SOME FAILINGS OF APL

Paul McJones
Computer Science Department
University of California, Berkeley

2 March 1973

1. Introduction

APL\360 is currently one of the most fashionable and widely used interactive programming languages. An examination of the language would seem to indicate that APL succeeds not on its own merits, but for lack of suitable competition. In the following sections we catalog some of the most serious problems. (Knowledge of APL is assumed.)

2. Control Structure

2.1. APL provides no conditional or repetitive constructs, forcing the programmer to rely on the tricky branch operator ("monadic right arrow"). In fact, most ways of writing a conditional transfer in APL have the side effect of creating an array!

2.2. The branch operator takes a line number as an operand, so inserting a new line into an APL function is liable to have some unexpected effects. A label in APL is just a symbolic name for a line number and is meaningful only in the function body containing its definition; passing a label as an argument to another function is allowed but not likely to be useful.

2.3. Much of the conciseness of the language is due to automatic extension of the scalar operators to arrays on an elementwise basis, and to the "composite functions" reduction, inner product, and outer product. Unfortunately these mechanisms are not applicable to user-defined functions.

3. Naming Structure

3.1. Free variables of APL functions are bound dynamically ("nesting by encounter" in APL parlance). This hinders program modularity, since to call a function one must be sure no global variable referenced by that function coincides with a local variable of the caller.

3.2. There is no "own storage" (storage private to a function and surviving from call to call, like local variables in FORTRAN or own variables in ALGOL 60).

3.3. The limit of two arguments to a user-defined function is very restrictive, and it is infuriating that some of the built-in functions cheat (e.g. A,[2]B or C[J1; J2; J3; J4; J5]).

3.4. Since there is no form of call-by-reference, a function is constrained to return only a single value.

4. Syntax

4.1. Right-to-left association of operators is unnatural; we read from left to right.

4.2. Some of the operators in APL are far from mnemonic. Examples:

1○X means SIN(X)
⌶25 means DATE()

5. Data Types and Structures

5.1. APL provides a very complete and elegant set of operations on multidimensional, homogeneous arrays of numbers and characters. But this is not enough if the language is to be considered "general purpose". We need to be able to construct lists, trees, and/or arbitrary directed graphs.

5.2. The only way for an APL program to decide whether a scalar is a character or a number is to compare it with every possible character value; the object is a number if and only if all the comparisons result in inequality.

6. Miscellaneous

6.1. The lack of type declarations in the language reduces beneficial redundancy and makes efficient implementation difficult.

6.2. No general input/output facilities are provided in the language.

6.3. The coercions (one-element array→scalar, etc.) are clumsy.

6.4. It is silly to decree that $0 \div 0$ is 1.

6.5. The index origin should not be a global variable.

6.6. There should be character constants for "carrier return", "backspace", etc.

6.7. The dependence on the IBM Selectric terminal (1050, 2741) and special typeball is pretentious. Teletypes are more widely available, and allow character-by-character interaction because of the full-duplex communications discipline.

6.8. There should be context edi ting capabilities in the editor.