Second edition *July 1998*

Beginning J

Keith Smillie

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 will 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.

In this article we shall give a brief introduction to **J** illustrated by a discussion of the coupon collector's problem which serves as a model for collecting a complete set of prizes included, one prize per package, in products such as breakfast cereal.

Some simple examples

IN THE FOLLOWING dialogue the **J** expressions we enter are indented whereas the responses from the computer begin at the left margin:

```
   3 + 5
8
   3 - 5
_2
   2 * 3
6
   15 % 6
2.5
   % 2.5
0.4
   % 15 % 6
0.4
   2 * 3 + 4
14
   (2 * 3) + 4
10
   % 1 2 3 4
1 0.5 0.3333333 0.25
   +/ % 1 2 3 4
2.08333
   4 * +/ % 1 2 3 4
8.33333
   0.5 + 4 * +/ % 1 2 3 4
8.83333
   <.0.5 + 4 * +/% 1 2 3 4
8
   >.0.5 + 4 * +/% 1 2 3 4
9
   i. 4
0 1 2 3
   >: i. 4
1 2 3 4
   pos=: >: @ i.
   pos 4
1 2 3 4
   w=: 2.3 5 3.5 6
   w
2.3 5 3.5 6
   +/w
16.8
   #w
4
   +/w % #w
4.2
   (+/ % #)w
4.2
   am=: +/ % #
   am w
4.2
   +/ % # w
0.25
```

The J language

THE MAIN CHARACTERISTICS of **J** are the following:

- The standard ASCII character set is used.
- Primitives are represented by a single character or a single character followed by a period or a colon.
- The terminology of English grammar is used rather than that of programming languages. Functions are referred to as *verbs* whose arguments are called *nouns* and *pronouns* instead of constants and variables, and 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*. Also **@** is the conjunction *atop* which, for example in the defined verb **pos**, applies the verb on the left after the verb on the right.
- Precedence amongst verbs is determined by parentheses, and in their absence the right argument is the entire expression on the right and the left argument is the noun immediately on the left. Adverbs and conjunctions take precedence over verbs with the left argument being the entire verb phrase on the left.
- Negative numbers are indicated by a preceding underbar _ which is considered to be part of the number as is, for example, the decimal point. Also the decimal point is necessarily preceded by at least one digit so that, for example, two-fifths as a decimal fraction is represented as **0.4**.
- Most function symbols represent one function when used with one argument and another function when used with two arguments. For example, **%** represents the monadic verb *reciprocal* and the dyadic verb *divided by*, and **/** represents the monadic adverb *insert* and the dyadic adverb *table*.
- Nouns may be single items or *atoms*, one-dimensional arrays or *lists*, two-dimensional arrays or *tables*, or arrays of higher dimension or *reports*. Thus the expression **a + b** is a valid sum as long as **a** and **b** are compatible arrays.
- Verbs may be defined in a *functional* or *tacit* manner without explicit arguments or control structures. However, *explicit* verbs may be defined where the arguments are specified and where the definition may extend over several lines and involve control structures similar to those in conventional programming languages.
- An uninterrupted sequence of three verbs is known as a *fork* and is a generalization of the notation of conventional mathematics where, for example, $(f+g)x$ represents the sum $f(x)+g(x)$. An example is the verb **am** for the arithmetic mean.

Coupon collector's problem

THE COUPON collector's problem involves sampling from a finite population until all of the items are represented in the sample. This sampling procedure, as was stated in the introductory comments, can serve as a model for collecting a complete set of prizes which are included in some product.

If there are *n* different prizes, the problem is equivalent to random sampling with replacement from the first *n* positive integers until all *n* integers are represented in the sample. It is a simple exercise in elementary probability theory to show that the average sample size is "*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, or 4 times 2.08333, or approximately 8.3. Therefore, on the average 8 or 9 purchases are required to collect all 4 prizes.

The corresponding calculations in **J** were included in the examples given previously and may be expressed as follows if the verb **pos** is used:

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

In general, if there are **n** prizes, the expression for the average number of purchases is

```
   n * +/ % pos n
```

which for **n** equal to **10** is approximately **29.3**.

In order to simplify these calculations we shall define the monadic verb

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

whose argument is a non-negative integer giving the number of prizes and whose result is the corresponding expected value. The conjunction *at* **@:** is used so that the sum is applied over the list of reciprocals rather than to each item in the list. For example, **cc 4** is **8.3** and **cc 10** is **29.3**. This verb has a two-verb sequence, called a "hook", consisting of the primitive verb ***** *multiply* and the composed 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**.

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.

Tables

IN ADDITION to being able to compute expected values for individual values of the number of prizes, we might wish to compute a table of expected values over an arbitrary range of number of prizes. This may be done very simply, but first we must introduce the dyadic adverb *table* **/** which gives the table formed by inserting the verb it modifies between all possible pairs of items chosen from the two arguments. For example, if **u** and **v** are the lists **1 2 3** and **1 2 3 4**, respectively, then **u+/v** and **u*/v** give portions of the addition and multiplication tables which may be displayed simply as

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

by the expression

```
   u (+/ ; */) v
```

where **;** is the dyadic verb *link*. This last expression is an example of a dyadic fork and is equivalent to

```
   (u +/ v) ; u */ v .
```

If we introduce the utility adverb **table**, the details of which need not concern us, which gives bordered tables, then the above addition table may be displayed by the expression

```
   u + table v
```

as

	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7

.

We shall introduce the verb

```
   rnd=: <.@(0.5&+)
```

where **<.** is the monadic verb *floor* and **&** is the conjunction *bond*, which rounds its non-negative argument to the nearest integer, so that, for example, **cc 20** is **71.9548**, and **rnd cc 20** is **72**.

Now define the lists

```
   rows=: 5 * i. 10
```

and

```
   cols=: pos 5
```

which have the values

```
   0 5 10 15 20 25 30 35
                  40 45
```

and

```
   1 2 3 4 5
```

respectively. Then a table of expectations rounded to the nearest integer for from **1** to **50** prizes is given by the expression

```
   rows (rnd@cc@+) table
                  cols
```

and is

	1	2	3	4	5
0	1	3	6	8	11
5	15	18	22	25	29
10	33	37	41	46	50
15	54	58	63	67	72
20	77	81	86	91	95
25	100	105	110	115	120
30	125	130	135	140	145
35	150	155	161	166	171
40	176	182	187	192	198
45	203	209	214	219	225

Simulation

IN ORDER to simulate the coupon collector's problem we shall require the monadic verb *roll* **?** for sampling with replacement. The expression, **? n** gives a uniform random sample from the population **i. n**, and, for example, the expression **? 3 3 3 3** could have the value **0 1 2 1**.

The explicit monadic verb **ccsim** defined by

```
ccsim=: 3 : 0
n=. y.
r=. i. 0
while. n > # ~. r do.
   r=. r, ?n
end.
>:r
)
```

simulates the coupon collector's problem for an arbitrary number of prizes. For example, **ccsim 4** could have the value

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

representing a simulation for four prizes.

A few remarks on the structure of this verb will be made. The first line is the heading which defines **ccsim** as a verb with a right argument **y.**. In the definition we have the monadic verb *nub* **~.** which gives the unique items in its argument, and the monadic verb *tally* **#** which gives the number of items in its argument. Thus, **n** is given the value of the right argument which is the number of prizes, the result **r** is first initialized to an empty list and then repeatedly has a random integer from **i. n** appended to it until all of the integers in **i. n** are represented in its nub. The last statement adds **1** to each item in the final value of the list **r**.

Repeated simulations for a given number of prizes may be done with the verb

```
ccs=: (#@ccsim)"0 @ #
```

and, for example,

```
s=. 10 ccs 3
```

where **s** might have the value

```
4 3 5 3 3 3 4 7 8 8
```

would represent **10** simulations with **3** prizes. (We remark that **#** is the dyadic verb *copy* so that **10 # 3** gives the list

```
3 3 3 3 3 3 3 3 3 3
```

and the conjunction *rank* **"** is used to apply the verb **#@ccsim** to each item of the list.) The largest sample is given by the expression **>./s**, where **>.** is the dyadic verb *larger of*, and is equal to **8**. The mean sample size is **am s** which is equal to **4.8** as compared to an expected value of **cc 3** which is **5.5**.

Frequencies

WE SHALL construct a frequency table for the simulation with the range of sample sizes in the first column and corresponding frequencies in the second with the range extending from **0** to the maximum sample size.

The range **r** for the list **s** of sample sizes is given by

```
r=. 0, pos >./s
```

whichis equal to

```
0 1 2 3 4 5 6 7 8
```

The distribution of **s** over the range relates its items to the range and is given by the expression **r =/ s** which gives a table of **0**s and **1**s with a **1** indicating that the row range value **r** is equal to the column value **s**. This table may be meaningfully displayed by the expression **r = table s** which has the value

	4	3	5	3	3	3	4	7	8	8
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	1	0	1	1	1	0	0	0	0
4	1	0	0	0	0	0	1	0	0	0
5	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	1

For example, the fifth row,

```
1 0 0 0 0 0 1 0 0 0
```

indicates that the first and seventh samples are of size **4**. The row sums of this table give the frequency of occurrence of each of the items in the range, and we have that

```
+/"1 r =/ s
```

is the list of frequencies

```
0 0 0 4 2 1 0 1 2
```

The rank conjunction **"** is used to give the row, rather than the column, sums.

The desired frequency table is given by the dyadic verb

```
frtab=: [,.fr
```

where

```
fr=: +/"1 @ (=/)
```

gives the frequencies. We note the use of the dyadic verb *stitch* **,.** in the verb **frtab** to form a two-column table from the lists for the range and the frequencies. Thus the frequency table is given by **r frtab s** which we shall display in transposed form as

```
0 1 2 3 4 5 6 7 8
0 0 0 4 2 1 0 1 2
```

by the expression **|: r frtab s**, where **|:** is the monadic verb *transpose* which interchanges rows and columns.

Instead of finding the frequencies over the range of sample sizes, we may wish to limit the rows of the table to those corresponding to non-zero frequencies. This may be accomplished simply by the verb

```
nubtab=: ~. ,. nubfr
```

where

```
nubfr=: +/"1 @ =
```

gives the nub frequencies. The numerical results of this section may be summarized as

0	0
1	0
2	0
3	4
4	2
5	1
6	0
7	1
8	2
3	4
4	2
5	1
7	1
8	2

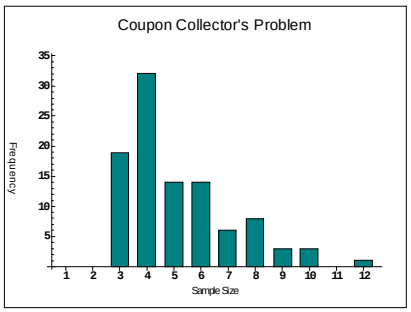
by the expression

```
(r frtab s);nubtab sort s
```

where the utility verb **sort** gives the items of the nub in sorted order.

J and graphics

A PLOTTING PACKAGE is available in **J** for constructing a variety of different graphics such as line and density plots, bar and pie charts, and three-dimensional surfaces. The following is a bar chart showing the frequency of sample sizes for 100 simulations for 3 prizes.

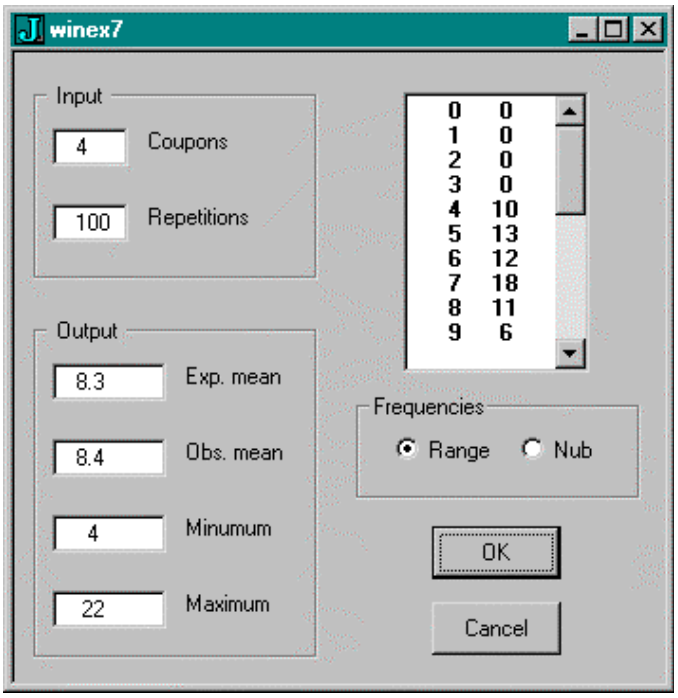


In addition, the Windows 95/NT versions of **J** support OpenGL which is an industry-standard software interface to graphics hardware for use in computer-aided design, medical imaging and film special effects.

J and Windows

J IS AVAILABLE for Windows 95/NT, Macintosh and Unix platforms with the same core **J** language on all machines. Windows systems combine the language with a large collection of utilities for simplifying the exchange of objects with other Windows applications and with an application development environment for the construction of graphical interfaces for **J** applications. All of the usual controls such as push buttons, scroll bars, edit boxes, etc. are available as well as controls for Dynamic Data Exchange (DDE), Object Linking and Embedding (OLE) and Visual BASIC Extended (VBX).

The Windows form shown here allows for the convenient use of some of the **J** verbs developed in this article. The user enters the number of prizes and number of repetitions in the appropriate boxes, selects either the "Range" or "Nub" radiobutton depending on whether the frequencies



over the whole range or only the nub are wanted, and then presses the OK button. The resulting frequency table is displayed and is also given in the global variable **`CCtable`**.

J and the World Wide Web

J IS A PRODUCT of Iverson Software Inc., which continues to develop the language, and is distributed by Strand Software Systems. The current version **J4.01** may be downloaded from the World Wide Web site *http://www.jsoftware.com* and comes with not only online help, tutorials and demos but also with online full-text versions of the several manuals published by ISI which are also available in printed form. The Web site contains also information about the latest developments in **J**, a list of selected **J** publications, and other items of interest to users of **J**.

What is J?

J HAS BEEN USED in this article both as a simple calculator and as a programming language in the study of the coupon collector's problem. However it is used, **J** hides many of the details that must be considered with conventional languages. 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 as a notation that will illuminate computational methods and 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 learning and using **J** is Kenneth Iverson's *J Introduction and Dictionary* published by Iverson Software Inc.

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.