July 1995

What is J?

Keith Smillie

WHAT are the characteristics of the **J** programming language? How does it differ from most other programming languages? How does it compare with APL from which it was derived? Why was **J** developed? What is the place of **J** with today's software packages and programming languages? It is questions such as these which we will try to answer in this short article.

Programming languages

SOMEONE adding up a list of numbers with a pocket calculator first must be sure that the register where the sum is to appear is set to zero and then must enter each number in turn and add it to the sum being accumulated. A computer program for finding a sum written in one of the conventional languages such as BASIC or Pascal must mimic this manual process of initialization and number-by-number accumulation.

One person commenting on the necessity in conventional languages to spell out the computational procedure in such detail has compared it to saying to a person something like "Go to the box of apples, pick up the first apple and examine it for goodness and if it is good set it to one side, pick up the second apple and ..." instead of saying simply "Go to the box and bring back all of the good apples." In **J**, to continue the analogy, we would simply say "Go to the box and bring back all of the apples that are good (or red, or large, or red and large, etc.)", and not bother at all with the details of the selection.

It is this subordination of detail that distinguishes **J** from almost all other programming languages.

A simple example

WE SHALL introduce **J** with a few calculations relating to a shopping trip where we are interested in finding how many different types of items are purchased, the totals spent on each type of item and on all items together, and the item for which the most money was spent. In the following dialogue with the computer the **J** expressions we enter are indented and the computer's responses begin at the left margin:

```
   item=. 'Apples';
        'Grapes';'CatFood'
   item
```

Apples	Grapes	CatFood

```
   qty=. 3 1 12
   pr=. 0.55 1.71 0.91
   #qty
3
   total=. qty * pr
   total
1.65 1.71 10.92
   +/total
14.28
   max=. >./total
   max
10.92
   pick=. total = max
   pick
0 0 1
   pick # item
```

CatFood

The J language

J IS A GENERAL-PURPOSE programming language developed by Kenneth Iverson, the originator of APL, and Roger Hui. It is intended to be a modern dialect of APL that would provide the simplicity and generality of APL while at the same time be readily and inexpensively available on a variety of computers and capable of being printed on standard printers.

The main characteristics of **J** are the following:

- The standard ASCII character set is used.
- Most primitives are represented by a single character or a single character followed by a period or a colon. For example, **<** represents *less than*, **<.** *lesser of* and **<:** *less than or equal*.
- There is no hierarchy of order of execution among functions as there is in most programming languages and in conventional mathematical notation.
- Most function symbols represent one function when used with two arguments and another function with a single argument. For example, in the last section we used the verb *tally* **#** which gives the number of items in its list right argument and the verb *copy*, also represented by **#**, which selects the name from the list of names on the right corresponding to the **1** in the list of **0**s and **1**s on the left.
- Arrays are considered as single entities so that **a + b** gives the sum whether **a** and **b** are single numbers, lists with the same number of items, tables with the same number of rows and columns, or arrays of higher dimension.
- The terminology of English grammar is used rather than that of programming languages and functions are referred to as *verbs* which may be modified by *adverbs* and *conjunctions*. For example, the verb **+/** which gives the sum over a list is derived from the verb **+** *plus* by the adverb **/** *insert* and **+/1 2 3** is **6** Also **3&***, where **&** is the conjunction *bond*, triples its argument so that **3&* 1 2 3** is **3 6 9**.
- A *functional* or *tacit* style of programming allows the definition of functions without explicit arguments or explicit control structures. For example, **sum=. +/** assigns the name **sum** to the verb **+/** so that **sum 1 2 3** is **6**.

Another example

AT ONE time Smarties were packaged in cardboard cylinders in which the top cap had one of the letters of the alphabet on the underside, the purpose being, of course, to encourage persons to continue buying the packages of the candy until all letters of the alphabet had been collected. The general problem is known in statistics as the coupon collector's problem where one is interested in the average number of purchases required to collect a complete set of whatever prizes are being given away.

The average number of purchases if there are *n* prizes may be found to be "*n* times the sum of the reciprocals of the first *n* positive integers". For example, if there are 4 prizes, then the average number of purchases is 4 times the sum 1 plus 1/2 plus 1/3 plus 1/4 which is 4 times 2.08333 or approximately 8.3. This means that on the average 8 or 9 purchases must be made to collect a complete set of 4 prizes.

We may perform this calculation in **J** as follows:

```
   % 1 2 3 4
1 0.5 0.3333333 0.25
   +/ % 1 2 3 4
2.08333
   4 * +/ % 1 2 3 4
8.33333
```

In the above dialogue `%` represents the verb *reciprocal*, `+/` is *sum*, and `*` is *multiply*. If there are 5 prizes, then the average number of purchases may be calculated in a similar manner by the expression

```
   5 * +/ % 1 2 3 4 5
```

which has the value `11.4167`.

Of course, we would not wish to have to enter such an expression every time we made a calculation of the expected number of prizes. Think of the tedium of entering the integers 1, 2, ... if there were 26 prizes or 100 or even more prizes. We would want to be able to represent the calculation for an arbitrary number of prizes by some name. In other words we would like to be able to define verbs for an arbitrary sequence of calculations.

In the next section we shall define a verb `cc` which gives the expected number of purchases for an arbitrary number of prizes. Thus we have that `cc 4` is `8.3` rounded to one decimal place, `cc 5` is `11.4` and `cc 26` is `100.2`. We also note the very reasonable results that `cc 1` is `1` and `cc 0` is `0` since if there is only one prize it is obtained with the first purchase and if there are no prizes there is nothing to purchase.

The reader may wish to omit on a first reading the following section in which the definition of several verbs for the coupon collector's problem is discussed since a knowledge of the details is not required in the remainder of the paper.

Programming in J

THE first verb in **J** we will define is

```
   pos=. >: @  i.
```

which gives a list of positive integers where, for example, `pos 4` is the list `1 2 3 4`. In this definition we have used the primitive verbs `>:` *increment* (by `1`) and `i.` *integers* which gives a list of non-negative integers, and the conjunction `@` *atop* (or *after*). Thus the definition of `pos` may be read as "increment each item of the list of the first 'so many' non-negative integers".

Now we may define the verb

```
   cc=. * +/ @ % @ pos
```

which is a two-verb sequence, called a *hook,* consisting of the primitive verb `*` "multiply" and the defined verb `+/ @ % @ pos` "the sum of the reciprocals of the first so many positive integers". Specifying a value for the argument of `cc` gives this value to both sides of the hook, so that, for example, `cc 4` is equivalent to

```
   4 * +/ @ % @ pos 4
```

which as we have seen is equal to `8.3`.

To the statistician, collecting all *n* prizes is equivalent to sampling with replacement from the first *n* positive integers until all *n* integers are represented in the sample. Since there are verbs in **J** for sampling with and without replacement, we may define a verb, `ccsim` say, the details of which need not concern us here, which will simulate the coupon collector's problem for an arbitrary number of prizes. Thus two evaluations of the expression `ccsim 4`, which could give the results

```
   2 3 4 4 3 4 3 1
```

and

```
   1 2 3 2 1 4  ,
```

would represent two simulations for four prizes.

The verb `ccsim` may be used in another verb - again the details need not concern us - that will give the sample size in each of a number of simulations for a given number of prizes. One hundred simulations for four prizes were performed and represented in the list `s`, the first few items of which were

```
   5 8 14 4 6 5 16 5 ... .
```

The largest sample was `21` and was found from the expression `>./s`, where `>./` is the verb "maximum over" which is derived from the verb `>.` *larger of.* The smallest sample size, given by the expression `<./s`, was found reasonably to be `4`. The average sample size may be found from the expression

```
   (+/ % #)s
```

to be `8.25` which we may compare with the expected value of `8.3`.

The above expression for the mean with its sequence of three verbs known as a *fork* is equivalent to the expression `+/s % #s`, which may be read as "the sum of the items of `s` divided by the number of items of `s`. If may define the verb

```
   am=. +/ % #  ,
```

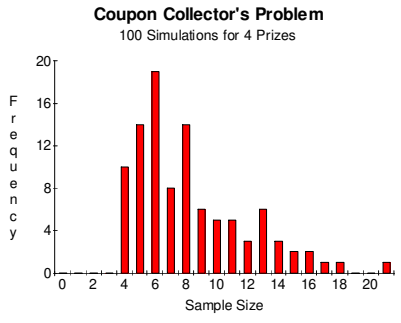
then the expression `am s` is equal to `8.25`.

Hooks and forks are examples of uninterrupted sequences of verbs which play an important role in **J** programming.

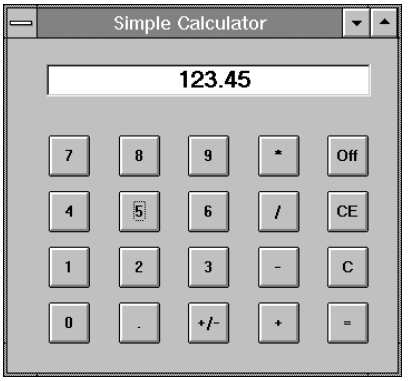
Implementation of J

J IS AVAILABLE on a large number of computers and operating systems including PC (both Windows and DOS), Macintosh and UNIX machines. The core **J** language is the same on all machines so that programs written on one machine may be run on another. **J** may be used either as a calculator as it was in the shopping example or as a programming language as in the coupon collector's problem. Programs and data may be entered into text or ASCII files, called script files, which may then be run so that the programs are executed. Changes to the programs and data are made by editing and rerunning the script file. In the professional version of **J** script files may be saved in a form that may be run independently of the **J** system.

J for Windows Release 2 which was used here provides a convenient graphical user interface and a large collection of utilities for simplifying exchange of objects with other Windows applications. For example, the barchart shown below for the simulation of the coupon collector's problem discussed in a previous section was produced in Microsoft Works from results generated and tabulated in **J**.



J provides a simple graphical interface which allows the construction of graphical interfaces for applications written in **J** without learning the details of Windows programming. All of the usual controls such as push buttons, scroll bars, edit boxes, etc. are available as well as controls for Dynamic Data Exchage (DDE), Object Linking and Embedding (OLE) and Visual BASIC Extended (VBX). The following figure shows a simple four-function calculator constructed in **J**:



J as a statistical companion

J CAN provide a simple yet powerful computational supplement to any statistical text. Consider, for example, Peter Sprent's *Understanding Data* (Penguin, 1988), a very readable introduction to elementary statistical methods. Almost all of the computations illustrated in the book may be handled with the primitive verbs in **J** together with a few readily available defined verbs for tabulating discrete and continuous data, constructing stem-and-leaf diagrams and barcharts, calculating means and variances and medians and quartiles, and performing least-squares fitting of straight lines. Graphing may be accomplished by a spreadsheet or graphics package used together with **J**. We shall give very briefly only two simple examples.

The topic of smoothing a series by moving averages is illustrated with some quarterly data on new houses completed in Scotland from 1981 to 1983. We shall use the public sector data where the number of new houses given in hundreds will be represented by the list **H**, the first few items of which are

```
24 20 19 13 12 ...
```

The successive triples of items used for a three-term moving average may be displayed by the expression **3<\H**, where **** is the adverb *prefix*, and are

```
┌──────────┬──────────┬──
│24 20 19  │20 19 13  │...
└──────────┴──────────┴──
```

The corresponding moving averages, given by **3 am\H** and rounded to one decimal place, are

```
21 17.3 14.7 11.7 ...
```

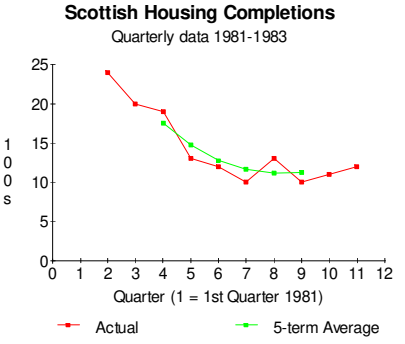
A more satisfactory five-term weighted moving average may be found from the expression

```
5 (1 2 2 2 1&AM)\H
```

where **AM** is a defined verb for a weighted mean with arbitrary weights, which gives the values

```
17.5 14.75 12.75 ...
```

which are shown in the following figure produced by Microsoft Works:



Another example is an empirical study of random sampling using the number of editorial corrections on each page of a nine-page text. First the 128 possible combinations of the 9 errors taken 4 at a time are found; secondly the means of the combinations are found; finally a five-number summary - minimum, first quartile, median, third quartile and maximum - of the means is calculated. The entire simulation may be accomplished simply in **J** by the expression

```
five am"1 (4 COMB C)
```

where **C** is the nine-item list of corrections and **COMB** and **five** are defined verbs whose purposes are suggested by their names.

J as a mathematical companion

J MAY serve also as a computational supplement to a variety of mathematics texts. As an example we shall take *Numbers Without End* by Cornelius Lanczos (Oliver & Boyd, 1968), a delightful little book intended to convey to the reader the fascination that this distiguished author has had with numbers. In this section there is space for only a very few brief remarks about the use of **J** in a discussion of number systems and divisibility. Some comments on prime numbers which may be relevant to a reading of the book are given in the following section.

Any discussion of number systems with arbitrary bases may be made clearer by use of the two verbs **#.** *base* and **#:** *antibase*. For example, **#: 13** is **1 1 0 1**, the binary or base-2 representation of the decimal number **13**, and conversely **#.1 1 0 1** is **13**. Also **8 8#: 35** is **4 3**, the octal or base-8 representation of the decimal number **35**, and **8#. 4 3** is **35**.

Much of the chapter on divisibility is based on the generalization of the familiar multiplication table. In **J** this concept may be illuminated by the adverb **/** *table* where, for example, if **a** is the list **1 2 3 4**, the addition and multiplication tables

2	3	4	5	1	2	3	4
3	4	5	6	2	4	6	8
4	5	6	7	3	6	9	12
5	6	7	8	4	8	12	16

are given by **a+/a** and **a*/a**, respectively. Also the remainder table

```
0 0 0 0
1 0 1 0
1 2 0 1
1 2 3 0
```

is given by the expression **a|/a** where the verb **|** *residue* gives the remainder when the first argument divides the second, and, for example, the third row **1 2 0 1** gives the remainders when the items of **a** are divided by **3**.

Finally, the concepts in the last two paragraphs may be combined, and, for example, the expression

a ((5 5&#: &. >)@ */)/a

gives the following base-5 multiplication table:

0	1	0	2	0	3	0	4
0	2	0	4	1	1	1	3
0	3	1	1	1	4	2	2
0	4	1	3	2	2	3	1

An exercise

A PRIME number is a positive integer that has no divisors other than itself and unity. According to the fundamental theorem of arithmetic a positive integer may be expressed in only one way as a product of primes. The verbs **p:** *prime* and **q:** *factor* are available for generating primes and for prime factorization so that, for example, **p: i. 5** gives the list **2 3 5 7 11** of the first five primes and **q: 75** is **3 5 5** so that ***/3 5 5** is **75**.

Primes may be generated by sieving methods in which non-primes are removed from a list of positive integers in an orderly manner. The following sequence of **J** expressions, due to Kenneth Iverson, implements a residue sieve based on the equivalent definition of a prime as a positive integer with exactly two divisors:

```
pn=. >: @ i.
rt=. pn |/ pn
dt=. 0&= @ rt
nd=. +/ @ dt
prt=. 2&= @ nd
prsel=. prt # pn
```

The verb names are abbreviations of the following terms: *p*ositive *n*umbers, *r*emainder *t*able, *d*ivisibility *t*able, *n*umber of *d*ivisors, *pr*ime *t*est, and *pr*ime *sel*ect.

By evaluating each of these expressions in turn the interested reader might wish to show that **prsel 12** gives the list **2 3 5 7 11** of primes less than or equal to **12**. (We note that the verb *equal* **=** gives a test for equality so that for example **3 = 3** is **1** or true and **4 = 5** is **0** or false.)

What is J?

WRITING in 1991 on the 25th anniversary of APL, Kenneth Iverson remarked that "... Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects." **J** as a modern dialect of APL should be regarded similarly. While its use as a programming language for commercial and scientific applications is to be encouraged, **J** might best be regarded as a notation that will both illuminate computational methods and also make them conveniently executable on a computer.

We shall let Roger Hui have the last word on **J**: "I called the language **J** because 'J' is easy to type."

Further reading

THE indispensible reference for **J** is Kenneth Iverson's ***J*** *Introduction and Dictionary* (Iverson Software Inc., 33 Major Street, Toronto, Ontario M5S 2K9) and is available online with **J** for Windows Release 2.05 and later. *Vector*, the Journal of the British APL Association, has an increasing number of articles on **J.** The author's *Some Notes on Introducing **J** with Statistical Examples* is available as a Postscript file by anonymous ftp at `ftp.cs.ualberta.ca` in the file `pub/smillie/intj.ps`.

Keith Smillie is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2H1. His email address is smillie@cs.ualberta.ca.