A Survey of "APL Thinking"

Murray Eisenberg
Mathematics and Statistics Department
Lederle Graduate Research Tower
University of Massachusetts
Amherst, MA 01003 USA
e-mail: murray@math.umass.edu
413-545-2859 (office), 413-549-1020 (home)

Howard A. Peelle
School of Education
Furcolo Building
University of Massachusetts
Amherst, MA 01003 USA
413-545-1114 (office), 413-259-1593 (home)

Introduction

During the APL86 conference in Manchester, England, we conducted a questionnaire survey of some aspects of "APL Thinking." We feel that the findings are as useful now as then, so to provide feedback to the APL community, we summarize the results here.

The purpose of the survey was to understand how people think while programming in APL. Our ultimate aim is to improve the learning, teaching, and dissemination of APL.

Respondents were specifically asked to concentrate on the thinking processes involved in APL programming, rather than on applications, language design and enhancements, or environment and implementation issues. (See the sample form on page 8 for a facsimile of the questionnaire.)

The Sample

The 70 respondents were involved with APL about 24 hours per week on the average (standard deviation = 13). Nearly half (32) described themselves as "expert" APL programmers and nearly half (32) as "experienced"; only 2 described themselves as "beginner." Furthermore, 8 were APL application users, 11 were APL administrators, and 22 had "other" involvement with APL—actuary, analyst, consultant, implementer, software designer, application developer, system developer, ex-user, user supporter, R&D manager, author, trainer, etc.

Among the 70 respondents, 48 learned APL on the job, 38 by themselves, 9 in school or college, 8 in a business course, and 4 in other ways, e.g., by "playing." (Note that these responses were not mutually exclusive.) While they mentioned some 15 different books and manuals used to learn APL, over half the respondents used Gilman and Rose, *APL: An Interactive Approach*—and over a third used it alone.

Aspects of APL Thinking

Respondents were asked to rate the importance of several aspects of APL thinking on a scale of 1 to 5, from low to high importance, with ⍳0 meaning "not relevant." (We coded 0 as well as ⍳0 as 0, loosely interpreting these ratings as respondents' intent to extend the scale.) The ratings are shown in Table 1, below, where n is the number who gave some rating, including ⍳0.

Table 1: Summary of Ratings of Aspects of APL Thinking

	n	Mean	St.Dev.
Modular Structure	68	4.2	1.0
Generalizing	66	4.0	1.0
Conceptual Wholes	68	3.9	1.3
Notation / Symbols	67	3.8	1.2
Parallel Array Processing	67	3.6	1.6
Mental Visualizations	67	3.6	1.5
Idioms	67	3.2	1.3
Imagery / Metaphors	64	2.9	1.5
Glass Box Approach	47	2.5	1.6
Identities / Proofs	65	2.0	1.4
Programming Tricks	65	1.6	1.3
One-Liners	65	1.4	1.1

It seems respondents believe that APL thinking is especially facilitated by constructing APL programs modularly and by the natural generality of many APL expressions, as well as by the ease of subordinating details and by the syntax and symbols of the language. By way of contrast, "One-Liners" and "Programming Tricks" are perceived as relatively unimportant.

Seven respondents added miscellaneous other aspects they considered important, including math-to-program compatibility, not having to worry about the operating environment, the ability to use function results as arguments to other functions, the workspace concept, and interactive use of the computer.

Styles in Writing APL Functions

When initially defining an APL function, 58 respondents said they write APL code directly; 25 write words first; 20, pseudo-code; 2, non-APL symbols; and 28, diagrams. (Again, these categories overlap.)

The question *"... how [do] you select and order things you write down...?"* elicited a variety of

responses such as: first try phrases and build up expressions; inside-out, back-and-forth on a line; high-level comments, then low-level comments and simple function definitions; mix words, pseudo-code, and APL code; structure diagram with snippets of APL code; (1) header syntax, (2) comments, (3) pseudocode outlining an algorithm; first a variable name, then bits of notation (often left-to-right); and (1) scratches on paper to describe problem, (2) pseudoprogram with empty blocks, (3) APL code.

Many respondents interpreted the selection/ordering question more broadly than we intended, as if it referred to the entire programming process. Hence some mentioned working top-down and/or bottom-up; others described the familiar *design-code-test-revise* cycle.

How and when do they use the computer when defining an APL function? Some use it from the start, testing phrases or lines of code and trying examples as they go. Others use the computer once they have written some notes, split the task into functions, written descriptions of global data structures, or sketched the "in- compute-out" structure. Still others do not use it until they have solved the problem mentally, or until they have written, debugged, and commented functions on paper.

Preferred Kinds of Function Definitions

When there is a choice, 62 respondents preferred array-oriented function definitions; 8, iterative; and 7, recursive. (Note that some chose more than one kind.)

Several preferred using array methods because it seemed "natural" to them, that is, in accord with the way they think. Some found that array solutions are easier to conceive, understand, visualize, or write, and result in more concise code with fewer errors—especially those due to flow of control. Others mentioned maintainability, efficiency, or elegance.

Additional reasons for preferring array methods included: *"it's easier to explain this way of thinking to others"*; it allows me to retain the core of the problem through several stages and focus on changes needed; *"it helps me think about what's really being done"*; and *"recursion gives me a headache."*

Counter-reasons and caveats included: array-oriented code must often be rewritten iteratively; there's no need to seek an array solution as long as iteration is obvious and adequate; iteration is simple and maintainable; special cases often negate compact array coding; and *"loops [do] aid my understanding of the problem."*

How APL *Helps* Problem-Solving

Besides general rhetoric about how APL helps problem-solving—such as *"[it] usually helps by giving [an] environment into which to put ideas"* and by extending thought—the benefits typically mentioned were: thinking in modules or *"chunks"*; working interactively, thereby seeing results at once and being able to display and manipulate data or to try alternate algorithms and make modifications readily; managing complex problems with just a little code; thinking visually; and having symbols do most of what one wants to do (one respondent referred to *"visions"*—suddenly seeing solutions in terms of APL symbols).

Additional ways that APL helps included: seeing patterns via generalizing solutions; avoiding natural language; using nested arrays to subordinate details, gathering thoughts in APL phrases and subfunctions; and prototyping in APL, then coding in assembler.

One example cited terse descriptions (in direct definition mode) of correlation, etc., as aiding understanding of key notions in statistics. Otherwise, the few specific examples of how APL helps had little or no explicit indication how it helps thinking and problem-solving. For example:

```
(+/ω)÷⍳¯1↑⍴ω
```
("Try that in any other notation!")

```
+/P×(1+R)*T-TI
```
(for compound interest)

```
V×(Z+.×⌽ω×4 2⍴V←1 ¯1)⌹Z←⍉ω-2⊖ω
```
(the intersection of two lines)

How APL *Hinders* Problem-Solving

Ways mentioned that APL hinders problem-solving were: *"array processing capability can sometimes lead to over-generalizing"*; *"[it's] too easy to get sidetracked by interesting aspects of the problem"*; *"[I] occasionally run into a syntax brick wall"*; *"lack of* IF-THEN *sometimes a hindrance"*; *"in the rush to coding I miss out on the analysis I would normally perform in mathematics or the ordering of processes that I would follow without a computer."*

There were also some irresistible reservations: *"[I] don't know that [APL] has an effect, but like to think it does!"*; *"[I'm] not convinced that it particularly helps...it just doesn't get in the way"*; *"I can no longer problem-solve... without APL"*; *"[APL] hurts thinking. No need to think. Just do it until it's right."*

Comparisons with Other Programming Languages

Most of the comparisons of APL with other programming languages suggested that people do think in a different way with APL. Several respondents said that APL allows them to see the big picture more easily by subordinating details. For example: *"I look at the whole problem more in APL or break it into major chunks. In BASIC I mostly think serially, with occasional need for subroutines."*; *"In FORTRAN or BASIC I used to make up flow diagrams with lots of branches and then translate it to code. In APL I make independent blocks of code ... and combine [them] into a simple main structure which describes what is happening."*; *"I think in chunks. Whether these require loops or not is incidental."*; and *"I tend to forget all but the highest level loops now.... To put [it] in another way, COBOL counts hydrogen and oxygen atoms in the vicinity; APL tells you if it's raining."*

Several appreciated APL's freeing the programmer from concern with the machine environment. For example: *"I think about the problem...not about internal system housekeeping functions."*; *"[no need for] reference to what the machine is actually doing"*; debugging in FORTRAN, for example, in contrast to APL, is difficult because of concern with memory locations, 3 to 10 times as much code, and nested loops; *"[In APL] I focus on the whole problem right away. In PL/I, COBOL, and FORTRAN the computer environment, declaration statements, etc., take up much more time."*

> *"...COBOL counts hydrogen and oxygen atoms in the vicinity; APL tells you if it's raining."*

Additional differences included: *"one function = one concept"*; *"[I can] think about the problem because of the wealth of primitives"*; *"the notation allows you to jump from a small part of the solution to the whole"*; *"[APL] avoids English words"*; *"APL supports [my visual thinking] better than any other language."*

One person said, *"I try not [to think differently]. Good design is still essential."* Another insisted he doesn't think differently: *"I just think less."*

Many respondents indicated they first solve problems in APL, then translate to other languages. (And they don't like others because of the need for looping and the absence of interactive capability there.) Some said they constructed programs in other languages in terms of APL, for example, *"I now write Pascal in a very APL-ish way."* One even claimed that someone else could tell from his assembler code that he used to program in APL! And, of course, a number declined comparisons because they don't use other languages.

Additional Comments

Here are a few of the more interesting additional comments: *"I throw away more code than I use. It's part of the thought process."*

"Computer people keep telling me that large arrays are bad, but I like 'em. They're wholesome. I think computers are limiting. What we need are brain coprocessors."

"The key to much creative thinking is to make trials... quickly—even systematically. Thomas Edison understood this well; so did Einstein with his thought experiments. APL supports this."

"[The] most [significant] threat to APL [is] not other languages but the laziness of man. [People] refuse to think and let themselves [be] led by software packages [that] put them in "jail" [with] limited possibilities."

Conclusion

Several respondents were skeptical about the value of the questionnaire. We realize that an instrument such as this is limited in eliciting reliable, useful information. Our questions were often fuzzy—deliberately so for an exploratory investigation of this kind. For that reason, we believe it inappropriate to analyze these results much further. Yet some information of this kind is desirable. Of course, one can read the few papers that purport to describe how APL programmers think—and the larger number that say how one ought to think. Still, there is little hard evidence, aside from the code they produce, about how APL programmers actually do think. It is therefore our hope that this survey and other explorations can lead to more systematic studies of APL thinking. ■

Sample Form: "Survey of APL Thinking" Questionnaire, As Used in Manchester

Purpose: to understand how people think while programming in APL—in order to improve the learning, teaching and dissemination of APL.

Please concentrate on the thinking processes involved in APL programming—not application uses, system/environment/implementation issues, or language design and enhancements. *Thank you.*

1. Describe yourself:

   ☐-APL Programmer (☐-Beginner ☐-Experienced ☐-Expert)
   ☐-APL Application User ☐-APL Administrator
   ☐-Other (specify)

   Average hours/week involved with APL: ____ hours/week

2. How did you learn APL?

   ☐-on the job ☐-by self ☐-in school/college
   ☐-business course ☐-other (specify)

   Book(s) used: _____

3. Rate each of the following aspects of "APL *Thinking*" on a scale of 1 to 5
   (1 = low importance; 5 = high importance; ⍬ = not relevant)

   ☐-Notation/Symbols ☐-Identities/Proofs
   ☐-Parallel Array Processing ☐-Glass Box Approach
   ☐-Generalizing ☐-Mental Visualizations
   ☐-Modular Structure ☐-Imagery/Metaphors
   ☐-One-Liners ☐-Working with Conceptual Wholes (subordinating details)
   ☐-Idioms ☐-Programming Tricks
   ☐-Other (specify)

4. When you initially define an APL function, what do you usually write down? (Check all that apply.)

   ☐-APL code ☐-words ☐-pseudo-code
   ☐-non-APL symbols ☐-diagrams

   Describe further how you select and order things you write down, as well as how and when you use a computer: _____

5. When there is a choice, which kind of function do you prefer to define?

   ☐-iterative ☐-recursive ☐-array-oriented

   Why? _____

6. How does APL help or hinder your thinking/problem-solving? _____

   Please give an example: _____

7. When you define a function in APL, do you think in a special way or a different way—compared with another programming language? _____

   _____

8. Additional Comments: _____

   _____

   _____