APL / J

by Seung-jin Kim and Qing Ju

What is APL and Array Programming Language?

APL stands for “A Programming Language” and it is an array programming language based on a notation invented in 1957 by Kenneth E. Iverson while he was at Harvard University[Bakker 2007, Wikipedia – APL]. Array programming language, also known as vector or multidimensional language, is generalizing operations on scalars to apply transparently to vectors, matrices, and higher dimensional arrays[Wikipedia - J programming language]. The fundamental idea behind the array based programming is its operations apply at once to an entire array set(its values)[Wikipedia - J programming language]. This makes possible that higher-level programming model and the programmer think and operate on whole aggregates of data(arrary), without having to resort to explicit loops of individual scalar operations[Wikipedia - J programming language].

Array programming primitives concisely express broad ideas about data manipulation[Wikipedia – Array Programming]. In many cases, array programming provides much easier methods and better prospectives to programmers[Wikipedia – Array Programming]. For example, comparing duplicated factors in array costs 1 line in array programming language J and 10 lines with JAVA. From the given array [13, 45, 99, 23, 99], to find out the duplicated factors 99 in this array, Array programing language J's source code is

```
+ / 99 = 23 45 99 23 99
```

and JAVA's source code is

```
class count{
   public static void main(String args[]){
      int[] arr = {13,45,99,23,99};
      int count = 0;
      for (int i = 0; i < arr.length; i++) {
         if ( arr[i] == 99 ) count++;
      }
      System.out.println(count);
   }
}
```

Both programs return 2.

Also Array programing is very well suited to implicit parallelization[Wikipedia – Array Programming, Howland 2005]. This topic is very much researched these days since currently new hardwares support multi-core processors even to personal use laptops[Wikipedia – Array Programming]. Intel and compatible CPU vendors introduced various instruction set extensions such as MMS, SSE3 and 3DNow! technologies since 1997[Wikipedia – Array Programming]. These technologies include rudimentary SIMD(Single Instruction, Multiple Data) array capabilities. And in higher level, this array processing is distinct from parallel processing which many instructions are carried out simultaneously. Parallel processing aims to split a larger problem into smaller ones (MMID - Multiple Instruction stream, Multiple Data stream) to be solved piecemeal by numerous processors[Wikipedia – Array Programming]. The APL language features a rich set of operations which work on entire arrays of data, like the vector instruction set of a SIMD architecture[XXXXXXX].

History of APL

Kenneth E. Iverson at Harvard University introduced a mathematical notation to help him delivering an lecture called “Automatic Data Processing” in years 1955 to 1960 and in 1962 he published the book “A Programming Language” which described his mathematical notation[Bakker 2007, Wikipedia – APL]. However, the mathematical notation described in the book “A Programming Language” was not quite close to the current APL in the market[XXXXX]. After his book, IBM was chiefly responsible for the introduction of APL to the marketplace[Wikipedia – APL]. In 1965, Kenneth E. Iverson joined IBM and IBM reworked a portion of the notation and implemented the language as a practical programming language for the market[Bakker 2007, Wikipedia - APL]. First available APL was in 1967 for IBM 1130 as APL\1130[Breed 2006]. At that time, software and hardware came together(very machine dependent). IBM 1130 was a machine type and APL\1130 was the system(programming language) for IBM 1130 machine(This is one of IBM 360 mainframe series machines). So APL a rather old language along with languages like ALGOL(1958), Basic(1965), COBOL(1960), Fortran(1953) and PL/I(1965)[Bakker 2007]. From late 1960s to 1980s, APL have lived and grown with IBM Mainframe system[Wikipedia - APL]. In 1969, STSC(Scientific Time-Sharing Corporation) was established with some of people who originally implemented APL at IBM and in 1970 SCSC created APL*PLUS(tm), a new version of the APL language with many extensions oriented toward allowing to develop business applications[Bakker 2007]. Since 1966 APL has been implemented mainly large scale machines such as IBM mainframe because APL required more resources (CPU, Storage, etc) than smaller personal computers at that time period[Bakker 2007]. First version of APL on the PC was APL*PLUS PC version 1 and was introduced by STSC[Bakker 2007]. APL*PLUS PC was launched in 1982 and was highly compatible with APL*PLUS Mainframe[Bakker 2007].

In 1984, the new version of APL with major extension: nested array was introduced by IBM[Bakker 2007, Wikipedia - APL]. Starting in early 1980s, under the leadership of Dr Jim Brown, IBM implemented a new version of APL language. At this time, IBM enhanced the concept of nested array where an array may contain other arrays[Wikipedia - APL]. Kenneth E. Iverson left IBM and he was no longer in control of the development of the APL language at IBM[Wikipedia - APL]. Kenneth E. Iverson joined I. P. Sharp Associates and directed the evolution of Sharp APL to be more according to his vision[Baker 2007, Wikipedia – APL]. Kenneth E. Iverson later developed the J Programming Language with Roger Hui at Jsoftware[Baker 2007, Wikipedia – APL].

Whereas APL interpreters had developed for Unix-based microcomputers, APL2 was almost always the standard chosen for new APL interpreter developments[Wikipedia - APL]. APL2 for IBM mainframe computers is still available today. The APL2 Workstation edition (Windows, OS/2, AIX, Linux, and Solaris) were available in the early 1990s[Wikipedia - APL]. In 1989-90 Kenneth E. Iverson and Roger Hui collaborated on an update to the APL language which laster release from Jsoftware and named as J[Wikipedia - APL]. J removed the requirements for APL's special symbols, which required a special greek character keyboard, using the ASCII character set for all functional symbols[Wikipedia - APL]. J also improved upon APL's function-level programming features by allowing true value-free algorithm definitions[Wikipedia - APL]. Currently the J language interpreter is freely available at Jsoftware for Windows, Linux(Unix) and Macintosh.

J (programming language)

In 1980, Kenneth E. Iverson left IBM and started to work for I.P.Sharp Associates which was using APL to establish a time-sharing service that became widely used in Canada, the United States and Europe[]. In 1987, Kenneth E. Iverson retired I.P.Sharp Associates and turned his attention to the development and promotion of a modern dialect of APL called simply J[]. Kenneth E. Iverson developed J due to provide a tool for writing about and teaching a variety of mathematical topics that was available either free or for a nominal charge, could be easily emplemented in a number of different computing environments, and maintained the simplicity and generality of APL[]. J was first implemented in 1980 and has undergone continuous development since then[]. Now J is available on a wide range of computers and operating systems and utilizes the latest developments in software including graphical user interfaces[].

The most obvious difference between APL and J is the use of the ASCII character set available on all keyboards[]. To avoid repeating the APL special character problem, J requires only the basic ASCII character set and resorting the digraphs formed using the dot or colon characters to extend the meaning of the basic characters available[]. As an array programming language, J is very powerful and most suited to mathematical and statistical programming, especially processes on matrices[]. J is, obviously, a MIMD language[]. J supports functional-level programming (known as higher-order functional programing), via its tacit programming features[]Wikipedia - J programming language. J's flexible hierarchical namespace shceme can be effectively used as a framework for both class-based and prototype-based object oriented programming[Wikipedia - J programming language].

Instruction of J

J supports interactive command line interface. The input line may be an expression and when the expression(s) is(are) entered, the values of the expression is computed and displayed its result on the next line.

Simple arithmetic expressions are

```
        2*3      returns 6,
        2+3     returns 5,
        3%4     returns 0.75,
        3-2     returns 1,
        2 -3    returns _1( underscore _ means negative symbol),
        *:3     returns 9 (*: means square root)
        3 * 4 +5        returns 27 (NOT 17!!)
```

As mentioned earlier, J is array programming language. Therefore, it has very efficient processing of array. Technically J interprets everything as an array object. Therefore list is also one type of array.

```
*: 1 2 3 4      returns     1 4 9 16      which is square root of each list value
1 + 10 20 30    returns     11 21 31      which 1 is added to each list value
10 20 30 + 1    returns     11 21 31
2 | 0 1 2 3 4   returns     0 1 0 1 0     | do the reminder operation
```

The English-language expression : let x be 100 is x=: 100 in J. A function taking a single argument on the right is called a monadic function or monad. “Square”, (*:) is a good example. A function taking two arguments, one on the left and one on the right, is called a dyadic function or dyad. For instance, “+”. Multiply together 2 3 4 is “* / 2 3 4” and its return value is 24. In the same manner, “+ / 1 2 3” returns 1+2+3 which is 6. Simply if F is a dyadic function and L is a list of numbers a,b,c,d,.....z then:

```
F / L     represents       a F b F c F .... F y F z
```

In J, “true” is represented by the number 1 and “false” by the number 0. So the expression “2 > 1” returns 1, “2=1” returns 0, and “2<1” returns 0. If x is a list of numbers, you can express it as “x =: 5 4 1 9” and simply typing “x > 2” means which number in list x are greater than 2? and its return values are “1 1 0 1”. Also * / x will do 5 * 4 * 1 * 9 and it returns 180. “+ / x > 2” will return 3 which it returns list of “1 1 0 1” from x >2 and it does “+ / 1 1 0 1” which means 1 + 1 + 0 + 1. If you do “x = x”, it will return “1 1 1 1“ because all list members are the same as its list members (i.e “5 4 1 9 = 5 4 1 9 -> true true true true).

Array processing in J / Types in J

As mentioned earlier, J is an array programming language which means generalizing operations on scalars to apply transparently to vectors, matrices, and higher dimensional arrays. In the other words, its operations apply at once to an entire array set(its values).

In J:

```
table =: 2 3  $  5 6 7  8 9 10
```

creates table

5	6	7
8	9	10

And now 10 * table means multiply by 10 to each individual factors in table. So

10 * table returns

50	60	70
80	90	100

0 1 * table returns

0	0	0
8	9	10

which is from

0*5	0*6	0*7
1*8	1*9	1*10

A table has two dimensions(rows and columns). In this sense, list can be one dimensioned table. Also we can think of table-like object with more than two mentions. The world "array" is defined by "data object with some number of dimensions"

So, in J:

3 $ 1 returns

1	1	1

2 3 $ 5 6 7 returns

5	6	7
5	6	7

2 2 3 $ 5 6 7 8 returns

5	6	7
8	5	6

7	8	5
6	7	8

Monatic function # gives the length of a list such as "# 5 7" returns 2 and "# 7 8 9" returns 3. Also monatic function $ gives the list-of-dimension of its argument such as

When Z =: 1 2 3, $Z returns 3, $ # Z returns 1

When T =: 2 3 $ 1, $T returns 2 3 , # $ T returns 2

When U =: 2, # $ U returns 0

When V =: 4 8 $ 2 3,

V =

1	2	3	4	5	6	7	8
9	1	2	3	4	5	6	7
8	9	1	2	3	4	5	6
7	8	9	1	2	3	4	5

# V	returns 4
# $ V	returns 2
$ V	returns 4 8
3 > V	returns (Array element which is smaller than 3)

1	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0

J treats arrays of characters just like arrays of numbers. A list of characters is entered between quotes, but will not displayed with quotes.

TABLE =: 2 3 $ 'a'

returns

a	a	a
a	a	a

TABLE =: 2 3 $ 'a' 'b'

returns

```
| syntex error
|           TABLE=: 2 3    $ 'a''b'
```

TABLE =: 2 3 $ 'ab'

returns

a	b	a
b	a	b

Selecting and matching in J is

When Y =: 'abcd' ,

"0 { Y"	returns "a"
"1 { Y"	returns "b"
"0 1 { Y"	returns "ab"
"3 0 1 { Y"	returns "dab"

Also a built-in function i. (letter-i dot) return successive integers from zero, expression (i,n)

“i. 4” returns “0 1 2 3”

So when x =: 'Arizona”, “#x” returns “7” and “i.#x” returns “0 1 2 3 4 5 6”

Built-in verb -: (minus colon, called “Match”) tests whether its two arguments have the same shapes and the same values for corresponding elements.

x =: 'university of arizona'

“x -: x” returns 1

y =: 1 2 3 4

“x -: y” returns 0

Another = (Equal) verb compares its arguments elements by elements and produces an array of boolenas of the same shape as the argument.

When x =: 1 2 3 4 and y =: 1 3 3 4, “x =y” returns “1 0 1 1”

“ , (comma)” is a built-in function which is called “Append” in J. It joins together to make lists.

When a =: 'University' and b=: ' of ' and c=: 'Arizona'

“a,b” returns “University of Arizona”

In J, there is Arrays of Boxes concept. A built-in function ; (semicolon, called 'Link') links together its two arguments to form a list. However, these two arguments can be different kinds. For example, “ A =: 'My student id' ; 00001 “ and this returns

My student id	00001

The result A is a list of length 2. and is called now a list of boxes. Elements of boxes can be called with “{“ operator. so “$A” returns 2, “1 { S” is “My student id”. A built-in function < (left-angle-bracket) is called “Box” and it create the box. For example, “< 'University of Arizona' returns a box

University of Arizona

The built-in function > (right-angle-bracket) is called “Open” and it unbox the box. For example, when “box =: < 'University of Arizona'”, “> box” returns “University of Arizona”(no box).

Basically, every data object in J is an array. Even simple expression “a =: 1” is creating an array which has one column and one row. An array may be numbers, or an array of characters, or an array of boxes. Of course, possibly it can be combinations of numbers, characters and boxes.

Functions in J

In J, existing built-in function can be renamed. For example “squre =: *:” makes “squre” to the same as “*:”. Therefore “*: 8” returns 64 but aldo “squre 8” returns 64. “+ / 2 3 4” returns 9 and this 9 comes from 2 + 3+ 4. Also this can be re-written as “sum =: +/”. Now “sum 2 3 4” also returns 9 which is 2+3+4. The expression “+/” can be interpreted as “Insert(/)” applied to the function + to produce a list-summing function. So “/” itself is a function which takes one argument, on its left. In J, “+” and “*” are called “verbs” and functions such as “/” which operates functions from functions are called “operators”.

Operators which should takes one argument are called “adverbs” such as “/”. The adverb is applied to the verb + to product a list-summing verb. “~” is also another adverb in J. “~” exchanges left and right arguments. For example, 'a','b' returns ab and 'a',~'b' returns ba. “&” operation forms a bond between a function and a value for one argument. For example, “double =: * & 2” defines new double function which returns double value of its argument. In the same manner, “cube =: ^ & 3” assgin new function cube which does arg^3. Let “double 3” return the 6 which is 3 * 2. The symbol “@:”(at colon) is called a “composition” operator. The sum of the squares of the list (1,2,3) is 14 which comes from 1+4+9. Based on pre-defined verb 'sum' and 'square', we can express this as “sum square 1 2 3” which square 1 2 3 first and it returns 1 4 9 and sum the (1 4 9) which is 1 + 4 + 9. This can be described as

```
sumsq =: sum @: square
```

Based on these verbs and operators, a temperature converter (from Fahrenheit to Celsius) can be written as

```
s          =: - & 32
m          =: * (5%9)
convert =: m @: s
```

Or, Simply

```
conv =: (* & ( 5 % 9 )) @: ( - & 32 )
```

* The function “sum of square” and its example comes from the Book – Learning J by Roger Stokes.

Program flow control in J is provided by words such as: if. else. while. and so on.

if. elseif.

```
  signum =: 3 : 0
if. y < 0 do. _1
elseif. y=0 do. 0
elseif. do. 1
end.
)  NB. This source is originally from A Brief J Reference by Chris Burke
```

New function signum return _1 when it gets negative numbers and +1 when it gets positive numebers. For example, signum &> 2 1 0 _1 would return 1 1 0 _1

Recursion is obviously averrable in J. Here is one of famous example function for recursion; Factorial.

```
   fact =: 3 : 0
if. y < 1 do. 1
else. y*(fact y - 1)
end.
)
```

Also one of famous mathematical functions in Computer science; Fibonacci:

```
    fibo =: 3 : 0
if. y < 2 do. y
else. +/i.(y+1)
end.
)
```

Object-Oriented Programing in J

J also supports Object-Oriented Programming. Mostly names beginning with “co” means “class and object” Class can be defined with “coclass” keyword

```
   coclass 'MatheFunctions'
   fact =: 3 : 0
if. y < 1 do. 1
else. y*(fact y - 1)
end.
)
   fibo =: 3 : 0
if. y < 2 do. y
else. +/i.(y+1)
end.
)
   destory =: codestroy
   cocurrent 'base'
```

This example creates Stack class and creates methods, push, top, pop and destroy (remove). cocurrent defines going back to the regular environment which means end of Stack class created with coclass keyword. Now new object from class Stack can be created.

```
  M =: conew 'MathFunctions'
  fact__M 3
6                         NB. Factorial 3 = 3 * 2 * 2
  fibo__M 10
55                        NB. Fibonacci 10 = 1+2+3+4+5+6+7+8+9+10
```

<u>Bibliography</u>

Breed 2006

Breed, Larry, How We Got To APL\1130, Vector (British APL Association) August 2006
http://www.vector.org.uk/archive/v223/APL_1130.htm

Bakker 2007

Bakker, Jan, APL, 1966
http://www.thocp.net/software/languages/apl.htm

Burke 2008

Burk, Chris, A Brief J Reference, J Software, Toronto, Canada, April 2008

Stokes 2006

Stokes, Rogers, Learning J, J Software, Toronto, Canada, June 2006.
http://www.rogerstokes.free-online.co.uk/book.htm

Howland 2005

Howland, John E, Functional Programming and the J Programming Language, Oct 2005.
http://www.cs.trinity.edu/~jhowland/math-talk/functional1/

Wikipedia – APL

Wikipedia, http://en.wikipedia.org/wiki/APL (programming_language), April 2008

Wikipedia - Array Programming

Wikipedia, http://en.wikipedia.org/wiki/Array_programming, April 2008

Wikipedia - J programming language

Wikipedia, http://en.wikipedia.org/wiki/J (programming_language), April 2008

Wikipedia – Von Neumann architecture

Wikipedia, http://en.wikipedia.org/wiki/Von_Neumann_architecture, April 2008

Smilie

Smilie, Keith, http://www.vector.org.uk/archive/v223/smill222.htm

Montalbano

Montablno, Michael, A Personal History of APL, San Jose, California, Oct 1982
http://ed-thelen.org/comp-hist/APL-hist.html

Weigang 1999

Weigang, Jim, Jim Weigang's APL Information, 1999
http://www.chilton.com/~jimw/