SOCIETY OF ACTUARIES

Article from:

Pension Section News

February 2001 – Issue No. 45

Why I Like J

by Brian Bambrough

I drive a late-model minivan. It has everything I need for my lifestyle. Lots of space. Seats which I can remove or install, depending on my need of the moment. Power windows, A/C, etc., etc. I am totally happy with it. Yet if an auto company said to me "we've got a new minivan with lots of new features including GPS, built-in computer and anti-gravity, and you can have it at a fraction of what you paid for your old minivan," I would switch in a heartbeat.

Such is my relationship with APL and J. APL is a wonderful language for actuarial work. It can manipulate arrays easily. It's an interpreter, so it's easy to debug code. It has powerful and flexible string handling capabilities with which to create output. Nothing could be simpler or more convenient than component files.

So why switch to J? Because J has everything APL has and a whole lot more. When I am asked to describe what J is like, and there is no time for a detailed answer, I simply say "J is APL on steroids." For some years I have been learning J in scraps of my spare time. Now I am working on my first commercial system. So I still have a lot to learn. With this as context, here are just a few of the reasons I like J:

J uses ASCII text instead of APL characters. For example, the Greek iota has been replaced by i. This simplifies my life in several ways. I can use a text editor to write and edit code. It is easier to send code to clients and collaborators. It allows for a richer set of primitives.

Like APL, J has only a few data types: number, character, and box. I find boxing to be more intuitive, consistent, and easier to use than APL's enclose and disclose.

There are many additions and extensions to the language that are marvelous conveniences. Some of these are possible because of the expansion of the number of primitives. For example, monadic {. is "head." This returns the first item in an array. There is also "tail," "behead," and "curtail." Others are brand new capabilities. For example, "infix" acts on successive groups of items in an array. In actuarial work, infix allows me to produce dxs from a survivorship group, lx, with: dx =: 2 -/\ lx. There are "nub" and "nub sieve" primitives. With them I can, for example, identify all the unique combinations of plan, issue age, duration, and underwriting class in a block of policies. The primitive i. is an example of an extension. i.4 returns a 4 element vector, 0 1 2 3, just like iota, in APL. But i.2 3 returns a matrix with two rows and three columns.

J has a symbol for infinity. It is the underline. Why would anyone living in a finite world need infinity? Well, J also has a power conjunction. This is similar to raising a number to a power. For example, 2 cubed means multiply 2 by itself three times. The power conjunction generalizes this to apply to any verb (the J name for function, or subprogram). The power conjunction instructs the verb to keep feeding its result back into itself a specified number of times. If this number is infinity, the verb will only stop if the verb's output is the same as the input, i.e., the process converges. This allows incredibly compact and efficient code to get the solution to problems such as finding the yield from a messy cash flow stream.

The power conjunction is also useful when its argument is finite. For example, to get second differences apply infix twice. Suppose q is a column of qxs that the user has just entered by hand. How can you help him check for errors? Get the second differences: diff =.2 –/\^:2 q. Then flag those that are not reasonable.

J is totally consistent. Concepts stretch across the entire language; for example, "item" is a technical term in J. The items of lx are the number of lives surviving at each age. The items of a table are the rows, the items of a 3 dimensional array are the tables, etc. This idea finds an application in J's "for" control structure. In BASIC we have "for i = 1 to n." In J we have "for_i. array do." In J, the for loop sequentially assigns to i the items of array. This is a powerful and useful generalization.

J has a very powerful grid feature. It's much more than a spreadsheet. It can be used to input, manipulate, and display data in any way that you can imagine. It can also handle infinite arrays, not much use in actuarial work, but intrinsically fascinating.

So, J has a lot of neat features, but what about building complete systems?

When I develop a system, I just write a script (the J term for program or module), run it, and look at the results, just like APL. When it's working right, I call it from the main module. A major feature of this approach is that each script can, if appropriate have its own locale, the J term for name space. Names can be global in a locale, but they don't collide with the same names in other locales. APL workspaces have their virtues, but in this respect J is superior.

The GUI for a system can be created in a manner similar to other languages: set up a form, drag and drop controls on it, and use point and click to set the controls' attributes. J also allows me to create controls and otherwise modify the form at run-time. That this can be done easily (or at all) distinguishes J from most other languages.

Is there any downside to J? Possibly a couple of things:

J, like any interpreter, runs slower than compiled languages. This is generally not a problem due to the array processing nature of the language. In those instances where a large amount of number crunching is required, and it must be done by looping, I use another language to compile this logic, then call it from my J code.

The only other problem with J is that its environment and add-ons constitute a huge amount of material. For an actuary who just wants to use the language to solve problems, it is critically important to delimit the amount of material he or she will try to absorb at first.

A good approach is to take a modest problem, say a spreadsheet that is becoming unwieldy, and learn enough to implement it in J as a single script. This can be done by learning a few primitives, how to write verbs and, possibly, some file handling. The material that can be ignored includes: most of the primitives; hooks, forks and trains; concepts like gerund and obverse; OOP; grids; ODBC; the Project Manager; mapped files and a whole lot more. Even locales and the GUI can be ignored at first.

My overall judgement of J is that I can be more productive with it than any other language, and it is a joy to work with. I plan to use it as my main programming language for the rest of my working life.

Brian Bambrough, FSA, is president of Bambrough & Associates Inc. in Kalamazoo, MI. He can be reached at b.bambrough@worldnet.att.net.