ED 226 728 IR 010 609

AUTHOR McIntyre, Donald B.; And Others
TITLE Teaching APL.
INSTITUTION Sharp (I. P.) Associates, Ltd., Toronto (Ontario).
PUB DATE Oct 80
NOTE 60p.; Papers presented at the APL Users Meeting (Toronto, Canada, October 1980).
PUB TYPE Collected Works - Conference Proceedings (021) -- Reports - Descriptive (141)

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS *Computer Oriented Programs; *Computer Programs; *Course Content; Foreign Countries; Higher Education; Induction; Program Implementation; *Programing Languages; Secondary Education; *Teaching Methods
IDENTIFIERS *APL Programing Language; Canada; Pomona College CA; United States; University of Guelph (Ontario)

ABSTRACT

This booklet contains four articles for use by instructors of the APL computer programming language. In the first paper, "APL in a Liberal Arts College," Donald McIntyre describes experiences in the implementation of an introductory computing course at Pomona College and the presentation of a faculty seminar in computing. Principles used in APL instruction are discussed, and specific instructional examples and a 13-item bibliography are included. Then, Gillian Wade, in "Teaching APL in an Academic Environment" describes the status, organization, and history of APL courses at the University of Guelph, Ontario, and presents examples of applications by students. "APL in the Classroom" by Ted Edwards outlines classroom applications for courses in which APL has been taught as a subject or used as a language of discourse to teach another topic. Examples of lesson material, reasons for APL success, limitations, and possible extensions are presented. Finally, Kenneth Iverson, in "The Inductive Method of Introducing APL," treats some of the lessons to be drawn from analogies with the teaching of natural language, examines details of their application in the development of a 3-day introductory APL course, and reports results. Implications for more advanced courses are also briefly discussed. (LMM)

ED226728

I.P. Sharp Associates

Teaching APL

IR010609

2

The following papers were reprinted from 1980 APL Users Meeting Proceedings October, 1980, Toronto, Canada.

APL IN A LIBERAL ARTS COLLEGE

Donald B. McIntyre
Geology Department
Pomona College
Claremont, California

Introduction

Iverson notation was introduced at Pomona College in 1964 when the "Formal Description of System/360" was published in the IBM "Systems Journal", Vol. 3, #2-3. Pomona, a liberal arts undergraduate college with 1300 students, ordered a 360/40 on April 7, 1964, and one of the first System 360's shipped by IBM was delivered in September, 1965. It had only 16K memory, and it had no console typewriter. The modern concept of a "System" was not recognized; all programming was in Basic Assembler (which required the card deck to be passed twice through the reader/punch), and the supposedly privileged instructions (such as LPSW) were freely available to any programmer.

The long-term advantage to me of so primitive and small a system was that it prepared me for the "Formal Description". If I had had a fully operational system, I would doubtless never have seen the IBM "Systems Journal", and would have remained in ignorance of Iverson's work. It was also fortunate that Chapter 2 of Iverson's "A Programming Language" used the IBM 7090 as the example of microcoding, because I was familiar with that machine through access to Western Data Processing Center on the U.C.L.A. campus. When I discovered the "Formal Description", and through it Iverson's book, I became greatly interested in the fact that a language had been developed that permitted the description of so complex a machine as a 360 computer. The intellectual potential of such a tool struck me as enormous and completely in the tradition of the historical development of mathematics. Quite apart from its obvious practical value, I was persuaded of its vital importance to the subject matter that is the core of liberal arts studies.

Although I had taught informal classes in Assembler and FORTRAN, I taught the first class for credit in computer science at Pomona in the academic year 1968-69. A feature of the class was a study of the comparative anatomy of computers, with special emphasis on the architecture of System/360 and its relationship to the structure of the IBM machines that preceded it. Iverson notation was used in the class, and students were encouraged to write formal descriptions of machines that they designed themselves. While I was teaching this class, I discovered to my delight that Iverson notation had been implemented as the language called APL.

Because no terminals were attached to our 360, we had to be content to use APL as a tool of thought, and this turned out to be very much to my advantage in appreciating its value. However, through the courtesy of our local IBM representatives, I was able to make some use of APL at the Thomas J. Watson Research Center, Yorktown

Heights, by means of terminals in IBM's Riverside office, California. On weekends and at night, some of my students were able to recast their Iverson notation into APL that they could execute on the computer at Yorktown Heights. The success of the class led me to attend the APL Users Conference ("The March on Armonk") at S.U.N.Y. Binghamton in July 1969, and I was fortunate also in being able to visit the APL group at Yorktown Heights.

From that time I worked to achieve an APL system for Pomona College, but this was not easy to accomplish. Although the configuration of our 360 grew, the administrative work on it also increased. It was never possible to attach terminals, and for many years we had access to APL only through the kindness of friends, until the College was able to lease a small number of ports on a DEC-10 owned by the other colleges in the Claremont cluster. During these years I did not teach the "Introduction to Computing" again, but on one occasion the College was fortunate in having Dr. W.J. Bergquist, of IBM, teach the course. In the meantime, I continued to use APL in my own work, and notably in my class in elementary crystallography, which is taken by all students who major in geology.

I was privileged also to spend the summer of 1971, at the invitation of Dr. Iverson, with the APL group, which was then under the management of Adin Falkoff at the IBM Scientific Center in Philadelphia.

In 1975, the Geology Department acquired the second 5100 desk-top computer shipped to a customer by IBM. Equipped with 4 video-monitors, the 5100 let me use APL in the classroom. Enormous progress resulted from the fact that students could get free use of APL 24 hours a day, and several developed into highly skilled users. Moreover other departments began to use the 5100 in Geology, and this machine was followed by a second 5100, in the Physics Department, and later by 5110's in Botany and Mathematics. All of these were used almost entirely as APL machines.

In 1977-78, through a grant from the Mellon Foundation for Faculty Development, and with the cooperation of the IBM Corporation, the College was able to have Dr. Don L. Orth teach courses both for faculty and students. This did much to lay the foundation at Pomona for a major advance in the appreciation of APL as an extremely important tool of thought.

On January 30, 1979, Pomona College ordered an IBM 4341, and an IBM 4331 was installed as an interim processor in May, 1979. This was the second 4300 System shipped to a customer. By this time it was generally understood on the campus that a good implementation of APL was essential, and this played a major role in our decision to purchase IBM equipment. During the first semester of 1979-80 we had limited access to VS APL under CMS through IBM 3277 terminals, which have since been replaced by 3278's. The Introductory Chemistry class, with about 150 students (which is a very large number for any class in Claremont), and some smaller classes in other departments, immediately began using APL on the 4331. During the second semester the APL Extended Editor was installed and 30 terminals were available to users. The number of active users rose to about 500. In addition to servicing the academic users, the 4331 continues to do most of the administrative computing for all the Claremont colleges formerly done by the 360/40. The administrative work is run as a virtual machine under the unsupported DOS-26 operating system, and there are, of course, times when the machine is overloaded.

Pomona College is a small liberal arts college with an exceptionally diverse curriculum, including Astronomy, Botany, and Geology. There are, however, no departments of engineering or computer science. Students who wish to do so are able to enroll in courses at our close neighbor, Harvey Mudd College, an undergraduate college strong in science and engineering. Harvey Mudd offers courses in computer science, but these have emphasized structured programming, notably ALGOL and PASCAL, in contrast to APL.

Because I had successfully demonstrated to colleagues in disciplines ranging from Economics to Chemistry that many practical problems can be solved remarkably easily be means of APL, I was asked to teach an "Introduction to Computing" in the second semester of 1979-80. The enrollment was about 140, making it one of the largest classes on the campus. Twenty-five different majors were represented by the students enrolled, and faculty members from such diverse areas as Biology, Geology, Art, French, and Physical Education audited the class. The impact of this class was sufficient to demand a two-week summer class in APL for faculty members, and it is probably safe to say that APL, using Direct Definition, is recognized as the principal language for computing at Pomona College.

The original System 360/40 was given to Pomona by Mr Frank Roger Seaver of Los Angeles, and the further generosity of the Foundation he created made it possible for the College to purchase the 5100 computers, the 4331 (soon to be replaced by the 4341), and all the peripheral equipment. Without this support, it would, of course, have been impossible for Pomona to provide the equipment needed for quality instruction.

"Introduction to Computing": A Semester Course

I offered the course in 1979-80 to all interested students, without pre-requisites of any kind. The number of senior students who enrolled would have made a class of substantial size at Pomona, but had the class been restricted to seniors, no progress would have been made towards alleviating the problem in future years. Now that all students have had an opportunity to take the class, I intend to restrict enrollment and give preference to freshmen and sophomores.

Including auditors, nearly 160 people took the class, and the size required our meeting in a large lecture room in which it was impossible to use video-monitors. Had the room been smaller I would have used monitors attached to a 5100, because we lack the equipment needed to display the screen of a 3278 terminal to a group. The class met for two 80 minute periods each week. In addition to teaching this class, I taught classes in geology and performed administrative duties for the Geology Department and the Computer Center as best I could.

The concept of IBM 3278 terminals is to provide clusters that share a printer, and the largest of our clusters consists of 11 terminals in a rather small room in the Mathematics Department. I could not schedule the use of this room, and my students had to compete with others for the use of the terminals. I had the help of only one assistant, Jeff Siegel, who was a senior premedical student and greatly interested in the power and proper use of APL. At night he monitored the cluster of 11 terminals, giving help to any students who needed it, while I travelled round the other clusters, like a doctor making the rounds of the wards. In the future I hope to be able to schedule the students' time on the terminals, in the way that laboratory work is scheduled in the Chemistry Department. I would then restrict enrollment based on the number of laboratory sessions that can be staffed and the number of terminals available. Students who have already taken the course should be able to act as assistants in the future.

Because there were some people on campus who were not persuaded of the utility of APL, I considered the course to be, as its title indicated, an Introduction to Computing in general, and not merely an introduction to APL. My object was to give the students an understanding of the principles of computers and computer languages, and their historical development, so that every one could judge independently about the proper role and usefulness of APL. It would, of course, have been impossible for me to conceal my own judgement on this, and I did not pretend to do so.

Because we were using VM, every student had his own "Virtual Machine". The Conversational Monitor System (CMS) permits each virtual machine to have a "Profile" that is executed whenever the user logs on, and it would have been possible to have given each student a Profile that would have automatically invoked the APL system, but I chose not to do so. This meant that the students became familiar with the management of their resources under CP (Control Program) and CMS. For example they learned how to define and attach additional (temporary) disks; how to link to virtual disks of other users; how to write CMS EXECs (including their own Profiles), how to use IBM's full-screen editor (EDGAR); how to rename, copy, pack, and erase CMS files, and how to send CMS files to other users or to the line printer. It is easy to teach this kind of information when the student has a terminal in front of him, but I had to teach by remote control, and to do this I wrote a manual with the following sections:

1. CP, CMS and APL. Three levels of operation
2. Interrupting your Virtual Machine
3. QUERY for information about your Virtual Machine
4. Copying CMS Files from disks on other Virtual Machines
5. Full-screen Editing and IBM's EDGAR editor
6. FLIST and the management of your resources under CMS
7. APL System Commands
8. IPF: IBM's full-screen Interactive Productivity Facility
9. Receiving and sending Mail
10. BROWSE
11. Reading and executing FORTRAN programs
12. Writing CMS EXECs
13. Attaching additional disk drives to your Virtual Machine
14. Linking to virtual disks on other machines
15. Using Program Function keys
16. Files created by the FORTGI compiler

17. FORTRAN subroutines

18. Management of APL Workspaces

19. APL Public Libraries

20. Direct Definition of APL functions

21. Relational Data Bases

22. QUERY: an APL simulation of IBM's "QUERY-BY-EXAMPLE"

23. Curve fitting (regression) and statistics using APL

24. Using Arrays in APL: an example from Economics

25. Logic: APL implementation of the work of Boole and Jevons

26. Linear Programming in APL

27. Introduction to plotting, using Tektronix 618 display and IBM's APL Graphpak

28. Data Bases with multiple-line entries, using APL

29. Storage of Floating-Point Numbers in the 4300

30. Introduction to Assembler Language

31. Introduction to the use of a memory dump

I asked each student in the class to attempt a project that would illustrate the application of computing to some area of personal interest. I particularly encouraged students to consult faculty members in their major fields, so that the projects would be as useful as possible in their academic programs. Several chose applications that related to their employment in college activities, such as the Student Union, the Admissions Office, the Development Office, and so on. A few concentrated on games. There were many projects dealing with statistics applied to psychology, government, and biology. Mathematics students developed functions for operations research and linear programming, and some created systems to differentiate and analyze given functions (see J.W. Bergquist, 1974). An interesting study of combinatorics made good use of recursive functions.

Many social scientists are highly dependent on a large black box called Statistical Package for the Social Sciences (SPSS), and several students successfully wrote APL functions that produce results identical to the output from SPSS.

The diversity was very great. Two students, impressed with the formal description of System/360, did a good job of simulating a 6800 microprocessor in APL. Another did an analysis of the practical question as to whether she should buy a new car, and if so, what kind. One biologist, who was using radio-transmitters on coyotes in the mountains, had the 5100 drive a Tektronix 4662 plotter to show on a contoured map the tracks of the animals by day and by night, and determine the statistics of their movements. Another analyzed tide tables to determine the length of time that the animals in the tide pools were below sea level. A French major used five-dimensional arrays to manipulate French verbs. And a geology student used eigenvalues to determine the fold axis in deformed rocks. Models in Economics were popular, and methods of text analysis were applied to English prose and to genetics.

After seeing a demonstration of IBM's "Query-by-Example", I simulated QBE in APL, largely in Direct Definition. I made the workspace available to the students, and in class I discussed how I had approached the problem. A large number of students took this as a model and created their own systems implementing simple but practical examples of relational data bases. An Art major prepared a system to catalog art slides, an anthropologist analyzed the distribution of cave art, a Music major simulated the R.I.L.M. bibliographic music data base, and an Asian Studies major created a complete interactive catalog of all known translations of Kabuki plays, which the faculty members in that field judge to be an important contribution.

Even those who did not complete a noteworthy project found the attempt to apply their computing knowledge to be a valuable experience.

Faculty Seminar in Computing

In June 1980, after the end of the second semester of the academic year, I taught a seminar for 12 of my colleagues, whose fields were Chemistry, Economics, Government, Mathematics, Philosophy, Sociology, and Zoology. The number of participants was limited to the number of terminals we could arrange to have in one room, for the reason that I wished each person to be at a terminal the entire time of the seminar. We worked intensively from 8:30 a.m. to 5:00 p.m. every day for two weeks. This was a very interesting experience. It involved total immersion in APL, and in my opinion the length of the seminar was just right. After 3-5 days most people find that they are becoming lost. They have learned much, but the parts are not yet seen as making a whole. As the second week progresses, it is obvious that comprehension (in the literal sense) is taken place rapidly; the mechanics of the operations cease to dominate, and attention can be given to topics of intellectual interest.

Although the group learned how to use the power of CMS, even learning how to manipulate real tapes, there was a clear emphasis on APL. Probably all who participated did so because they had heard of APL and wished to learn about it for themselves. Because they constantly learned by doing, they covered the more mechanical and machine-oriented material quickly, and this gave maximum time for instruction on the proper use of APL.

Many of the participants were from the Social Sciences, and I therefore covered two topics that I had not been able to include in the semester course; namely, the use of Shared Variables to access data in CMS files not created by APL, for example by EDGAR; and the use of IBM's 124X auxiliary processor for creating full-screens under the control of APL functions.

The faculty members who were in the seminar are busy people, and it seemed desirable to cover as much material as possible in the limited time, knowing that they could get the experience of practice later, on their own. I believed it was important that they ended the two weeks with the feeling that they had accomplished a lot, and that in dealing with students and colleagues in the future, they would be confident in knowing that although they might not remember exactly how to perform some task, they would remember that they had actually done it. In this way the fear and mystery associated with the jargon and the symbols was effectively dispelled.

Principles Used in APL Instruction

I always begin by showing that APL is immediately useful, for example, by using +/1 2 3 4 5 6 to add up a string of numbers. After two or three class meetings, when the students had had time to see how easily they could achieve results with APL, I was asked "when will we start writing programs?", or, in other words, "when will this start to hurt?" I decided then that it was worth taking time to let the students learn enough FORTRAN to write simple programs and subroutines. This ensures an understanding of the fact that there are different kinds of programming languages. Before I had finished with this exercise, several students asked me why we were wasting time on FORTRAN when it was so easy to achieve the same results in APL. I explained that there are those who claim that APL is inefficient in comparison to a compiled language, such as FORTRAN; and I directed them to colleagues who hold this view. The result was remarkable. The students asked whether the object was to save their effort or the effort of the machine! In view of the obvious decrease in cost in hardware, and the simultaneous increase in the cost of getting qualified people, the argument about supposed efficiency was not found to be very convincing. This was even more the case after we had discussed the "efficiency" of the FORTRAN program that we had written to do sorting. An important lesson learned was that inefficient programs can be written in any language, and that the efficiency of the interpreter or compiler might be taken into account also.

As the course continues I try constantly to make what we are doing appear obviously practical. I do this by taking examples that I know to be relevant to fields represented by members of the class. But I also emphasize the evolution of mathematical thought and notation through time, demonstrating that APL today is a continuation of the tradition represented by such men as Napier, Oughtred, Leibniz, Euler, Boole, Jevons, De Morgan, Cayley, Sylvester, Peano, and many more of the intellectual giants of past centuries. For example, we see how Iverson extended De Morgan's Law and why these are useful; how compression implements Boole's selection function; how the inner product of APL enormously extends the power of the matrix multiplication of Cayley; and how APL helps the non-mathematician to understand and use concepts such as zero, emptiness, and recursion.

Index Origin

The discovery of zero was one of the triumphs of human intellect. But, even today, many people, even mathematicians, seem not to have fully accepted zero into their thinking. Although my students were, of course, free to work in either origin, every expression that I shared with the class was written in Origin Zero. *X*∘.⋆⍳*N* is the obvious way to generate the powers of *X* for polynomials and regression, and the first (leading) power is 0. The word "first" is, perhaps, the greatest stumbling block in the general acceptance of origin zero, but nearly all students chose to work in origin zero when they realized that graphs are normally drawn so that the axes are labelled starting at 0, and when they saw such commonly occurring expressions as:

N+⍳*M*	to generate *M* integers starting with *N*
A[*B*]	where *B* is logical (Boolean); i.e., 0's or 1's. This is an expression useful for giving pictorial output, as in

```
' *'[⊖(⍳⌈/V)∘.≤V←3 2 1 0 2 4 6]
```

```
(⍴M)⊤(V=⌈/V)/⍳⍴V←,M
```

which gives the indexes of all occurrences of the largest value in the matrix *M*

Right-to-Left Execution

Not a single student as much as remarked about the right-to-left order of execution. At the outset I simply said that, as in the expression log sin square-root X, every function in APL takes as its right argument everything to its right, unless parentheses change the order. I pointed out that this was the normal order in mathematics beyond the most elementary level, for example in a sequence of rotation matrices, and that it leads to expressions that use the minimum of parentheses. I stressed that it is never necessary to end an expression with a right parenthesis, or to use two adjacent right parentheses within any expression.

The non-commutative functions minus and divide, which may require parentheses if the left argument is composite, can often be converted into the related commutative functions plus and multiply, with the elimination of parentheses:

```
(A F B)-1     ↔    ¯1 + A F B
(A F B)÷10    ↔    .1 × A F B
```

Moreover, this brings out the analogy between the negative sign and the decimal point as part of the name of a number.

When these simplifications are ignored, there is ground for questioning whether the concepts of negative and fractional numbers have been understood.

Although expressions can be EXECUTED only from right-to-left, it is good to adopt the practice of READING expressions from left-to-right and well as from right-to-left. For this reason (as originally pointed out to me by Larry Breed) no variable should be specified more than once in one expression.

I asked my students to be conscious of the style of their APL, just as they should be of their English composition. As with English, it is important to study good examples. The publications of the APL Press were pointed out as containing admirable models.

Direct Definition

The word "function" was first used in the modern sense by Leibniz and Bernoulli before 1700. They wrote in Latin and simply used the word meaning "perform". A function performs the task of combining its arguments together in some specified (formal) way. If the arguments are thought of as nouns, then the function is an imperative verb.

In 1753, Euler used the notation ϕ:(x,t) for a function ϕ whose arguments are x and t, and in 1754 he wrote the analogous notation f:(a,n) for the function f of a and n.

K.E. Iverson, the principal inventor of APL, introduced a variant of Euler's notation in 1976 in his book "Elementary Analysis". Like Euler, Iverson began by writing the name of the function separated from what followed by a colon. Whereas Euler simply named the arguments (separated by commas and enclosed in parentheses), Iverson wrote the APL expression that constitutes the formal definition of the function. "A formal definition is one which can be interpreted by a mechanical application of known rules, requiring no judgement or subtle interpretation" (Iverson, 1976, p. 10).

Following the conventions of elementary mathematics (illustrated in the examples -*X*, 2+3, 2-3, 2×3, and 2÷3), Iverson permitted definition of functions with either one or two arguments, one on the right and, if necessary, one on the left of the function name. To distinguish between these within the formal definition, he used the symbol α to stand for the left argument (if any), and ω to stand for the right argument. These are "dummy" arguments or "place-holders", for which actual values are substituted when execution takes place.

Thus, we can define the function *YF* as follows:

```
      YF:(ω∘.*⍳⍴α)+.×α
```

where α is a vector of length 2 whose elements are the y-intercept and the slope of a straight line, and where ω is a vector (or scalar) of values of x.

Then *YF* is the function that gives the values of y corresponding to the values of x denoted by ω; for example,

```
      .5 2 YF 1 2 3 4 5
```

gives the values of y corresponding to the values 1 2 3 4 5 for x when the y-intercept is .5 and the slope is 2. The colon may be read as "is", thus, *F*:α*ω may be read as "*F IS* α raised to the power ω".

Iverson extended this notation to permit definition of recursive functions; i.e., functions that are defined in terms of themselves (Iverson, 1976, Chapter 10). The recursive definition of a factorial requires the following three pieces of information:

A primary expression:	ω×*FAC* ω-1
A proposition:	ω=0
A secondary expression:	1

In a formal definition, these three items are presented in order, with colons separating them; thus

```
      FAC:ω×FACω-1:ω=0:1

      FAC 4
24
```

You can read the formal definitions as: "The *FAC*torical *IS* ω times the factorial of ω-1, UNLESS ω=0, IN WHICH CASE it is 1".

A proposition is an expression that yields either 0 (meaning false) or 1 (meaning true). When there are three colons, the proposition is executed first; the primary expression is executed if the proposition yields 0, and the secondary expression is executed if the proposition yields 1.

Iverson's αω method of function definition is called "Direct Definition". In order to implement it on a computer, it is necessary to provide a compiler that can take the character string giving the formal definition and convert it into executable code. I.P. Sharp's system can detect function definition from its unique syntax (the name being followed by colon) and does the conversion in a manner transparent to the user.

Direct Definition enforces a discipline that makes it far easier to write good APL. It is also a natural way to teach, because functions are introduced (on the blackboard or on the terminal) as the subject requires them and without digressions. Any expression already developed is simply given a name and used. Documentation is straightforward, because every distinct concept is a one-line function with no side effects, i.e., without pretending to do one thing but doing another. If the functions have been well chosen and well named, it is easy to state what each does. Interactive documentation with appropriate data can illuminate the individual functions and illustrate typical ways in which the functions interact as a system.

As Paul Berry has said (Berry, et al., 1971), "When APL functions are executed at an APL system, the user should have to state mathematical kernel of his function, and nothing more". Direct Definition permits just that. Nothing superfluous is required. I believe that the use of Direct Definition is also consistent with the exhortation of John Backus, in his Turing Award Lecture, that the user should function as much as possible in the world of expressions rather than in the world of control statements.

Throughout both my semester course and my two-week seminar, I used only Direct Definition of functions. The only exceptions were when I needed "house-keeping" functions, for example to handle shared variables or to provide annotated output. Despite my example and admonition, several students "discovered" that the del (∇) form of function definition permitted them to write multi-line programs filled with loops and branches. Although I had never once spoken about branching, students who already knew something about languages such as BASIC seemed to have a serious disadvantage in being almost incapable of escaping from the intellectual tyranny of "word-at-a-time processing" and the consequent superstructure of control statements. In an attempt to combat this unfortunate tendency, I played the tape of John Backus' Turing Award Lecture, in which he says that this style of programming has held up progress for 20 years. My experience persuades me that the widespread use of BASIC on home-computers is a dangerous threat to the intellectual development of many bright people.

Familiarity with All Primary Functions

I try to let my students share the intellectual pleasure of seeing the surprising and diverse uses of such functions as ⊥ and ⊤. I want them to feel at home with expressions like 0⊥*V* (for the scalar representation of the last element of the numeric vector *V*), or *M*⊥1 (for 1 plus the number of 1s before the first 0, reading the rows of the logical matrix *M* from right to left). On the first encounter with such expressions, they may appear awkward and unnecessarily subtle, but this is because of unfamiliarity with the primary functions. Like any other new concept, once the strangeness of novelty wears away, these functions can become very natural as well as useful intellectual tools.

As an example of how the decode, or base-value, function (⊥) can be introduced and made familiar, consider the following:

It is easy to understand how the vector 1 9 8 4 is reduced to a scalar by the function ⊥ using the base 10

```
      10⊥1 9 8 4
1984
```

Now create a function that provides a formal definition of what we have just performed, and we see that although we commonly write a sequence of numbers from left to right, we write the digits of a single number so that they increase in value from right to left; thus, the vector resulting from the function ⍳ must be rotated before it can be used to generate the needed weights.

```
      BASE:⍵+.×⌽⍺*⍳⍴⍵
      V←1 9 8 4
      10 BASE V
1984
```

The function BASE will always give a scalar result, and its left argument can be 0:

```
      0 BASE V
4
      0⊥1 9 8 4
4
```

That the result is a scalar can best be demonstrated by:

```
      ⍴⍴0⊥V
0
```

In contrast, the result of take (↑) is always a vector:

```
      ⍴⍴¯1↑V
1
```

Now consider the treatment of a polynomial such as:

$$a + bx + cx^2 + dx^3$$

In the first place we can rewrite this as:

$$a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3$$

so that the pattern is more evident by having a uniform and consistent treatment of all terms. We have made it obvious that there are only two arguments: a vector of coefficients and a scalar, X. The first term is brought into conformity with the pattern of the others, once we assimilate zero into our thinking. Moreover, we find that many disciplines, such as Economics, already use the subscript 0 for the first term. Thus origin zero is in fact in common use, though not always recognized. I encourage my students to be consistent in their use of origin zero.

If we take the following values for the coefficients and the scalar:

```
      C←1.5 2 ¯3 0.5
      X←2.5
```

the polynomial can be written in APL thus:

```
      C[0] + (C[1]×X) + (C[2]×X*2) + C[3]×X*3
¯4.4375
```

This is a direct translation of the algebra into APL. We can improve it by rewriting so that the parentheses are not needed:

```
      C[0]+X×C[1]+X×C[2]+X×C[3]
¯4.4375
```

But notice that although the coefficients exist as an array, we refer to them and use them, in both expressions, as individual scalars. In order to take advantage of the fact that *C* is an array, and that the vector of powers to which *X* is raised is itself a function of *C*, we write:

```
      +/C×X*ιρC
¯4.4375
```

Now, whenever we see the expression +/*A*×*B*, we should consider whether this can be rewritten as *A*+.×*B*. In this case, as so often happens, we have a sum of products, and the necessary compatibility enables us to write:

```
      C+.×X*ιρC
¯4.4375
```

We can therefore use the function *BASE* to achieve the same result:

```
      X BASE ⌽C
¯4.4375
```

Or, what is the same thing:

```
      X⊥⌽C
¯4.4375
```

It should be noted that because *C* is a vector (ρρ*C* ↔ 1), its leading axis is also its last, and hence ⌽*C* ↔ ⊖*C*.

We have looked at the case where the *X* of the polynomial is a scalar, but if *X* is a vector, then the polynomial must be evaluated for each value of *X*:

```
      POLY:(α∘.*ιρω)+.×ω

      X←2.5 3.5

      X POLY C
¯4.4375 ¯6.8125
```

If we use the base function to achieve the same result, *X* must be a matrix. If it were a vector, its elements would be used together (like 24 60 for hours and minutes) instead of separately.

```
      (X∘.+(ρC)ρ0)⊥⌽C
¯4.4375 ¯6.8125
```

Or, as we will see is still better, (*X*∘.+(ρ*C*)ρ0)⊥⊖*C*

Now suppose that several polynomials are to be evaluated, each with its own set of coefficients:

```
      C←C,[⍟.5]2 1.3 ¯1.5 .25
      C
1.5   2
2     1.3
¯3    ¯1.5
0.5   0.25
```

In conformity with the rules for matrix multiplication, where the rows of the left argument are combined with the columns of the right argument, the coefficients of the individual polynomials must be columns. Now that the *C* is a matrix, ⌽*C* would simply change the order in which the polynomials are given — rotating about the last axis. As long as *C* was a vector it had only one axis, but now there are two, and to change the order of the terms within each polynomial, we must rotate about the leading axis, ⊖*C*.

```
      (X∘.+(1↑⍴C)⍴0)⊥⊖C
¯4.4375   ¯0.21875
¯6.8125   ¯1.1063
```

Each polynomial is evaluated for each term in *X*.

To see how the *BASE* function must be modified to produce the same result, we can proceed as follows:

```
      BASE1:((1↓⌽×\⌽α),1)+.×ω
      ⍝   α⊥ω where α is vector
```

Or rearranging to eliminate unneeded parentheses:

```
      BASE2:(1↓⌽1,×\⌽α)+.×ω

      24 60 60 BASE2 1 2 3
3723
```

The number of seconds in 1 hour, 2 minutes, and 3 seconds

If α is a matrix and ω is a scalar, vector, or matrix:

```
      BASE3:(0 1↓⌽1,×\⌽α)+.×ω
```

If *A* and *B* are specified as matrices, as follows:

```
      A
  24  60  60          Hours, minutes, seconds
1760   3  12          Yards, feet, inches

      B
1 4
2 5
3 6

      A BASE3 B
3723 14706
  63   210

      A⊥B
3723 14706
  63   210
```

The function *BASE*3 will work if α is a matrix, but not if α is a vector. To allow for this, we define:

```
      BASE4:((⌽(⍴⍴α)↑1)↓⌽1,×\⌽α)+.×ω
```

Or the clumsier, but for some students more easily understood, form:

```
      BASE5:((((¯1+⍴⍴α)⍴0),1)↓⌽1,×\⌽α)+.×ω

      24 60 60 BASE4 B
3723 14706
```

But now this will not work if α is a scalar. A simple way out of the difficulty is to decide which algorithm is appropriate by using the so-called "recursive" form, even although the function is not itself recursive:

```
      BASE6:αBASE4ω:0=⍴⍴α:(α*⌽⍳1↑⍴ω)+.×ω

      A BASE6 B
 3723 14706
   63   210

      10 BASE6 B
123 456
```

If both α and ω are scalars, then the result is ω (weighting factor of 1):

```
      BASE7:αBASE6ω:0∧.=(⍴⍴α),⍴⍴ω:1×ω
```

After exploring the base function ⊥ in this way, the student should have no difficulty in understanding why 0⊥1 2 3 4 is the scalar 4, or why 1 0 1 1⊥1 ↔ 3. It should also be clear why the following function will right adjust a character matrix:

```
      RAJ:(1-(ω=' ')⊥1)⌽ω
```

and why the following will give the vector of indexes that will alphabetize a character matrix provided that ∘⊥⍴ω is not too large:

```
      ASRT:⍋(⍴⎕AV)⊥⍉⎕AV⍳ω
```

Roy Sykes (1978) has given the following good example of ⊥.

```
      SPS:(,α)[(⍴α)⊥(1⌽⍳⍴⍴ω)⍉ω]
      ⍝ Scattered-Point Selection from array α by indices ω
```

To illustrate the utility of ⊤ we might introduce a function that will find the indexes of all occurrences of ω in a matrix α:

```
      IXM:(⍴α)⊤(V=ω)/⍳⍴V←,α
```

Those who began by thinking that because base 10 had been good enough for their fathers it should be good enough for them, are often excited to discover that there are many ways in which ⊥ and ⊤ can accomplish very practical tasks. They are then ready to read the sections on Decode and Encode in Falkoff and Orth (1979, p. 435-436).

It was interesting that, after conducting this exploration in the faculty seminar, a zoologist realized that in his work with tide tables he converts matrices of times in days, hours, minutes into vectors of minutes, to learn how long the tidepools are submerged over a given period. He saw that he could get his result using the function ⊥ with the left argument 0 24 60, and with the right argument a matrix whose rows are days, hours, minutes, and whose columns are points in time.

We have taken the decode function as an example, because it is often not well understood by beginners, but every primary function deserves similar study.

Using Arrays in APL

"It is the constant aim of the mathematician to reduce all his expressions to their lowest terms, to retrench every superfluous word and phrase, and to condense the Maximum of meaning into the Minimum of language." (1877)

"A matrix of quadrate form .. emerges ... in a glorified shape — as an organism composed of discrete parts, but having an essential and undivisible unity as a whole of its own. The conception of multiple quantity thus rises upon the field of vision. [Matrix] ... dropped its provisional mantle, its aspect as a mere schema, and stood revealed as bona-fide multiple quantity subject to all the affections and lending itself to all the operations of ordinary numerical quantity. ... The Apotheosis of Algebraical Quantity." (1884)

These quotations, taken from the writings of James Joseph Sylvester (1814-1897), the self-styled Mathematical Adam, who introduced the term Matrix in 1850, summarize for me much of the spirit of APL, and testify to the importance of APL in the development of mathematics.

When APL is well written, it takes full advantage of the algebra of multiple quantity, and would surely have greatly pleased Sylvester. When FORTRAN employs a loop, it is likely that this can be avoided in APL by increasing the rank of a variable from a scalar to a vector. If FORTRAN has nested loops, APL probably increases the rank still further. Thus the concept of rank must be thoroughly grasped from the beginning, and the student should be trained to think about the ⍴⍴ of his variables.

I have found it important to demonstrate how one approaches a problem by considering the shapes and compatibilities of the variables. For example, to create a histogram you must have two variables: the data to be classified and the bounds that define the classes. Because there is no necessary compatibility between these, an outer product is required:

```
HIST:PWD +/α∘.>ω          Where α gives the bounds and ω the data
PWD:(1↓ω)-¯1↓ω            Pairwise differences
```

Any function written for scalars only should be considered as a candidate for extension so that it can work with vectors, and possibly with matrices and higher arrays. The person familiar with the subject matter should enquire what meaning is to be attached to such proposed extensions. Scalars are simply scalars, and compatibility between them is always guaranteed, but arrays can differ in both rank and shape.

The following example is a function written by an economist as part of a system to compute Double Declining Balance. The formula is cumbersome, and the APL reflects this; but the important point is that the function as written works only with scalars.

```
    ∇ Z←R DDB1 T
[1]   Z←((2÷T)÷R+2÷T)×1-((1-2÷T)÷1+R)*⌈T÷2
    ∇
```

The following functions, in Direct Definition, take into account the necessary compatibilities of the arguments, and outer products are generated between the vectors of different lengths.

```
      DDB2:(((ρM)ρ2÷ω)÷M←α∘.+2÷ω)×⍉αPVAω

      PVA:1-((1-2÷ω)∘.÷1+α)*⌈.5×ω∘.+(ρα)ρ0        Present Value (part A)
```

Where compatibilities necessarily exist between the variables, then inner products are probably called for. Students should know that

```
      if Z←A∘.×B          then    ρZ ↔ (ρA),ρB

but   if Z←A+.×B          then    the condition for compatibility of arrays is that
                                  ¯1↑ρA ↔ 1↑ρB

                          and     ρZ ↔ (¯1↓ρA),1↓ρB
```

An interesting example of extending a function that worked only on scalars arose in an ecological study. Brillouin's measure of diversity of species requires the logarithms of the factorials of large numbers. The direct APL expression for small numbers is:

```
      ⍟!5
4.7875
```

Because this fails for large numbers, the following recursive function was defined:

```
      LFA:(⍟ω)+LFAω-1:ω=0:0
      ⍝ Log of factorial for large scalar ω

      LFA 5
4.7875
```

But Brillouin's Index uses the sum of the logarithms of the factorials, and *LFA* only works with scalars, hence the functions:

```
      SLF:+/⍟(V≠0)/V←,M×+\M←ω∘.>⍳+/ω
      ⍝ Sum logs of factorials (for BDI)

      BDI:((⍟10)÷+/ω)×(SLF+/ω)-SLFω
      ⍝ Brillouin's Diversity Index

      SLF 5
4.7875

      SLF 4 2 5
8.6587

      +/⍟!4 2 5
8.6587
```

Taking the data from Robert W. Poole (1974, p. 390):

```
      V
235 218 192 87 20 11 11 8 7 4 3 2 2 1 1

      H←BDI V
      H
3.8063          Brillouin's measure is designated H

      H÷⍟10
1.653           natural bels per individual
```

the number of individuals is:

```
      +/V
802

      B←H×+/V
      B
1325.7          The total information content of the collection, in natural bels
```

The following example, proposed by an economics student, illustrates the use of arrays in defining functions for practical applications.

The economist Paul Douglas (later U.S. senator) wanted to relate the Quantity of output to the Capital (number of machines) and the Labor (number of workers). Together with a mathematical colleague, Cobb, he defined an expression that is known as the Cobb-Douglas Production function (Douglas, 1976):

$$Q = A.K^{\alpha}L^{\beta}$$

where *K* is the Capital (e.g., number of tractors),
L is the Labor (number of farm workers),
and *Q* is the Quantity produced (e.g., amount of wheat)

A is a "shift parameter", which enables us to change the value of *Q* without changing *K* or *L*.

α and β are numbers (usually between .25 and .75) that describe the way in which *Q* changes as *K* and *L* are changed. There is a special interest in the case where $\alpha + \beta = 1$; i.e., where $\beta = 1-\alpha$

If ω is the vector α, *A*, *K*, *L*, then *Q* is given by the function

```
CDP:ω[1]×(ω[2]*ω[0])×ω[3]*1-ω[0]
```

For example,

```
      CDP .3 10 2 1
12.311
```

However, when the economist writes the function as *Q*=f(*K*,*L*), which can be read as "*Q* is a function of *K* and *L*", he shows that he thinks of *K* and *L* as being fundamentally different from α and A. We might therefore separate the "parameters" α and A as the left argument, and let K and L together form the right argument of the function.

```
CDP0:α[1]×(ω[0]*α[0])×ω[1]*1-α[0]

      .3 10 CDP0 2 1
12.311

      .5 15 CDP0 3 4
51.962
```

But this function permits only scalar values for K and L, and we would like to see the effect on Q of varying K and L. The function ought to accept vectors of K and L. Now for every value of K there must be an associated value of L; so the right argument must be a matrix. Let us assume that the columns are K and L. Then, instead of raising K and L separately to the appropriate powers, we use the dyadic * with matrix arguments:

```
CDP1:α[1]××/ω*(ρω)ρα[0],1-α[0]

      .3 10 CDP1 2 2 3 3,[.5]1 2 3 4
12.311 20 30 36.693

      .3 10 CDP1 2,[.5]1 2 3 4
12.311 20 26.564 32.49
```

We might modify this function so that it could accept a vector of values for A. One way to do this is to make the economists' parameter α a global variable, A, thus reserving the left argument for the "shift parameter" or "constant".

Whereas the left argument of the ~~X~~ was previously a scalar, and so was automatically extended to be compatible with the vector right argument of X, the left argument is now a vector and incompatible in length with the right argument; consequently an outer product is needed:

```
CDP2:α∘.××/ω*(ρω)ρA

      A←.3 .7
      10 15 20 CDP2 2,[.5]1 2 3 4 5
 12.311 20      26.564 32.49  37.983
 18.467 30      39.846 48.735 56.974
 24.623 40      53.128 64.98  75.966
```

When we saw the expression (A*B)×(C*D) in the first form of the function, we knew we could make A and C into the *COLUMNS* of a matrix M, and C and D into the *COLUMNS* of a compatible matrix N. We then wrote ×/M*N. But this form, f/ M g N (where f and g are scalar dyadic functions, taking scalar arguments and extending to element-by-element functions on arrays), can always be rewritten as an inner product, M f.g N; thus we have:

```
CDP3:α∘.×ω×.*A̲

      A̲←.3 .7
      10 15 20 CDP3 2,[.5]1 2 3 4 5
 12.311 20      26.564 32.49  37.983
 18.467 30      39.846 48.735 56.974
 24.623 40      53.128 64.98  75.966
```

This has the advantage of automatically extending to a matrix of A̲, whose *COLUMNS* are successive pairs of exponents. It is worth noting that the function is unchanged if a THIRD variable is added to *K* and *L*.

```
      A̲←X,[¯.5]1-X←.25 .3 .5
      A̲
0.25 0.3  0.5
0.75 0.7  0.5

      R←10 15 CDP3 2,[.5]1 2 3 4 5
      ρR
2 5 3
```

The result is rank 3; i.e. it has three axes for indexing. The axes are:

0	Length 2	2 ↔ ρ10 15	the economists' parameter *A*
1	Length 5	5 ↔ ρ*K* ↔ ρ*L*	the economists' variables
2	Length 3	3 ↔ 0⍳ρA̲	the number of exponent pairs

It is important to be able to identify any element in *R*, *R*[*I*;*J*;*K*], in relation to the original data. The function *CDP* (which takes only scalar arguments) should be used to check your navigation through the 3-dimensional array *R*.

```
      R
11.892 12.311 14.142
20     20     20
27.108 26.564 24.495
33.636 32.49  28.284
39.764 37.983 31.623

17.838 18.467 21.213
30     30     30
40.662 39.846 36.742
50.454 48.735 42.426
59.645 56.974 47.434
```

The sequence of the axes is determined by the sequence of the inner and outer products in the function *CDP*3, but it would probably be more convenient to rearrange the axes so that axis 0 corresponded to the exponent pairs, axis 1 to the shift parameter *A*, and axis 2 to the successive pairs of values of *K* and *L*. This is done by the dyadic transpose. To understand this, consider that the transpose of a matrix is the result of interchanging the rows and columns; i.e. the result of taking axis 1 as the leading axis and axis 0 as the last axis.

The monadic transpose of a 2-dimensional array (i.e. of a matrix) is merely a special case of the general (dyadic) transpose of ANY array. However, because a vector has only one axis, *V* ↔ ⍉*V*, and a scalar has NO axes to be permuted.

```
      M ↔ 0 1 ⍉ M
      ⍉M ↔ 1 0 ⍉ M
```

Consequently, we can rearrange the axes of R in this way:

```
      Q←1 2 0⍉R
      ρQ
3 2 5

      Q
 11.892 20       27.108 33.636 39.764
 17.838 30       40.662 50.454 59.645

 12.311 20       26.564 32.49  37.983
 18.467 30       39.846 48.735 56.974

 14.142 20       24.495 28.284 31.623
 21.213 30       36.742 42.426 47.434
```

In order to explore the properties of the Cobb-Douglas Production function, it might be useful to provide more structure in the variable. This can be done by raising its rank from 2 to 3; for example:

```
      I←4 5 2ρ⍉1+0 5⊤ι20

      A←10 15 20

      I
 1 1
 1 2
 1 3
 1 4
 1 5

 2 1
 2 2
 2 3
 2 4
 2 5

 3 1
 3 2
 3 3
 3 4
 3 5

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

Then the product of the powers is:

```
      R←I×.*X,[¯.5]1-X←.3 .5
      ρR
4 5 2
```

Notice that if

```
      R←A+.×B
```
then
```
      ρR ↔ (¯1↓ρA),1↓ρB
```

The last axis of the first argument must equal the first axis of the second argument, and it is these two equal axes that are eliminated in the inner product.

The rank is raised to 4 when we use the outer product to multiply by the vector of shift parameters. The new leading axis has length 3, equal to ρA:

```
      Q←A∘.×R
      ρQ
3 4 5.2
```

Once again rearranging the axes, we have:

```
      Q←1 2 3 0⍉Q
      ρQ
2 3 4 5
```

We therefore have 2 sets of results, corresponding to the 2 sets of exponents; each set being divided into 3 groups, corresponding to the 3 values of the shift parameter; and each group being a 4 by 5 matrix, corresponding to that part of the structure of the variables in array *I* that is not lost in the inner product.

```
      Q
10      16.245 21.577 26.39  30.852
12.311  20     26.564 32.49  37.983
13.904  22.587 30     36.693 42.896
15.157  24.623 32.704 40     46.762

15      24.368 32.365 39.585 46.278
18.467  30     39.846 48.735 56.974
20.856  33.88  45     55.039 64.344
22.736  36.934 49.056 60     70.144


20      32.49  43.153 52.78  61.703
24.623  40     53.128 64.98  75.966
27.808  45.174 60     73.385 85.792
30.314  49.246 65.408 80     93.525


10      14.142 17.321 20     22.361
14.142  20     24.495 28.284 31.623
17.321  24.495 30     34.641 38.73
20      28.284 34.641 40     44.721

15      21.213 25.981 30     33.541
21.213  30     36.742 42.426 47.434
25.981  36.742 45     51.962 58.095
30      42.426 51.962 60     67.082
```

```
20      28:284 34.641 40     44.721
28.284  40     48.99  56.569 63.246
34.641  48.99  60     69.282 77.46
40      56.569 69.282 80     89.443
```

Gathering up the individual pieces, we have the following result:

```
    1 ↔ ∧/, Q = 1 2 3 0 ⍉ A∘.×I×.*A̲,[¯.5]1-A̲
```

Values of *Q* can be contoured on a plot of Capital, *K*, against Labour, *L*. Different contoured maps correspond to different values of the shift parameter, *A*, and α.

A perhaps simpler example of the dyadic transpose is the case of an array *A*, such that ρ*A* ↔ 5 10 12. The three axes correspond to an account extending over 5 years, for 10 line items, and 12 months. If *A* is displayed, 5 tables will appear, each with 10 rows and 12 columns.

Now suppose that we wish to have a separate table for each line item, then *B*←2 0 1 ⍉ *A* and ρ*B* ↔ 10 12 5; i.e. there are 10 tables (one for each line item), each with 12 rows (the months) and 5 columns (years).

Or we might have chosen to create the array *C*←1 0 2 ⍉ *A*, so that ρ*C* ↔ 10 5 12; ρ*C* ↔ 10 5 12 i.e. there are 10 tables, each with 5 rows (years) and 12 columns (months).

An Example of the Use of APL in Logic

The most amusing and readable book on Logic was written by Lewis Carroll, the author of "Alice in Wonderland" and "Through the Looking Glass". Although its title is "Symbolic Logic", it has rather little to do with the symbolism found in modern texts on that subject, but Carroll does use formal methods of solving syllogisms and sorites. The original edition is rare, but a reprint by Dover Publications Inc. is readily available.

Carroll defines a "sorites" as a set (literally a heap) of three or more propositions related in such a way that two of them together yield a conclusion, which, taken with another of them, yields another conclusion; and so on until all have been taken. If the original set is true then the last conclusion must also be true.

One of Carrolls's examples (#47 on p. 120) is as follows:

(1) Every idea of mine, that cannot be expressed as a Syllogism, is really ridiculous;
(2) None of my ideas about Bath-buns are worth writing down;
(3) No idea of mine, that fails to come true, can be expressed as a Syllogism;
(4) I never have any really ridiculous idea, that I do not at once refer to my solicitor;
(5) My dreams are all about Bath-buns;
(6) I never refer to any idea of mine to my solicitor, unless it is worth writing down.

The "Universe of Discourse" is "My ideas", and the "Dictionary of Terms" is:

1. able to be expressed as a Syllogism
2. about Bath-buns
3. coming true
4. dreams
5. really ridiculous

6. referred to my solicitor
7. worth writing down

Using negative numbers to represent "Not", we can write the 6 bilateral propositions in this way:

```
 ¯1   5
  2  ¯7
 ¯3  ¯1
  5   6
  4   2
  6   7
```

Thus the pair ¯1 5 can be read:

All my ideas that are	NOT	able to be expressed as a Syllogism
are all		really ridiculous

and the pair 2 ¯7 can be read:

All my ideas that are		about Bath-buns
are all	NOT	worth writing down

This is an example where origin 1 is preferred to origin 0, because we wish to distinguish between two related cases (true and false) by the use of the sign of the number, and we therefore cannot use 0 for one of our cases.

Now if the proposition ¯1 5 is true, then the proposition ¯5 1 must also be true.

All my ideas that are	NOT	really ridiculous
are all		able to be expressed as a Syllogism

Hence every proposition can be inverted with a change of signs.

Inspection of the numeric table representing the six propositions shows that, with two exceptions, each number (irrespective of its sign) occurs exactly twice. The two exceptions are the terms that occur in the final conclusion.

To implement the analysis, suppose that the propositions given above are assigned to the variable *P*. Then the complete set of propositions is:

```
      M←P,[0]-⌽P
      M
¯1  5
 2 ¯7
¯3 ¯1
 5  6
 4  2
 6  7
¯5  1
 7 ¯2
 1  3
¯6 ¯5
¯2 ¯4
¯7 ¯6
```

The beginning and end of the sequence are found by the function:

```
    BES:(2=+/N∘.=V)/N←NUB V←|,ω
    ⍝  Beginning and end of sorites

    NUB:((ω⍳ω)=⍳ρω)/ω

         BES M
3 4
```

The start of the sequence is given by:

```
    STS:ω[;0]⍳1↑BESω
    ⍝  Start for sorites

         STS |M
2
         M[2;0]
¯3                    Begin with NOT coming true'
```

The end of the sequence is given by:

```
    ENS:ω[;1]⍳1↓BESω
    ⍝  End of sorites

         ENS |M
10
         M[10;1]
¯4                    End with NOT dreams
```

Hence ¯3 ¯4 or, alternatively, 4 3; i.e. All my dreams come true.

In order to move from the starting proposition to the end, we can proceed as follows:

```
         U←M[;0]=¯3
         U/M[;1]
¯1
         U←M[;0]=¯1
         U/M[;1]
5
         U←M[;0]=5
         U/M[;1]
6
         U←M[;0]=6
         U/M[;1]
7
         U←M[;0]=7
         U/M[;1]
¯2
         U←M[;0]=¯2
         U/M[;1]
¯4                    That this is the end, is shown by:
      M[ENS|M;1]
¯4
      U←M[;0]=¯4
```
If we proceed, the result is empty:
```
      U/M[;1]
```

Consequently, we can get the sequence by recursion, stopping either when we reach the known end proposition or when the result is empty:

```
  RSR:(α,(ω[;0]=0⊥α)/ω[;1]RSRω:A=0⊥α:α
  ⍝ Recursive sorites

  SRS:ω[STS|ω;0]RSRω:0,0ρA←ω[ENS|ω;1]:0
  ⍝ Sorites

      SRS M
¯3 ¯1 5 6 7 ¯2 ¯4
```

The sequence needs to be obtained as a function of P, and it must be reversed if it starts with a negative:

```
  IXS:Z:0>1↑Z←SRSω,[0]-ϕω:-ϕ7
  ⍝    Indexes of sequence of sorites

      IXS P
4 2 ¯7 ¯6 ¯5 [illegible] 3
```

Finally, to translate the result into English we need the following text data, with 'Not' appended to the end of each line.

```
      DATA
able to be expressed as a SyllogismNot
about Bath-buns                    Not
coming true                        Not
dreams                             Not
really ridiculous                  Not
referred to my solicitor           Not
worth writing down                 Not
```

The function Carroll will then reorder the terms, and rotate where a NOT is to be inserted:

```
  CARROLL:0 ¯4↓(¯4×I<0)ϕα[¯1+|I←IXSω;]
  ⍝  α is text. ω is numeric matrix of propositions

      DATA CARROLL P
dreams
about Bath-buns
Not worth writing down
Not referred to my solicitor
Not really ridiculous
able to be expressed as a Syllogism
coming true
```

The conclusion is: All my dreams come true

Data Bases with Multiple-Line Entries

A surprising number of students wished to create a simple data base system. I had provided a workspace with a fairly powerful simulation of IBM's product "Query-by-Example", and had explained the principles involved in its implementation. The students could use this as a model, both to get experience in using such a system and to understand how such a system can be created in APL. However, I wanted them to write their own functions. Although the lessons I learned in writing the system in Direct Definition are relevant to the subject of this paper, a listing of the functions is too long to include here.

One of the problems encountered by students was how to handle the case where some items consist of multiple-lines. We did not have time to implement a system of arrays of arrays, but the following illustrates what might be considered to be the first step in dealing with the problem.

Consider the following data base:

```
      D
0     0    2    2 1915
1     2    4    1 1860
2     6    3    0 1826
0     9    1    2 1912
2    10    1    0 1830
3    11    1    0 1835
2    12    2    0 1837
0    14    2    2 1909
2    16    4    0 1820
2    20    2    1 1818
```

This matrix represents a bibliographic file giving Author, Title, Type of publication, and Date. With the exception of the date, *D*[;4], the numbers in any column are pointers to supporting character arrays. *D*[;0] points to a table of Authors, *AU*; *D*[;1] points to the START of the Title in a table of Titles, *TI*; *D*[;2] gives the number of lines to be taken from *TI*, so that ¯1++/*D*[;1 2] points to the LAST line of each title in *TI*; and *D*[;3] points to a table of Types, *TYPE*. The supporting character arrays (which were conveniently entered by the use of IBM's Extended APL Editor) are the following for this example:

```
      AU
Shaw, G.B.
Tennyson, A.
Scott, W.
Dickens, C.

      TI
Good King Charles' Golden Days:
A play for 6 characters.
Idylls of the King:
A poem describing the Life and Times
of King Arthur and the Knights of
```

```
the Table Round.
The Heart of Midlothian:
being an attempt at a Historical
Tale of the Olden Times.
Major Barbara.
The Antiquary.
Nicholas Nickelby.
The Tale of Two Cities:
Paris and London.
The Doctors Dilemma:
the problems of medical practice.
Waverley:
or Tis 60 Years Since;
the story of what followed the
late attempt at Revolution.
Marmion:
being Prelude to the Battle of Flodden.

      TYPE
Novel
Poem
Play
```

The problem, of course, is that whereas the Author, the Type, and the Date each take a single line, the Title takes a variable number of lines. Consequently we must expand the selected Authors by inserting blank lines to maintain compatibility with the multi-line titles. The Type and Date must be expanded in the same way. This is complicated by the fact that when we select from *D*, we rearrange the order of the Titles, and thus change the pattern of multi-lines. However, all the information needed for the solution is in D[;1 2]. Note that the result is an "Unnormalized Relation" in Relational Data Base terminology.

The functions needed are as follows:

```
RIX:((ω≠0)/ιρω)[¯1++\¯1⌽∨⌿(+\ω)∘.=1+ι+/ω] ⍝ Repeated indexes

CTI:((RIX α[;2])∊ω/ιρα[;2])⌿TI ⍝ Compress titles

EXP:¯1⌽(ι1↑ρω)∊¯1++\α
⍝ Expansion vector needed for compatibility with ω as prescribed by α

PRINT:(V⍀AU[ω/α[;0];]),M,(V←(ω/α[;2])EXP M←αCTIω)⍀ω⌿TYPE[α[;3];],⍕α[;,4]
```

- -

In order to understand how these functions work, consider the following illustrations:

```
      RIX 2 4 3          "Repeated indexes"
0 0 1 1 1 1 2 2 2

      W←D[;0]=2          Author with index 2
      W
0 0 1 0 1 0 1 0 1 1
```

```
        D CTI W                Titles of works by Author 2
The Heart of Midlothian:
being an attempt at a Historical
Tale of the Olden Times.
The Antiquary.
The Tale of Two Cities:
Paris and London.
Waverley:
or Tis 60 Years Since;
the story of what followed the
late attempt at Revolution.
Marmion:
being Prelude to the Battle of Flodden.

        W/D[;2]                         Numbers of lines in the titles
3 1 2 4 2                               of works by Author 2.

        V←(W/D[;2]) EXP D CTI W        Expansion vector for compatibility
        V                                                with the titles.
1 0 0 1 1 0 1 0 0 0 1 0

        V\AU[W/D[;0];]                 Expand the list of selected Authors.
Scott, W.



Scott, W.
Scott, W.

Scott, W.




Scott, W.
```

- -

1. To print the entire data base, we make the selection with a vector that is all 1's. The scalar 1 will extend as needed for compatibility.

```
      D PRINT 1
Shaw, G.B.         Good King Charles' Golden Days:     Play  1915
                   A play for 6 characters.
Tennyson, A.       Idylls of the King:                 Poem  1860
                   A poem describing the Life and Times
                   of King Arthur and the Knights of
                   the Table Round.
Scott, W.          The Heart of Midlothian:            Novel 1826
                   being an attempt at a Historical
                   Tale of the Olden Times.
Shaw, G.B.         Major Barbara.                      Play  1912
Scott, W.          The Antiquary.                      Novel 1830
Dickens, C.        Nicholas Nickelby.                  Novel 1835
Scott, W.          The Tale of Two Cities:             Novel 1837
                   Paris and London.
```

```
Shaw, G.B.          The Doctors Dilemma:                 Play  1909
                    the problems of medical practice.
Scott, W.           Waverley:                            Novel 1820
                    or Tis 60 Years Since;
                    the story of what followed the
                    late attempt at Revolution.
Scott, W.           Marmion:                             Poem  1818
                    being Prelude to the Battle of Flodden.
```

2. To select all works by Scott (Author 2):

```
      D PRINT D[;0]=2
Scott, W.           The Heart of Midlothian:             Novel 1826
                    being an attempt at a Historical
                    Tale of the Olden Times.
Scott, W.           The Antiquary.                       Novel 1830
Scott, W.           The Tale of Two Cities:              Novel 1837
                    Paris and London.
Scott, W.           Waverley:                            Novel 1820
                    or Tis 60 Years Since;
                    the story of what followed the
                    late attempt at Revolution.
Scott, W.           Marmion:                             Poem  1818
                    being Prelude to the Battle of Flodden.
```

3. To select all works of poetry (Type 1):

```
      D PRINT D[;3]=1
Tennyson, A.        Idylls of the King:                  Poem  1860
                    A poem describing the Life and Times
                    of King Arthur and the Knights of
                    the Table Round.
Scott, W.           Marmion:                             Poem  1818
                    being Prelude to the Battle of Flodden.
```

4. To select Plays and Novels (Types 0 and 2) published after 1830:

```
      D PRINT (D[;3]∊0 2)∧D[;4]>1830
Shaw, G.B.          Good King Charles' Golden Days:      Play  1915
                    A play for 6 characters.
Shaw, G.B.          Major Barbara.                       Play  1912
Dickens, C.         Nicholas Nickelby.                   Novel 1835
Scott, W.           The Tale of Two Cities:              Novel 1837
                    Paris and London.
Shaw, G.B.          The Doctors Dilemma:                 Play  1909
                    the problems of medical practice.
```

Conclusion

Some people object to APL, claiming either that it is "mathematical" or "inefficient". APL is rich in symbols, and there are those who seem to have real difficulty in using symbols as a tool of thought. This appears to be the reason why one hears objections to APL's mathematical content. Such people prefer to spell out every detail of a process (as one is obliged to do in a language like FORTRAN), or prefer PLUS to +, because they have not yet learned the lesson that Sylvester so eloquently taught a century ago. It takes time and patience to work with students who have this difficulty, to convince them with simple and meaningful examples that they can learn to use symbols to add power to their thinking. This is an old problem. As I pointed out at the APL Users Conference in 1978, Oughtred faced the same difficulty when he introduced the symbol × for multiplication. Students are greatly helped when they manipulate arrays that are meaningful to them; a student of language can transpose a multidimensional array of verb endings (where the axes are conjugation, tense, singular-plural, person, and individual letters), whereas a student of business will gain a similar experience from transposing an array representing financial accounts (where the axes are year, month, and line-item).

The objection on the grounds of supposed "efficiency" is best answered by demonstrating that practical results can be obtained far more quickly in APL than by any alternative. It is important that students have some experience with another language (preferably FORTRAN or COBOL) because these are still so commonly used. But, although some will be compelled by an employer to work in these languages, there will be few who will do so by choice after they have used APL. Moreover, if machine efficiency is really the issue, then it is hard to see why the programs are not written in machine language. I included an introduction to Assembler in my course, so that students would know what this means and would understand a little better what goes on inside the machine. But for a Biologist, an Economist, or other scientist or humanist to be working at any level other than APL seems to me to be where real inefficiency of valuable resources is serious.

Quite apart from its obvious utility as an aid in our solution of practical problems, APL can claim a key position in the curriculum of a Liberal Arts College. Benjamin Franklin, among others, said that man is a tool-making animal. The modern computer is the most powerful and universal tool so far created, and as such its study has a special importance. APL is not only the best way to learn about computers, for example with APL the student can design and operate a simple computer of his own, but it is part of the historical tradition and evolution of mathematics and symbolic thinking. This is what makes APL an exciting subject that should be included wherever the Liberal Arts are taught and respected.

Acknowledgements

I am grateful to K.E. Iverson (I.P. Sharp Associates) and P.B. Yale (Pomona College) for suggestions that have been incorporated in this paper.

Bibliography

Bergquist, J.W.	Algebraic Manipulation. Proc. 6th International APL Users Conference, Anaheim, California. May 14-17, 1974. p.45-49.
Berry, P. et al.	APL and Insight. 1978. APL Press.
Carroll, Lewis	Symbolic Logic. 1958 reprint. Dover Publications, Inc.
Douglas, Paul H.	The Cobb-Douglas production function once again: its history, its testing, and some new empirical values. Journal of Political Economy, 84 (1976) 903-915.

Falkoff, A.D. and D.L. Orth	Development of an APL Standard. APL Quote Quad 9, No. 4-Part 2 (June 1979), p.409-453.
Iverson, K.E.	Elementary Analysis. 1976. APL Press.
Iverson, K.E.	APL in Exposition. 1976. APL Press.
Iverson, K.E.	Introducing APL to Teachers. 1976. APL Press.
Iverson, K.E.	Programming Style in APL. Proc. APL Users Meeting, Toronto, 1978, p.200-224. I.P. Sharp Associates.
McIntrye, D.B.	Experience with Direct Definition One-liners in Writing APL Applications. Proc. APL Users Meeting, Toronto, 1978, p.281-297. I.P. Sharp Associates.
McIntyre, D.B.	The Architectural Elegance of Crystals made Clear by APL. Proc. APL Users Meeting, Toronto, 1978, p.233-250. I.P. Sharp Associates.
Poole, Robert W.	Quantitative Ecology. 1974. McGraw-Hill, Inc.
Sykes, Jr., Roy A.	Collected Whiz Bangs: An Anthology of Tutorials on APL Programming Techniques, Vol.I, 1978. STSC, Inc.

TEACHING APL IN AN ACADEMIC ENVIRONMENT

Gillian Wade
Institute of Computer Science
University of Guelph
Guelph, Ontario

Introduction

The unfortunate division between scientists and non-scientists first explored so perceptively by C.P. Snow in his seminal work **The Two Cultures** (in 1959) has not appreciably narrowed in the intervening years; rather, the continuing technological revolution has, if anything, exacerbated this seemingly unbridgeable chasm. The paradox, of course, is that the mounting wave of technological achievement is largely due to specialization — in universities and industries, research labs and businesses — and, in turn, its maintenance and further development demands an even higher degree of specialization. One of the most important and frustrating gaps today is between those people (and not all of them are scientists, by any means) who feel comfortably at home with computers and those who don't.

As computers play an increasingly important role in all our lives (which critics, unfamiliar with these machines, darkly hint is an ominous intrusion), it becomes imperative that we strive to dismantle the barriers that exist and neutralize the growing polarization between "**them**" and "**us**", and work towards a more widespread acceptance of the computer as another powerful and highly useful tool, which should not be beyond the grasp of any reasonably intelligent person to use.

Those of us who are already involved in this exciting and ever-expanding field must remember, as one of my students pointed out in his feedback comments, that there is "life beyond computing", and it is up to us, to all of us, to reach out beyond our ivory towers and computing centres and spread the knowledge and share the experience we have gained. We have an obligation to our non-computing colleagues to do our best to dispel the modern myth of the computer as a mysterious and infinitely complex "black box with flashing lights" that can only be used by a highly-trained and specialized elite. Computers are for everyone to share! Indeed, the rapid growth of the home-based microcomputer industry is hopefully an early clue to the new direction, but there is still much for us to do.

For many years, computer professionals have too often been their own worst enemies in forging the negative image so many people have of computers, for everything they did was couched in highly technical jargon and complicated buzz-phrases. Simply to use the machines required more than a passing familiarity with what made them run, and an extensive knowledge of the intricate computerese of COBOL, FORTRAN, ALGOL and so on... but there is no longer any excuse for this situation, for now we have an ideal language for non-computer people: APL.

APL at Guelph

The joy of APL is that it allows so many people, with so many diverse requirements, to use the resources made available through the computer with a minimum of technical know-how. The comparatively easy acceptance of APL is as apparent in the commercial and industrial world as it is in the academic environment with which I am most familiar — perhaps even more so, for unfortunately, many computing science professors still seem to regard APL as a rather fun toy but not the first choice as a language one would select for any serious work.

One of the most gratifying aspects of my own role at the University of Guelph is the opportunity I have to introduce young school-children to the world of computers. Each year, several special groups, many with children as young as the age of ten, come to the Institute of Computer Science for an introductory course in APL. These young people have no pre-conceived notions or fears of the machine and within hours of their first acquaintance with the system they are happily using APL at their terminals with far greater ease than I remember my own generation acquiring the three "R's", which is a promising indication of future trends.

Older students — and I don't just mean those enrolled in university undergraduate courses — are a somewhat different matter, and the manner in which these people are introduced to computers in general, and APL in particular, is of great importance. The wrong approach can even aggravate rather than overcome the latent suspicion and even dread of computers that, as I have remarked, is still so widespread among those with little or no computing experience. There are probably as many ways to teach APL as there are rival theories for the teaching of conventional foreign languages, but I will outline here what we are doing with marked success at the University of Guelph, and hopefully you will find at least parts of our approach to be useful in your own situation if you are faced with the job of arranging an introductory APL course for students, employees or clients.

At Guelph, degree courses in computing are offered by the Department of Computing and Information Science, but despite the dramatic growth in the use of time-sharing on campus over the past few years they still have no credit course where students can learn APL. The procedural languages have traditionally been dominant for it is felt that they give students a better understanding of the technological sub-structure. One or two of the introductory lectures, as well as courses offered by the Department of Mathematics and Statistics, do include a brief discussion of APL. But as one leading faculty member confessed to me, the primary reason that this situation has not changed is because of bureaucratic inertia.

However, students in such diverse disciplines as Economics, Zoology and Hotel Administration are often required to use APL for some of their assignments; in addition, an increasingly large group among the faculty and administrative staff are discovering the convenience of APL to help with their research or data processing. To meet this demand, all formal APL training has become the responsibility of the Institute of Computer Science, the organization which handles the bulk of computing services for the campus.

The institute offers a regular non-credit introductory course in APL, as well as many ad-hoc seminars to inform users of new features, tricks and techniques we've developed, new public library programs, and more sophisticated APL topics. Some people, of course, have managed to learn APL on their own, with just a little help from Institute staff, as a means of solving problems in their particular discipline, but the majority of our users have at some time taken our introductory course. I'd like to outline now how our course is organized, and why we have chosen to do it in this particular way.

Our APL Teaching Program

Our first course in APL was offered in September of 1970, at which time some 30 students were introduced to this versatile language. A decade later, we are teaching well over 300 people a year, and indications are that demand will continue to be high. In this ten-year period the course has gone through several changes. We have now arrived at a format which we feel best suits the needs of our users, though of course we are constantly refining the content in line with the evolution of the language itself.

Obviously, some of the considerations we have had to bear in mind won't necessarily apply in your case. For example, our courses are taken by an extremely diverse group of people, including undergraduates, graduate students, faculty members, researchers, administrators and technicians. Their computer expertise ranges from none at all to several years experience with some of the more traditional languages and systems. A company organizing an APL course for some of its employees, for instance, can expect a more uniform group of students. We are also fortunate in that all the individuals we teach are highly self-motivated. Some of them merely want to acquire a minimum amount of computer expertise to apply to their own field of interest, and have chosen APL as the most convenient tool, while others are interested in extending their computer skills and becoming APL programmers themselves.

It is most important to first analyze your potential audience in order to achieve the right level of presentation. It's very easy to oversimplify and spend a disproportionate amount of time on what may be a trivial topic; and it's equally easy to take for granted concepts you have already mastered but which may be more difficult for a novice to grasp at the first introduction, skimming over points which may deserve more detailed explanation. Once you have lost your students' attention, whether from oversimplification or overcomplexity, it can be hard to regain it. The dividing line is not at all clear, nor is it always possible to achieve, but it does make the task easier if some initial assessment of this kind can be made.

Another factor that has influenced the presentation of Guelph's APL course is the available technical facilities and teaching personnel. Because of physical limitations, and because we have found that a smaller group is more rewarding both for the teacher and the students, we have an enrollment ceiling in each of our classes. At the moment this ceiling is 30 people, which we feel is still too high, but because of resource constraints we are unable to lower it. The ideal arrangement, perhaps, is to have every student seated at his or her own terminal so that they can immediately try out each new concept as it is presented by the instructor. Unfortunately, we are far from achieving this one-to-one goal, and so we have had to adapt to what is available. There is a CRT in the classroom for the instructor's use, and our computing laboratory is equipped with 26 terminals for everyone to use. In addition, several public terminal pools and departmental machines ensure that no one has difficulty in getting access to the computer at times of their own choosing to try their hand at APL.

The demand for places is high, and so each semester (that is, three times a year), we teach at least three separate courses. Indeed, last summer overflow registrations necessitated the scheduling of two additional sessions. The timing is important, since many of our students need to learn their APL early in the semester in order to apply it to their other work. So we arrange overlapping rather than consecutive courses, and each session is taught by a different staff member, none of whom, incidentally, is a professional lecturer.

It was decided early on that uniformity in presentation was important, since we wanted to ensure that students taking any of the sessions all covered the same material, and we wanted to make it easy for a student who was unable to attend, say, the second class of his particular group to sit in on the same lecture with another group at another time without missing anything. Obviously, the personal enthusiasm and knowledge of the individual instructors will have a decisive influence on his teaching style and his audience's appreciation of the subject, but by providing a uniform framework for the course we believe we can supply our students with the best instruction while at the same time making it easier for newcomers to our APL team to teach the same course by giving them the benefit of our cumulative experience. Everyone uses a common set of notes, slides and assignments, which naturally reduces time spent on class preparation. Introductory manuals on APL and the file subsystem are given to all attendees. There is no required textbook, though we are frequently asked if there is one available. We recommend "APL: An Interactive Approach" by Gilman and Rose, and Paul Berry's "SHARP APL Reference Manual" is also becoming popular.

Originally, three separate mini-courses were offered, but since most people took all three anyway, it seemed more sensible to amalgamate them into a single introductory course. This course is divided into six modules, each three hours long, given twice a week over a period of three weeks. So students receive a total of 18 hours of formal instruction, 12 in the classroom and 6 hours of supervised practice in the laboratory. This format is well suited to the needs of our students, and again, our experience has indicated that dividing the material into smaller "chunks" with breathing space for practice in between makes it easier for them to learn than presenting a concentrated course jammed into a couple of days. The threshold of "mental indigestion" is fairly low, and though crash courses can be effective, we do not feel this approach is appropriate for our purposes at the introductory level. We have recently experimented with spreading the course over a six-week period, but the consensus is that a seven-day gap between classes is too long, and the present scheduling seems to be the most effective.

During the two-hour classroom period the instructor demonstrates each point using a terminal hooked into TV monitors, and students are encouraged to ask questions and discuss various points at any time. Immediately after the lecture there is a one-hour lab session in which students can take turns at the terminals to get some "hands-on" experience while the teacher is present to give answers and deal with problems or misunderstandings. Some simple assignments are given out and the students are expected to spend time at the terminal on their own to try out what has been covered in class, for only when the theory is put into practice is it possible to see if the concepts were understandably presented. Any difficulties or "fuzzy" areas are discussed at the beginning of the next session. Because of the students' differing interests it isn't possible to tailor assignments specifically to everyone's needs, and so instead we choose exercises and examples from the standard textbooks. 81% of the students complete most of the assignments, and of those unable to do them all, 88% said they were unable to find the time. This is an insoluble problem, however, for many were still unable to devote more time even when the course was spread out over six weeks.

The six modules have been constructed so that each builds on the others, forming a continuous whole, while at the same time being sufficiently self-contained to allow those people who do not wish to attend the full course to sit in on just those topics they wish to cover either to get started or for revision purposes. The first session introduces the concept of time-sharing in general and the basic elements of the APL language, including: monadic and dyadic scalar functions; the concept of scalars, vectors and matrices; assignment; selection using indexing, take and drop; the reduction and scan operators; and rho.

In the second class, students learn workspace management, the more frequently used system commands, system functions and system variables, and the APL public library structure. For some, this is as much as they will need. Some 70% of the students, however, proceed to the third module which covers more of the language including the relational and logical primitives, character data, compression and expansion, selection on matrices, inner and outer products, and a variety of other array manipulations. The fourth and fifth sessions cover writing, editing and debugging programs, branching, mixed output, quad and quote-quad. Students are given a programming assignment which is demonstrated in class. The final lecture is devoted to a brief overview of the APL file system.

A recent survey of all the Institute's non-credit courses provided some interesting statistics about how readily most people accept APL. 94% of those responding who had taken our APL course stated that their interest in computing had been increased, and 92% now found it easier to use the computer. Nor were these all novices; about half had had other exposure to computers. 87% stated that they had been able to apply their new-found knowledge ~~to~~ their own work. One reseacher lauded APL as giving him a "quantum leap" in his ability to analyze his data.

A look at some of the APL projects now in use or being developed on campus also offers an intriguing glimpse of the rich variety of applications to which the language is being put. APL is used for diet analysis, for the simulation of a hotel's back and front office operation, to teach perspective drawing for landscape architecture, wildlife management, and to maintain personal bibliographies and collections of references. On the administrative side, an APL system maintains all the University's complicated budgeting. A veterinary professor, delighted at the new prospects opened up to him, bought a terminal and a flat-bed plotter and quickly wrote some simple APL programs to produce graphs of equine electrocardiograms and other research data of a sophistication that previously had been almost impossible to achieve. For several years, a group of physicists had been attempting to translate some extremely complex matrix convolutions for analyzing X-ray spectra into FORTRAN. Within three months of taking our introductory APL course, they had successfully produced the results they wanted. True, they spent ten thousand dollars to do it, but they have written several major research papers about their findings, and they now have a system which, as far as we are aware, is the first to apply these APL techniques to atomic physics.

Conclusion

We strive constantly to improve our courses to provide the best training we possibly can, and feedback from our students is important to us. Each person attending is asked to complete a course evaluation form, and these are used to define our areas of weakness and try to improve them. We spend a great deal of time consulting with and giving advice to our students after they have completed the introductory course to help them develop their own APL skills. To provide easy access to the computer, free account numbers with a modest spending ceiling are provided to users who have no other source of funding. As an indication of the wide use of APL, these free numbers accounted for some 30,000 computing dollars in the past fiscal year even at our comparatively low internal rates.

Where do we go from here? A common problem facing all academic institutions in the coming years will be tackling an increasing workload with the same number of personnel, and with even less in some cases, in the face of cutbacks in educational spending. In business too, individual productivity will assume ever greater importance and since this is an area in which APL excels, its use will inevitably continue to grow. At Guelph, we are planning to upgrade our APL self-teaching courses so that the computer itself can play a more prominent role in teaching its own use, and we would like to develop pre-recorded audio-visual presentations, thereby reducing some of the load on our human resources.

As several of the other papers presented here have clearly demonstrated, APL offers many technical advantages over rival computing systems, but I hope this account of our training methods at the University of Guelph has demonstrated yet another virtue of APL, in that its comparative simplicity and flexibility make it the ideal language for teaching computing to those people who need it as another tool for use in their own particular area of interest. And in offering a relatively painless introduction to the enormous benefits of computer resources to people with little or no previous experience, it is building a sturdy bridge across that gap which divides so many specialists in both academic and non-academic environments.

APL IN THE CLASSROOM

E.M. (Ted) Edwards
Simon Fraser University
Burnaby, B.C.

Abstract

The author has used APL as a teaching tool in courses directed at students ranging from high school to senior year at university. Topics in these courses have included computer programming, curve fitting, mathematical logic, set theory, predicate calculus, computer hardware design and others. APL has been used as a language of discourse, as a programming tool for student use, and to produce software for editing lesson material and for presenting examples "on-line" in the classroom. Examples of lesson material are included and the reasons for the success of APL in this environment are discussed. Some limitations are also examined and a few possible extensions suggested.

Introduction

The author has used APL in the classroom in two distinctly different modes. In the first, the objective has been to teach APL and in the second, to teach some other topic. In this second case, APL has been used as a language of discourse. That is, having examined various notations used in the subject field, the author felt that APL offered the students advantages that more than offset the overhead of learning a new notation.

Regrettably, many teachers of elementary mathematics courses overlook the power of array notations to convey functional concepts through the examination of patterns (something the human mind is very good at). This oversight occurs since it is generally believed that arrays and array operations are much more difficult to understand. This is, in fact, true if one restricts oneself to conventional notation. APL provides a notation in which the generalization from single element operations to array operations occur in a straightforward and comparatively simple fashion. Some examples will be given below. Fortunately, APL has (to a very large extent) the property that "what you don't know won't hurt you". That is APL may readily be subsetted or, in other words, the teacher need only introduce as much as is needed to meaningfully discuss the subject matter at hand. Further, even this subset may be introduced only as each aspect is needed. In any course the author has every taken or taught, this has occurred anyway as the instructor found it convenient to introduce special symbolisms to expedite the discussion. It is not, then, as drastic a step as it may appear to introduce APL as a notation. In every class the author has taught in which APL has been involved, the concept of scalar functions applied to arrays has been introduced early in the first lecture without apparent difficulty. (See sample lessons included in this paper.)

At the present time, few textbooks are available in which APL has been used as notation. This can mean a considerable amount of extra work for the instructor as it must be recognised that the order of topics and their presentation must often be radically altered. The problem is akin to that of translating poetry. The idioms available for expression are different in different languages. A good translation must present the material in idiomatic form in the target language. In the context of this discussion, that often means considerable effort rearranging the material for presentation in APL. It is the opinion of this author that the investment is often warranted.

Teaching APL

Shown below is the outline for a three week course in computer programming taught by the author. This course is offered as part of the Simon Fraser University Summer Computer Institute. It is directed primarily towards high school students as an enrichment program. Students from grades eight through twelve plus a few of their teachers have taken the course. The primary objective is to learn to program a computer in APL, although the concept of the notational use of APL is introduced early and used throughout. Those students who have some computing background in another language (usually BASIC) present some of the greatest difficulties in terms of misconceptions about computing that need to be dynamited. The worst of these are with regards to modularity and the nature of iteration as distinct from loops used to express what are in reality disguised array operations.

Easy as APL 01
An Introduction To APL
Course Outline

LESSON 1
- Expressions and results
- Some APL primitive functions
- Vectors
- Names for data objects
- Defining your own functions

LESSON 2
- Tables and matrices
- More primitive functions
- Indexing
- Character strings
- Arrays of truth values
- Graph plotting as an example of array thinking

LESSON 3
- Some more primitive functions
- Automatic scaling for a graph plotter
- Manipulating arrays

LESSON 4
- Character strings and their handling

LESSON 5
- What is inner product?

LESSON 6
Simultaneous equations the easy way

LESSON 7
Polynomials and curve fitting

LESSON 8
An example of visualization of array operations
Projections in N-dimensional spaces

The remaining time will be spent assisting students to implement a project of their own.

The first eight to ten days are spent going through the material of the eight lessons. Mornings are spent in classroom lectures and afternoons are spent gaining "hands-on" experience. An MCM APL computer connected to two large screen video monitors is used to present the material of the lessons in class. A set of hard copy notes is also given to the students. The technique of preparation and presentation of the lesson material is the subject of another paper (Edwards 1). Students have access to the university's MTS APL system running on an IBM-370/148 and to some MCM APL stand-alone systems for their practice sessions. The final week of the course is spent on individual projects with the instructor and an assistant available to provide help and to suggest appropriate programming techniques. Portions of lessons 1, 5 and 8 are shown below as an indication of the presentation.

```
      ⍝          EASY AS APL 01 (LESSON 1)
      ⍝EXPRESSIONS AND RESULTS
      2+3
5
      2+3+4+5
14
      (2+3)×(4+5)
45
      ⍝MAXIMUM ('⌈' IS READ 'MAX')
      2⌈3
3
      7⌈5
7
      ¯3⌈¯1
¯1
      ⍝AS YOU WOULD EXPECT, '⌊' IS MINIMUM.
      2⌊3
2
      ⍝RIGHT TO LEFT EVALUATION ORDER
      2+3×4+5
29
      2+5⌈3+4
9
      9÷2
4.5
      4+6÷2+1
6
```

```
        ⍝VECTORS
        2 3 1+3 4 5
5 7 6
        2 2 2+3 4 5
5 6 7
        2+3 4 5 6
5 6 7 8
        5⌈3 4 5 6 7
5 5 5 6 7
        2 3 4×5 6 7
10 18 28
        2 3 4⌈5 3 2
5 3 4
        2 3 4⌊5 3 1
2 3 1
        3⌊1 2 3 4 5
1 2 3 3 3
        ⍝
        ⍝'⍝'IS READ LAMP.  THE STATEMENT THAT FOLLOWS IT
        ⍝IS FOR ILLUMINATION ONLY (I.E. A COMMENT).
        ⍝NAMES
        DATA←2 3 4 5 6
        ⍝NAME: A STRING OF LETTERS OR NUMBERS BEGINNING WITH A
        ⍝LETTER.  '←' IS READ 'IS' AND ASSIGNS A VALUE TO A NAME.
        DATA
2 3 4 5 6
        2×DATA
4 6 8 10 12

        ⍝              EASY AS APL ○1 (LESSON 5)
        ⍝
        ⍝      INNER PRODUCT
        ⍝
        ⍝AN EXAMPLE OF AN INNER PRODUCT EXPRESSION:
        2 3 4+.×3 2 1
16
        ⍝OBSERVE THAT WE OBTAINED A ONE ELEMENT RESULT FROM TWO
        ⍝VECTOR ARGUMENTS.  YOU SHOULD VERIFY THAT IT WAS IN FACT A
        ⍝SCALAR.  CONSIDER THE TWO EXPRESSIONS:
        2 3 4×3 2 1
6 6 4
        +/2 3 4×3 2 1
16
        ⍝FOR VECTORS α AND ω, AND SCALAR DYADIC FUNCTIONS F AND G,
        ⍝(αF.Gω)=F/αGω.  CURRENT APL'S ALLOW INNER PRODUCTS TO BE
        ⍝FORMED FROM PRIMITIVE FUNCTIONS ONLY.  INNER PRODUCTS MAY
        ⍝BE APPLIED TO HIGHER RANK ARRAYS BY REGARDING THEM AS
        ⍝ARRAYS OF VECTORS.  STUDY THE EXAMPLES BELOW TO SEE HOW
        ⍝THIS IS DONE.

        ⍝                  2  5  4  3  1
        ⍝     +.×          3  5  4  2  2     B
        ⍝                  5  4  3  1  3
        ⍝                  5  4  2  2  5
        ⍝     A
        ⍝ 1  2  3  1      28 31 23 12 19
        ⍝ 2  3  1  2      28 37 27 17 21     A+.×B
        ⍝ 3  1  2  3      34 40 28 19 26
        ⍝
        +/ 2 3 1 2 × 4 4 3 2
        ⍝+.× IS 'ORDINARY' MATRIX MULTIPLICATION
        ⍝             EASY AS APL ○1 (LESSON 8)
        ⍝
        ⍝     PROJECTIONS
        ⍝
        ⍝SAY WE HAVE A VECTOR IN 3-SPACE
        L←1 2 3
        ⍝WE WOULD LIKE TO FIND THE PROJECTION OF L ONTO THE X-Y
        ⍝PLANE I.E. THE PLANE SPANNED BY
        M←1 0 0
        N←0 1 0
        ⍝WE COULD PUT THESE TWO VECTORS INTO A MATRIX
        ⎕O←⍉2 3⍴M,N
1 0
0 1
0 0
        ⍝ANY VECTOR IN THE PLANE SPANNED BY THE COLUMNS OF O CAN BE
        ⍝OBTAINED BY (O[;1]×P[1])+O[;2]×P[2] I.E. O+.×P FOR SOME 2
        ⍝ELEMENT VECTOR P.  THUS WE CAN SPEAK OF THE PLANE, O.  IF
        ⍝L LAY IN THE PLANE O, THEN WE COULD FIND P BY SOLVING
        ⍝L=O+.×P.  SIMULTANEOUS EQUATIONS AGAIN!  WELL, LET'S TRY
        ⍝SOLVING WITH ⌹ (OR QD∆) EVEN IF L ISN'T IN O.
        ⎕P←L QD∆ O
1 2
        ⍝IF WE ADD 1 OF M AND 2 OF N WE GET:
        M+2×N
1 2 0
        ⍝WHICH IS THE DESIRED PROJECTION!  BUT THIS IS JUST:
        (O[;1]×P[1])+O[;2]×P[2]
1 2 0
        ⍝WHICH IS:
        O+.×P
1 2 0
        O+.×L QD∆ O
1 2 0
        ⍝WOULD IT MATTER IF THE COLUMN VECTORS OF O WERE NOT OF
        ⍝LENGTH 1?  LET'S MAKE O[;1]=2 0 0 AND FIND OUT.
```

Defined functions are introduced in lesson 1 and an example is included in which one function calls another. Outer product is dealt with in lesson 2. Direct definition (Iverson 1) (the alpha/omega notation) is used throughout. 'Del" definition is explained near the end of the course. As may be seen, array concepts are introduced early and used throughout. In fact, the students are given no mechanism to create a loop until well on.

Teaching With APL

Some texts have been produced in which APL has been used as the language of discourse. Excellent examples are Blaauw 1, Iverson 1, 2, Orth 1 and Spence 1. Iverson's texts deal with high school algebra and elementary analysis, Orth's deals with calculus and Blaauw's and Spence's with electronics. In the spring semester of 1980, the author was asked to teach a course in discrete mathematics. This course has previously been taught in the conventional manner and covers the material shown in the outline below but in a different order. The author decided that this material was a natural place for the use of APL as notation. As no suitable text was available, the author developed a set of notes for the course as a first step to producing such a text. A sample of some of the material is shown below.

The usual presentations seem to treat the topics of logic, set theory and relations almost as totally separate topics with a different notation for each. Eventually, students spot the connections between these topics, but usually not while they are taking the course. It is this author's opinion that set theory is most readily introduced via the characteristic vector used to indicate the truth or falsity of the proposition "these elements of the universe of discourse belong to this set".

Relations may be introduced by generalising the characteristic vector to a characteristic array with a matrix representing a set of sets. Boolean matrices are a natural way to deal with cartesian product sets, partitions, and other topics where sets of sets are the objects of study.

CMPT 205-3 - FALL 1980
Introduction To Formal Topics in Computing Science

INSTRUCTOR: E.M. Edwards
C.C. 7318
291-3761

This course introduces some of the theoretical tools and concepts used in computing science. The purpose of the course is to provide the student with some facility at recognizing that some problems may be modelled in formal systems, and in using such formal systems to advantage in solving problems and designing algorithms.

APL is used as notation for developing concepts and proving theorems as well as for programming examples of the theory. The material has been arranged so as to gradually introduce the necessary subset as required. No prior knowledge of APL is assumed.

The main topics to be covered are:

1. Logical statements, truth values and truth tables

2. Logical functions and identities
3. Introduction to set theory
4. Predicate calculus
5. Relations and graphs
6. Equivalence relations and partitions
7. Ordering relations and the basis for sorting
8. Functions
9. Some aspects of number theory

TEXT:
"Computing Science 205 Notes", E.M. Edwards
"Easy As APL 01", E.M. Edwards

```
                           CMPT 205 - NOTES 10
        ⍝                RELATIONS (CONTINUED)
        ⍝
        ⍝PARTITION AND COVERING
        ⍝LET U BE A SET (WITH NO REPEATED ELEMENTS) AND LET S
        ⍝BE A SET OF SETS REPRESENTED BY A MATRIX, EACH COLUMN
        ⍝OF WHICH IS A CHARACTERISTIC VECTOR FOR A SUBSET OF U.
        ⍝FURTHER, LET S BE SUCH THAT THE UNION OF ALL THE SETS
        ⍝IN S IS U.  E.G.
        U←⍳5
        ▼S←⍉3 5⍴ 1 1 0 0 0  0 0 1 0 0  1 0 0 1 1
1 0 1
1 0 0
0 1 0
0 0 1
0 0 1
        S[;1]/U
1 2
        S[;2]/U
3
        S[;3]/U
1 4 5
        ⍝THE UNION OF THE SETS IN S IS GIVEN BY
        S[;1]∨S[;2]∨S[;3]
1 1 1 1 1
        ⍝WHICH IS
        ∨/S
1 1 1 1 1
        ∧/∨/S
1
        ⍝SUCH A SET OF SETS IS CALLED A COVERING OF U.
        ⍝IF, IN ADDITION, ALL THE SETS OF S ARE DISJOINT, S IS
        ⍝CALLED A PARTITION OF U.  SETS ARE DISJOINT IF THEY
        ⍝CONTAIN NO ELEMENTS IN COMMON.  I.E THEIR INTERSECTION
        ⍝IS EMPTY.  FROM THE ABOVE IT IS CLEAR THAT IF S IS A
        ⍝PARTITION OF U, EACH ELEMENT OF U WILL BE IN EXACTLY
        ⍝ONE OF THE SETS OF S.  I.E.
        ∧/1=+/S
0
        ⍝IN THIS CASE THIS FAILS. 1 IS IN TWO OF THE SETS.
        +/S
2 1 1 1 1
```

```
      ⍝IF WE MODIFY S SLIGHTLY, IT BECOMES A PARTITION
      S[1;3]←0
      ⎕S
1 0 0
1 0 0
0 1 0
0 0 1
0 0 1

      ∧/1=+/S
1
      ⍝S IS A PARTITION SINCE ∧/1=+/S.  THE ELEMENTS OF A
      ⍝PARTITION ARE CALLED BLOCKS.  S HAS THREE BLOCKS.
      ⍝OTHER PARTITIONS OF U ARE POSSIBLE.  E.G.
      5 1⍴1
1
1
1
1
1
      ⍝THE PARTITION CONSISTING OF ONE BLOCK CONTAINING ALL
      ⍝THE ELEMENTS OF U.
      (⍳5)∘.=⍳5
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
      ⍝THE PARTITION CONSISTING OF ALL SETS WHICH CONTAIN
      ⍝EXACTLY ONE ELEMENT OF U.
      ⍝LET B BE A SUBSET OF U REPRESENTED BY ITS
      ⍝CHARACTERISTIC VECTOR.  THEN B AND ITS COMPLEMENT
      ⍝PARTITION U.
      ⎕B←U∊2 4
0 1 0 1 0
      ~B
1 0 1 0 1
      ⍝WE MAY CONVENIENTLY GENERATE THIS PARTITION BY
      ⎕P←B∘.=1 0
0 1
1 0
0 1
1 0
0 1
      ⍝TO SEE WHY THIS WORKS, CONSIDER THAT B=1 LEAVES B
      ⍝UNCHANGED WHILE B=0 IS THE SAME AS ~B.
      ∧/1=+/P
1
      ⍝THUS P IS A PARTITION OF U.
      ⍝
      ⍝LET B AND C BE SUBSETS OF U.
      B←U∊3 4
      C←U∊2 4 5
      ⍝THEN THE FOUR SETS
      ⎕I0←(~B)∧~C
1 0 0 0 0
      ⎕I1←(~B)∧C
0 1 0 0 1
```

```
      ⍞I2←B∧~C
0 0 1 0 0
      ⍞I3←B∧C
0 0 0 1 0
      ⍝FORM A PARTITION OF U.
      ⍞I←⍉4 5⍴I0,I1,I2,I3
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
0 1 0 0
      ∧/1=+/I
1
      ⍝AS CAN BE SEEN, I IS A PARTITION OF U.  THE SET I IS
      ⍝CALLED THE COMPLETE INTERSECTIONS OR THE MINTERMS
      ⍝GENERATED BY B AND C.  NOTE THAT THE POSITIONS OF THE
      ⍝'~' SYMBOLS IN THE EXPRESIONS FOR I0, I1, I2 AND I3
      ⍝CORRESPOND TO THE 0'S IN THE 2 DIGIT BINARY
      ⍝REPRESENTATION OF 0, 1, 2 AND 3.  WE MAY EXTEND THIS
      ⍝TO N SETS AND ELIMINATE EMPTY SETS AS FOLLOWS:
      MINTERMS:(∨⌿R)/R←ω∧.=BITS ¯1↑⍴ω
MINTERMS
      BITS:(ω⍴2)⊤¯1+⍳2*ω
BITS
      ⍝TO SEE THIS, CONSIDER THAT ∧/ IS THE INTERSECTION OF A
      ⍝SET OF SETS AND =1 OR =0 GIVES THE SET OR ITS
      ⍝COMPLEMENT (AS SHOWN PREVIOUSLY).  IF THIS IS APPLIED
      ⍝TO THE INNER PRODUCT DIAGRAM, THE RESULT IS CLEAR.
      ⍝REMEMBER THAT (N⍴2)⊤¯1+⍳2*N GIVES THE BINARY NUMBERS
      ⍝FROM 0 TO ¯1+2*N AS ITS COLUMNS.
      U←⍳10
      ⍞S←⍉3 10⍴(U∊1 2 5 6 9),(U∊2 3 4 5 9),U∊4 5 6 7
1 0 0
1 1 0
0 1 0
0 1 1
1 1 1
1 0 1
0 0 1
0 0 0
1 1 0
0 0 0


      ⍞MS←MINTERMS S
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
      ∧/1=+/MS
1
      ⍝SO MS IS A PARTITION OF U.
      ⍝EMPTY SETS MAY BE ADDED TO OR REMOVED FROM A PARTITION
      ⍝WITHOUT AFFECTING ITS BEING A PARTITION SINCE AN EMPTY
      ⍝SET IS DISJOINT FROM EVERY SET AND CONTAINS NO ELEMENTS.
```

Below is a sample of a proof using APL notation. Since the notation is machine executable, an example is run in parallel with the steps of the proof to provide a partial check against errors (e.g. a wrong sign).

```
        ⍝THEOREM: (EQUIV R) ≤ (~R[A;B]) = ∧/~R[A;]∧R[B;]
        ⍝IN WORDS: LET R BE AN EQUIVALENCE RELATION.
        ⍝THEN A AND B ARE NOT RELATED IFF THE R-RELATIVES
        ⍝OF A AND THE R-RELATIVES OF B ARE DISJOINT.
        A←2∘B←5
        ⍝PROOF THAT (∧/~R[A;]∧R[B;]) ≤ ~R[A;B]
        (∧/~R[A;]∧R[B;]) ≤ ~R[A;B]∧R[B;B]     ⍝INSTANCE
1
        R[B;B]                                ⍝REFLEXIVE
1
        (∧/~R[A;]∧R[B;]) ≤ ~R[A;B]            ⍝A=A∧1
1
        ⍝PROOF THAT (~R[A;B]) ≤ ∧/~R[A;]∧R[B;]
        P←∧/~R[A;]∧R[B;]
        P = ~∨/R[A;]∧R[B;]          ⍝DE MORGAN
1
        P = ~∨/R[A;]∧R[;B]          ⍝SYMMETRY
1
        (~P) = ∨/R[A;]∧R[;B]        ⍝(P=Q) = (~P)=~Q
1
        (~P) ≤ R[A;B]               ⍝TRANSITIVE
1
        (~R[A;B]) ≤ P               ⍝(P≤Q) = (~Q)≤~P
1
        ⍝QED
```

Some Extensions to APL

Some of these (or similar) suggestions have appeared elsewhere. They are included here since their absence has been particularly noticed in classroom applications.

1. Allow the comment symbol to appear anywhere on a line. The processor would ignore everything after its first appearance.

2. Specification into selection expressions. An extremely potent feature of APL is the facility with which selection of sub-arrays may be done. Unfortunately, this aspect of current implementations is not symmetric with respect to "read" and "write" operations. It would be desirable to replace elements of an array with the same facility that allows them to be tested or examined.
   e.g.

```
      ⍝INSERT TWO 1'S ON THE DIAGONAL OF A 2×2 MATRIX SITUATED
      ⍝WITH ITS UPPER LEFT CORNER AT M[2;3] WITHOUT CHANGING ANY
      ⍝OTHER VALUES.
      M←4 6⍴0
      (1 1⍉2 2↑1 2↓M)←[illegible]
      M
0 0 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 0 0

      ⍝GIVEN EQUAL LENGTH VECTORS OF ROW AND COLUMN INDICIES, I
      ⍝AND J, INSERT VALUES X IN POSITIONS OF M GIVEN BY
      ⍝M[I[1];J[1]],M[I[2];J[2]],...
      (1 1⍉M[I;J])←X

      ⍝INSERT 1'S IN M[1;2], M[2;4], M[3;6], ...
      ((¯1+2×⍳1↑⍴M)⌽M)[;1]←1
```

3. Extension of operators to defined and composite functions. The principle of syntactical equivalence (i.e. defined functions and primitive functions are syntactically equivalent) currently fails for operators since defined or composite functions may not be indexed nor may they be used as arguments for reduction, scan and inner and outer product. The major difficulty to be overcome in proposing this extension is the handling of identities for reduction of empty arrays.

```
      ⍝THE SYMBOL '¨' WILL BE USED IN THE DEFINITION FOR A
      ⍝FUNCTION INDEX.
      AVG:(+/[¨]ω)÷(⍴ω)[¨]
      ▼X←3 4⍴⍳12
 1  2  3  4
 5  6  7  8
 9 10 11 12
      AVG X
 2.5 6.5 11.5
      AVG[1] X
 5 6 7 8
```

```
⍝IF A RELATION IS REPRESENTED BY A SQUARE BOOLEAN MATRIX,
⍝R, THE TRANSITIVE CLOSURE, S, OF R IS GIVEN BY:
S←∨.∧/[⎕IO](3⍴⍴R)⍴R

⍝MONTHLY IS A DEFINED SCALAR DYADIC FUNCTION WHICH RETURNS
⍝THE MONTHLY PAYMENT REQUIRED TO AMORTISE A $1 LOAN OF
⍝DURATION ⍵ AND MONTHLY INTEREST FACTOR ⍺.  PRODUCE A
⍝TABLE OF MONTHLY PAYMENTS FOR A $1200 LOAN AT MONTHLY
⍝INTEREST RATES OF 1, 1.5, AND 2 PERCENT WITH DURATIONS OF
⍝1, 2, 3, AND 5 YEARS.
TABLE←.01×⌈100×1200×.01 .015 .02∘.MONTHLY 12×1 2 3 5
```

4. Extended scalar conformability. Scalar dyadic functions should accept arguments of equal rank provided that, element by element, either their dimensions are equal or at least one of the pair of values is 1. In the event the ranks of the arguments differ by 1, the dimension to be coerced is supplied by indexing the function with the usual default of last dimension. This would permit such operations as:

```
⍝ADD ⍳5 TO EACH ROW OF A 5×5 MATRIX, X.
X←X+[⎕IO]⍳5

⍝MANY OF THE USES OF THIS EXTENSION LIE IN COMBINATIONS
⍝OF A REDUCTION AND A SCALAR FUNCTION.  E.G.

⍝SCALE THE ROWS OF A MATRIX SO THAT EACH LIES IN THE RANGE
⍝0 TO 1.
SX←SX÷S+0=S←⌈/SX←X-⌊/X

⍝FORCE THE MEAN OF EACH COLUMN OF X TO 0 (SEE AVG ABOVE).
X←X-[⎕IO]AVG[⎕IO]X
```

Conclusion

Materials have been presented from two courses in which APL plays a role, either as the subject being taught or as notation to aid in the discussion of another subject.

Many areas of Mathematics and Physics could benefit from the use of APL as a teaching notation if well planned and executed teaching materials are made available. Maximum benefit is gained when the required APL sophistication is developed gradually along with the subject matter. An APL system should be available to students for examples and problems. Problem sets should mix exercises which require notational and computational use of APL. The advantages of APL are completely lost if use is not made of array concepts such as outer and inner product, reduction, etc. Attempts to transliterate algorithms from other languages on the grounds that "loops are easier to understand" completely miss the point.

References

1. Blaauw, G.A., **Digital System Implementaton**, Prentice-Hall, 1976.
2. Edwards, E.M., **ECHO — A Smart Electronic Blackboard**, 3rd Canadian Symposium on Instructional Technology, 1980.
3. Iverson, K.E., **Elementary Analysis**, APL Press, 1976.
4. Iverson, K.E., **Algebra — An Algorithmic Treatment**, APL Press, 1972.
5. Orth, D.L., **Calculus in a New Key**, APL Press, 1976.
6. Spence, R., **Resistive Circuit Theory**, McGraw Hill, 1974.

THE INDUCTIVE METHOD OF INTRODUCING APL

Kenneth E. Iverson
I.P. Sharp Associates
Toronto, Ontario

Because APL is a language, there are, in the teaching of it, many analogies with the teaching of natural languages. Because APL is a **formal** language, there are also many differences, yet the analogies prove useful in suggesting appropriate objectives and techniques in teaching APL.

For example, adults learning a language already know a native language, and the initial objective is to learn to translate a narrow range of thoughts (concerning immediate needs such as the ordering of food) from the native language in which they are conceived, into the **target** language being learned. Attention is therefore directed to imparting effective use of a small number of words and constructs, and not to the memorization of a large vocabulary. Similarly, a student of APL normally knows the terminology and procedures of some area of potential application of computers, and the inital objective should be to learn enough to translate these procedures into APL. Obvious as this may seem, introductory courses in APL (and in other programming languages as well) often lack such a focus, and concentrate instead on exposing the student to as much of the vocabulary (i.e., the primitive functions) of APL as possible.

This paper treats some of the lessons to be drawn from analogies with the teaching of natural languages (with emphasis on the inductive method of teaching), examines details of their application in the development of a three-day introductory course in APL, and reports some results of use of the course. Implications for more advanced courses are also discussed briefly.

1. The Inductive Method

Grammars present general rules, such as for the conjugation of verbs, which the student learns to apply (by **deduction**) to partieular cases as the need arises. This form of presentation contrasts sharply with the way the mother tongue is learned from repeated use of particular instances, and from the more or less conscious formulation (by **induction**) of rules which summarize the particular cases.

The inductive method is now widely used in the teaching of natural languages. One of the better-known methods is that pioneered by Berlitz [1] and now known as the "direct" method. A concise and readable presentation and analysis of the direct method may be found in Diller [2].

A class in the purely inductive mode is conducted entirely in the target language, with no use of the student's mother tongue. Expressions are first learned by imitation, and concepts are imparted by such devices as pointing, pictures, and pantomime; students answer questions, learn to ask questions, and experiment with their own statements, all with constant and immediate reaction from the teacher in the form of correction, drill, and praise, expressed, of course, in the target language.

In the analogous conduct of an APL course, each student (or, preferably, each student pair) is provided with an APL terminal, and with a series of printed sessions which give explicit expressions to be "imitated" by entering them on the terminal, which suggest ideas for experimentation, and which pose problems for which the student must formulate and enter appropriate expressions. Part of such a session is shown as an example in Figure 1.

SESSION 1: NAMES AND EXPRESSIONS

The left side of each page provides examples to be entered on the keyboard, and the right side provides comments on them. Each expression entered must be followed by striking the RETURN key to signal the APL system to execute the expression.

```
      AREA←8×2                      The name AREA is assigned to the result
      HEIGHT←3                      of the multiplication, that is 16
      VOLUME←HEIGHT×AREA
      HEIGHT×AREA                   If no name is assigned to the result, it
48                                  is printed
      VOLUME
48
      3×8×2
48
      LENGTH←8 7 6 5                Names may be assigned to lists
      WIDTH←2 3 4 5
      LENGTH×WIDTH
16 21 24 25
      PERIMETER←2×(LENGTH+WIDTH)    Parentheses specify the order in which
      PERIMETER                     parts of an expression are to be
20 20 20 20                         executed
      1.12×1.12×1.12                Decimal numbers may be used
1.404928
      1.12*3                        Yield of 12 percent for 3 years
1.404928
```

SAMPLE PORTION OF SESSION

Figure 1

Because APL is a formal "imperative" language, the APL system can execute any expression entered on the terminal, and therefore provides most of the reaction required from a teacher. The role of the instructor is therefore reduced to that of tutor, providing explicit help in the event of severe difficulties (such as failure of the terminal), and general discussion as required. As compared to the case of a natural language, the student is expected, and is better able, to assess his own performance.

Applied to natural languages, the inductive method offers a number of important advantages:

1. Many dull but essential details (such as pronunciation) required at the outset are

acquired in the course of doing more interesting things, and without explicit drill in them.

2. The fun of constantly looking for the patterns or rules into which examples can be fitted provides a stimulation lacking in the explicit memorization of rules, and the repeated examples provide, as always, the best mnemonic basis for remembering general rules.

3. The experience of committing error after error, seeing that they produce no lasting harm, and seeing them corrected through conversation, gives the student a confidence and a willingness to try that is difficult to impart by more formal methods.

4. The teacher need not be expert in two languages, but only in the target language.

Analogous advantages are found in the teaching of APL:

1. Details of the terminal keyboard are absorbed gradually while doing interesting things from the very outset.

2. Most of the syntactic rules, and the extension of functions to arrays, can be quickly gleaned from examples such as those presented in Figure 1.

3. The student soon sees that most errors are harmless, that the nature of most are obvious from the simple error messages, and that any adverse effects (such as an open quote) are easily rectified by consulting a manual or a tutor.

4. The tutor need only know APL, and does not need to be expert in areas such as financial management or engineering to which students wish to apply APL, and need not be experienced in lecturing.

2. The Use Of Reference Material

In the pure use of the inductive method, the use of reference material such as grammars and dictionaries would be forbidden. Indeed, their use is sometimes discouraged because the conscious application of grammatical rules and the conscious pronunciation of words from visualization of their spellings promotes uneven delivery. However, if a student is to become independent and capable of further study on his own, he must be introduced to appropriate reference material.

Effective use of reference material requires some practice, and the student should therefore be introduced to it early. Moreover, he should not be confined to a single reference; at the outset, a comprehensive dictionary is too awkward and confusing, but a concise dictionary will soon be found to be too limited.

In the analogous case of APL, the role of both grammar and dictionary is played by the reference manual. A concise manual limited to the core language [3] should be supplemented by a more conprehensive manual (such as Berry [4]) which covers all aspects of the particular system in use. Moreover, the student should be led immediately to locate the two or three main summary tables in the manual, and should be prodded into constant use of the manual by explicit questions (such as "what is the name of the function denoted by the comma"), and by glimpses of interesting functions.

3. Order Of Presentation

Because the student is constantly striving to impose a structure upon the examples presented to him, the order of presentation of concepts is crucial, and must be carefully planned. For example, use of the present tense should be well established before other tenses and moods are introduced. The care taken with the order of presentation should, however, be unobtrusive, and the student may become aware of it only after gaining experience beyond the course, if at all.

We will address two particular difficulties with the order of presentation, and exemplify their solutions in the context of APL. The first is that certain expressions are too complex to be treated properly in detail at the point where they are first useful. These can be handled as "useful expressions" and will be discussed in a separate section.

The second difficulty is that certain important notions are rendered complex by the many guises in which they appear. The general approach to such problems is to present the essential notion early, and return to it again and again at intervals to reinforce it and to add the treatment of further aspects.

For example, because students often find difficulty with the notion of literals (i.e., character arrays), its treatment in APL is often deferred, even though this deferral also makes it necessary to defer important practical notions such as the production of reports. In the present approach, the essential notion is introduced early, in the manner shown in Figure 2. Literals are then returned to in several contexts: in the representation of function definitions, in discussion of literal digits and the functions (⍕ and ⍎) which are used to transform between them and numbers in the production of reports; and in their use with indexing to produce barcharts.

Function definition is another important idea whose treatment is often deferred because of its seeming complexity. However, this complexity inheres not in the notion itself, but in the mechanics of the general del form of definition usually employed. This complexity includes a new mode of keyboard entry with its own set of error messages, a set of rules for function headers, confusion due to side-effects resulting from failure to localize names used or to definitions which print results but have no explicit results, and the matter of suspended functions.

SESSION 4: LITERALS

```
      JANET←5                  Janet received 5 letters today
      MARY←8

      MARY⌈JANET               The maximum received by one of them
8
      MARY⌊JANET               The minimum
5
      MARY>JANET               Mary received more than Janet
1
      MARY=JANET               They did not receive an equal number
0
```

What sense can you make of the following sentences:

JANET has 5 letters and *MARY* has 8

JANET has 5 letters and *MARY* has 4

'*JANET*' has 5 letters and '*MARY*' has 4.

The last sentence above uses quotation marks in the usual way to make a literal reference to the (letters in the) name itself as opposed to what it denotes. The second points up the potential ambiguity which is resolved by quote marks.

```
      LIST←24.6 3 17
      ρLIST
3
      WORD←'LIST'
      ρWORD
4
      SENTENCE←'LIST THE NET GAINS'
```

INTRODUCTION OF LITERALS

Figure 2

All of this is avoided by representing each function definition by a character vector in the direct form of definition [5 6]. For example, a student first uses the function *ROUND* provided in a workspace, then shows its definition, and then defines an equivalent function called *R* as follows:

```
      ROUND 24.78 31.15 28.59
25 31 29

      SHOW 'ROUND'
ROUND:⌊.5+W

      DEFINE 'R:⌊.5+W'

      R 24.78 31.15
25 31
```

The function *DEFINE* compiles the definition provided by its argument into an appropriate del form, localizes any names which appear to the left of assignment arrows in the definition, provides a "trap" or "lock" appropriate to the particular APL system so that the function defined behaves like a primitive and cannot be suspended, and appends the original argument in a comment line for use by the function *SHOW*.

This approach makes it possible to introduce simple function definition very early and to use it in a variety of interesting contexts before introducing conditional and recursive definitions (also in the direct form), and the more difficult del form.

4. Teaching Reading

It is usually much easier to read and comprehend a sentence than it is to write a sentence expressing the same thought. Inductive teaching makes much use of such reading, and the student is encouraged to scan an entire passage, using pictures, context, and other clues, to grasp the overall theme before invoking the use of a dictionary to clarify details.

Because the entry of an APL expression on a terminal immediately yields the overall result for examination by the student, this approach is particularly effective in teaching APL. For example, if the student's workspace has a table of names of countries, and a table of oil imports by year by country by month, then the sequence:

```
      N←25
      B←+/[1]+/[3] OIL

      COUNTRIES,'.⎕'[1+B∘.≥(⌈/B)×(⍳N)÷N]
```

produces the following result, which has the obvious interpretation as a barchart of oil imports:

```
ARABIA   ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕.....
NIGERIA  ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕.......
CANADA   ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕..........
INDONESIA⎕⎕⎕⎕⎕⎕⎕⎕⎕................
IRAN     ⎕⎕⎕⎕⎕⎕⎕⎕.................
LIBYA    ⎕⎕⎕⎕⎕⎕⎕⎕.................
ALGERIA  ⎕⎕⎕⎕⎕⎕⎕⎕.................
OTHER    ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Moreover, because the simple syntax makes it easy to determine the exact sequence in which the parts of the sentence are executed, a detailed understanding of the expression can be gained by executing it piece-by-piece, as illustrated in Figure 3. Finally, such critical reading of an expression can lead the student to formulate his own definition of a useful related function as follows:

```
      DEFINE ⍞
BARCHART:'.⎕'[1+ω∘.≥((⍳α)÷α)×⌈/ω]
```

5. Useful Expressions

As remarked in Section 3, some expressions are too useful and important to be deferred to the point that would be dictated by the complexity of their structure. In APL such expressions can be handled by introducing them as defined functions whose use may be grasped immediately, but whose internal definition may be left for later study.

For example, files can be introduced in terms of the functions *GET*, *TO*, *RANGE*, and *REMOVE*, illustrated in Figure 4. These can be grasped and used effectively by the student at an earlier stage and with much greater ease than can the underlying language elements from which they must be constructed in most APL systems.

A further example is provided by the function needed to compile, display, and edit the character vectors used in direct definition of functions. For example, an editing function which deletes each position indicated by a slash, and inserts ahead of the position of the first comma any text which follows it (in the manner provided for del editing in many APL systems) is illustrated in Figure 5.

```
      N                               The width of the barchart
25
      Q←(⍳N)÷N                        Numbers from 0 to 1 in 25 equal steps
                                          (display if desired)
      ⌈/B                             The largest value to be charted
      C←(⌈/B)×Q                       Numbers from 0 to the largest value to be
                                      charted

      S←B∘.≥C                         Comparison of each value of B with
      S                               each value in the range to be charted
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

      3 21↑1+S                        Examine a piece of 1+S
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1
```

DETAILED EXECUTION OF AN EXPRESSION

Figure 3

If the first dimension of an array (list, table, or list of tables) has the value *N*, (for example, `1↑ρOIL` is 7), then it may be distributed to *N* items of a file by a single operation. For example:

```
      OIL TO 'IMPORTS 72 73 74 75 76 77 78'
```

*Use the function *GET* to retrieve individual items from the *IMPORTS* file to verify the effect of the preceding expression.

```
      COUNTRIES TO 'IMPORTS 1'
```
Non-numeric data may be entered

The functions *RANGE* and *REMOVE* are useful in managing files:

```
      RANGE 'IMPORTS'           Gives range of indices
1 72 73 74 75 76 77 78'

      REMOVE 'IMPORTS 73 75 77'  Removes odd years

      RANGE 'IMPORTS'
1 72 74 76 78
```

FUNCTIONS FOR USING FILES

Figure 4

Deferral of the internal details of the definition of these essential functions can, in fact, be turned to advantage, because they provide interesting exercises in reading (using the techniques of Section 4) the definitions of functions whose purposes are already clear from repeated use. For example, critical reading of the following definition of the function *EDIT* is very helpful in grasping the important idea of recursive definition:

```
EDIT:EDIT(A DELETE K↑ω),(1↓K↓A),(K←+/∧\A≠',')↓ω:0=ρA←⍞,0ρ⎕←ω:ω

DELETE:(~(ρω)↑'/'=α)/ω
```

Analysis of the complete set of functions provided for the compilation from direct definition form also provides an interesting exercise in reading, but one which would not be completed, or perhaps even attempted, until after completion of an introductory course. Extensive leads to other interesting reading, of both workspaces and published material, should be given the student to encourage further growth after the conclusion of formal course work.