APL

PROGRAMMING LANGUAGE

Rajat Acharya(904567457)

Nitin E.Pereira (904567481)

MIT-01, SPJIMR

INDEX

INTRODUCTION

APL (for **A Programming Language**, or sometimes **Array Processing Language**) is an array programming language invented in 1962 by Kenneth E. Iverson while at Harvard University. Iverson received the Turing Award in 1979 for his work. As with all programming languages that have had several decades of continual use, APL has changed significantly from the original language described by Iverson in his book *A Programming Language* in 1963. One thing that has remained constant is the interpretive nature of the APL programming environment, a feature much appreciated by its users. Conversely, its initial lack of support for both structured and modular programming has been solved by all the modern APL incarnations. One much criticized aspect still remains, though the use of a special character set .

APL is one of the most powerful, consistent and concise computer programming languages ever devised.

<u>The main features of APL are as follows:</u>

- APL is an interactive language returning answers immediately.
- APL is easy to learn and use; if you make a mistake, it is easy to fix.
- APL code takes less time to write and debug than any other high level language. This makes it the most efficient and cost effective language from the combined personnel and hardware point of view.

Because of these features, APL is the ideal language for all types of people:

- Those people who want to learn a programming language
- Those who need a computer which works for them, not they for the computer
- Managers interested in being managers, not programmers
- Programmers who want to program

APL is a language for describing procedures in the processing of information. It can be used to describe mathematical procedures having nothing to do with computers, or to describe (to a human being) the way a computer works. It is more of a mathematical notation that can be interpreted by a computer than a traditional computer language, and handles data in groups (arrays) rather than just in small pieces. Most commonly, however, at least at this time, it is used for programming in the ordinary sense of directing a computer to process numeric or alphabetic data.

APL is a user-oriented notation, particularly well suited for communication from people to other people or to computers. The notation consists of a set of symbols (letters, numbers, punctuation, algebra, and special shapes), with a very simple set of rules (syntax) for putting them together to describe the processing of data. The data can be either numeric or literal (which includes words and text handling).

In fact there are about a hundred different "primitive (i.e. fundamental to APL) operations which can be performed. This can be compared to having a calculator with over 100 different function keys. Following the old Chinese proverb that a picture is worth a thousand words, the *APL symbol set is the equivalent of many words* in describing algorithms or procedures. Often one or a few APL symbols (function) can have the same result as several lines of code in another computer language or of several paragraphs in English. An example is shown later in this article.

APL allows the computer to be used in the same way as many hand-held calculators. Any operation which can be performed on a small calculator can be performed just as easily on a computer using APL. The answer is returned immediately, just as with the ordinary calculator. With just a few keystrokes you can easily: perform addition, subtraction, multiplication and division

- take reciprocals, square roots or even factorials
- sort up (or down) a set of numbers
- raise any number to any power
- take any root of a number
- calculate logarithms and exponentials
- examine separately the integer and decimal parts of a number
- convert number bases
- take absolute values of numbers
- perform trigonometric functions
- generate random numbers

- rearrange arrays of numbers into different size and shape arrays, i.e. vectors, matrices, and multi-dimensional "cubes".

How Does APL Compare with other Languages

APL is a very concise language. It is normal for APL programs to have one-fifth as many instructions as programs written in other languages, and one-tenth to one-twentieth the number of instruction lines. Among the reasons for this is that APL is a symbolic language, and single symbols often replace multiple lines of code. Here is a simple comparison of a numerical sort in BASIC, FORTRAN, and APL:

BASIC	FORTRAN	APL
10 DIM X(100), (100)	DIMENSION X(100),Y(100)	X [⍋X]
20 READ N	READ(5,10) N,(X(I),I=1,N)	
30 FOR I =1 TO N	10 FORMAT (15/(F10.2))	
40 READ X(I)	DO 20 I=1,N	
50 NEXT I	A=X(I)	
60 FOR I=1 TO N	L=1	
70 LET A=X(I)	DO 15 J=1,N	
80 LET L=1	IF(A-X(J))15,15,12	
90 FOR J=1 TO 10	12 A=X(j)	
100 IF A<X(J) THEN 130	L=J	
110 LET A=X(J)	15 CONTINUE	
120 LETL=J	Y(I)=A	
130 NEXT J	20 X(L)=100000.	
140 LET Y(I)=A	WRITE(6,30) (Y(I),I=1,N)	
150 LET X(L)=1000000.	30 FORMAT(E15.2)	
160 PRINT Y(I)	END	

170 NEXT I 180 DATA ...... XXX END		

In fact, the APL language has about 100 primitive symbolic functions. The ability to pre-store independently existing variables eliminates the need for creating them within a program. Also, APL permits many statements to be combined into a single line of code. Since fewer lines are coded, programming time decreases. More important, the modularity of APL encourages the generation of many small interdependent programs. As a result debugging time is decreased by approximately 50%. Since half of all programming time is devoted to debugging, the time saved is significant.

APL is very different from Fortran, C, Forth, or Lisp. The symbols, syntax, and natural algorithms all take some getting used to. The first hurdle is the keyboard layout. Unshifted characters give upper-case letters, shifted characters and numbers give the APL symbols. You'll need to keep the keyboard map handy for reference at first, though most of the symbols are easy to find once you know the mnemonics (i.e. È (iota) is shift-I, ® (rho) is shift-R). If you can't stand it, there is an ASCII mode that substitutes keywords and ASCII punctuation for the symbols, but if you plan to read standard APL books, or trade programs with other APL users, it's better to do it right. The real notation is also more compact and easily read - the symbols stand out.

The next difficulty is the syntax. There is no hierarchy of operators - all APL expressions are evaluated from right to left. If you need to change the order of operation, you must use parentheses. For example, the expression 5-6-2 is equivalent to 5-(6-2) = 1 in APL, not (5-6)-2 = -3 as in most languages. And 5x6-2 equals 20, not 28.

People who do not know APL may feel that it is not practical to learn the approximately 35 special shapes in the APL "vocabulary" which represent certain powerful primitive function. But only a few need be learned for any one application. And the ease of learning them is evidenced by the thousands of people from Junior High School student to non-computer professionals who have learned to use APL quickly and efficiently. People who say that it is easier to program in COBOL than in APL omit to say that COBOL (and other computer languages) is complicated by computer-related restrictions and words reserved for special use, which in the case of COBOL amounts to a list of about 500 such words.

Some people believe that the ultimate goal is to program computers in English. However, experience to date shows that symbols may be better than English for specific cases of communication. For example, consider the programming and documentation of music. Can you imagine trying to write the score for Jingle Bells in words? "Three E's, the third one held twice as long as either of the other two; repeat; etc...!" English (or any other language) has evolved over centuries as the best form of oral and written communication between people. But there are better ways for communicating between people and computers and the APL symbol set seems to be the most productive system yet devised.

CONTROL STRUCTURE

Over a very wide set of problem domains (math, science, engineering, computer design, robotics, data visualization, actuarial science, traditional DP, etc.) APL is an extremely powerful, expressive and concise programming language, typically set in an interactive environment. It was originally created as a way to describe computers, by expressing mathematical notation in a rigorous way that could be interpreted by a computer. It is easy to learn but APL programs can take some time to understand. Unlike traditional structured programming languages, code in APL is typically structured as chains of monadic or dyadic functions and operators acting on arrays. Because APL has so many nonstandard *primitives* (functions, operators, or features built into the language and indicated by a single symbol or a combination of a few symbols), APL does not have function or operator precedence. The original APL did not have control structures (loops, if-then-else), but the array operations it included could simulate structured programming constructs. For example, the iota function (which yields an array from 1 to N) can simulate for-loop iteration. More recent implementations of APL generally include explicit control structures, so that data structure and control-flow structure can be more clearly separated.

The APL environment is called a ***workspace*.** In a workspace the user can define programs and data, i.e. the data values exist also outside the programs, and the user can manipulate the data without the necessity to define a program, for example:

$N \leftarrow 4\ 5\ 6\ 7$

Assign the vector values 4 5 6 7 to N.

$N + 4$

Add 4 to all values (giving 8 9 10 11) and Print them (absence of the assignment arrow means "show")

$+/N$

Print the sum of N, i.e. 22

The user can save the workspace with all values and programs. In any case, the programs are not compiled but interpreted.

APL is well-known for its use of a set of non-ASCII symbols that are an extension of traditional arithmetic and algebraic notation. These cryptic symbols, some have joked, make it possible to construct an entire air traffic control system in two lines of code. Indeed, in some versions of APL, it is theoretically possible to express any computable function in one expression, that is in one line of code. You can use the other line for I/O, or constructing a GUI. Because of its condensed nature and non-standard characters, APL has sometimes been termed a **"write-only language"**, and reading an APL program can at first feel like decoding an alien tongue. Because of the unusual character set, many programmers used special APL keyboards in the production of APL code. Nowadays there are various ways to write APL code using only ASCII characters. Indeed most if not all modern implementations use the standard keyboard, displaying APL symbols by use of a particular font.

Advocates of APL claim that the examples of **"write-only"** code are almost invariably examples of poor programming practice or novice mistakes, which can occur in any language. APL has perhaps had an unusually high percentage of users who are subject-matter experts who know some APL, rather than professional programmers who know something about a subject. Iverson designed a successor to APL called J which uses ASCII "natively". So far there is only a single source of J implementations: **http://www.jsoftware.com/** Other programming languages offer functionality similar to APL. A+ is an open source programming language with many commands identical to APL.

Here's a program that finds all prime numbers from 1 to R:

$$(\sim R \in R \circ . \times R)/R \leftarrow 1 \downarrow \iota R$$

Here's how to read it, from right to left:

1. ι creates a vector containing integers from 1 to R (if R = 6 at the beginning of the program, ιR is 1 2 3 4 5 6)
2. Drop first element of this vector ($\downarrow$ function), i.e. 1. So $1 \downarrow \iota R$ is 2 3 4 5 6
3. Set R to the vector ($\leftarrow$, assignment primitive)
4. Generate outer product of R multiplied by R, i.e. a matrix which is the *multiplication table* of R by R ($\circ . \times$ function)
5. Build a vector the same length as R with 1 in each place where the corresponding number in R is in the outer product matrix ($\in$, set inclusion function), i.e. 0 0 1 0 1
6. Logically negate the values in the vector (change zeros to ones and ones to zeros) ($\sim$, negation function), i.e. 1 1 0 1 0

7. Select the items in R for which the corresponding element is 1 (/ function), i.e. 2 3 5

APL program to find all prime numbers <= specified integer

```
PRIMES : (~R∊R∘.×R)/R←1↓⍳R
```

Notes: APL processes expressions from right to left, except when the order of evaluation is changed by using parentheses.

1. Select the highest number in desired range, and assign to *R* (e.g., 6; R equals scalar value: 6)
2. Generate vector of all integers 1 thru R (1, 2, 3, 4, 5, 6). [*iota* function]
3. Drop first element from the vector (drop: 1, retain: 2, 3, 4, 5, 6). [*down arrow* function]
4. Set R to the vector (R equals: 2, 3, 4, 5, 6). [*left arrow* function]
5. Generate "outer product" of R multiplied by R [i.e., the multiplication table for R by R]. [*R circle dot* × *R* function] (*See right*)
6. Build a vector the same length as R, but with 1 in each place where the corresponding number in R is in the table [true=1], and 0 where the corresponding number is not [false=0] (0, 0, 1, 0, 1). [*set inclusion* function]
7. Logically negate the values in the vector (change zeros to ones and ones to zeros) (1, 1, 0, 1, 0). [*negation* function]
8. Select the items in R, for which the corresponding element in vector from #7 is one (2, 3, 5). [*slash* function, evaluated now due to parentheses] -- *Done!*

DATA STRUCTURE

The structure of the data is classified into scalars, vectors, and larger arrays. An argument to any function can be a scalar, vector, or array, with a few exceptions. It is irrelevant (again with some exceptions) whether the data is a character string, an integer, or a set of Boolean values. A scalar is a single number or character, without dimension or length. A vector is a set of data that forms a one dimensional array, a matrix has two dimensions. Arrays can have up to 63 dimensions in PortaAPL, though you'll never use that many - if you had two bits per dimension, you'd need a billion Gigabytes to hold the array. All elements of an array must have the same data type, so if you put a single floating point number in a large array, every element will be floating point. Type conversions are taken care of automatically and transparently at the interpreter's discretion, though there are ways to force a conversion to a particular type. On the Mac, Boolean values are stored as bits, characters as bytes, integers as 4 byte words, and floating point as 8 byte long words. A character string is a vector of characters; a character matrix is a set of strings, one per row.

In addition to dividing functions into classes according to how many arguments they take, they can be divided into three groups that depend on what their effect is. Scalar functions change the data in an array, but not the shape or size of the array. Restructuring functions change the shape or dimension of an array, but not the data values. Mixed functions change both the data and the shape, and will have to wait for a later article. A scalar function can operate on arrays, not just scalars, but they operate element by element. For example, you can add two vectors of the same length:

```
 2 4 6 11 8  +  7 43 1 6 31

9 47 7 17 39
```

Iota (È) and rho (®) are two of the most important reshaping functions. Iota in it's monadic form creates a vector of integers in ascending order from a scalar: È6 gives the vector 1 2 3 4 5 6. Monadic rho gives the length of each dimension of an array: ® 1 3 5 7 is 4. Dyadic rho creates an array with the dimensions given by the left argument, filled with the values given in the right argument. The right argument is repeated as many times as necessary to fill the array:

```
 2 4 ® 1 2 3

1 2 3 1
```

```
2 3 1 2
```

Some of the scalar functions will look deceptively familiar to programmers. + and - do what you expect, but * means the exponential or power, not multiply, and / is not divide, but something more complicated - it can be either a restructuring function (compress) or an operator (reduction). When used as the reduction operator, f/ inserts the function f between each element of a vector:

```
+/ 1 2 3 4
10

 ª/ 1 0 0 1
1

 -/ 1 2 3 4
™2
```

CALCULATION

APL was unique in the apparent speed with which it could perform complex matrix operations. For example, a very large matrix multiplication would appear to take only a few seconds on a machine which was much less powerful than those today. There were some technical and other economic reasons for this advantage:

- Commercial interpreters delivered highly-tuned linear algebra library routines.
- Very low interpretive overhead was incurred per-array—not per-element.
- APL response time compared favorably to the runtimes of early optimizing compilers.

A widely cited paper "The APL Machine" perpetuated the myth that APL made pervasive use of lazy evaluation where calculations would not actually be performed until the results were needed and then only those calculations strictly required. Although this technique was used by just a few implementations, it embodies the language's best survival mechanism: not specifying the order of scalar operations. Even as eventually standardized by X3J10, APL is so highly data-parallel, it gives language implementers immense freedom to schedule operations as efficiently as possible. As computer innovations such as cache memory, and SIMD execution became commercially available, APL programs ported with little extra effort spent re-optimizing low level details.

TERMINOLOGY

APL makes a clear distinction between *functions* and *operators*. Functions take values (variables or constants or expressions) as arguments, and return values as results. Operators take functions as arguments, and return related, derived, functions as results. For example the "sum" function is derived by applying the "reduction" operator to the "addition" function. Applying the same reduction operator to the "ceiling" function (which returns the larger of two values) creates a derived "maximum" function, which returns the largest of a group (vector) of values. In the J language, Iverson substituted the terms 'verb' and 'adverb' for 'function' and 'operator'.

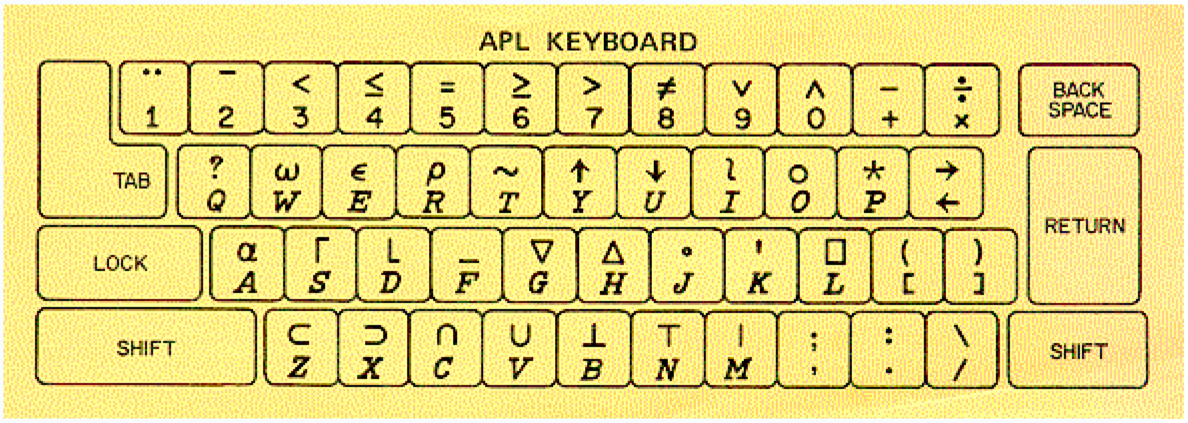
APL also identifies those features built into the language, and represented by a symbol, or a fixed combination of symbols, as *primitives*. Most primitives are either functions or operators. Coding APL is largely a process of writing non-primitive functions and (in some dialects of APL) operators. However a few primitives are considered to be neither functions nor operators, most noticeably assignment.

CHARACTER SET

APL has always been criticized for its choice of a unique, non-standard character set. The fact that those who learn it usually become ardent adherents shows that there is some weight behind Iverson's idea that the notation used does make a difference. In the beginning, there were few terminal devices which could reproduce the APL character set — the most popular ones employing the IBM Selectric print mechanism along with a custom type element. With the popularization of the Unicode standard, which contains an

APL character set, the eternal problem of obtaining the required particular fonts seems poised to go away.

APL SYMBOLS AND KEYBOARD LAYOUT



Much of the syntax consists of Greek letters and specially invented characters. Note the mnemonics associating an APL character with a letter: *question mark* on *Q*, *power* on *P*, *rho* on *R*, *base value* on *B*, *eNcode* on *N*, *modulus* on *M* and so on. This makes it easier to type APL on a non-APL keyboard providing you have visual feedback on your screen. All APL symbols are present in Unicode (In computing, **Unicode** provides an international standard which has the goal of providing the means to encode the text of every document people want to store on computers. This includes all scripts in active use today, many scripts known only by scholars, and symbols which do not strictly represent scripts, like mathematical, linguistic and APL symbols).

Who Uses APL?

APL is widely used in the United States, Canada and Western Europe. It is used by at least half the Fortune 500 industrial concerns. It is used in single departments in very small companies. It is provided as a centralized service in some companies and purchased from outside sources in others. APL applications cover the gamut of computer uses: business, scientific, accounting, engineering, education, actuarial, text processing, pension valuations, simulation, graphics, and many more.

Computers are programmed in APL by financial analysts, actuaries, business managers, clerks, administrative assistants, professional programmers, etc. The common bond among all users is that at last the computer has been made directly available to the end user, to enhance his/her own creativity without being bogged down with a list of computer details and restrictions.

APL encourages direct end-user involvement. In addition to being extremely powerful, APL is very easy to learn. The rules of the language allow non-programmers to create short working programs almost instantly, thereby providing immediate positive reinforcement. Together with the ability to create programs with English language names, this allows the end-user to become directly involved and reduces or eliminates the communication problem between the end-user and the professional programmer. In turn, this shortens turnaround and yields a more accurate representation of the user's needs.

ADVANTAGES

- APL is an interactive language returning answers immediately.
- APL is easy to learn and use; if you make a mistake, it is easy to fix.
- APL code takes less time to write and debug than any other high level language. This makes it the most efficient and cost effective language from the combined personnel and hardware point of view.

DISADVANTAGES

Since it is an interpreter, APL runs slower than most compilers. But it runs much faster than other interpreters such as Basic. The programs are much shorter, so APL spends most of its time working on the problem, not interpreting the commands. You also get the advantages of an interpreter - interactive writing and debugging. Functions are defined by name as in Forth or Logo, and you can define local variables, so it's easy to define new functions as you go along.

APL is extremely powerful. It's recursive and self-modifying, with dynamic allocation of data storage. It does calculations on entire arrays at once, not just element by element. This power has it's disadvantages, the primary one being memory limitations. Arrays require lots of memory. On a 512k Mac, the largest two dimensional array of integers you can deal with is about 225 by 225, which isn't big enough for many problems. Sometimes you have to use a less elegant algorithm to solve a problem. But then, who has ever had enough memory?