November 1996

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 an analysis 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 with the computer 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 + 5
15
   +/1 2 3 4 5
15
   % 1 2 3 4 5
1 0.5 0.3333333 0.25 0.2
   +/ % 1 2 3 4 5
2.28333
   5 * +/ % 1 2 3 4 5
11.4167
   i. 5
0 1 2 3 4
   >: i. 5
1 2 3 4 5
   pos=. >: @ i.
   pos 5
1 2 3 4 5
   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. For example, **>** represents *Larger Than*, **>.** *Larger Of* and **>:** *Increm*, i.e., increase by **1**.
- 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 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*. 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.
- Most function symbols represent one function when used with one argument and another, usually related, function when used with two arguments. For example, **%** represents the monadic verb *Reciprocal* and the dyadic verb *Divided By*, and the adverb **/** gives the monadic verb *insert* and the dyadic verb *table*.
- Arrays are considered as single entities so that **a + b** gives the sum whether **a** and **b** are single numbers or *atoms*, *lists* with the same number of items, *tables* with the same number of rows and columns, or compatible arrays of higher dimension.
- 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 conventional selection and repetition.
- 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 defined above. An example of a *hook* which is an uninterrupted sequence of two verbs will be given in the next section.

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. In this section and the next we shall investigate the average sample size, i.e., the average number of purchases, required to collect a complete set of items.

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 5 prizes, then the average number of purchases is 5 times the sum 1 plus 1/2 plus 1/3 plus 1/4 plus 1/5, or 5 times 2.28333, or approximately 11.4. Therefore, on the average 11 or 12 purchases are required to collect all 5 prizes.

The corresponding calculations in **J** were included in the examples given previously and are as follows:

```
   % 1 2 3 4 5
1 0.5 0.3333333 0.25 0.2
   +/ % 1 2 3 4 5
2.28333
   5 * +/ % 1 2 3 4 5
11.4167
```

These expressions may be simplified by using the verb **pos** also introduced in the examples to give a list of positive integers, so that the last expression, for example, becomes

```
   5 * +/ % pos 5
```

.

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. For example, **cc 5** is **11.4** and **cc 10** is **29.3**. This verb is a two-verb sequence, called a "hook" in the last section, 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 5** is equivalent to

```
   5 * +/ @ % @ pos 5
```

which as we have seen is equal to **11.4**.

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*.

The following table gives the expected number of purchases for the number of prizes from **1** to **50**, inclusive:

	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

The left and top margins are given by the expressions

```
   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. The body **t** of the table is given by

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

where the verb

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

rounds the expected values to the nearest integer. The required table may be assembled by the expression

```
   rows by cols over t
```

where

```
   by=. ' '&;@,.@[ ,. ]
```

and

```
   over=. ({.;}.)@":@,
```

are utility verbs, as is **rnd**, the details of which need not concern us.

To simplify the construction of such tables of expected values we define the dyadic verb

```
   cctable=. [ by ] over
          [ (rnd@cc@+)/ ]
```

where the verbs *Left* **[** and *Right* **]** give the left and right arguments. The required table is then given by

```
   rows cctable cols
```

.

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, two evaluations of the expression **ccsim 4**, which might 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.

Only 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 smallest and largest samples are **3** and **8** and are given by the expressions **<./s**, where **<.** is the dyadic verb *Lesser Of*, and **>./s**, respectively. The mean sample size is **am s** which is equal to **4.8** as compared to an expected value of **5.5**.

Another example with **100** simulations for **4** prizes gave the sample sizes

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

Calculations similar to those in the last paragraph gave smallest and largest sample sizes of **4** and **21**, and a mean of **8.25** as compared with the expected value of **8.3**.

Frequencies

IN ORDER TO construct a table of sample sizes and corresponding frequencies we shall construct a list giving the range of sample sizes from 0 to the maximum sample size and also a list of corresponding frequencies and then join the lists together to form a two-column table.

The range **r** of the list **s** in the last section is

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 range value corresponding to the row is equal to the item of **s** corresponding to the column. This table may be meaningfully displayed by the expression

```
r by s over r =/ 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 fourth row in the table,

0 1 0 1 1 1 0 0 0 0 ,

indicates that a sample size of **3** occurs in the second, fourth, fifth and sixth samples. The row sums of the distribution 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 **"** was used in the above expression to give the row, rather than the column, sums.)

The above calculations for the frequencies may be summarized by the following verbs:

```
frtable=. range ,. fr
range=. i. @ >: @ (>./)
fr=. +/"1 @ dis
dis=. range =/ ]
```

We note the use of the dyadic verb *stitch* **,.** in the verb **frtable** to form a two-column table from the lists for the range and the frequencies. Thus the required frequency table is given by

```
frtable s
```

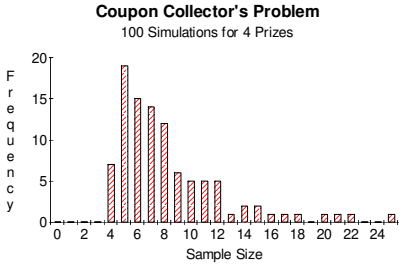
which has the value

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

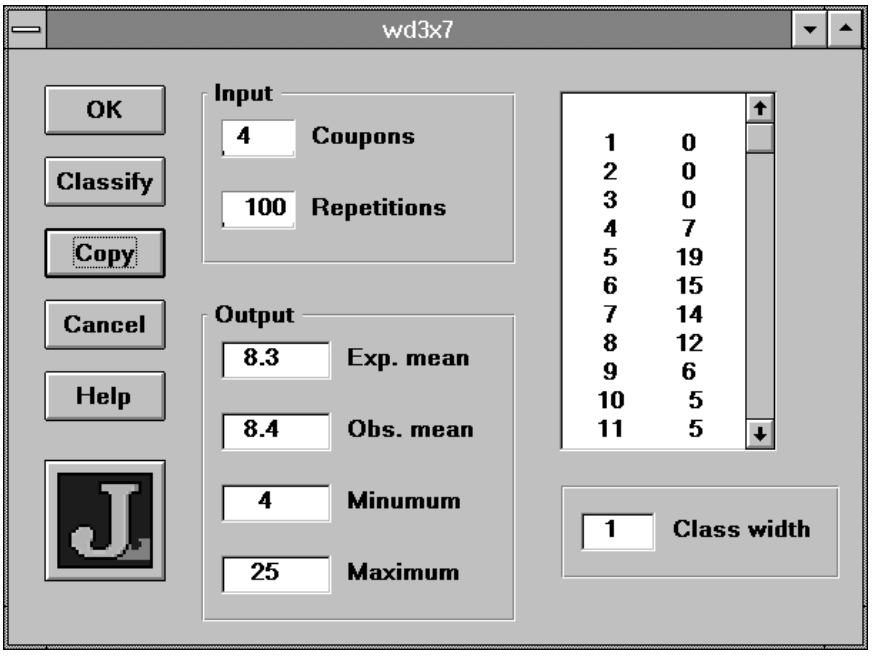
J and Windows

J IS AVAILABLE on a number of computers and operating systems including PC (both Windows and DOS), Macintosh and UNIX systems 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 opposite allows the use of the **J** verbs developed in this article without any knowledge of the language. The user merely enters the number of prizes and number of repetitions in the appropriate boxes and presses the "OK" button. The simulations are performed and the results are displayed as shown. The "Classify" button groups the rows of the frequency table according to the specified class width, and the "Copy" button copies the frequencies, grouped or not, to the Windows Clipboard. The barchart given below has been obtained from the results shown in the form and was produced by copying the frequency table to the Clipboard, pasting it into a spreadsheet in MS Works, and finally pasting the resulting chart into this document which has been prepared with MS Word.



Runtime executable files have been made freely available and may be distributed with the **J** script files, which are ASCII files, for Windows forms which may then be run independently of the development system.



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. These organizations maintain a World Wide Web site at http://www.jsoftware.com which gives up-to-date information about the language, **J** summaries, a list of ISI publications, and access to downloadable copies of not only the freeware Windows version of **J** but also the full texts of some ISI publications as standard Windows Help files.

J as an executable notation

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

TWO RECOMMENDED references for **J** are Kenneth Iverson's ***J*** *Introduction and Dictionary* which is indispensible at any level of learning or using **J** and the more recent ***J*** *Primer* by Eric Iverson, both of which are 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.