APL TEACHING BUGS

Howard A. Peelle
School of Education
University of Massachusetts
Amherst, MA 01003 USA
(413) 545 - 0496

Murray Eisenberg
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003 USA
(413) 545 - 2859

Abstract

This paper discusses "APL teaching bugs", in three senses: (1) issues inherent in the teaching of APL that confront the instructor with difficult choices; (2) potential mistakes sometimes made by instructors teaching APL; and (3) problematic aspects of the design of APL that are especially difficult to explain. These teaching bugs are presented as provocative questions, but the "answers" are left to individual instructors. By facing these questions, teachers may make APL more comprehensible and hence foster its acceptance and growth.

Introduction

How should one teach APL, especially to novices? Advice on teaching APL is hardly lacking, whether it be statements of precepts (e.g., [2], [6], [8], [12], [13]) or actual examples of APL pedagogy (e.g., [7], [8], [14], [17]). Such advice typically carries an explicit or implicit message that the author's approach is the correct way, at least for the particular audience being taught. Before accepting or rejecting any such advice, it would be wise first to examine undogmatically some of the issues involved.

In the first part of this paper, we address a number of teaching issues (focusing on APL, of course) and the quandaries--or "bugs"--they present to students and teachers alike. In the second part, we expose some common practices that are or, if carried to extremes, could be mistakes--"bugs"--in the ways instructors teach APL. In the third part, we mention a few problematic aspects of the design of APL itself--"bugs"?--that pose substantial difficulties to instructors who try to explain them.

While we pose problems and elucidate potential flaws and pitfalls in teaching APL, we do not recommend specific remedies in this paper. For the students' sake it may not even matter greatly which way some of these problems are resolved by an individual instructor! Perhaps what really matters is that the instructor follow some coherent approach. In any case, each instructor at least needs to be cognizant of these issues and to confront them openly, in order to debug and refine his or her own approach.

Issues Inherent in Teaching APL

Let us begin with some general educational issues--not specific to APL, but pertinent to teaching any programming language (and, indeed, many other subjects). For emphasis we express these and the other issues as dichotomies, whereas in reality they entail entire continua of possibilities. Not included here are issues contingent upon audience or time allocation, much less any claim about what is "natural" to learn or teach.

Tutorial vs. Discovery:

Which overall pedagogical approach is appropriate? Present factual information directly (based on what the instructor's wisdom and experience deem best for stu-

dents to know)--or--let students explore freely, experience errors, and develop their own debugging techniques?

Logical Sequence vs. Ad-hoc Learning:

In what order should material be covered? Always build on previously mastered material--or--present an overview first, allow skipping around, give sneak previews, etc.? Is there any one best way to learn how to program--or--are there multiple, alternative learning paths, differing from student to student? And, how much does it matter where one starts?

For example, is it effective to begin teaching about defined functions in APL by showing briefly that they may take arguments and return results, but defer thorough coverage of that until after students learn the mechanics of function editing?

Answers vs. Learning Skills:

Tell students the answers they want when they ask--or--avoid giving direct answers to their questions but encourage them to develop learning skills, that is, learn how to learn what they need when they need it (cf. [8])? Correct students' programs for them (showing them shorter, faster, or "better" ways) and thereby ensure success for them--or--just support what they have done, as is, and let them take responsibility for their own work?

Instructor Errors:

Openly acknowledge errors made by the instructor while teaching (whether accidental, incidental, or contrived)--or--strive for perfection, showing only the best model of how it should be done?

Use of References:

Have students avoid reference manuals (which are usually not well-designed for learning)--or--encourage students to learn to use a reference manual (as in [8]) or other sources? One pundit has said, "School is a textbook; life is a reference manual." Should instructors then insulate beginners from the rigors of using reference manuals--or--help them prepare for the "real world" where a reference manual may be the only printed information available?

Use of Computer:

Learn programming "hands-on", at a computer or terminal-- or--acquire fundamentals first by reading books and writing programs (on paper or a blackboard)? Recall that APL was originally conceived as a mathematical notation, not as an actual computer programming language. Yet students will eventually have to use APL on real machines. Can postponing their first encounters with a computer mitigate the distress of techno-phobic students?

Mathematics Background:

Rely on students' mathematical background--or--expect that programming (especially in APL) will help crystallize their previous, perhaps muddled, mathematical understanding? Teach programming--or--teach mathematics?

Comparison with Mathematics:

Explicitly contrast the precedence-free, unambiguous notation of APL with the precedence-using, at times ambiguous notation of mathematics, perhaps denigrating the latter--or--let APL notation speak for itself, without reference to mathematics?

Comparison with Other Languages:

Deliberately compare APL with other programming languages, e.g., BASIC, PL/I, FORTRAN, Pascal, or LOGO, and connect to the previous programming experience of students--or--tell them to forget what they know about other languages and introduce programming anew in APL?

Principles vs. Features:

Take a seemingly "theoretical" approach by emphasizing the underlying principles of the language, which guided its design (APL is array-oriented, variables have shape as well as value, etc. [13])--or--be "practical" by emphasizing the role of the language's features and capabilities (e.g., system functions) in getting quick solutions to problems?

Algorithmic Efficiency vs. Creativity:

Pay careful attention to amounts of time and space, symbol table size, etc., consumed by algorithms--or--suppress questions of efficiency in favor of creative algorithm development? Is it proper, for example, to encourage a beginner to write a recursive function for finding determinants (using expansion by minors), when an iterative solution (using row reduction) executes more quickly? At a more basic level, may a looping solution be tolerated, for example, to the problem of deleting all blank rows of a matrix of names, when a non-looping solution using compression is available? Is it true that "Inexperienced programmers should almost never attempt to optimize"? [18]

Programming Style:

Teach elements of style--or--because style is inherently so idiosyncratic, let it be developed individually? For example, insist on a While-Do form, as in

```
      I←0
LOOP:→(N<I←I+1)/END
      .
      .
      .
      →LOOP
END: ...
```

because one can more readily discern the loop's structure--or--permit a Repeat-Until form, as in

```
      I←1
LOOP: ...
      .
      .
      .
      →(N≥I←I+1)/LOOP
```

when the novice prefers that for the problem at hand? (And, which of these two forms should be introduced first?)

Next are a number of high-level teaching issues which are especially significant in relation to APL.

Variant and Enhanced Versions:

There are many different versions of APL on the market, including several microcomputer implementations and three notable enhanced versions: IBM APL2, SHARP APL, and STSC APL*PLUS. There are differences on many levels and of many kinds: in underlying philosophies (e.g., floating vs. grounded arrays), in functions and operators available (e.g., a primitive Each operator vs. a primitive Rank operator), in choice of symbols (e.g., Enclose and Disclose), in system variables and system functions (e.g., Execute Alternate vs. Trap), in file handling (e.g., component files vs. shared variables), in notation (e.g., Trace and Stop--`⎕TRACE 'FNAME'` in most versions, as in the draft ISO standard, but still `T∆FNAME` in some), in whether a comment may appear on the same line as an executable expression and whether the statement separator ◊ is allowed, and in specific forms and effects of system commands (e.g., when to use `)SAVE WSNAME` instead of just `)SAVE`), etc.

With so many differences, the question arises: Which APL should one teach? This is a real question which is usually resolved quickly by whichever one is available. Does this mean that APL must be taught as a machine and implementation-specific language (it wasn't designed that way)--or--can some common core of APL be taught? If your students are using one particular version, should you stick exclusively to that version to avoid confusing them--or--should you expose students to differences they will meet in case they use other versions later? This issue certainly creates problems for APL students and instructors (and especially for authors of APL textbooks).

Array-Thinking:

"Thinking arrays" is often espoused as desirable in APL programming (see [6], e.g.). But, when should this be taught? In the very first lesson--or--after the student has seen non-array solutions and can appreciate array-oriented solutions? How should it be taught? Should students be encouraged to devise a sequential solution first and then transform it into an array-oriented solution ([11] and [13])? Can array-thinking be taught--or only learned? And just what is "array-thinking"?

Readability vs. Efficiency:

Define functions as clearly as possible, so that they may be read easily by someone else (or yourself) later on--or--define functions to run quickly and economize on space? For example, should learners be required to use distinctive variable names, as in

```
∇AMOUNT←PERIODS COMPOUND RATE
```

--or--reuse conventional names, as in

```
∇Z←A COMP B
```

to save space in the symbol table? And, when should you explain the trade-offs between readability and efficiency?

Idioms:

Furnish students with "cover functions" to name commonly used phrases--or--encourage recognition of "idioms" [12]? Introduce such idioms early because they are so useful, for example, `((V⍳V)=(⍳⍴V))/V` to select the unique members of a list--or--wait until students are in a position to analyze the phrase into its constituents? (See [8].)

Double-duty Symbols:

When should it be pointed out that many symbols on the keyboard have dual meanings? Introduce monadic and dyadic primitive functions together from the start to highlight the differences, even when some might not be well-motivated or of immediately obvious utility (e.g., monadic + or monadic |)--or--introduce monadic and dyadic forms well-separated in time, thereby obscuring the connection between them?

Classifications:

APL has a rich taxonomy: scalar vs. mixed functions, primitive vs. derived functions, etc. (This has become even more complex with enhanced versions--see [1], e.g.). When should such classes of functions be distinguished: as soon as examples appear--or--not until students

have had the opportunity themselves to experience what the distinctions mean?

Terminology:

APL terminology was carefully chosen, in part to avoid computer jargon ([3], [9]). Should an instructor use official, precise APL terms such as "function", "argument", "vector", and "matrix", which may intimidate students who are not mathematically inclined--or--use colloquial, perhaps less threatening terms such as "program" (at least for niladic functions returning no result), "input", "list", and "table"? And what about all the esoteric Greek and Latin terms, like "monadic" and "dyadic" and "modulo"?

Some issues pervade the pedagogical approach, that is, once an instructor decides what to do and when, it has implications for the rest of his or her teaching. (Such issues may be more critical in writing a book than in teaching a class!)

Higher-dimensional Arrays:

Introduce vectors, matrices, and higher-dimensional arrays early to reveal their underlying importance and thereby use them in more powerful examples--or--postpone higher-dimensional arrays until fundamental concepts and primitives are amply illustrated with scalars? For example, introducing vectors too early can have untoward consequences: 3+4 5 suggests 7 5 (see [4]); using matrices early forces the issue of syntax with Reshape, as in $(R,C)\rho V$ as compared to $3\ 4\rho V$.

Direct Definition:

Introduce and use direct definition form, because it is so concise, avoids the complexities of the ∇-editor, and encourages modular programming--or--don't, because the α and ω are additional, seemingly cryptic, Greek symbols, because it just adds another notation to learn, or because the available system may not have *DEF* implemented?

Comments:

Recommend generous use of comments in all function definitions to instill good habits of proper documentation and as an aid to comprehension--or--encourage minimal or no commenting, because defined functions are supposed to be short and intelligible? Should students have to learn to analyze functions without seeing comments? Are comments in defined functions crutches that fail to force extra effort toward a clear style? Should an APL function have to speak for itself?

APL Syntax:

How early should APL syntax be formalized? Should the instructor deliberately hold off for as long as possible (as in [14], e.g.), say by assigning intermediate results to variables instead of forming expressions with multiple functions, in order to protect the student from dissonance with their former mathematical or programming experience--or--get it over with right away, admitting that APL is unusual but asserting that syntax is the heart of APL?

Redundant Parentheses:

Encourage--or--discourage the use of redundant parentheses in formulating APL expressions? Redundant parentheses in expressions such as $(A+B)\times(C+D)$ can reduce error and help readability by the uninitiated. What about nested parentheses, as in $((\iota 1\uparrow\rho M)-1)\phi M$, which can be removed by slight reformulation, as in $(^{-}1+\iota 1\uparrow\rho M)\phi M$? Such extra parentheses might imply a lack of APL sophistication. They entail a bit of extra interpreter expense. Does the maxim "Never teach what you (or someone else) will later have to un-teach" (see [2]) apply here?

Function Definition:

Which of the six forms of defined functions should be introduced first? Niladic functions with no result because they are simplest--or--functions with arguments and explicit results because they are generic and behave not only like APL primitive functions, but also like mathematical functions (cf. [13])? Does starting with niladic functions with no result risk misleading students into imitating that form, or does trying to ameliorate the very real difficulties involved in learning the mechanics of function definition (see [4]) outweigh that risk? Does availability of a friendly full-screen editor make initial exposure to the ∇-editor unnecessary and affect the answers to these questions?

When should the concept of explicit result be explained? Early because it is important--or--later because it seems to present a stumbling block for APL learners (both neophytes and experienced programmers).

Powerful Functions:

When should the especially complex and powerful primitive functions, such as Matrix Divide and Base Value, be introduced? Early, to provide students with convenient tools for getting the job done quickly--or--only after they understand how they work (and perhaps have themselves defined functions to simulate them)? The same question arises even for some simpler primitives, such as Residue (which is handy, for example, for parity checking or extracting the fractional part of a number).

Operators:

When should the presence of operators in the language be formalized? At the first use of +/ because students should know what / really is--or--later, perhaps along with other operators, with all the incumbent terminology and syntax?

Branching:

When should branching be introduced? Show examples of branching first to motivate the use of arrays--or--avoid branching as often and as long as possible? Will delaying branching make students too uneasy--especially those who already know another programming language?

Nested Arrays:

If nested arrays are implemented in the system being used, should the instructor introduce them early, because they actually simplify solving many problems (for example, by avoiding iterative solutions)--or--wait, because APL is complicated enough without them? If, to the contrary, nested arrays are not available in the system at hand, should they nonetheless be mentioned early, because they represent an important general concept which is expected to be included (in some form) in standard APL in the near future?

The final two issues are "local" in that they primarily affect only the particular topic in which they occur.

Scalar Extension:

Present scalar extension before--or--after parallel processing (as a special case)? That is, start with something like 2+4 5 involving only one non-scalar array (because, perhaps, it postpones the syntactical issue that arises when both arguments are vector constants)--or--with something like 2 3+4 5 instead?

Conditional Branch Form:

Which form of conditional branching should be introduced first: using / or ⍴ or $\times_1$ or ↑ and ↓, or a defined cover function *IF*? Should just one be chosen and used consistently (until much later)--or--should several alternate forms be introduced more or less together? Should multiple branching from a single statement (e.g., →(*L*1,*L*2,*L*3)[*I*]) be introduced right away?

Common Mistakes in Teaching APL

Some of the approaches already mentioned can, if pushed to the extreme, become blatant mistakes. Suppose, for example, that comparing APL notation with conventional mathematical notation should turn into insistently denigrating the latter. Then the student may, in effect, be confronted with an uncomfortable choice between, on the one hand, rejecting the familiar ground of traditional mathematics from which he or she might build toward the unfamiliar and, on the other hand, rejecting APL with its "peculiar" new notation altogether. Such an approach is evidently counterproductive.

Here are more practices that are mistakes to a lesser or greater degree, depending on how extreme their application. They are expressed--for irony, of course--as imperatives.

Snow the Student:

Present as much of APL as possible, all at once.

Teach "Logically":

Present all the primitive functions; then, show how to apply them. Present all the dyadic functions first, then the monadic ones; or all the scalar functions, then all the mixed ones. Make sure they are presented in a totally organized and logical order (e.g., alphabetically!). In other words, take the approach of a reference manual.

Here are some mistakes that manifest APL "cultism" [16]. They concern instructor attitudes that can "turn off" students and thereby interfere with learning.

APL Elitism:

Be condescending. Instill a sense of inferiority in 98% of the students by continually implying that one has to be intellectually erudite or at least "mathematically minded" in order to learn APL. Warn, "Only a few of you will become true APLers," as if to imply, "After all, APLers are better people."

Brainwash:

Say, "APL requires a completely new way of thinking." Insist that there is no hope of connecting APL comfortably with what students already know.

"APL is Superior" Syndrome:

Tell the students the sort of thing you sometimes hear at APL conferences: "APL is superior to all other programming languages (for any purpose). APL is <u>the</u> way. And, the acronym really should be <u>TPL</u> (The Programming Language)."

I Love APL:

Make students feel inadequate by comparison with the instructor's enthusiasm for APL. Don an "I ♥ APL" button.

The next three mistakes involve terminological warfare.

Babel:

Don't teach just the APL language itself, but from the start emphasize an entire meta-language for describing the new concepts, new symbols, and new terms. Be sure to insist on careful distinctions, as between "statement" and "expression". Introduce as many unfamiliar and esoteric terms as possible: "token", "identifier", etc.

"Right-to-Left" Rule:

Mislead the students by characterizing APL syntax as the "right-to-left rule". (If students do not understand that "right-to-left" concerns just the order in which functions are executed, then they sometimes think it means that 8÷4 is 0.5. Or, they wonder why, after A←⍳10, the value of A[3] isn't 8, or why scan works from left to right.)

Names of Symbols vs. Names of Functions:

Confuse the names of symbols with the names of functions they denote. Call | "absolute value" or "residue" instead of "stile" (or "vertical bar") when you first introduce it for one of those two functions, so that students are perplexed when they see it again, for the other function. Note that instructors as well as students sometimes mistakenly call ⌈/ "ceiling reduction"--perhaps because ⌈ has two names or perhaps because ⌈/ itself is monadic, or because they don't know the unfamiliar name "up-stile" for ⌈.

Finally, we have one serious strategic blunder and several tactical trip-ups.

Code First, Think Later:

Insist that students define a complete function to solve a problem as soon as possible, rather than encourage them to start by exploring the problem (whether with computer, paper and pencil, or brain alone) and developing a solution piece-by-piece (on the computer, in immediate execution mode).

Ambiguous or Unhelpful Examples:

Unwittingly select examples like

```
      V←5 ¯8 3 16
      ¯2⌽V
```

(which fails to distinguish positive from negative direction of rotation). Or,

```
      ⍋3 1 5 2 4
```

(which has the same result as 3 1 5 2 4 ⍳ 1 2 3 4 5). Or,

```
      V←1 2 3 4
      V[1+2]
3
```

(which could suggest adding up the first two elements of V). Or--to serve as a reminder of order of execution--

```
      1=1∧2=3
```

(which is equivalent to the expression (1=1)∧(2=3) that the naive student may read it as). Or, (if you are especially unlucky),

```
      ⎕←V←5?100
39 52 84 4 6
      ⍒V
3 2 1 5 4
```

(where the last result just happens to be the same as ⍋⍒V, the rank order of elements of V from largest to smallest).

Conceptual Morasses

Our third and last type of "APL teaching bugs" concerns specific aspects of the design of APL that are hard to explain, especially in short order.

Order of Execution:

Why does APL execute from right to left? Why not from left to right (as in NIAL)? Do pragmatic reasons really justify that, or is there, in the final analysis, only an arbitrary choice?

Explicit Result:

What exactly is an explicit result, anyway?

Bracket Notation:

Is [ ] a function? It requires two symbols, which act as delimiters. And what are the semicolons when they appear inside brackets? (See [5] and [15].) Bracket notation seems to be popular for indexing, yet students get into some trouble when using it, e.g., in not seeing why (⍴M)[2] does not need to be surrounded with parentheses when part of a larger expression. Should this anomalous notation, despite its familiarity from mathematics or other programming languages, be displaced by ⌷ or { or other consistent notation (see [1] and [10])? Moreover, what about indexed reassignment A[ ]←...? And is [ ] also an operator (as in Sharp APL)?

Assignment:

What, exactly, is ←? Is it actually a function (as it is designated in APL2 [1, pp. 15, 17]), or is it an "operation" of a type wholly unlike any primitive or defined function?

Branch-to-null:

How does one explain convincingly why →⍳0 causes execution to continue with the next line of a defined function, whereas →0 exits the function? Or why →0 does not cause execution to resume from the top of the function?

Index-of:

In *L⍳R*, why is *L* restricted to be a vector (see [4])? Why is the universe of values there on the left and the "control" on the right, whereas in most mixed functions (e.g., *L/R*) it is the reverse (cf. [13])?

Strand Notation:

Strand notation seems deceptively simple and useful, but does it create serious problems for students' understanding of APL (as well as problems for language designers)? (Cf. [2] and [4].)

Implications for the Future of APL

For the future existence and growth of APL, the most devastating APL teaching bug is, of course, not to teach it at all! Rather, one could offer an excuse: "It's not implemented on our computers...it's different from what we usually do...it would be too difficult for us to change... it would take too much time to learn...we are supposed to train students for what they will encounter in the 'real' world... we can't take chances experimenting with controversial languages or new approaches to computing."

If APL is going to be taught in ways likely to confuse or intimidate students, then that could be a justifiable excuse for not teaching APL at all. For that reason, confronting APL teaching bugs is important for the future of APL. Instructors can then do a better job teaching APL at all levels: from elementary school (where it is hardly taught at all) to business data processing training courses (where APL is growing but still controversial). Students can become sensitive to the issues in the design of APL and appreciate that there were reasons behind design decisions. And developers of enhanced APL systems, or even more advanced programming languages, can make design decisions guided in part by a better understanding of how people teach and learn.

Acknowledgement

This paper is based in part upon work of the second-named author supported by the Local Course Improvement Program of the National Science Foundation under Grant No. SER-8160887.

References

[1] APL2 Language Manual, IBM Corp., 1982.

[2] Paul Berry, "What the User Really Learns", lecture at APL83; reprinted by I. P. Sharp Associates, Toronto, 1983.

[3] Paul Berry, Gottfried Bach, Michel Bouchard, Roland Pesch, Margerete Buch, and Sachiko Berry, "Word, Image and Metaphor to Name APL Concepts in Many Tongues", in APL84 Conference Proceedings, APL Quote Quad 14 (1984), no. 4, pp. 63-69.

[4] Murray Eisenberg and Howard A. Peelle, "APL Learning Bugs", in APL83 Conference Proceedings, APL Quote Quad 13 (1983), no. 3, pp. 11-16.

[5] A. D. Falkoff, "Semicolon-Bracket Notation: A Hidden Resource in APL", in APL82 Conference Proceedings, APL Quote Quad 13 (1982), no. 1, pp. 113-116.

[6] Garth H. Foster, "Motivating Arrays in Teaching APL", in Proceedings of the Fifth International APL Users' Conference (May 15-18, 1973), APL Technical Committee, 1973, pp. 3.1-3.8.

[7] Kenneth E. Iverson, Introducing APL to Teachers, APL Press, 1976.

[8] Kenneth E. Iverson, "The Inductive Method of Introducing APL", in APL Users Meeting (Toronto, October 6, 7, 8, 1980), I. P. Sharp Associates, pp. 211-220; reprinted in A Source Book in APL, APL Press, Palo Alto, 1981, pp. 131-139.

[9] Kenneth E. Iverson, "APL Terminology", report of talk at APL82 International Conference, Heidelberg, reprinted in I. P. Sharp Associates Newsletter, vol. 10, no. 6, 1982.

[10] Kenneth E. Iverson, "Rationalized APL", Sharp Research Reports, no. 1, I. P. Sharp Associates, January, 1983.

[11] Robert C. Metzger, "APL Thinking--Finding Array-Oriented Solutions", in APL81 Conference Proceedings, APL Quote Quad 12 (1981), no. 1, pp. 212-218.

[12] Alan J. Perlis and Spencer Rugaber, "Programming with Idioms in APL", in APL79 Conference Proceedings, APL Quote Quad 9 (1979), no. 4-part 1, pp. 232-235.

[13] Raymond P. Polivka, "The Impact of APL2 on Teaching APL", in APL84 Conference Proceedings, APL Quote Quad 14 (1984), no. 4, pp. 263-269.

[14] Allen J. Rose, "Making APL Palatable", in APL in Practice, Allen J. Rose and Barbara A. Schick, eds., Wiley, New York, 1980, pp. 312-322.

[15] Karl Fritz Ruehr, "A Survey of Extensions to APL", in APL82 Conference Proceedings, APL Quote Quad 13 (1982), no. 1, pp. 277-314.

[16] John R. Searle, "The Future of Programming--Whither APL", in APL84 Conference Proceedings, APL Quote Quad 14 (1984), no. 4, 291-296.

[17] John E. Suwara, "APL Tutorial for General Management", in APL in Practice, Allen J. Rose and Barbara A. Schick, eds., Wiley, New York, 1980, pp. 121-129.

[18] Roy A. Sykes, Jr., "Optimization of APL Code", in APL Users Meeting (Toronto, October 6, 7, 8, 1980), I. P. Sharp Associates, pp. 309-314.