able to type Nub and not have to think about the construct means that you don't lose the 'flow' of thinking about the problem at hand.

Ken Iverson wrote a couple of papers during the nineties about areas where there could be some unification in the APL area. Nub is mentioned in at least one of them, published in Quote Quad, I believe. It is respectfully suggested that in memory of the originator of APL, one of his later ideas be incorporated as a new primitive 'Nub'... in all flavours of APL.

A Lifetime of Working with Ken

from Roger Hui

I am indebted to Chris Burke, Eric Iverson, Eugene McDonnell, Donald McIntyre, Roland Pesch, Joey Tuttle, and Arthur Whitney for reading earlier drafts of the paper.

0. Beginnings

I first met Ken Iverson early in 1981 in Toronto, when he invited me to dinner, along with Mrs. Jean Iverson, and Eugene McDonnell, Lib Gibson, David Keith, and Jane Minett of I.P. Sharp Associates. I believe the dinner happened because Eugene mentioned me to Ken, mainly due to *The N Queens Problem paper* [0] I had submitted to the Recreational APL column that he wrote and edited.

So now not only was I to meet the great Ken Iverson, inventor of APL, but I was also to get a free dinner. The latter was a consideration: at the time I was a graduate student at the University of Toronto. At the appointed hour I went up to Ken's apartment on the 47th floor of the Manulife Center. I was so in awe of the great man that I took scant notice of the spectacular view – from the apartment one can see Niagara Falls, a 90 minute drive away – until Ken directed me to do so.

We then went for dinner at a restaurant on King Street. I don't remember much about the dinner, except for the way Ken left the tip. The change came back from paying the dinner check, in the form of the dingiest looking bills you ever saw. Ken took out his wallet, extracted some crisp clean bills, and crumpled them up before leaving them on the tray.

In another sense, I had "met" Ken Iverson years before, when I learned APL during my undergraduate studies at the University of Alberta, from reading the *APL\360 User's Manual* [1] and from working at the computer terminal. Since I did not have direct access to Ken, I read his papers carefully, principally *The Design of APL* [2], *The Evolution of APL*[3], and *Notation as a Tool of Thought*[4]. I also further acquainted myself with APL during the summers of 1975 and 1976, when Lib Gibson hired me to work as a summer student at I.P. Sharp in Calgary.

1. From APL to J

The late 70's and early 80's were an exciting time for APL. APL was a commercial success. APL was taught in schools and universities. APL conferences were well-attended. (I attended my first APL conference in 1979 in Rochester, and my recollected impression is that the parallel sessions at the conference each had audiences of over a hundred.)

Ken had published his seminal paper *Operators and Functions* [5] in April 1978. He went from IBM to I.P. Sharp in Toronto in 1980, and there collaborated with Arthur Whitney on *Practical Uses of a Model of APL* [6] in 1981-82, leading to *Rationalized APL* [7] in January 1983, multiple editions of *A Dictionary of the APL Language* between 1984 and 1987, and *A Dictionary of APL* [8] in September 1987. Within I.P. Sharp, the phrase "dictionary APL" came into use to denote the APL specified by *A Dictionary of APL*, itself referred to as "the dictionary".

At that time, the main APL vendors were IBM, STSC, and I.P. Sharp, and all were active in developing and extending the language. IBM had APL2, based on the work of Trenchard More and Jim Brown [9], [10], [11]. Work on APL2 proceeded intermittently for 15 years [12], with actual coding starting in 1971 and APL2 becoming available as an IUP in 1982. STSC had an experimental APL implementation called NARS [13]. NARS and APL2 differed in fundamental respects from dictionary APL (and differed from each other).

I.P. Sharp implemented the new APL ideas in stages, complex numbers, enclosed (boxed) arrays, match, and composition operators (on, over, under) in 1981 [14, 15], determinant in 1982 [16], and the rank operator, lev (left), dex (right), and link in 1983 [17]. However, the domains of operators were restricted to the primitive functions or subsets thereof. I.P. Sharp also had SHARP APL/HP [18], principally the work of Arthur Whitney with the assistance of Rob Hodgkinson.

An important stepping stone from APL to J was SAX [19], SHARP APL/Unix, written in C and based on an implementation by STSC. An alpha version of SAX became available within I.P. Sharp around December 1986 or early 1987. From that time to August 1989, when the first J source line was written, I had access to SAX, and in the later part of that period used SAX on a daily basis.

SAX implemented dictionary APL in part, and was upward-compatible with mainframe SHARP APL in order to run applications ported from there. It permitted derived or user-defined functions as arguments to operators, and had the special APL characters, workspaces, and the del-form function definition. However, it did not permit functions and operators to be named using the assignment arrow, as in the phrase `sum←+/` , precluding what later came to be

called tacit definition. (The dictionary specified such assignment.) It also did not have hooks and forks [20], for those were not yet invented. For Ken, an important drawback to SAX was that it was not widely and freely available.

From a personal perspective, another important stepping stone from APL to J was the writing of my APL87 paper *Some Uses of { and }* [21]. At the time both Ken and I were working for I.P. Sharp in Toronto, and the ideas in the paper evolved over many conversations with Ken. Writing the paper refined and sharpened my understanding of dictionary APL.

Arthur Whitney, from collaborating with Ken on *Practical Uses of a Model of APL* [6] in 1981-82, inventing the rank operator while on the train ride to the APL82 conference in Heidelberg in 1982 [22, 23], and implementing SHARP APL/HP in 1986 [18], went on to Morgan Stanley in 1988 and there invented A [24]. He later invented k [25] in 1993 and q [26] in 2003. (Arthur is a classmate from the University of Alberta; over the years our paths have crossed many times, and each time I had benefitted. But that is a story for another time.)

In the wider world, the PC revolution was in full swing. It was now possible to have a computer (for example, the AT&T 3B1 UNIX PC or the IBM PC/AT) at home that was a reasonable platform for developing an APL system. Not too long ago, such a platform would be under lock and key in a room with a raised floor, inaccessible to mere mortals and not readily available for experimentation.

The conditions were ripe for...

2. J

The story of how J began has been told in detail by Donald McIntyre [27]. Ken himself wrote in *A Personal View of APL* [28]:

> Work began in the summer of 1989 when I first discussed my desires with Arthur Whitney. He proposed the use of C for implementation, and produced (on one page and in one afternoon) a working fragment that provided only one function (+), one operator (/), one-letter names, and arrays limited to ranks 0 and 1, but did provide for boxed arrays and for the use of the copula for assigning names to any entity.
>
> I showed this fragment to others in the hope of interesting someone competent in both C and APL to take up the work, and soon recruited Roger Hui, who was attracted in part by the unusual style of C programming used by Arthur, a style that made heavy use of preprocessing facilities to permit writing further C in a distinctly APL style.

> Roger and I then began collaboration on the design and implementation of a dialect of APL (later named J by Roger), first deciding to roughly follow "A Dictionary of APL" and to impose no requirement of compatibility with any existing dialect. We were assisted by suggestions from many sources, particularly in the design of the spelling scheme (E.B. Iverson and A.T. Whitney) and in the treatment of cells, items, and formatting (A.T. Whitney, based on his work on SHARP APL/HP and on the dialect A reported at the APL89 conference in New York).
>
> E.E. McDonnell of Reuters provided C programs for the mathematical functions (which applied to complex numbers as well as to real), D.L. Orth of IBM ported the system to the IBM RISC System/6000 in time for the APL90 conference, and L.J. Dickey of the University of Waterloo provided assistance in porting the system to a number of other computers.

It's funny, but my recollection is that at the time I thought *I recruited Ken*. Whoever recruited whom, I won a great prize when Ken decided he and I should work together.

The final impetus that got J started was the one-page interpreter fragment that Arthur wrote, recorded in Appendix A of *An Implementation of J* [29] and also reproduced in Appendix A below. My immediate reaction on seeing the page was recoil and puzzlement: it looked nothing like any C code I had ever seen. ("Is it even C?") However, Ken counselled that I should reserve judgment. As recounted in *An Implementation of J*, it then happened that:

> I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this document.

The name "J" was chosen a few minutes later, when it became necessary to save the interpreter source file for the first time.

I consciously designed and built the system around "Table 2: Parsing Process" in *A Dictionary of APL* [8], which I had studied carefully and modelled at least two different ways between 1986 and 1989. The C data and program structures were designed so that the parse table in C corresponded as directly as possible to the parse table in the dictionary. I set as an objective of being able to show Ken the C source for the parser, and have him verify that the syntax being implemented was correct. We never did carry out the exercise (of Ken reading the C source), but I reckon I met my objective, because eventually Ken replaced the table in the dictionary with the parse table from the C source [30]. Other effects of trying to meet the objective were entirely beneficial.

Years after the initial implementation, I happened to re-read Ken's APL87 paper *APL87* [31]. In retrospect, the paper, in five pages, prescribed all the essential steps in writing an APL interpreter, in particular the sections on word formation and parsing.

Ken and I had in mind to implement *A Dictionary Of APL* [8] together with hooks and forks (phrasal forms) [20]. For years, Ken had struggled to find a way to write `f+g` as in calculus, from the "scalar operators" in *Operators and Functions* [5, section 4], through the "til" operator in *Practical Uses of a Model of APL* [6] and *Rationalized APL* [7, p.18], and finally forks. Forks are defined as follows:

```
(f g h) y ← (f y) g (h y)

x (f g h) y ← (x f y) g (x h y)
```

Moreover, `(f g p q r) ← (f g (p q r))`. Thus to write `f+g` as in calculus, one writes `f+g` in J. Ken and Eugene McDonnell worked out the details on the long plane ride back from APL88 in Sydney, Australia, with Ken coming up with the initial idea on waking up from a nap.

The choice to implement forks was fortuitous and fortunate. We realized only later [32] that forks made tacit expressions (operator expressions) complete in the following sense: any sentence involving one or two arguments that did not use its arguments as an argument to an operator, can be written tacitly with fork and `@:` (compose) and `[` (left) and `]` (right) and constant functions. If `@:` were replaced by the equivalent special fork `[: f g`, then a sentence can be written as an unbroken train (sequence of forks).

Because explicit definition (the analogue of del-form definitions in APL) was relatively complex, it was not implemented until some months had elapsed. As well, the restriction in APL that only nouns (arrays) can be assigned never made it past the first day, for there was no way other than by assignment to name verbs, adverbs, and conjunctions (functions and operators). Therefore, for some months *all* verbs were tacit; that is, all functions were defined as operator expressions. These circumstances forced us to explore tacit definition extensively.

Meanwhile, Ken was concerned about the usefulness of forks, and worked hard at finding examples of forks beyond those in *Phrasal Forms* [20]. After a while, it seemed that *everything* was a fork. The explanation lies in the proof of completeness for tacit definition [32]: if the root (last) function in a sentence is applied dyadically, then a fork is required to write the sentence tacitly. Since we wrote tacit definitions exclusively, it was no wonder that we ran into forks constantly.

As well, we had little trouble expressing computations tacitly because of the completeness.

Eric Iverson founded Iverson Software Inc. in February 1990 to provide an improved SHARP APL/PC product. It quickly became obvious that we had shared interests and goals, and in May 1990 Ken and I joined Iverson Software Inc. The company soon became J only. The name was changed to Jsoftware Inc. (www.jsoftware.com) in April 2000.

Eric adapted the session manager from SHARP APL/PC for J in time for the first release. Over the years, he was responsible for everything that was not the interpreter proper – the integrated development environment, the Windows interface, and all other interfaces – and made contributions to the language itself, including locatives, control structures, and memory-mapped files. Eric also handled product packaging and release details. Chris Burke joined us in early 1994 and wrote the application libraries, in particular the plot package, the debugger, the project manager, and the performance monitor.

Work proceeded rapidly from August 1989. The first public presentation took place on 1990-02-26 when Ken and I gave a talk complete with a live demo to the Toronto APLSIG at the Toronto Board of Trade. The first release of the software was at APL90 [33] in Copenhagen in August 1990, when J was made available as shareware to the Software Exchange. (You can still get this version from various archives.) The first J conference was held on 1996-06-24 to -25 at the Koffler Institute of the University of Toronto. There were 123 attendees and 12 contributed papers in the proceedings[34].

In working with Ken, I soon realized that Ken had a flexibility of mind that is breathtaking. Two examples illustrate this point.

Example 0: For the first few months, the special APL characters and the ASCII spelling co-existed in the system. It was Ken who first suggested that I should kill off the special APL characters. I myself resisted for a few weeks longer, until the situation became too confusing, for reasons described in *J for the APL Programmer* [35, p.11].

Example 1: It came time to implement \ ("scan"). On that day (in the first half of 1990), during lunch at Josie's Cafe on Yonge Street, I asked Ken innocently, "Have you ever wanted to scan something other than reduction?" The question was not even well-formed, because the APL paradigm is so powerful that it usually pushes aside alternative thoughts. Nevertheless, the discussion that then proceeded was productive, and quickly led to the realization that [35, p.16]:

Some APL expressions apply reduction implicitly. For example, APL scan applies its left argument reduction to successive prefixes of the right argument. The use of reduction ensured that computations such as sum scan can be effected by primitive function arguments to the operator, and that the overall result could be assembled in APL\360.

With J's more permissive assembly rules (and the use of boxed arrays), reduction is no longer necessary, and in many cases, inappropriate. Therefore in J, if a reduction is required, it must be specified. Thus `+/\` in J computes *sum scan* and `+/\.` computes *suffix sum scan*. The following J example does not use reduction, and is therefore not readily expressed using APL scan:

```
   <\ 'abcdef'
+-+--+---+----+-----+------+
|a|ab|abc|abcd|abcde|abcdef|
+-+--+---+----+-----+------+
```

Much recent work involved "recognizing" phrases and implementing them with special code. The expressive power of tacit expressions makes this a fruitful approach. For example, if `f` is a primitive proposition, then:

`f i. 0:`	first place where not `x f y`
`f i. 1:`	first place where `x f y`
`f i: 0:`	last place where not `x f y`
`f i: 1:`	last place where `x f y`
`+ /@:f`	number of places where `x f y`
`+./@:f`	`x f y` anywhere?
`*./@:f`	`x f y` everywhere?
`I. @:f`	indices where `x f y`

are supported by special code. A primitive proposition is one of the relational atomic verbs (scalar functions) `= ~: < <: > >:` or the verbs `E.` or `e.` (respectively, "substring match" and "member of"; a legend of the J words used in the text is available in Appendix B below.) For example:

```
   'boustrophedonic' (e.i.1:) 'aeiou'     NB. index of first vowel
1
   'boustrophedonic' (e.i:1:) 'aeiou'     NB. index of last  vowel
13
```

If the target is found near the beginning of the search, the result is computed instantaneously (and in time independent of the lengths of the arguments for atomic verbs `f`). Even if the target is found only at the end of the search, the improvement in time is by a factor of two or more. In either case the space used is constant for atomic verbs `f`.

Another recent development is a port to the Linux and Windows XP 64-bit operating systems on the AMD64 and Intel EM64T processors [36]. J64 has 64-bit integers and a 64-bit address space.

Ken had started to write a companion to the Abramowitz and Stegun classic, *Handbook of Mathematical Functions* [37]. Such writing was the motivation behind the extended-precision integer and rational number facilities; to the same end, extended-precision floating-point numbers will be also added. The following examples illustrate extended-precision integers and rational numbers:

```
   (+%)/\ 10 $ 1
1 2 1.5 1.66667 1.6 1.625 1.61538 1.61905 1.61765
1.61818
   (+%)/\ 10 $ 1x
1 2 3r2 5r3 8r5 13r8 21r13 34r21 55r34 89r55
   0j40 ": (+%)/ 100 $ 1x
1.6180339887498948482045868343656381177203
```

The hook x(+%)y is x plus the reciprocal of y, and so (+%)/ is the continued fraction. The first phrase computes convergents to the golden ratio phi in 64-bit IEEE floating-point numbers; the second computes rational convergents to phi; and the third computes phi to 40 decimal places.

3. Remembering Ken

The last photograph of Ken Iverson, with Rachel Hui, Jean Iverson, and Nicholas Hui, on Sunday, 2004-09-05, at the roof-top garden of the Manulife Center, Toronto, Canada. [photo by Stella Hui]

Ken and I worked on J in our homes in Toronto, I in my apartment in the High Park area and Ken in his apartment on Erskine Avenue and subsequently in the Manulife Center (a different apartment from the one in 1981, with an even more spectacular view). The arrangement placed me uniquely under Ken's influence.

Oftentimes I found myself at Ken's house, delivering the latest J version or working on a computer, and many times Ken would invite me to stay for lunch or dinner, explaining that "thou shalt not muzzle the ox when he treadeth out the corn". Little did I imagine in 1981 that the dinner I had with Ken and Jean then would be the first of many.

It is said that if you keep playing chess with a grandmaster, your skills can not help but improve. I had been in almost daily contact with the master since late 1986, and the following are a few of the things I have learned.

Credit

When Ken was at Harvard a fellow student habitually stamped his papers "Copyright" or "Confidential". Howard Aiken, head of the Comp Lab and Ken's thesis supervisor, advised, "Don't worry about people stealing your ideas. If it's any good, you'd have to ram it down their throats!" (A less picturesque version of this story is found in [38, p.240].)

Blame

Ken and I sometimes kidded each other about whose fault it was when something went wrong. He complained in jest that I as implementor had the attitude of "just tell me where to pour the concrete". And I, in turn, complained in jest that he as

designer had the attitude that he was never to blame: If it's a mistake in the implementation, then it's obviously my fault; and if it's a mistake in the design, well then I should have caught it during the implementation.

In truth, I am profoundly grateful to Ken for not complaining in the times when I failed him, even when I failed him the same way more than once. I believe that if necessary, he would have forgiven me for seventy times seven times (although even I would probably catch on before too long).

Secret to Success

In January 1992, I was in the final struggles of writing my book *An Implementation of J* [29], having difficulty and under time pressure. Around that time, Ken was polishing off several books and papers, seemingly effortlessly. I asked him what his secret was. His reply was, basically, "First, write 500 papers."

(Ken subsequently gave me some more immediately useful advice about writing.)

Reading and Writing Carefully

In the early days I did not have direct access to Ken, and therefore read his papers carefully. Another example is Arthur Whitney's "one page thing", which was the final thing I studied (for one week) before writing the first line of source code for J. Over the years, I have benefitted enormously from such careful reading.

I later realized that Ken wrote carefully, in the expectation that what he wrote would be read carefully. An exemplar of such writing is the description of "cut" in *Rationalized APL* [7, pp. 19-20]. It annoyed Ken to no end when accused of being "too terse", for example as in "the dictionary is too terse", when "terse" means "effectively concise".

Words

As others have recounted, Ken was deeply interested in words, their use and their etymology. He indeed did read the dictionary, and kept a copy of the American Heritage Dictionary [39] (along with other dictionaries) by his easy chair for ready reference. He especially encouraged me to consult the section on Indo-European roots in the back of the AHD, which makes deep and uncommon connections between words.

Many of Ken's relatives and friends received from him the AHD as a present. I myself did not because I'd already owned one years before I met Ken. In fact, *I* gave *him* the third edition of the AHD as a present.

I think nothing I had ever done impressed Ken quite as much as when I found the word "rhematic" (meaning, pertaining to the formation of words), a word he had been searching for for some time. Thus the phrase "rhematic rules of J" made its way into the J dictionary [30].

Books

Before long, Ken and I discovered that we shared a love of books. We often recommended reading material to each other. A sample of our recommendations:

I to Ken: *The Language Instinct* by Steven Pinker; *Climbing Mount Improbable* by Richard Dawkins; *Lincoln at Gettysburg* by Garry Wills. Ken to me: *From Dawn to Decadence* by Jacques Barzun; *Calvin and Hobbes*; *The New York Review of Books*.

And, of course, both of us had read *The Book of J* [40].

Politics

Unsurprisingly, Ken stood up for what he believed in. He once spent a few hours in jail after being arrested during a protest against the Vietnam War. In the same cell was Garry Wills, the historian and author.

Respect for People

Ken liked to tell the story of the stoic service agent: at an airport counter an irate traveller was berating the service agent over something or other, which the agent took with stoic forbearance. After the traveller went on his way, the next person in line told the agent, "I am amazed at how well you took that abuse." The agent smiled thinly and replied, "Oh, the gentleman is flying to Chicago, but his luggage is going to Moscow."

Ken also suggested, "Be kind to someone trying to make a living." I try to remember this every time a taxi or truck driver cuts me off on the road.

A Questioning and Flexible Mind

The flexibility of mind that Ken brought to his work (noted above) was also demonstrated in his everyday life. For example, it astounded me how often and how thoroughly Ken rearranged the furniture in his home. His explanation was, our needs have changed, and we rearranged our surroundings to meet those needs.

References

[0] Hui, R.K.W., "The N Queens Problem", *APL Quote-Quad*, Volume 11, Number 3, 1981-03.

[1] Falkoff, A.D., and K.E. Iverson, *APL\360 User's Manual*, IBM Corporation, 1968-08.

[2] Falkoff, A.D., and K.E. Iverson, "The Design of APL", *IBM Journal of Research and Development*, Volume 17, Number 4, 1973-07.

[3] Falkoff, A.D., and K.E. Iverson, "The Evolution of APL", *ACM SIGPLAN Notices*, Volume 13, Number 8, 1978-08.

[4] Iverson, K.E., "Notation as a Tool of Thought", *Communications of the ACM*, Volume 23, Number 8, 1980-08.

[5] Iverson, K.E., *Operators and Functions*, Research Report #RC7091, IBM Corporation, 1978-04-26.

[6] Iverson, K.E., and A.T. Whitney, "Practical Uses of a Model of APL", APL82, *APL Quote-Quad*, Volume 13, Number 1, 1982-09.

[7] Iverson, K.E., *Rationalized APL*, I.P. Sharp Associates, 1983-01-06.

[8] Iverson, K.E., "A Dictionary of APL", *APL Quote-Quad*, Volume 18, Number 1, 1987-09.

[9] Brown, J.A., "A Generalization of APL", Ph.D. Thesis, Department of Computer and Information Sciences, Syracuse University, 1971.

[10] Brown, J.A., *The Principles of APL2*, Technical Report 03.247, IBM Santa Teresa Laboratory, 1984.

[11] *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation, 1988.

[12] Falkoff, A.D., "The IBM Family of APL Systems", *IBM Systems Journal*, Volume 30, Number 4, 1991-12.

[13] Smith, R., "Nested Arrays, Operators, and Functions", APL81, *APL Quote-Quad*, Volume 21, Number 1, 1981-09.

[14] McDonnell, E.E., *Complex Numbers*, SHARP APL Technical Notes 40, I.P. Sharp Associates, 1981-06-20.

[15] Iverson, K.E., *Composition and Enclosure*, SHARP APL Technical Notes 41, I.P. Sharp Associates, 1981-06-20.

[16] Iverson, K.E., *Determinant-Like Functions Produced by the Dot-Operator*, SHARP APL Technical Notes 42, I.P. Sharp Associates, 1982-04-01.

[17] Bernecky, R., K.E. Iverson, E.E. McDonnell, R.C. Metzger, and J.H. Schueler, *Language Extensions of May 1983*, SHARP APL Technical Notes 45, I.P. Sharp Associates, 1983-05-02.

[18] Hodgkinson, R., "APL Procedures", APL86, *APL Quote-Quad*, Volume 16, Number 4, 1986-07.

[19] Steinbrook, D.H., *SAX Reference*, I.P. Sharp Associates, 1986.

[20] Iverson, K.E., and E.E. McDonnell, "Phrasal Forms", APL89, *APL Quote-Quad*, Volume 19, Number 4, 1989-08.

[21] Hui, R.K.W., "Some Uses of { and }", APL87, *APL Quote-Quad*, Volume 17, Number 4, 1987-05.

[22] Whitney, A.T., private e-mail, 2004-11-11.

[23] Pesch, R.H., private e-mail, 2004-11-11.

[24] Whitney, A.T., "A", plenary session, APL89, 1989-08.

[25] Orth, D.L., *K Reference Manual*, Kx Systems, 1998.

[26] Whitney, A.T., *Abridged Q Language Manual*, Kx Systems, 2004.

[27] McIntyre, D.B., "A Tribute to Roger Hui", *APL Quote-Quad*, Volume 27, Number 1, 1996-09.

[28] Iverson, K.E., "A Personal View of APL", *IBM Systems Journal*, Volume 30, Number 4, 1991-12.

[29] Hui, R.K.W., *An Implementation of J*, Iverson Software Inc., 1992-01-27.

[30] Hui, R.K.W., and K.E. Iverson, *J Introduction and Dictionary*, Jsoftware Inc., 2004.

[31] Iverson, K.E., "APL87", APL87, *APL Quote-Quad*, Volume 17, Number 4, 1987-05.

[32] Hui, R.K.W., K.E. Iverson, and E.E. McDonnell, "Tacit Definition", APL91, *APL Quote-Quad*, Volume 21, Number 4, 1991-08.

[33] Hui, R.K.W., K.E. Iverson, E.E. McDonnell, and A.T. Whitney, "APL\?", APL90, *APL Quote-Quad*, Volume 20, Number 4, 1990-07.

[34] *J User Conference Proceedings*, Iverson Software Inc., 1990-06-24.

[35] Burke, C., and R.K.W. Hui, "J for the APL Programmer", *APL Quote-Quad*, Volume 27, Number 1, 1996-09.

[36] Iverson, E.B., "J64 Beta Available", J Forum message, 2004-07-30.

[37] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964-06.

[38] Cohen, I.B., *Howard Aiken: Portrait of a Computer Pioneer*, MIT Press, 1999.

[39] *The American Heritage Dictionary of the English Language*, Third Edition, Houghton Mifflin Company, 1982.

[40] Bloom, H., and D. Rosenberg, *The Book of J*, Vantage Books, 1990.

Appendix A. Incunabulum

From *An Implementation of J* [29]

One summer weekend in 1989, Arthur Whitney visited Ken Iverson at Kiln Farm and produced – on one page and in one afternoon – an interpreter fragment on the AT&T 3B1 computer. I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this document.

Arthur's one-page interpreter fragment is as follows:

```
typedef char C;typedef long I;
typedef struct a{I t,r,d[3],p[2];}*A;
#define P printf
#define R return
#define V1(f) A f(w)A w;
#define V2(f) A f(a,w)A a,w;
#define DO(n,x) {I i=0,_n=(n);for(;it=t,z->r=r,mv(z->d,d,r); R z;}
V1(iota){I n=*w->p;A z=ga(0,1,&n);DO(n,z->p[i]=i);R z;}
V2(plus){I r=w->r,*d=w->d,n=tr(r,d);A z=ga(0,r,d);
 DO(n,z->p[i]=a->p[i]+w->p[i]);R z;}
V2(from){I r=w->r-1,*d=w->d+1,n=tr(r,d);
 A z=ga(w->t,r,d);mv(z->p,w->p+(n**a->p),n);R z;}
V1(box){A z=ga(1,0,0);*z->p=(I)w;R z;}
V2(cat){I an=tr(a->r,a->d),wn=tr(w->r,w->d),n=an+wn;
 A z=ga(w->t,1,&n);mv(z->p,a->p,an);mv(z->p+an,w->p,wn);R z;}
V2(find){}
V2(rsh){I r=a->r?*a->d:1,n=tr(r,a->p),wn=tr(w->r,w->d);
 A z=ga(w->t,r,a->p);mv(z->p,w->p,wn=n>wn?wn:n);
 if(n-=wn)mv(z->p+wn,z->p,n);R z;}
V1(sha){A z=ga(0,1,&w->r);mv(z->p,w->d,w->r);R z;}
V1(id){R w;}V1(size){A z=ga(0,0,0);*z->p=w->r?*w->d:1;R z;}
pi(i){P("%d ",i);}nl(){P("\n");}
pr(w)A w;{I r=w->r,*d=w->d,n=tr(r,d);DO(r,pi(d[i]));nl();
 if(w->t)DO(n,P("< ");pr(w->p[i]))else DO(n,pi(w->p[i]));nl();}

C vt[]="+{-<#,";
A(*vd[])()={0,plus,from,find,0,rsh,cat},
```

```
 (*vm[])()={0,id,size,iota,box,sha,0};
I st[26]; qp(a){R  a>='a'&&a<='z';}qv(a){R a<'a';}
A ex(e)I *e;{I a=*e;
 if(qp(a)){if(e[1]=='=')R st[a-'a']=ex(e+2);a= st[ a-'a'];}
 R qv(a)?(*vm[a])(ex(e+1)):e[1]?(*vd[e[1]])(a,ex(e+2)):(A)a;}
noun(c){A z;if(c<'0'||c>'9')R 0;z=ga(0,0,0);*z->p=c-'0';R z;}
verb(c){I i=0;for(;vt[i];)if(vt[i++]==c)R i;R 0;}
I *wd(s)C *s;{I a,n=strlen(s),*e=ma(n+1);C c;
 DO(n,e[i]=(a=noun(c=s[i]))?a:(a=verb(c))?a:c);e[n]=0;R e;}

main(){C s[99];while(gets(s))pr(ex(wd(s)));}
```

Appendix B. J Words Used In The Text

Full reference in *J Introduction and Dictionary* [30].

=	equal
<	box • less than
<:	less than or equal
>	greater than
>:	greater than or equal
+	plus
+.	or
*.	and
%	reciprocal
$	reshape
~:	not equal
/	insert (reduce)
\	prefix (scan)
\.	suffix
[	left (the left argument)
[:	cap (special fork)
]	right (the right argument)
":	format (thorn)
@:	at (compose)
E.	member of interval (substring search mask)
e.	member of
I.	indices (from boolean)
i.	index of
i:	index of last
0:	constant function 0
1:	constant function 1

A Tribute to Ken Iverson

from Donald B. McIntyre

A progression of great thinkers has moved the human race towards the adoption, first of an economical and efficient number system containing zero and based on place value, and then of a universal algebra, APL, which operates on arrays or multiple quantities, and is totally devoid of words.

There have also been those who resisted the inevitable progress, who found it difficult to adopt new and improved tools for thought. In our own time we hear appeals to revert from this high intellectual level and use English words, and to submit to the tyranny of scalars, as if Sylvester's eloquence a century ago had fallen on deaf ears.

Unlike its predecessors, APL is an executable notation. APL represents, in a phrase used by Babbage, the "triumph of symbols over words." As so many of our distinguished predecessors predicted, it makes reasoning easier. APL is the result of brilliant insight, careful thought, and hard work through at least 5,000 years. Iverson is the latest in a succession that includes Peano, Sylvester, Cayley, De Morgan, Boole, Newton, Leibniz, Napier, Stevinus, Fibonacci, Diophantus, and the unknown Egyptian whose work was copied by Ahmes the scribe.

In 1866 Sylvester proclaimed that: "To attain clearness of conception, the first condition is 'language,' the second 'language', The third 'language' – Protean speech – the child and parent of thought."

In reflecting on the significance of APL I have adopted a historical approach. Having done so I find that Sylvester had something to say on that subject also. The occasion was his Presidential Address to the British Association in 1869 when he said: "the relation of master and pupil is acknowledged as a spiritual and lifelong tie, connecting successive generations of great thinkers with each other in an unbroken chain."

[Ken Iverson is our master and we are his pupils.] We think in a different way because of APL.

The preceding paragraphs are from p.576 of my paper: **Language as an Intellectual Tool: From Hieroglyphics to APL** (IBM Systems Journal Vol.30, No.4 1991, p.554-581). They are copyright and are incorporated here with permission of IBM. The paper, which is referenced in the J

Dictionary, is available on *www.mcintyre.mc.uk*. It was written in part to justify the change from traditional APL symbols to the new notation used in J. The references to APL include J, as was stated explicitly by Ken Iverson in his important **A Personal View of APL** – the paper that immediately followed mine. (IBM Systems Journal Vol.30, No.4 1991, p.582-593)

An article in *Datamation* (May 1975, p.13) said "that the efficiency of APL today is due to Iverson's tight control of enhancements and to his insistence on 'bending only to what was right, not what would sell.' Associates talked of Iverson as a 'highly moral and genuine person' who appealed to people's sense of good taste, and convincingly so." Three decades on we echo that judgement.

On learning of Ken's death, a former colleague at Pomona College – an economist whose students use APL – reminded me of an APL course that I taught for Faculty Members some thirty years ago. He recounts how, in that distant past, a colleague had commented that perhaps I could be likened to Paul in relation to Ken's Jesus! It was 7 April 1964 that I saw the light on the road to my Damascus. On that day new doors opened when I attended the *Announcement of IBM System/360* and initiated one of the first purchase-orders! As a result I received a copy of the *IBM Systems Journal*, Vol.3, Nos.2&3, 1964, containing Ken's *A formal description of System/360*. Just as, when a stone is thrown in the air, you can determine its position at any moment of time provided you know its mass and initial velocity – so the formal description, like a mathematical equation, answers every question regarding System/360's behaviour. To discover that such a complex machine could be described in this way seemed miraculous.

For years I have pored over this remarkable document and shared my enthusiasm with everyone who would listen. Moreover, the *formal description* sent me to Ken's *A Programming Language* (1962) – there was an unused copy in the College library! Being already familiar with the IBM 7090, I was delighted to find that this computer featured as principal example.

Pomona College was, I believe, the first general customer site where System/360 was installed. Although ordered on the day System/360 was unveiled, we waited until October 1965 for delivery. For part of the time I had access to a System/360 at IBM's Data Center in Los Angeles, and enjoyed informal tuition from IBM system engineers. This helped me understand the machine's architecture and master System/360 Assembler Language. With Ken's formal description at hand I was in a favoured position.

Late in 1968, our IBM representative (Don Stanger), noticing *A Programming Language* on my desk, told me that IBM had implemented Iverson Notation and

called it APL. W.J. [Bill] Bergquist gave me a demonstration of APL at IBM in Los Angeles. In early 1969 two of my students, who had designed a small computer, wrote a formal description using Iverson Notation. We rewrote this in APL, and Don Stanger arranged for us to access the computer at Yorktown Heights and run a working model of the students' projected computer! What a thrill it was to see that Ken Iverson was signed on to the system we were using!

Ken, to whom I sent a copy, was surprised that it came from the geology department of a small liberal arts college! He was delighted and invited me to visit him at the Thomas J. Watson Research Center – timing the visit to attend the 1st APL Users Conference, *The March on Armonk* 1969, at Binghamton. Only one other attended from California.

In this way an extraordinary relationship was born – so that now I feel Ken's death as though I had lost an older brother whom I will always remember, respect and admire. From then on Ken was a patient guide, always willing to answer questions and never showing irritation with a slow learner. He replied to my last questions on 15 October 2004 at 19:48. The following afternoon Ken suffered a fatal stroke while preparing a J tutorial on his laptop.

In 1969 I was en route to Scotland for sabbatical leave. There, without access to an APL system, I used an APL ball on an IBM Selectric typewriter and wrote a paper on *APL and the study of Data Matrices*. I presented the paper in Philadelphia at the Geological Society of America's Annual Conference, using an acoustic coupler to link to the APL system at the Thomas J. Watson Center. Following the Conference I gave the Matthew Vassar lecture – *Hooked on APL!* – at Vassar College, and visited Ken and Adin Falkoff again at Yorktown.

McIntyre, Donald B., *Introduction to the Study of Data Matrices*, In: Models of Geologic Processes - An introduction to Mathematical Geology, P. Fenner, Editor, Washington D.C.: American Geological Institute. [Typed with an APL ball on an IBM Selectric typewriter] Invited speaker at Short Course on Models of Geologic Processes, American Geological Institute, Philadelphia, 1969.

In 1971, when Ken had become an IBM Fellow, he invited me to join him at the IBM Scientific Center in Philadelphia. It was a wonderful summer experience. From then on, wherever we were in the world, we never lost contact – in New York, Toronto, California, or Scotland.

1978 McIntyre, Donald B. *Experience with Direct Definition One-Liners in Writing APL Applications*, An APL Users Meeting. Toronto, September 18, 19, 20, 1978., Proceedings, I.P. Sharp Associates, Ltd., p.281-297 [IPSA Users Conference 1978, 1980, 1982]

1978 McIntyre, Donald B. *The Architectural Elegance of Crystals Made Clear by APL*, Toronto: September 18, 19, 20, 1978. Proceedings, , I.P. Sharp Associates, Ltd. [With Ken Iverson in the Chair]

1983 Keynote speaker:The first *Tool of Thought Conference*, Association for Computing Machinery, New York.

1983 Banquet Address: International APL83 Conference, Washington, DC.

After my retirement Ken visited us twice in Scotland and on several occasions I greatly enjoyed being Ken and Jean's guest in Toronto and at the famous "Farm" where J was born.

On May 30, 1990, Ken sent me the original J system and the *Dictionary of J*. After APL90 in Copenhagen, he came to Scotland to share his enthusiasm for J – his new dialect of APL, brilliantly implemented and named by Roger Hui. Ken came in order to persuade me to write a paper entitled *Mastering J* – Ken provided the title – for APL91 at Stanford. As the paper had to be submitted in the last days of December, it was a hectic rush, but Eugene McDonnell saved the day by shepherding the paper for inclusion in the APL91 Proceedings. It was probably the first paper on J written by a user rather than by a team member. There is a copy on my website.

I took Ken to Edinburgh airport, let him off, and parked the car, but when I rejoined him Ken broke the news that he had misread the ticket – the flight had left the previous day! We rushed to Glasgow airport for an alternative flight. Ken's credit card failed to register – fortunately mine worked. Although I was very distressed, Ken's philosophic response was characteristic: "It's only money" he remarked – a balanced attitude that more of us should espouse!

In 1992 I conducted a *Tutorial Day: J Workshop* for the British APL Association, London. As a result of the success of this workshop, Ken proposed that I spread the word by going on tour in North America. Of course I agreed and in less than 2 weeks I ran J workshops in Toronto, The University of Waterloo, Lawrence KS, Dallas TX, Phoenix AZ, Claremont CA, Palo Alto CA, Washington DC, and the College of Wooster OH. I gave more J workshops in Toronto in 1993.

Ken had a genuine interest in people. When he asked how you were, he really wanted to know. He always inquired after our son, Ewen, whose life is complicated by cerebral palsy, and Ken went out of his way to ask questions when he heard how Ewen is able to use a computer without a mouse. Ken and Jean have always been generous and big-hearted. They opened their home as a halfway

house to troubled youths, with as many as six living with their own four children. Some have remained life-long friends.

I have always been struck how Ken never indulged in hero-worship, and I have come to understand that this is consistent with his Norwegian heritage. It is a national characteristic of Norway that there is no place for heroes. Prince and citizen are equal.

Ken had a profound interest in the use of language. When told something, Ken was always interested in the speaker; he was honest, direct and straightforward. If he were bored his body language would show it. Ken used words as a fencer uses a rapier! He would have made a fortune had he received a commission on the countless copies of the American Heritage Dictionary's Indo-European Glossary that he persuaded people to use!

For Ken, a sentence is like a mathematical expression. He was laconic – using few words – and readers often complain of his terseness, as if this were a fault. [terse: AHD, Brief and to the point. OED, Neatly concise.]

Ken was generous in sharing unusual books that had interested him, and the use of words was a common thread. Here are some of the books Ken sent or recommended – all relate to words!

- Terrance Deacon: The Symbolic Species: The Co-Evolution of language and the human brain.
- K.M. Elizabeth Murray: Caught in the Web of Words: (The making of the Oxford English Dictionary).
- R.E. Friedman: Who Wrote the Bible.
- Simon Winchester: The Surgeon of Crowthorne: A Tale of Murder, Madness, and the Love of Words.
- Steven Pinker: The Language Instinct: How the Mind Creates Language.

I was happy to contribute to Ken's library: e,g, the Complete OED on CD, and Vol.2 of Cajori's History of Mathematical Notations – he already had Vol.1.

Ken had a fine sense of humour and often surprised with the pithy remark. I had given Ken a McIvor [son of Ivor] tartan tie, and when the APL Group in Philadelphia had a night out, and I was moved to give a spontaneous exposition of the Scottish Enlightenment – demonstrating to my own satisfaction that most good things (APL perhaps included?) were invented by Scots. This explains why

in August 2003 Ken sent me a copy of *How the Scots invented the modern World* inscribed succinctly "For DBM @80 KEI".

Ken's great joy was to devise novel ways of introducing mathematics to the general public – using the spirit of his much admired Hogben's *Mathematics for the Million* coupled with his own J notation. While working at the frontier of mathematical notation and computer language, Ken devoted enormous thought and energy to devise ingenious ways of tutoring non-mathematicians. On my last visit I was fortunate to see Ken in operation at his Toronto home as he led an on-line session of *Mathematics for the Layman* for five or six adults with little background in the subject.

Such a many-faceted person! It is no wonder we all miss him so much.

We pay tribute also to Jean Iverson, whose dedicated support throughout Ken's professional life made his splendid contributions possible. Ken acknowledged Jean's assistance in preparing the final draft of *A Programming Language*. Anyone who has opened that book will recognise the magnitude of her task! For a while in the 1970s Jean also ran APL Press and produced an APL Newsletter. Jean quietly and efficiently kept Ken and all the extended family afloat. We who benefit from Ken's work owe her much gratitude.

30 November 2004

Kenneth Iverson, APL and J: Some Personal Recollections

from Keith Smillie

The *American Heritage Dictionary* gives as a definition of **epiphany** the following: "A sudden manifestation of the essence or meaning of something." Its etymology is "Middle English *epiphanie*, from Old French, from Late Latin *epiphania*, from Greek *epiphaneia*, manifestation, from *epiphainesthai*, to appear : *epi-*, forth; see EPI- + *phainein*, *phan-*, to show; see **bhā-**[1] in Appendix." One of the best known epiphanies in Western culture is possibly that of Saul on the way to Damascus.

My own epiphany in programming languages was much less dramatic than Saul's. I was not going anywhere, there was no blinding light or voice from above, nor did I fall down. It was in 1965 or 1966, and I was sitting quietly in my office in the Department of Computing Science at the University of Alberta thinking about the commonly occurring statistical problem of classifying a number of observations given the left-hand end of the first frequency class, the class width and the number of classes. This is a problem I had already handled in other languages but now I was trying to program it in a new language called Iverson's notation or APL.

I had heard of Iverson's notation in a lecture just before I came to the University of Alberta in 1963 and had even read a little of *A Programming Language*. On arriving at the University, I found that my friend and colleague Bill Adams, whom I had known for several years, had just completed an MSc thesis in which he had used it. However I promptly forgot about all of this work for a couple of years while establishing an academic career, which involved amongst other duties teaching a course in probability, statistics and numerical analysis and another in Fortran.

I first met Ken Iverson in the Department during one of his visits to his native Alberta. I was soon caught up in Ken's enthusiasm which he had already imparted to Bill, and we soon had an IBM 1050 terminal linked by a dial-up connection to a computer in the IBM Research Center in Yorktown Heights. So Bill and I were able, with the cooperation of an indulgent Department which paid the telephone bills, to experiment with APL as an executable, rather than just a paper-and-pencil, notation.

Now to return to my own undramatic epiphany. After some work on the frequency classification problem, I realized I could express the algorithm in a one-line APL function. Gone were the do- and if-then-else-statements and the type

and array declarations of Fortran. I was so excited by this discovery that I carefully wrote the "one-liner" on a small strip of cardboard which I glued to a tie clip. I'm not sure whether I ever wore it outside my office or Bill's.

And so from such inauspicious beginnings began my association with APL, and my later association with J. Bill and I both used APL in courses, and in addition Bill used it in his work with computer architecture and I in my work with statistical algorithms. The technical details although important and exciting at the time may be passed over briefly: after the single 1050 there were one or two IBM 2741 terminals located in a hallway and connected to the new IBM 360/67, then one or two laboratories equipped with them, then a 2741 in the classroom with a connection to the main computer, in the mid 1970s an IBM 5100 for both classroom and office use, and of course from the early 1980s the personal computer. Then there were the conferences, only a few of which we went to, starting with the 1969 conference in Binghamton , the celebrated "March on Armonk". Reading the conference Proceedings today gives a rare glimpse of the beginnings of APL, and a look at how some of us appeared when we were 35 years younger.

My solution to the frequency distribution problem which kindled my enthusiasm for APL marked the beginning of many years of the use of array languages in statistical calculations. On looking over the lectures notes for my first APL lectures – given in about 1966 or 1967 and carefully handwritten in pencil on yellow paper – I see functions for mean, variance and standard deviation, frequency distributions in one, two and an arbitrary number of dimensions, contingency tables, probability distributions, and also various calendar problems which would dog me for years. This work would result in a package of statistical functions named *STATPACK* which would keep appearing for many years in various reincarnations with extensions into such areas as analysis of variance and non-parametric methods, always carefully documented in technical reports with red covers and black bindings and distributed by postal mail with a 5¼-inch floppy disk included.

Apart from adapting this work to the personal computer, I set APL aside for most of the 1980s while many of us in academia grappled with the seemingly insatiable demand for courses in "computer literacy", whatever that term might have meant. Fortunately this period was enlivened, even made bearable, by a close and most enjoyable association with Trenchard More and his Nested Interactive Array Language Nial during which I wrote amongst other things a Nial equivalent of *STATPACK*.

In 1991, a year before retirement, I chanced upon J which Ken was then introducing as a modern dialect of APL. I forget the paper which drew my attention to J. My earliest work in J was a one-page listing of some 60 J verbs for statistical calculations which I can remember showing to Ken on a visit to Toronto in August of the same year. I continued my work with J throughout the 1990s and into the early 2000s during which I had the pleasure of using J in its increasingly convenient implementations while enjoying the great luxury of retirement.

During all of my work with APL and J, Ken Iverson was always ready, by precept and godly example, to educate, encourage, correct, and even chastise as he considered necessary. I believe that the most important lesson I learned from Ken was that programming languages should be learned and taught as we learn natural languages, i.e., by using them in the real world to solve real problems and not by studying the grammar. I can still remember Ken saying at the Binghamton conference something like "No more manuals!" Ken's writings have influenced me very much and I would like to mention just a few of them.

The *APL\360 User's Manual*, which he wrote with Adin Falkoff, in its several editions was a constant companion and guide for many years. The technical report *APL in Exposition* was a source of splendid examples. I can still remember Ken asking me what I thought of the 15-page section "The Computer: A Device for the Automatic Execution of Algorithms" which he said was his "summary of computer science". (I loved it!) Then there was his landmark Turing Award Lecture "Notation as a Tool of Thought" which impressed upon me the vital importance of a good notation. Alfred North Whitehead's remark quoted on the first page set the tone for the entire paper: "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." The paper in the 1991 *IBM Systems Journal*, "A personal view of APL" provided an excellent transition from APL to J. One phrase which appears on the first page, "When I retired from paid employment, ...", I have quoted many times to explain my continued work after retirement, much to the mystification of a few friends and colleagues who consider work and money to be inseparable. And, finally, will I ever understand everything in the *Introduction and Dictionary* which Ken considered the only necessary reference for J? No, never!

I deliberately began this paper with a definition from the *American Heritage Dictionary* which was a great favourite of Ken's and which he introduced to countless users of APL and J. I find it indispensable in my reading and writing, and have four copies: on my computer desk (one of three dictionaries there), in the living room, in my study, and in my campus office. I know I don't refer to the

Indo-European roots as much as Ken would have liked, but I would be lost without the *AHD* with its definitions, etymology, Synonyms and Usage notes.

I do little programming these days, and spend most of my time on the history of computing. From time to time I use J as a very convenient means to model, say, a Turing machine or a simple machine-language computer. Then once again I become entranced by the power and beauty of the language as I develop, test and polish my programs. I also think of all the people I have met and worked with during my association with array languages, and the influence they have had on my life and work. I cannot imagine what my professional career would have been like without APL and J. I cannot imagine, either, how these languages would have been developed without the insight and perseverance of a person like Kenneth Iverson. To him I am profoundly grateful.

Thank you, Ken.

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.

What's Wrong with my Programming?

from Joey Tuttle

The first time I ever heard of APL or Ken Iverson was during a lunchtime discussion in Boulder, Colorado, when my manager, Bob Weiss, asked me and my co-worker, "What's APL?" And I said, "In what context?", and he said, "It's a programming language. I'm taking this course in programming for managers and the instructor keeps talking about it. But he never tells us any more about it. He made one remark: he said programmers who use it like it better than their wives. So I think we should find out something about it."

On my way back from lunch to my office, I had to walk past the IBM Library. One of the nice things about IBM in those days was they were very careful to keep good libraries, a wonderful asset. I hope we don't lose those to the world by the Internet. There was a magazine on the rack and it had a picture of Kenneth Iverson and Adin Falkoff and this new language APL. It was a tabloid-size thing, and I went in the library, sat down and read the article.

And I said, "Wow, this pretty interesting!"

At the time we were working on analyzing patterns in data from magnetic tapes. The guy I worked for had this patented idea for testing punched cards, which he transferred to tapes by damaging the data and looking at the patterns of failures. So we had these patterns of data to look at and APL looked perfect. You could analyze huge chunks of data all at once: I mean, up to sixteen thousand bytes, because you had a 32,000-byte workspace. It was amazing!

I immediately started pestering Larry and other people in Yorktown to become a user of this fledgling system, even though I didn't have an APL type element and had to learn to use it with a Correspondence Courier type element — which

perhaps explains why I didn't completely stumble when I transferred stuff to J later on. (That's an in-joke, I guess.)

I'd been teaching some physics courses at IBM in their internal education and I felt a need to evangelise about APL. I started a course in Boulder and with the students in this course quickly came to realise that knowledge of this stuff had changed our lives. I was talking just last night to one of those people. John Armitage was a PhD physicist working on rotating optical storage devices at IBM in Boulder in 1968, so this was pretty early work in that field. He said, "Yes, it absolutely changed my life."

Another one was a technician who came to the course and said, "Well, I don't know any math. I've never had math. I trained as a tech in the Navy." He was a test technician who read oscilloscopes and collected data. He came to me shortly after we were all fledgling students, and said, "I've just figured out how to display that data that we've been struggling with. It's really very simple, just a collapsing dyadic transpose of a four-dimensional matrix." I thought: how many mathematicians would be comfortable with a collapsing dyadic transpose of a four-dimensional array? But he thought it was just common sense. So it was that, and that kind of experience, that caused me to say this is a different way of thinking and that's what became an important theme in APL for me.

We were working with all this data and we wanted to read these tapes. Unfortunately at that time the only way you could get data into APL other than typing it was to punch a paper tape and read it on a 1050, which wasn't really too cool. (Although you could send very long messages to the operator because it had separate line-feed and carriage-return controls. That was fun.) But I called and complained about this. And Ken said, "Well, we have this new facility called shared variables which solves that problem." I was immediately interested in that and pestered them until at last we got the second installation of such a system in Boulder.

Another fellow there, Bob Shively [?], and I conspired to make a program which could read data from a card reader, or a magnetic tape of all things, or a disk drive. We were tinkering with this and that got me and Bob invited to Philadelphia (where the APL group had moved from Yorktown) to show this to Ken and Adin. We did that, and they were suitably impressed. But Bob had called this program APLIO, the name on the first card of the assembly deck. Adin was incensed by this and Ken wasn't very happy either. He said, "This has nothing to do with APL, this stuff, nothing to do with it." So Bob Shively became very agitated and went off and changed the name of the program to EIEIO. Ken thought this was fine, a pretty good description, but Adin still wasn't very happy,

and said in some exasperation, "Why don't you call it the TSIO — the Tuttle-Shively Input Output?" We went ahead and did that; the name stuck and and we won some IBM awards for it, which was nice.

A couple of years later, in 1972, my family moved to Philadelphia and I joined the group there. It was great fun working with all these large amounts of data. But the best thing about it was that every day at work if I had an idea or a question, even if it was a stupid idea, I could walk into Ken's office and say, "Ken, what about this?" I had the gift of this easy interaction with someone who clearly saw things way way above what I usually thought about. And occasionally he'd steal ideas from me.

I say I could drop in on him any time but not... just any time. Because then, and later, when Ken came to work for me, one of his requirements in his office was to have a couch, large enough to take a good nap on. Which he did, with regularity. When he felt a little bit tired, the door was closed: don't knock, I'm napping. Those times were definitely off limits.

That was the high point of being around Ken, was you could just interact with him, and you frequently got a listen and then a rebuke, or a 'clarification on your thinking'.

In 1973 or 1974 a manager at Kodak invited Ken to come and talk to Kodak corporate programmers, who were all required to work only in PL/1. Ken seized this opportunity, as he always did, and said to me, "Come along with me and tell them about all this new stuff you guys are doing." We met in Rochester with about forty programmers to talk about using APL, and what it was. I sat and listened to Ken talk. I had a list of things I was going to talk about and each thing that Ken said I marked it off my list. I was just marking off the last thing that I had anything prepared to say about when Ken said, "And now here's Joey Tuttle. He'll tell you about all the *good* stuff that's in APL." So I got up and said, "Well, I don't have very much to say, but are there any questions that we can answer?" A woman in the audience stood up, I think she was a senior staff member, and said "If I understand what you people are saying, you're suggesting that we should adopt a new way of thinking." Ken jumped up out of his chair, and said, "Yes! That's exactly what I'm saying. Yes, you need to do this. You know, there are very great benefits from that." And she said, "But that's very scary. That's like walking into a dark room and not knowing what's there. Very scary." He said, "Well, think of it this way. Walking into a dark room heightens your senses, and you'll get benefits from that too." I don't think she was very convinced by this, and later as we were leaving the lecture hall our host walking beside us said, "You know, of all the people in that room, she could use a new way of thinking."

By 1977 Ken's group had moved back to Yorktown, so I was working in Yorktown but living in Palo Alto. I left IBM and joined I.P. Sharp Associates.

In 1978 I attended a lecture by Don Knuth at Stanford campus about TeX, his typesetting system. Now I'd never met Dr Knuth, but I was interested in the typesetting and saw an announcement so I made sure to go. I was interested in what he had to say about typesetting, but towards the end of his talk he said,

> You know, if you begin to use TeX, you'll be joining a sub-culture. People will look at you and say 'What are you doing? Why are you using that strange notation?' And you'll try to explain how powerful it is. They'll be very sceptical and look at you very strangely, much as we do at people who use APL. We don't understand what it is that APL users are doing but it's obviously very powerful.

So I went up to him afterwards and we had a nice discussion, and we had more over the years since then. Knuth gives credit to Ken for many things, concepts and notations and so on, that he adopted from APL for his books. But, true to his word, while he respects Iverson, APL, and the people who understand it, he has no time for learning anything about it. It's not a part of his thought process. Many years later in a social conversation with Ken I said, "You know, Ken, you're my favourite language designer, and Don Knuth is my favourite programmer." And Ken said immediately, "What's wrong with my programming?"

After I left IBM to join I.P. Sharp Associates (IPSA), people asked me, "What's the worst thing about the change?" And I always replied that it was not being able to drop in casually on Ken and have conversations with him. I was overjoyed to end that when Ken retired from IBM to join IPSA in Toronto. Life was good.

By that time he'd worked out pretty radical changes in APL and published a paper in 1983 called "Rationalised APL". The APL development group there, which was under my charge at that time, undertook retrofitting Ken's new ideas to the SHARP APL system. I was a strong advocate for those at the time.

In 1986 I was involved in initiating a project called SAX, for "SHARP APL under Unix". I didn't invent the name, it was Paul Jackson in Palo Alto who chose the cute name, but it was a Rationalised APL system. Ken liked that but was unhappy it didn't have wide distribution, so people could see it.

Ken went further in his thinking and in 1987 published "A Dictionary of APL", a part of the thinking that a couple of years later led to the J programming language, which Ken continued to nurture and develop for the rest of his life.

About that same time, IPSA was acquired by Reuters, the British news and data services company, and like many acquisitions, this one had a very dampening effect on the fun we were all enjoying. As a senior manager in this new enterprise it gave me great satisfaction to help shelter the efforts of Ken, Roger Hui, Eugene McDonnell and others from the 'realities' of the corporate world. After all, it was a perfect payback for the same shelter Ken provided me fifteen years earlier at IBM.

I can say without any reservation that Ken Iverson was the single biggest influence on my professional life. I have long considered him a mentor and he will be sorely missed.

ANECDOTES

Ken Iverson Quotations and Anecdotes

compiled and edited by Roger Hui

- Ken's ancestors came from Trondheim, Norway. He remarked on the serious outlook of Norwegians by retelling a story he had heard on Garrison Keillor's radio show about Lake Wobegon.

  The couple had just returned from their honeymoon.

  "How was your honeymoon, Gullik?"

  "Could have been worse," replied young Gullik. On seeing the crestfallen expression of his bride, he hastened to add, "Could have been a lot worse."

  – Roger Hui and Eugene McDonnell

- Ken dropped out of school after Grade 9 because it was the height of the Depression and there was work to do on the family farm, and because he thought further schooling only led to becoming a schoolteacher and he had no desire to become one.

During World War II, Ken first served in the Canadian army and then in the R.C.A.F. as a flight engineer in PBY *Catalina* flying patrol boats. He finally learnt about universities from his Air Force mates, many of whom planned to return to university, thanks to government support for servicemen. While in the service he completed high school courses by correspondence. After the war he graduated *summa cum laude* with a B.A. from Queen's University and the M.A. and Ph.D. degrees from Harvard.

– Roger Hui

- Ken didn't get tenure at Harvard. He did his 5 years as an assistant professor and the Faculty decided not to put him up for promotion. I asked him what went wrong and he said, well, the Dean called me in and said, the trouble is, you haven't published anything but the one little book.

  The one little book later got the Turing Award.

  I think that is a comment on the conventional mindset of promotion procedures rather than a comment on Ken; it's a comment on academic procedure and on Harvard.

  – Fred Brooks, *A Celebration of Kenneth Iverson*, 2004-11-30

- Applied mathematics is largely concerned with the design and analysis of explicit procedures for calculating the exact or approximate values of various functions. Such explicit procedures are called algorithms or *programs*. Because an effective notation for the description of programs exhibits considerable syntactic structure, it is called a *programming language*.

  Much of applied mathematics, particularly the more recent computer-related areas which cut across the older disciplines, suffer from the lack of an adequate programming language. It is the central thesis of this book that the descriptive and analytic power of an adequate programming language amply repays the considerable effort required for its mastery. This thesis is developed by first presenting the entire language and then applying it in later chapters to several major topics.

  – overture of the Preface to *A Programming Language*, 1962

- Dijkstra: How would you represent a more complex operation, for example, the sum of all elements of a matrix M which are equal to the sum of the corresponding row and column indices?

  Iverson: $+ + /(M = \iota^1 \circ . + \iota^1)//M$

  – presentation of *Formalism in Programming Languages*, Working Conference on Mechanical Language Structures, 1963

[*Editor's note*: Ken's one-liner is found on page 25 of *A Source Book in APL*. According to the notation in *A Programming Language*, a valid solution would be $+/(M = \iota^1(\mu(M)) \overset{\circ}{+} \iota^1(\nu(M)))/M$. (Two slashes instead of one in front of the final M would also work.) Dijkstra posed a simple problem, as any conversant APL or J user would attest. The printed version probably contains clerical errors.]

- My first acquaintance with the notation that has since become APL (for several years it was either "Iverson Language", "Iverson Notation", or just plain "Iverson") started with an IBM Research Report by Kenneth Iverson called, *The Description of Finite Sequential Processes*.

  I don't have the paper handy at the moment so what I'm about to tell you is all memory; it may be mistaken in details but not in essence. I seem to remember that the first page was mostly given over to heading material, possibly an abstract, so that there were only two short columns of reading matter on it. And, again memory, it took me several hours to understand what those two short columns were all about.

  The author's approach was so different from anything I'd ever encountered that I had a difficult time adjusting to his frame of reference. At the end of the first page, a fair assessment of my state of mind would be that I had glimmerings but no hope.

  The second page took about as much reading time as the first but, since it had twice as much matter, I was clearly improving. The glimmerings were now fitful gleams. One thing had definitely changed, however. I had no doubts about the value of what I was reading. I was now virtually certain that the author had something to say and that I'd better find out what it was. The third page had an illustration that, in a few short lines, described George Dantzig's simplex algorithm simply and precisely.

  That was the overwhelming, crucial experience.

  In the previous thirteen years, I had participated in so many murky discussions of what was here presented with crystal clarity that I knew that what I was reading was of enormous significance to the future of computing.

  – Michael Montalbano, *A Personal History of APL*, 1982

- Before APL was called APL, it was called "Iverson notation". Ken mused that it should be called simply "the notation". After all, we don't say "God's grass", just "the grass".

  – Paul Berry

- I remember quite well the day I first heard the name APL. It was the summer of 1966 and I was working in the IBM Mohansic Laboratory, a small building in Yorktown Heights, NY. The project I was working on was IBM's first effort at developing a commercial time-sharing system, one which was called TSS. The system was showing signs of becoming incomprehensible as more and more bells and whistles were added to it. As an experiment in documentation, I had hired three summer students and given them the job of transforming "development workbook" type of documentation we had for certain parts of the system into something more formal, namely Iverson notation, which the three students had learned while taking a course given by Ken Iverson at Fox Lane High School in Mount Kisco, NY. One of the students was Eric Iverson, Ken's son.

  As I walked by the office the three students shared, I could hear sounds of an argument going on. I poked my head in the door, and Eric asked me, "Isn't it true that everyone knows the notation we're using is called APL?" I was sorry to have to disappoint him by confessing that I had never heard it called that. Where had he got the idea it was well known? And who had decided to call it that? In fact, why did it have to be called anything? Quite a while later I heard how it was named. When the implementation effort started in June of 1966, the documentation effort started, too. I suppose when they had to write about "it", Falkoff and Iverson realized that they would have to give "it" a name. There were probably many suggestions made at the time, but I have heard of only two. A group in SRA in Chicago which was developing instructional materials using the notation was in favor of the name "Mathlab". This did not catch on. Another suggestion was to call it "Iverson's Better Math" and then let people coin the appropriate acronym. This was deemed facetious.

  Then one day Adin Falkoff walked into Ken's office and wrote "A Programming Language" on the board, and underneath it the acronym "APL". Thus it was born. It was just a week or so after this that Eric Iverson asked me his question, at a time when the name hadn't yet found its way the thirteen miles up the Taconic Parkway from IBM Research to IBM Mohansic.

  – Eugene McDonnell, *A Source Book in APL*, 1981

- [D]esign really should be concerned largely, not so much with collecting a lot of ideas, but with shaping them with a view of economy and balance. I think a good designer should make more use of Occam's razor than of the dustbag of a vacuum cleaner, and I thought this was important enough that it would be worthwhile looking for some striking examples of sorts of overelaborate design. I was surprised that I didn't find it all that easy, except perhaps for the [designs] of programming languages and American automobiles. I think that designers seem to have this feeling, a gut feeling of a need for parsimony.

  – from of *The Evolution of APL*, HOPL Conference, 1978

- I asked Ken, I think it may have been at the HOPL Conference, "What is the touchstone to making an elegant programming language?" He said, "The secret is, it has to do what you expect it to do."

  If you stop and think about APL and if you stop and think about J and if you think about Ken's work generally, it is that high degree of consistency which is the product of an exceptionally clean mind, and a fierce determination not to invent any new constructs, until you have to.

  – Fred Brooks, *A Celebration of Kenneth Iverson*, 2004-11-30

- Algebra is the language of mathematics. It is therefore an essential topic for anyone who wishes to continue the study of mathematics. Moreover, enough of the language of algebra has crept into the English language to make a knowledge of some algebra useful to most non-mathematicians as well. This is particularly true for people who do advanced work in any trade or discipline, such as insurance, engineering, accounting, or electrical wiring. For example, instructions for laying out a playing field might include the sentence, "To verify that the corners are square, note that the length of the diagonal must be equal to the square root of the sum of the squares of the length and the width of the field", or alternatively, "The length of the diagonal must be $\sqrt{l^2 + w^2}$ ". In either case (whether expressed in algebraic symbols or in the corresponding English words), the comprehension of such a sentence depends on a knowledge of some algebra.

  Because algebra is a language, it has many analogies with English. These analogies can be helpful in learning algebra, and they will be noted and explained as they occur. For instance, the integers or counting numbers (1, 2, 3, 4, 5, 6, ...) in algebra correspond to the concrete nouns in English, since they are the basic things we discuss, and perform operations upon. Furthermore, functions in algebra (such as `+` (plus), `-` (subtract), and `*` (times)) correspond to the verbs in English, since they *do* something to the nouns. Thus, `2+3` means "add 2 to 3" and `(2+3)*4` means "add 2 to 3 and then multiply by 4". In fact, the word "function" (as defined, for example, in the *American Heritage Dictionary*), is descended from an older word meaning, "to execute", or "to perform".

  – *Algebra: An Algorithmic Treatment*, 1972, Chapter 1

- The inclusion of too many simple exercises may bore the quick student, but their exclusion may leave unbridgeable gaps in the experience of some. The student should therefore learn to use discretion in the doing of exercises, ranging ahead and skipping detail, but being prepared to return to do earlier exercises whenever unintelligible difficulties arise in later ones. The most serious difficulty most students find with this approach is psychological; one must learn to treat exercises as a potential source of light and delight rather than as a capriciously imposed drudgery.

– *Elementary Analysis*, 1976, Chapter 1

- The importance of nomenclature, notation, and language as tools of thought has long been recognized. In chemistry and in botany, for example, the establishment of systems of nomenclature by Lavoisier and Linnaeus did much to stimulate and to channel later investigation. Concerning language, George Boole in his *Laws of Thought* [p.24] asserted that "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

  Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought. Recognition of the important role of notation in mathematics is clear from the quotations from mathematicians given in Cajori's *A History of Mathematical Notations* [pp. 332, 331]. They are well worth reading in full, but the following excerpts suggest the tone:

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

  The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid.
  Charles Babbage

– overture of *Notation as a Tool of Thought*,
Turing Award Lecture, 1979-10-29

- The practice of first developing a clear and precise definition of a process without regard to efficiency, and then using it as a guide and a test in exploring equivalent processes possessing other characteristics, such as greater efficency, is very common in mathematics. It is a very fruitful practice which should not be blighted by premature emphasis on efficiency in computer execution.

Measures of efficiency are often unrealistic because they concern counts of "substantive" functions such as multiplication and addition, and ignore the housekeeping (indexing and other selection processes) which is often greatly increased by less straightforward algorithms. Moroever, realistic measures depend strongly on the current design of computers and of language embodiments. For example, because functions on booleans (such as `∧/B` and `∨/B`) are found to be heavily used in APL, implementers have provided efficient execution of them. Finally, overemphasis of efficiency leads to an unfortunate circularity in design: for reasons of efficiency early programming languages reflected the characteristics of the early computers, and each generation of computers reflects the needs of the programming languages of the preceding generation.

– finale of *Notation as a Tool of Thought*,
Turing Award Lecture, 1979-10-29

- In 1973 or 1974 Ken and I gave a talk at Kodak in Rochester to a group of 40 to 50 programmers who were required to work in PL/I. In the question period a senior staff member said, "If I understand what you people are saying, you are suggesting that we should adopt a new way of thinking."

  And Ken jumped up out of his chair and said, "Yes! That's exactly what I am saying!"

  – Joey Tuttle, *A Celebration of Kenneth Iverson*, 2004-11-30

- When I retired from paid employment, I turned my attention back to this matter and soon concluded that the essential tool required was a dialect of APL that:

  Is available as "shareware", and is inexpensive enough to be acquired by students as well as by schools

  Can be printed on standard printers

  Runs on a wide variety of computers

  Provides the simplicity and generality of the latest thinking in APL

  The result has been J, first reported in Reference 5.

  Work began in the summer of 1989 when I first discussed my desires with Arthur Whitney. He proposed the use of C for implementation, and produced (on one page and in one afternoon) a working fragment that provided only one function (+), one operator (/), one-letter names, and arrays limited to ranks 0 and 1, but did provide for boxed arrays and for the use of the copula for assigning names to any entity.

I showed this fragment to others in the hope of interesting someone competent in both C and APL to take up the work, and soon recruited Roger Hui, who was attracted in part by the unusual style of C programming used by Arthur, a style that made heavy use of preprocessing facilities to permit writing further C in a distinctly APL style.

– *A Personal View of APL*, 1991

- It's funny, but my recollection is that at the time I thought *I recruited Ken.* Whoever recruited whom, I won a great prize when Ken decided he and I should work together.

The final impetus that got J started was the one-page interpreter fragment that Arthur wrote, recorded in Appendix A of *An Implementation of J* and also reproduced in Appendix A below. My immediate reaction on seeing the page was recoil and puzzlement: it looked nothing like any C code I had ever seen. ("Is it even C?") However, Ken counselled that I should reserve judgment. As recounted in *An Implementation of J*, it then happened that:

> I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this document.
>
> The name "J" was chosen a few minutes later, when it became necessary to save the interpreter source file for the first time.
>
> – Roger Hui, *Remembering Ken Iverson*, Vector, 2004

- When Reuters took over I.P. Sharp in 1987, Ken retired from I.P. Sharp. But did he rest? Of course not!

In 1990 I was the Chairman of the annual APL conference in Copenhagen.

Not hampered by an installed user base and desiring to get rid of the character set problem, Ken had invented J, and submitted papers with the announcement to the APL90 conference.

Well – nothing Ken ever did left people cold – but the resulting discussion in the program committee was probably the hottest I have ever experienced. Finally I had to draw a line, seeing that the rest of the programme would not come into existence if this debate was not stopped.

"We will not decline a paper from Ken Iverson, and it will go into the main conference stream."

Not that a refusal to accept Ken's paper at APL90 would have done anything to slow him down, of course ...

– Gitte Christensen, *Ken Iverson in Denmark*, Vector, 2006

- My first name starts with J and my last name starts with K, followed by APL. So you know were I stand.

– Joel Kaplan

- Implementers I: Usually it was Adin who pulled down on the enthusiasm a little bit. Very often it was the APL programmers, the APL implementers which at that time were me and Dick Lathwell. We had I think an unfair advantage, because if we liked an idea, it would show up in the implementation overnight; and if we didn't like the idea, it might take a *really* long time to show up.

  At the "March on Armonk" conference Ken said this: "I would like to address a word to the implementers: God bless them. I say God bless them with the same mixture of admiration, appreciation, and exasperation as we say to the ladies, God bless them."

  And I recall what he snarled once when I was around, "Programmers are like women: you can't live with them, and you can't live without them."

  If that sounds inappropriate today, all I can say is, this was a long time ago.

– Larry Breed, *A Celebration of Kenneth Iverson*, 2004-11-30

- Implementers II: Stephen Wolfram gave a plenary presentation on Mathematica at the APL89 Conference in NYC, and sat at the same table as Ken in the banquet that evening. Ken asked Wolfram why it was that in Mathematica propositions don't have values 0 and 1 as in APL instead of `True` and `False`. Wolfram replied that he had no objections, but the Mathematica implementers were against it.

– Roger Hui

- Implementers III: The trouble with you, Ken complained to me in jest, is that you have the attitude of "just tell me where to pour the concrete".

– Roger Hui, *Remembering Ken Iverson*, Vector, 2004

- Brothers Brian and Barry Silverman made a 360 hardware and host OS emulator for the PC, running the original APL\360 source code. They demonstrated their APL\360 emulator to the Toronto APLSIG on 1998-05-26.

Ken and Roger Moore (and I) were in the audience. After seeing that O backspace U backspace T worked, Roger asked to see F overstruck with L, getting an E. The Silvermans were surprised by this last result.

– Roger Hui

- John Lawrence and Al Rose were returning at JFK from a trip. John was talking with the porter while waiting for Al to fetch the car. He tried explain that the two large suitcases held a 2741 computer terminal which the porter understood as a "computer". Al arrived and the porter had some difficulty getting the 2741 suitcases into the Ford Skyliner. Al asked the porter to wait while he put the roof down. As the Skyliner relays did their magic with the roof, the porter said: "Your son, he is a genius!"

  Ken often told this story which perhaps only makes sense to those who have seen a Skyliner roof operate.

  – Roger Moore

- I recall another story about Ken, witnessed firsthand, when he came to the National Security Agency, around 1970, to give a lecture on APL to an audience of several hundred people.

  He was introduced by someone who gave quite a complete description of Ken's work and personal history. Then Ken got up to speak. He started by saying that normally when he gives a lecture the host approaches him a few minutes before the start of the lecture and furtively scribbles a few notes about where he went to school, etc. Ken said he was quite surprised that no such discussion occurred this time, until he realized where he was! The audience had a great laugh.

  – Bob Smith

- At the AMS meetings in Washington DC in the late 1990's, Ken wandered over to the NSA booth.

  Ken: I don't suppose there's any point in asking you any questions, since you can't answer anyway.

  NSA guy: Sure I can. But then I'll have to kill you.

  – Roger Hui

- Later, Ken visited the Maple booth. He asked the young people staffing the booth, "How do you find the number of elements of a vector that are greater than a hundred?" (That is, `+/x>100` in J or APL.) He left them a few minutes later, still scratching their heads.

  – Roger Hui

- Ken interviewed a job applicant once and reported back, "I don't think we want him. I told him what we were doing and he didn't argue. He just listened."

  – Larry Breed, *A Celebration of Kenneth Iverson*, 2004-11-30

- One time I was complaining about an impertinent customer who was making suggestions, and Ken said, "You know, Arthur, when I was working at IBM, I learnt that it's very important, very important to listen to your customers. It's even more important to disregard what they say and do the right thing."

  – Arthur Whitney, *A Celebration of Kenneth Iverson*, 2004-11-30

- In a social conversation with Ken, I said, "You know, Ken, you are my favorite language designer and Don Knuth is my favorite programmer." And Ken said immediately, "What's wrong with my programming?"

  – Joey Tuttle, *A Celebration of Kenneth Iverson*, 2004-11-30

- The Judge's Prize was awarded to the isi entry, submitted by a team of five from Iverson Software:

  Chris Burke
  Roger Hui
  Eric Iverson
  Ken Iverson
  Kirk Iverson

  The isi entry was written in a language called J, Ken Iverson's successor to APL. ...

  Congratulations to the isi team! Without a doubt, "a bunch of extremely cool hackers" – and an extremely cool programming language.

  – The Judges, *ICFP Functional Programming Contest*, 1998

- In developing J, Ken accepted that APL had shortcomings and it was time to move on. New features in J had to be supported on their own merits. "Because it is in APL" was not a good argument. This led to a rift between the J group and the traditional APL community, that Ken simply ignored.

His careful writing meant that, in turn, he was a good editor. Papers or labs submitted to Ken for review provoked immediate, thorough, and careful responses.

He was willing to discuss APL and J with anyone, treating them as an equal – and often mistakenly assuming that the person he was speaking to had thought about the matter as much as himself.

He could talk on almost any topic under the sun.

– Chris Burke

- Ken believed passionately that brevity is essential to clarity. "Be concise," he'd say. For me, this made for endless tension in my role as popularizer or explainer. I never got over my belief that in English, readability is mostly redundancy. Ken would look at what I have drafted and say, "Long-winded. You can cut out two-thirds."

  – Paul Berry, *A Celebration of Kenneth Iverson*, 2004-11-30

- For me personally, the biggest contribution in my life was that Ken taught me to write. Ken wanted things very concise. When in doubt, leave it out. Do not put runways for your prose to take off or to land at the end of a chapter; when you're finished, quit.

  As a matter of fact, the first chapter of *Automatic Data Processing* was far and away the very densest, because we edited it and we edited it and we edited it.

  Ken taught me some useful productions: If it's a clause, turn it into a phrase. If it's a phrase, turn it into an adjective or an adverb. If it's an adjective or an adverb, omit it. And you apply these recursively.

  – Fred Brooks, *A Celebration of Kenneth Iverson*, 2004-11-30

- The secret to writing well? "First, write 500 papers."

  – Roger Hui, *Remembering Ken Iverson*, Vector, 2004

- On the first night of the course that Ken Iverson was giving in his notation at the IBM Watson Research Center, in 1962, I saw him writing on the chalkboard, first with one hand, later with the other, and thus back and forth throughout the evening.

Many years later Ken told me the tale of his ambidexterity. He was naturally left-handed, but when he went to the grade school in the small town of Camrose, in Alberta, his teacher made him learn to write right-handed, and, if I remember correctly, by tying his left arm behind him. He did this, and for many years wrote with nothing but his right hand.

While on the Harvard faculty, he saw that when a fellow teacher injured his writing arm, he was unable to teach until it healed. Ken thought about this, and, for insurance, recalling his natural left-handedness, set about developing writing skills with the left hand. Thereafter, he wrote with either hand, as he felt inclined, one way or the other. I don't recall which hand he favored, but it was probably his right.

– Eugene McDonnell

- I remember being surprised at finding the line "I sing of Olaf glad and big" in some examples that Ken wrote in Phrases1.A (1996-8) (and J Version 7 *Introduction & Dictionary*, (1993), p. 32). I knew his source, since I was interested in its poet – e.e. cummings – and my surprise was how Ken had ever run across it. The poem tells of Olaf, a conscientious objector – presumably in WWI – and the foul way he is tortured by the military.

  In *Programming in J* (1991) page 33 is: "Do you love me/ or do you not/ you told me once/ but I forgot" is anonymous – maybe by a greeting card poet. Also on page 40 is "With blackest moss", "Mariana", Tennyson.

  In *An Introduction to J* (1992), 17, "Nobly, nobly, Cape St. Vincent" is from Robert Browning's "Home Thoughts from the Sea". This is also on p. 17 of *J Introduction & Dictionary* (1993)

  Another poem Ken liked was Clarence Day's: "When eras die, their legacies / Are left to strange police. / Professors in New England guard / The glory that was Greece." This is on the ; : page of the J dictionary.

  He greatly admired Robert W. Service, the Canadian poet, who wrote "A bunch of the boys were whooping it up in the Malemute saloon" and many other Yukon poems.

  -- Eugene McDonnell

- Ken loved poetry and had a really good memory for it. A poem that he recited from time to time, with laughter in his voice, was *When Adam Day by Day* by A.E. Housman.

When Adam day by day
Woke up in paradise,
He always used to say
"Oh, this is very nice."

But Eve from scenes of bliss
Transported him for life.
The more I think of this
The more I beat my wife.

– Roger Hui

- I don't think of Ken as a person who told a lot of jokes but when he did, there was usually something more than just a funny story – there was something to learn or something to make you think. One story he told stands out in my memory and I've repeated it many times because in addition to being a good story, it also describes the central fallacy of communism. What's wrong with "From everyone according to their ability and to everyone according to their need"? And the answer is that you're dealing with people! Here's the story, a conversation between an interviewer and a farmer:

  I: If you had two acres of land and a friend had none, would you give him an acre of land?

  F: If I had two acres of land and a friend had none, then for the greater good of the state, I would give him an acre.

  I: If you had two horses and your friend had none, would you give him one of your horses?

  F: If I had two horses and a friend had none, then for the greater good of the state, I would give him a horse.

  I: If you had two cows and your friend had none, would you give him one of your cows?

  F: No!

  I: I don't understand, you would give him an acre and a horse, why not a cow?

  F: Well, I *have* two cows.

  – Jim Brown, *Remembering Ken Iverson*, Vector, 2004

- There is an 8-word Chinese phrase describing a well-ordered society, where the citizens are so good that they don't pick up items on roads that were dropped by accident, or need to lock their doors at night.

  夜不閉户 路不拾遺

  I once described this phrase to Ken. He immediately responded that he probably wouldn't like the degree of control and coercion necessary for that kind of result. Human nature being what it is, I think Ken was right.

  – Roger Hui

- Ken illustrated the importance of treating people with respect by telling the following story: At an airport counter an irate traveller was berating the service agent over something or other, which the agent took with stoic forbearance. After the traveller went on his way, the next person in line told the agent, "I am amazed at how well you took that abuse." The agent smiled thinly and replied, "Oh, the gentleman is flying to Chicago, but his luggage is going to Moscow."

  – Roger Hui, *Remembering Ken Iverson*, Vector, 2004

- Pascal's Wager is an argument that one should believe in God: If there is a God, then you better believe in Him in order to go to heaven; if there is no God, then believing in Him does no harm.

  Ken argued that there should be a special place in Hell for people who believe for this reason.

  – Roger Hui

- Ken aphorisms:

  [Beware of anyone who says,] let's you and him fight.

  Good fences make good neighbors.

  Who gets upset depends on whose ox is being gored.

  Ask the question: Who benefits?

  Trying to ease the pain by carrying out an unpleasant procedure a bit at a time, is like cutting off a dog's tail an inch at a time.

  Never give more than one reason for anything – the last one is always the real one. (recalled by Roland Pesch)

  – Roger Hui

- Ken's Erdös number is at most 3, achieved in at least two different ways:

Paul Erdös and Jeffrey Shallit, *New bounds on the length of finite Pierce and Engel series*, Séminaire de Théorie des Nombres de Bordeaux 3, 1991, pp. 43-53.

Eugene McDonnell and Jeffrey Shallit, *Extending APL to Infinity*, Proc. APL 80 International Conf., North-Holland, 1980, pp. 123-132.

Roger Hui, Kenneth E. Iverson, Eugene McDonnell, and Arthur Whitney, *APL\?*, APL90, 1990, pp. 192-200.

Paul Erdös, Nathan Linial, and Shlomo Moran, *Extremal Problems on Permutations under Cyclic Equivalence*, Discrete Mathematics 64, 1987, pp. 1-11.

Oscar Ibarra, Shlomo Moran, and Roger Hui, *A Generalization of the Fast LUP Matrix Decomposition Algorithm and Applications*, Journal of Algorithms 3, 1982, pp. 45-56.

Roger Hui, Kenneth E. Iverson, Eugene McDonnell, and Arthur Whitney, *APL\?*, APL90, 1990, pp. 192-200.

– Roger Hui

- The founder, Ken Iverson, inspired great loyalty the old-fashioned way: he earned it.

– Bill Clinton, *My Life*, 2004, p. 321

- I was having dinner with Ken and Jean at their house, and the story of grains of rice on a chessboard came up in the conversation. As a reward for some outstanding service to the Emperor, a magician was granted any wish. He asked the Emperor for one grain of rice on the first square of an 8 by 8 chessboard, two grains on the next square, four grains on the next, and so on, doubling on each square. The Emperor, once he realized how much rice was involved, got so upset that he chopped off the magician's head.

  The total was of course `(2^64)-1` grains of rice. So how much rice was that? I guess Ken didn't want to think about it during dinner, because he quickly said that it would cover the earth to a significant distance to the sun.

  I estimated that a grain of rice was roughly one-eighth of an inch on each side, remembered from grade school that the radius of the earth was about 4000 miles, and so on, and while continuing to eat dinner, did some mental calculations. After a while I was able to tell Ken and Jean, "There are a lot of cubic inches in a cubic mile; that amount of rice wouldn't even cover the earth."

  – Roger Hui

- A "Kenecdote" is an item in this collection.

  – Eugene McDonnell

- In an early talk Ken was explaining the advantages of tolerant comparison. A member of the audience asked incredulously, "Surely you don't mean that when `A=B` and `B=C`, `A` may *not* equal `C`?" Without skipping a beat, Ken replied, "Any carpenter knows that!" and went on to the next question.

  – Paul Berry

- The conjunction *under* `f&.g` in J (`f¨g` in SHARP APL) is defined as

```
f&.g y      ·     gi f g y

x f&.g y    ·     gi (g x) f (g y)
```

  where `gi` is the inverse of `g`. "Under" elucidates the important but often mysterious concept of duality in mathematics.

  Ken liked to use the "under anaesthetics" example to introduce the idea. Several steps were composed:

```
apply anaesthetics
      cut open
            do procedure
      sew up
wake up from anaesthetics
```

  The audience never failed to see that the inverse steps were pretty important! Ken also used the "pipe laying" example: dig a trench, lay the pipe, cover the trench.

  – Roger Hui

- I recall in the late 1980s a discussion with Ken, Arthur, and me around Ken's kitchen table, querying new aspects of the composition operator in Sharp APL and new variations we were contemplating in Sharp APL/HP (the composition operator is analogous to `&.` in J).

  Ken resisted agreeing to new variations of composition, wanting to "keep it simple". As part of his explanation he asked me where I have used a function and its inverse in real life ... I hesitated, pausing ...

  He enlightened me:

  Washing the dishes: Fill the sink / wash the dishes / empty the sink

Food from the fridge: Open the door / get the food / close the door

Have dinner: Set the table / eat dinner / clean up the table

Go to work: Drive to work / work in the office / drive home

Rubbish bins: Put out bins / council collects rubbish / bring in bins

Filing: Find appropriate file / update contents / restore file in order

The aspects were endless, but underlined Ken's desire to keep the mathematical implementations as simple as possible, pointing out that we need to picture these in terms of our own real life experiences to better understand them.

It was an interesting discussion and gave me some guidelines of where Ken was coming from in so much of his language design.

– Rob Hodgkinson

- In J, an isolated sequence of verbs `f g h` derives a *fork*, defined as follows:

```
   (f g h) y     ·     (f y) g (h y)

x (f g h) y      ·     (x f y) g (x h y)
```

In March 2005, I thought of an extension where the isolated sequence `m g h` with `m` being a noun, previously an error, is defined to mean `m"_ g h`. (`m"_` is a constant verb producing `m` as its result.) Many verbs in J (and APL) were designed so that fixing the left argument of the dyad made a sensible monad, and there is a *commute* adverb `~` where `x f~ y · y f x`. Therefore, the new fork is very useful. Two examples:

0. Lower case from upper case.

```
   U   =: a. {~ (i.26)+a.i.'A'
   L   =: a. {~ (i.26)+a.i.'a'
   lfu=: (L,a.) {~ (U,a.) i. ]
   lfu 'PROTASIS apodosis'
protasis apodosis
```

1. An identity from mathematics.

```
   sin=: 1 o. ]
   cos=: 2 o. ]


   (^@j. = cos + 0j1 * sin) 1 2 3 0.1j_0.2
1 1 1 1
```

The extension is "obvious". Ken or I (or anyone else) could have thought of it any time between 1989 and 2004, but we didn't. Of all the new work in J since his passing, I wish I could tell Ken about this one. I think it would have made his day.

– Roger Hui

- ... one of [Howard Aiken's] graduate students was becoming a little paranoid that somebody might steal one of the fantastic ideas from his thesis, and so he was stamping every copy "Confidential". Aiken just shook his head and said, "Don't worry about people stealing your ideas. If they're any good, you'll have to ram them down their throats!"

– presentation of *The Evolution of APL*,
HOPL Conference, 1978

- Once launched on a topic it's very easy to forget to mention the contributions of others, and although I have a very good memory, it is, in the words of one of my colleagues, very short, so I feel I must begin with acknowledgments, ...

– presentation of *The Evolution of APL*,
HOPL Conference, 1978

Advertising in Vector

All queries regarding advertising in VECTOR should be made to Gill Smith, at 01439-788385, Email: apl385@compuserve.com.

Submitting Material to Vector

The Vector working group meets towards the end of the month in which Vector appears; we review material for issue n+1 and discuss themes for issues n+2 onwards. Please send the text of submitted articles to the Vector Working Group via the Editor:

The Editor, Vector
81 South Hill Park
London
NW3 2SS

Email: editor@vector.org.uk

Please supply as much material as possible in machine-readable form. Our preferred format is HTML, using simple markup – we apply Vector's style sheet. Source code can be tagged with `<pre>` or `<code>` tags as usual. You can further distinguish language within these tags (which would be important if, for example, you were comparing J and APL expressions) with the class attribute. For example: `<code class="J">MEAN=:+/%#</code>` would be typical. Authors wishing to use MS Word should install a copy of the APL385 Unicode TrueType font and use Unicode APL symbols if their interpreter permits 'copy as Unicode' from the APL session. Otherwise we can convert from Dyalog APL, APL+Win, APLX, and APL2 mappings if necessary.

Camera-ready artwork (e.g. advertisements) and diskettes of 'standard' material (e.g. sustaining members' news) should be sent to Vector Production, Brook House, Gilling East, YORK YO62 4JJ. Please also copy us with all electronically submitted material so that we have early warning of possible problems.

Subscribing to Vector

Your Vector subscription includes membership of the British APL Association, which is open to anyone interested in APL or related languages. The membership year runs from 1st May to 30th April. The British APL Association is a special interest group of the British Computer Society, Reg. Charity No. 292,786

Name: _____
Address: _____
_____
Postcode / Country: _____
Telephone Number: _____
Email Address: _____

UK private membership	£20	❑
Overseas private membership	£22	❑
Airmail supplement (not needed for Europe)	£4	❑
UK Corporate membership	£100	❑
Corporate membership overseas	£110	❑
Sustaining membership	£500	❑
Non-voting UK member (student/OAP/unemployed)	£10	❑

PAYMENT – in Sterling or by Visa/Mastercard only

Payment should be enclosed with membership applications in the form of a UK Sterling cheque to "The British APL Association", or you may quote your Mastercard or Visa number.

I authorise you to debit my Visa/Mastercard account

Number: ⌊⌊⌊⌊ ⌊⌊⌊⌊ ⌊⌊⌊⌊ ⌊⌊⌊⌊ Expiry date: ⌊⌊ | ⌊⌊

for the membership category indicated above,

❑ annually, at the prevailing rate, until further notice
❑ one year's subscription only

Data Protection Act:
The information supplied may be stored on computer and processed in accordance with the registration of the British Computer Society.

Signature: _____

Send the completed form to:
BAA, c/o Rowena Small, 12 Cambridge Road, Waterbeach, CAMBRIDGE CB5 9NJ, UK

VECTOR

VECTOR is the quarterly Journal of the British APL Association and is distributed to Association members in the UK and overseas. The British APL Association is a Specialist Group of the British Computer Society. APL stands for "A Programming Language" – an interactive computer language noted for its elegance, conciseness and fast development speed. It is supported on most mainframes, workstations and personal computers.

SUSTAINING MEMBERS

The Committee of the British APL Association wish to acknowledge the generous financial support of the following Association Sustaining Members. In many cases these organisations also provide manpower and administrative assistance to the Association at their own cost.

Causeway Graphical Systems Ltd
Brook House, Gilling East
YORK YO62 4JJ
Tel: 01439-788413
Web: www.causeway.co.uk

Dyalog Ltd
Grove House, Lutyens Close,
Chineham Court, BASINGSTOKE,
Hants, RG24 8AG
Tel: 01256-338461
Email: sales@dyalog.com
Web: www.dyalog.com

Insight Systems ApS
Nordre Strandvej 119G
DK-3150 Hellebæk, Denmark
Tel: +45 70 26 13 26
Email: info@insight.dk
Web: www.insight.dk

MicroAPL Ltd
The Roller Mill, Mill Lane
Uckfield, E.Sussex TN22 5AA
Tel: 01825-768050
Email: MicroAPL@microapl.demon.co.uk
Web: www.microapl.co.uk/apl

Kx Systems
555 Bryant St., No. 375
Palo Alto CA 94301 USA
Tel: +1 866 kdb fast
Email (general enquiries): info@kx.com
In Europe: Simon Garland, simon@kx.com

Optima Systems Ltd
Optima House, Mill Court, Spindle Way, Crawley,
West Sussex, RH10 1TT
Tel: 01293 562700
Email: paul@optima-systems.co.uk

APL Italiana
via Monferrato 1, 144 Milano, Italy.
Email: stf@apl.it

Compass Ltd
Compass House
60 Priestley Road
GUILDFORD, Surrey GU2 5YU
Tel: 01483-514500

APL2000 Inc
One Research Court
Suite 140
Rockville MD 20850, USA
Tel: +1 (301) 208 7150
Email: sales@apl2000.com
Web: www.apl2000.com

HMW Computing
Hamilton House,
1 Temple Avenue,
LONDON EC4Y 0HA
Tel: 0870-1010-469
Email: HMW@4xtra.com

Soliton Inc
44 Victoria Street, Suite 2100
Toronto, Ontario M5C 1Y2, Canada
Tel: +1 (416) 364 9355
Email: sales@soliton.com
Web: www.soliton.com

First Derivatives plc
First Derivatives House
Kilmorey Business Park, Kilmorey Street,
Newry, Co. Down, N. Ireland BT34 2DH.
Tel: 028 3025 2242
Email: enquiries@firstderivatives.com

The Carlisle Group
544 Jefferson Avenue, Scranton PA, 18510
USA.
Tel: +1 (570) 963-2036
Web: www.carlislegroup.com

The British APL Association

The British APL Association is a Specialist Group of the British Computer Society. It is administered by a Committee of officers who are elected by a postal ballot of Association members prior to the Annual General Meeting. Working groups are also established in areas such as activity planning and journal production. Offers of assistance and involvement with any Association matters are welcomed and should be addressed in the first instance to the Secretary.

2005/2006 Committee

Chairman	Paul Grosvenor 01293-562700 paul@optima-systems.co.uk	Optima House, Mill Court Spindle Way, CRAWLEY West Sussex, RH10 1TT
Secretary	Anthony Camacho 0117-9730036 acam@blueyonder.co.uk	11 Auburn Road Redland BRISTOL, BS6 6LS
Treasurer	Nicholas Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB5 9NJ
Journal Editor	Stephen Taylor sjt@vector.org.uk	81 South Hill Park LONDON NW3 2SS
Education	Dr Alan Mayer 01792-295779 a.d.mayer@swansea.ac.uk	European Business Management School, Swansea University, Singleton Park, SWANSEA SA2 8PP
Activities	Ray Cannon 01252-874697 ray_cannon@compuserve.com	21 Woodbridge Rd, Blackwater Camberley, Surrey GU17 0BS
Projects	Ian Clark earthspot2000@hotmail.com	
Administration	Rowena Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB5 9NJ

Journal Working Group

Editor:	Stephen Taylor	see above
Production:	Adrian & Gill Smith	Brook House, Gilling East, YORK (01439-788385)
Advertising:	Gill Smith	Brook House, Gilling East, YORK (01439-788385)

Support Team: Anthony & Sylvia Camacho (0117-9730036),
Ray Cannon (01252-874697), Stephen Taylor (077-1340 0852)
Bob Hoekstra (01483-771028), Marc Griffiths, Ian Clark

Typeset by APL-385 with MS Word for Windows
Printed in England by Short-Run Press Ltd, Exeter