DANIEL BROCKLEBANK

# APL – THE LANGUAGE

Computer programming languages, once the specialized tools of a few technically trained people, are now fundamental to the education and activities of millions of people in many professions, trades, and arts. The most widely known programming languages (Basic, Fortran, Pascal, etc.) have a strong commonality of concepts and symbols; as a collection, they determine our society's general understanding of what programming languages are like. There are, however, several languages of great interest and quality that are strikingly different. One such language, which shares its acronym with the Applied Physics Laboratory, is APL (A Programming Language).

## A SHORT HISTORY OF APL

Over 20 years ago, Kenneth E. Iverson published a text with the rather unprepossessing title, *A Programming Language.*[1] Dr. Iverson was of the opinion that neither conventional mathematical notations nor the emerging Fortran-like programming languages were conducive to the fluent expression, publication, and discussion of algorithms—the many alternative ideas and techniques for carrying out computation. His text presented a solution to this notation dilemma: a new formal language for writing clear, concise computer programs. To demonstrate the value of such a language, he used it to present a variety of interesting, nontrivial algorithms (such as those for sorting lists of numbers or for "parsing" and analyzing statements written in other programming languages).

The APL language, which is used extensively throughout the world today,[2] looks only remotely like the notation presented in Iverson's seminal text, although it does retain many of the fundamental concepts. The original notation could not be entered into a computer to be acted upon; it was confined to blackboards, pencil and paper, and the published literature. Only through manual translation into Fortran or some other language could Iverson's notation actually "come to life" in the real world of computing.

Through collaboration with others, most notably Adin Falkoff, Dr. Iverson designed another APL[3] that could be used in a practical way on time-shared interactive computer systems. By 1970, the APL language service was in use by thousands of people, including the research and design staff of IBM (whose various System 360 computers hosted the APL services), the staff of the NASA Goddard Space Flight Center, and the financial planners of a number of major corporations. The elegant yet practical APL won the praise of many of its users, just as it evoked stern criticism from the established data processing and academic computing communities.

APL received only a modicum of software engineering support from the principal computer vendors as it struggled through the 1970s. Its international contingent of enthusiasts was continuously hampered by inefficient machine use, poor availability of suitable terminal hardware, and, as always, strong resistance to a highly unconventional language.

At the Applied Physics Laboratory, the APL language and its practical use have been ongoing concerns of the F. T. McClure Computing Center, whose staff has long been convinced of its value. Addressing the inefficiency problems, the Computing Center developed special APL systems software for the IBM 360/91 computer. That software provided a dramatic improvement in the speed of APL language programs, achieved largely by the thorough use of the high-speed "vector" processing capabilities of the machine. APL at APL was the fastest known implementation of the language anywhere in the free world.

When the Computing Center was upgraded to the IBM 3033 processors, the speed of the 360/91 was partially sacrificed, but a new opportunity arose: to build an APL system to handle very large applications (using virtual memory technology and bulk data handling techniques). Since its introduction in 1978, this system[4] has been used regularly by hundreds of Laboratory staff members for many purposes.

Now, in 1984, APL is available for nearly every computer in the marketplace. It is also a featured service of numerous commercial time-sharing companies, including one that specializes in up-to-date (hourly) international financial databases. Most significantly, APL is now available for personal computers,[5] thus bringing it within reach of everyone. The inexorable trend toward more powerful, less expensive computers is bringing APL out into the world it was designed for: the world of learning, intellectual satisfaction, human productivity, timely right answers, and smooth integration of computing and education.

## A BRIEF LOOK AT THE APL LANGUAGE

APL is international in character. It uses symbols instead of words borrowed from natural language (although some of the symbols are indeed derived from

the Greek alphabet). In this sense, APL is comparable to traditional notations of mathematics. The symbols require a special keyboard (Fig. 1) and display device for interacting with the computer. (Today, numerous manufacturers offer APL-equipped terminals and personal computers.)

Much of APL is familiar and reminds us of our fundamental learnings in arithmetic and algebra. In fact, APL can be used very effectively as a "laboratory" for learning algebra.[6] When being taught French, American students often spend hours speaking the language in a laboratory. When learning mathematics, however, there is usually no such laboratory in which to experience and experiment with the concepts. When computers are introduced, ostensibly for that purpose, the languages used (almost always Basic) have but a weak relationship to the substance of the material being taught. APL offers an attractive, substantive alternative.

Consider the familiarity of the following APL symbols:

| | |
|---|---|
| + | Addition |
| − | Subtraction |
| × | Multiplication |
| ÷ | Division |
| ! | Factorial |
| < ≤ = ≥ > ≠ | Comparisons |
| ~ | Logical negation |
| ∧ | Logical AND (intersection) |
| ∨ | Logical OR (union) |
| ∈ | Set membership |
| \| | Absolute value |

The Fortran-like languages use unfamiliar symbols (such as * for multiply, / for divide, and .LT. for less than). Instead of restricting his notation to the symbols available on the keypunch machines of the 1960s, Iverson chose symbols that were and are used to teach arithmetic and that appear today on nearly every handheld calculator.

APL is rooted in the mathematical concept of function, and most of its capabilities are implemented as functions. The language is rich in interesting identities involving many combinations of its intrinsic functions, and it is conducive to formal symbolic manipulation and proofs.

The familiar symbols in the previous list all imply the performance of the corresponding familiar functions. Other functions, familiar and otherwise, are denoted by well-chosen, easily learned symbols. The following are some examples:

| | |
|---|---|
| ⌈ ⌊ | Maximum and minimum |
| * | Power |
| ⍟ | Logarithm |
| ⌹ | Matrix inverse, matrix division |
| ○ | Circular, Pythagorean, trigonometric |
| ⍋ ⍒ | Ascending and descending "sort" |
| ⍉ | Transpose about the diagonal |
| ⌽ | Reverse |

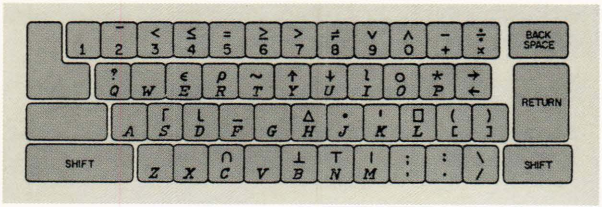


**Figure 1**—Typical APL keyboard.

APL functions are defined upon rectangular arrays of data, not just upon individual scalar values. A rectangular array contains data arranged along zero or more axes or dimensions, not necessarily all having the same length. A simple list of values is a rectangular array with only one dimension (a vector); a square matrix is a rectangular array with two equal dimensions. A typical financial report often contains tables of numbers that can easily be recognized as rectangular arrays. This characteristic results in what may be the single most important contribution of the language, particularly in light of evolving supercomputer techniques: those who "think in APL"[7] naturally think in terms of vector and array processing.

APL achieves generality through the simplicity and uniformity of its rules (as contrasted with achieving generality through exhaustive enumeration of features, which is typical of computer paraphernalia). As a strong source of generality, APL uses the concept of an operator, permitting one to build functions that are variants of other functions. The / operator, for example, can be used to create a summation function out of the ordinary addition function. The resultant summation function looks like +/ and is read as "plus over." We consider this to be highly general because it is simple and has many latent possibilities for which traditional mathematics (and Fortran) has inconsistent ad hoc notations. Consider these variations:

| | | |
|---|---|---|
| +/ | "plus over" | Summation |
| ×/ | "times over" | Product |
| ⌈/ | "max over" | Maximum value |
| ⌊/ | "min over" | Minimum value |
| ∧/ | "and over" | True for all |
| ∨/ | "or over" | True for any |

When applied to arrays, these reduction functions may be applied along any axis (i.e., down the columns or across the rows).

APL promotes human productivity by thoughtfully automating most irrelevant aspects of the computer's inner workings and by nearly eliminating the need for declarative statements (DIMENSION, DECLARE, etc.). Unless facing extreme limits, APL users do not need to consider how numbers are represented internally (as binary, decimal, hexadecimal, Boolean, byte, word, floating-point, etc.), nor do they need to announce in advance the sizes, shapes, or types of arrays. Furthermore, most APL implementations provide for convenient interactive testing and

debugging capabilities, entirely in APL terms (no core dumps or other machine-related details).

## APL AND EXPRESSIVE DIRECTNESS

For a broad class of problems, APL provides a very high dgreee of expressive directness. By this we mean that the language is conducive to the fluent, rapid, and clear expression of many algorithms, ideas, and definitions. To illustrate, let us take a simple concept and evolve an APL program to represent it. The concept we have chosen is that of a palindrome: a word, verse, or sentence that reads the same backward or forward. This definition, like many stated in natural language, elides many of the detailed points that one would have to consider in constructing a program to embody the concept. In the sequence of steps below, we start with the literal definition and then account for an unstated but important assumption.

Our goal is to construct an APL function named P (for palindrome), which takes a list of characters of arbitrary length as its right argument (input) and determines if that list represents a palindrome. It should return the value 1 (true) if the input is a palindrome; otherwise it should return the value 0 (false). Our first solution is P:∧/R= ⌽ R. This means "define a function P whose result is computed by: Is it true for all elements that R (the right argument) is equal to the reverse of R." This function is easily derived from the English definition, and it directly and succinctly expresses the concept of palindrome.

After the definition[8] is typed into an APL language system (thus making it a complete, executable computer program), it can be exercised immediately by typing the function name followed by a sequence of characters to be tested. The computer prints the answer immediately at the left margin:

```
      P 'APL'
0
      P 'GLENELG'[9]
1
      P 'ABLE WAS I ERE I SAW ELBA'
1
      P 'ABLE WAS I ERE I SAW ELBA.'
0
      P 'A MAN, A PLAN, A CANAL,
      PANAMA.'
0
```

Usually, the last two examples would be accepted as palindromes – neither punctuation marks nor blanks should be considered detrimental. To account for this previously unstated rule, let us develop another APL function, named A, which takes a list of characters and returns that list with all the nonletters removed:

```
A: (R∈'ABCDEFGHIJKLMNOPQRSTUVWXYZ')/R
```

This means "define a function A whose result is computed by: For every R that is a member of the set 'ABC. . .XYZ', keep that R." In this context, the / symbol denotes an APL function named "compress," which uses a list of 1's and 0's (a Boolean vector) as its left argument to indicate which elements of its right argument are to be kept (1) or deleted (0).

Now let us test function A, first by itself and then in combination with function P:

```
      A 'A MAN, A PLAN, A CANAL,
      PANAMA.'
AMANAPLANACANALPANAMA
      P A 'A MAN, A PLAN, A CANAL,
      PANAMA.'
1
      P A 'HE LIVED AS A DEVIL, EH?'
1
```

Rather than always having to apply function A before applying function P (as in the above tests), we could merge them by incorporating the invocation of A as part of a new definition for P:

```
P:∧/(A R)= ⌽ (A R)
```

Thus, when tested:

```
      P 'A MAN, A PLAN, A CANAL,
      PANAMA.'
1
      P 'MUCH MADNESS IS DIVINEST SENSE
      TO THE DISCERNING EYE, MUCH
      SENSE THE STARKEST MADNESS'
0
```

Alas, we do not know of any clever alterations to the function P that would enable it to detect the slant allusion to palindromic style in these words from Emily Dickinson.

An important observation about the first palindrome function, P:∧/R= ⌽ R, is that it is highly general. It will work correctly for arguments that are bit strings (Boolean data, digital signals, etc.):

```
      P 1 0 1 0 0 1 0 1
1
      P 1 1 1 0 0 0
0
```

It will work correctly for arguments that are real numbers (integral or otherwise):

```
      P 1 3 5 7 5 3 1
1
      P 31 28 31 30 31
0
      P .1 .4 99 .4 .1
1
```

It will also work correctly for any rectangular array of data, operating across the rows (unfortunately constrained to be of uniform length):

```
      P AAAXBBB[10]
        GLENELG
        5966955
        1100011
1 1 0 1
```

## NEW DIRECTIONS FOR APL

Both the formal definition of APL and its actual practice are being stretched and challenged in many ways, and they are receiving unprecedented levels of attention and support. Some of the major directions[11] include

1. Extension of the language to rectangular arrays that may contain other arrays, nested recursively to achieve arbitrarily complex data structures (including structures of considerable interest to researchers in artificial intelligence and advanced database design);
2. Extension of the language to the complex number system (instead of just the real number system);
3. Introduction of new functions, new operators, and greater flexibility into the language;[12]
4. Implementations that (a) exploit advances in processor performance and memory capacities, (b) use sophisticated formal techniques for understanding optimal execution of APL programs, and (c) are available on more computers, both large and small;
5. Appearance of APL in homes and schools;
6. Extension of the language for use in very large software systems for which current APL modularization techniques are not adequate;
7. Further adoption of APL as a fundamental tool for research, design, database systems, teaching, and corporate management.

While the Basics and Fortrans of the world continue as popular models of what programming languages are, the unconventional APL will persist as a model of how much better programming languages could be and as an increasingly capable tool—the first choice of many.

REFERENCES and NOTES

[1] K. E. Iverson, *A Programming Language*, John Wiley and Sons, Inc., New York (1962).
[2] Emerging ISO Standard TC97/SC5 WG6 N28 (1984).
[3] *APL Language*, IBM Corp. GC26-3847 (1978).
[4] D. Brocklebank and W. T. Renich, "The APL/MVS Interactive Computing System," in *APL Developments in Science and Technology, Fiscal Year 1980*, JHU/APL DST-8.
[5] They include such popular machines as the IBM PC, the TRS-80, and the family of machines based upon the Motorola 68000 microprocessor. APL language products are available from many sources in addition to the machine vendors.
[6] K. E. Iverson, *ALGEBRA, An Algorithmic Treatment*, Addison-Wesley Publishing Co., Menlo Park, Calif. (1972).
[7] K. E. Iverson, "Notation as a Tool of Thought," ACM Turing Award Lecture, *Commun. ACM* **23**, 444-465 (1980).
[8] The exact technique for entering the function definition varies among systems.
[9] Glenelg is a small town near APL.
[10] Note that this is not how two-dimensional arrays are entered; the notation shown is for convenience of exposition only.
[11] Most new APL language extensions will soon be operational features of F. T. McClure Center software systems.
[12] See, for example, *APL2 Programming: Language Reference,* IBM Corp. SH20-9227 (1984).