

# Contents

Quick reference diary		2
Editorial	Stephen Taylor	3
<b>News</b>		
Remembering Eugene McDonnell	Roger Hui	5
Industry news: Dyalog, Kx Systems, Optima, IBM		10
APL2010 Berlin – several reports		16
BAA AGM 2010	Anthony Camacho	29
AGM addendum	Paul Grosvenor	32
BAA AGM 2010 – Chairman’s report	Paul Grosvenor	33
<b>Discover</b>		
The PhraseBook project on the APL Wiki	Phil Last	37
Bring something beautiful	Roger Hui	40
<b>Learn</b>		
Treetables in q	Stevan Apter	42
A commentary on the formulator	Neville Holmes	72
Understanding font embedding	Kai Jaeger	83
Financial math in q: The price of bonds	Jan Karman	89
Punctuation and rank	Norman Thomson	96
<b>Profit</b>		
Backgammon tools in J: 2. Wastage	Howard Peelle	106
Functional calculation: 4. The year 1998	Neville Holmes	110
Simulating the Enigma	Keith Smillie	117
Subscribing to Vector		128

## Quick reference diary

14-15 Mar      Cologne, Germany      GSE Meeting and APL Germany

## Consultants

Note that the Consultants page has been transferred to the web:

<http://vector.org.uk/?area=consultants>

That seemed to be the right thing to do in order to make sure that the data provided is up-to-date.

## Dates for future issues

*Vector* articles are now published online as soon as they are ready. Issues go to the printers at the end of each quarter – as near as we can manage!

If you have an idea for an article, or would like to place an advertisement in the printed issue, please write to [editor@vector.org.uk](mailto:editor@vector.org.uk).

The big event of the autumn was APL2010 in Berlin, another encouraging revival of an international conference series.



*Eugene McDonnell*

Roger Hui delivered a eulogy in Berlin for Eugene McDonnell, stalwart of the APL Press, which we are pleased to reprint here. *Vector* readers will know Gene for his long-running column “At Play With J”, collected and republished by Vector Books, which he lived to see run into its second edition. We salute Eugene McDonnell’s life and work.

This issue includes several reports from Berlin, including an unapologetically contentious one from Graeme Robertson, and a review of 2010 from our chairman, Paul Grosvenor.

Published literature on the *q* language has been scarcer than hens’ teeth, with most practitioners swaddled in corporate non-disclosure agreements. Fittingly, it was Jan Karman in the Netherlands who broke the dam last issue with the first of his series “Financial math in *q*”. We have part two in this issue.

Even more exciting, we start a new occasional column “No Stinking Loops” by the legendary Stevan Apter. Apter, who works in a cabin deep in the woods of upstate New York, is one of the programmers Jeffrey Borror (author of *q for Mortals*) dubbed “the *q* gods”. His first article explores some sophisticated techniques for handling tree tables.



*Stevan Apter*

We have another rich offering of J material: Thompson’s “J-ottings”, Peelle on backgammon, Holmes on functional calculation, Smillie on cryptography and John McInturff on odd-order magic squares. Plus news of the BAA polyglot Phrasebook project and Kai Jaeger, recipient of this year’s BAA award for outstanding contribution to APL, on how to display web pages with APL to browsers with no APL fonts installed.

*Stephen Taylor*

# NEWS

# Remembering Eugene McDonnell

by Roger Hui



**Eugene McDonnell**  
**1926 – 2010**

Originally presented by Roger Hui at the APL2010 conference in Berlin, 13 September 2010.

When I met Eugene McDonnell in 1981, there were traces of humidity behind my auricular orifices.

Since my pronunciation may be a bit off, and since I don't want you to think that I thought of the witticism all by myself, let me show you where it comes from:

When I met Clark Wiedmann in 1968, there were traces of humidity behind his auricular orifices.

– Eugene McDonnell,  
Minnowbrook APL Workshop 1985,  
*APL Quote-Quad*, 16.3, 1986-03.

## Recreational APL

Magic Squares and Permutations

Spirals and Time

How Shall I Transpose Thee? Let Me Count the Ways

The Story of ◦

How the Roll Function Works

The Caret and the Stick Functions

The Point of No Return

Sauce for the Gander (or Adding a Vector to a Matrix)

Making a Calendar  
 1980 Year's Digits Problem  
 Puzzle of the Year 1980 (solution)  
 Pyramigram  
 Numbering Crossword Squares  
 Pyramigram (solution)

I first knew of Eugene as the "Recreational APL" editor. In those days, on receiving an issue of the *APL Quote-Quad*, I would inevitably and eagerly first turn to the "Recreational APL" column. Through these columns I learned that it was possible for technical writing to be erudite, educational, and entertaining, and through them I learned a lot of APL. Jeffrey Shallit, professor of mathematics at the University of Waterloo, recently wrote[1] of his similar experience with "Recreational APL".

Language Contributions  
 circle  $x \circ y$   
 signum  $x \vee y$   
 extension of  $x | y$   $x \tau y$  for negative  $x$   
 $\lceil y \lfloor y$   $x | y$   $x \tau y$  for complex  $x$  and  $y$   
 extension of  $x \vee y$   $x \wedge y$  as GCD and LCM  
 $0 = 0 \div 0$   
 infinity and infinite arrays  
 hook and fork

Later, I found out that Eugene was also a key contributor to the development of APL. I don't have time here to go into the details of all the language contributions, so I'll just touch on two.

#### The Story of $\circ$

$x \circ y$	$x$	$(-x) \circ y$
$(1-y*2)*0.5$	0	$(1-y*2)*0.5$
$\sin y$	1	$\arcsin y$
$\cos y$	2	$\arccos y$
$\tan y$	3	$\arctan y$
$(1+y*2)*0.5$	4	$(-1+y*2)*0.5$
$\sinh y$	5	$\operatorname{arcsinh} y$

cosh y	6	arccosh y
tanh y	7	arctanh y

Anyone who has ever used the circle function probably wondered about the numbering of the left argument  $x$ . The explanation usually hinges on the fact that  $x \circ y$  is an odd or even function depending on whether  $x$  is an odd or even integer, then goes on to explain what is an odd or even function. But the *real* explanation is as follows. In Eugene's own words in *The Story of  $\circ$*  [2]:

Actually, 1 and 3 were chosen first, more or less by accident, for the sine and tangent, along with 2 for the cosine function, by listing the functions in the order in which they were taught me in high school, and then the observation was made about sine and tangent being odd functions. The hyperbolic functions simply fell into place afterwards.

#### Extending APL to Infinity – McDonnell and Shallit, APL80.

- infinity
- infinite arrays

The other language contribution that I want to touch on was proposed by Eugene and Jeff Shallit in the APL80 paper *Extending APL to Infinity*. The proposal had two parts: infinity as a number, and infinite arrays.

NARS2000	J
$\lfloor / \quad ''$	$< . / \quad ''$
$\infty$	$-$
$\div 0$	$\% 0$
$\infty$	$-$
$2 * 3333$	$2 ^ 3333$
$\infty$	$-$

To date, infinity has been implemented in NARS2000 and J, denoted as a 'sideways 8' ( $\infty$ ) in NARS2000 and as the underscore ( $\_$ ) in J. For example, the identity element of minimum is infinity instead of a finite number; the reciprocal of 0 is infinity instead of an error; and 2 to a large power is infinity instead of an error.

i . 4
0 1 2 3

```
i. _
0 1 2 3 4 5 ...

+ / 3 ^ - i. _
1.5
```

To date, no APL has implemented infinite arrays. When they are implemented, you can do the following: The index generator function on  $n$  gives the natural numbers less than  $n$  ; the same function on infinity gives all the natural numbers. Having infinite arrays facilitates working with infinite series and limits of sequences.

```
p: i. _
2 3 5 7 11 13 ...

R=: + / (1 + i. _)^-s
E=: * / % 1 - (p: i. _)^-s

R = E
1
```

In J,  $p:$  is a function and  $p: n$  is the  $n$ -th prime, therefore  $p:$  on the index generator on infinity are all the primes. So the Riemann zeta function can be computed as

```
R=: + / (1 + i. _)^-s
```

and the Euler product of all the primes can be computed as

```
E=: * / % 1 - (p: i. _)^-s
```

That these are equal was shown by Euler in 1737 using high-school mathematics. The identity is one of the most beautiful things that humanity has to offer. (Suitable for presentation to the Galactic Emperor[3].)



### At Play With J

41 Columns in *Vector*

Written between 1993 and 2006

Published by  in 2009



Eugene remained active after he retired from paid employment in 1990. He wrote 41 *At Play with J* columns for *Vector* between 1993 and 2006. These columns have been collected together and published as a book. For the duration of the conference you can order this book from Lulu[4] at cost, or download the electronic version for free. (Thank you British APL Association and Vector Books, for this generous offer.)

```
eem=: 1926 10 18 ,: 2010 8 17
daynum eem
46310 76929
--/ daynum eem
30619

kei=: 1920 12 17 ,: 2004 10 19
daynum kei
44179 74801
--/ daynum kei
30622
```

Eugene wrote about calendar calculations in a “Recreational APL”[5] column. Using a descendant of those functions, we see that Eugene’s lifespan was 30619 days. For Ken Iverson, it was 30622 days. So Eugene and Ken, whose careers and lives are so intertwined, have one more connection.

I smile in my heart when I think that in heaven, Ken now has an ally in his debates with the Almighty on the finer points of language design. I positively chuckle at the thought that, in the extremely unlikely event that they are in that other, warmer, place, 0-origin indexing now has another advocate against the entity in charge.

## References

1. <http://www.jsoftware.com/papers/eem/anecdotes.htm#jos>
2. <http://www.jsoftware.com/papers/eem/storyofj.htm>
3. See page 40 or online <http://www.vector.org.uk/?vol=24&no=3&art=hui>
4. <http://www.lulu.com/product/paperback/at-play-with-j-%5Bedn-2%5D/6073726>
5. <http://www.jsoftware.com/papers/eem/qq101.htm>
6. Papers and Articles: <http://www.jsoftware.com/papers/eem>
7. Quotations and Anecdotes: <http://www.jsoftware.com/papers/eem/anecdotes>

## Industry News

### **Dyalog**

Although we did not manage to get any new releases out the door during this calendar year (v12.1 was released in the last quarter of 2009 and v13.0 will be out in Q1 of 2011), we have had a very busy year!

### **Version 13.0**

Version 13.0 of Dyalog APL will be Beta-testing in January and should be released in March 2011. As usual, it will be available in 32- and 64-bit versions under Microsoft Windows, AIX and Linux (and for Solaris on demand). V13.0 will break significant new ground on a number of fronts:

First (because it caused us the largest headaches), we have been working on adding support for 128-bit decimal floating-point numbers, implemented according to the IEEE-754 2008 (DPD) standard – with hardware support on recent IBM POWER and ‘z’ processors and software emulation on other platforms. The current 64-bit binary floating-point representation (corresponding to roughly 16 decimal digits) is sufficient for the vast majority of applications, but financial reporting involving multiple currencies can run into problems due to the limited precision (16 digits) and the fact that many decimal numbers cannot be precisely represented, leading to multiple small errors which can become significant if financial portfolios are sufficiently large. Version 13.0 allows the optional use of 34-digit decimal numbers, which avoid all such inaccuracies at the cost of slower calculations.

If you are willing to spend about \$10,000 on memory chips, and have enough coffee to get you through the time it takes to )LOAD it from disk, you can now have nearly 128Gb of workspace in memory on ordinary hardware. In version 13.0, most primitive functions are no longer subject to the limit of 2 billion elements per simple array that existed in earlier 64-bit versions. In version 13.1, we plan to remove the size limits for all primitive functions, upgrade our test machine from 32 to 128Gb, and revisit other strategies and limits in the interpreter that might come into play when workspaces grow extremely large.

For technical and educational users, we have added complex numbers, which are implemented as a pair of 64-bit binary floating-point numbers (we do not think there is much of a market for 2x128-bit decimal complex numbers). Our

implementation follows the extended ISO APL standard (ISO/IEC 13751:2001), and should be identical to that found in IBM APL2, SHARP APL or J.

Although APL applications tend to have a numerical focus, APL is also a terrific language for connecting things together; massaging, cleaning and redistributing data for use by APL code or indeed applications written in other languages. An important missing link has been integrated support for regular expressions. Regexp have become a tool that most young computer scientists, software engineers or hackers pretty much take for granted and miss when they move to APL. Version 13.0 contains our first system operators, one for searching and one for replacing using regular expressions as implemented by PCRE, one of the most widely used regular expression engines available today.

Version 13.0 also adds a number of good ideas pioneered by other array language dialects. *Take* ( $\uparrow$ ), *drop* ( $\downarrow$ ) and *squad indexing* ( $\square$ ) allow short left arguments (you only need specify what to select on leading dimensions, as in SHARP APL). We have implemented the functions *right*  $\leftarrow$  and *left*  $\rightarrow$ . We have also been inspired to implement a much simpler yet more powerful application profiling tool, after seeing Richard Nabavi demonstrate `□PROFILE` in a recent release of APLX.

The addition of the two new numeric data types mentioned above has forced us to revisit a number of algorithms, in particular *index of* (dyadic  $\iota$ ), *matrix inverse* ( $\boxminus$ ) and the *gamma* and *beta* functions (monadic and dyadic  $!$ ), with the result that they are not only significant faster (for all data types) but often produce more accurate results.

### Other projects

While the production of new releases of Dyalog APL still consumes most of our efforts, those of you who visited the successful APL2010 conference in Berlin in September will be aware that we have a number of other irons in the fire. The two most important ones are APL# ('APL Sharp') and RIDE.

APL# is the name of a new APL Interpreter which will be a native ('managed code') Microsoft .Net language. We published the first description of this new language at APL2010, and are inviting feedback from the entire APL community. The .Net framework implements a virtual machine which has significant benefits, such as being able to run APL# applications off a web page in the same way that Javascript applications can run without first being installed – but also significant drawbacks, like making it hard to implement the extremely simple and efficient data model that native APL interpreters provide.

APL# will not be a replacement for Dyalog APL, but a slightly different APL system that will allow APL to be used in ways which are not possible today. We hope to make the first prototypes of APL# available for experimentation (as a free download) around the middle of 2011, and keep APL# as an experimental system for at least a year after that before considering a commercial release.

The Remote Integrated Development Environment (RIDE) is the name of a new front end that we are building to provide an identical graphical development environment for APL# and Dyalog APL, on all platforms where either interpreter is available. This will allow us to unify the user experience for developers running APL or APL# locally – or on remote machines running Windows, UNIX and Linux – even when APL is running on servers which would otherwise not be accessible for debugging. Early versions of the RIDE will start to appear in the late spring or summer of 2011, but general availability is unlikely before the second half of the year.

You can watch John Daintree introduce the RIDE at APL2010 – and a number of other interesting presentations which we recorded at that conference – at <http://video.dyalog.com/APL2010/>.

### **Manpower**

We have recruited two more developers, and our good fortune in finding excellent developers has continued! Jay Foad joined the team in May and is now working on the core interpreter, and Liam Flanagan who came on board in September is working on .NET, GUI and other interfaces. 2011 will get off to a perfect start as we provide a little balance to the newcomers by welcoming an array language legend into the team: Roger Hui has been helping us with performance enhancements for some years; we are incredibly pleased and proud that Roger will be working for Dyalog in an almost full time capacity from January – with a little of his time fenced off to continue his work as the chief architect and implementer of the J programming language!

From the middle of November to February of 2011 Ronald Chan, the first winner of the Dyalog Programming Competition back in 2009, is braving the (record) English winter in order to be an intern at Dyalog during his New Zealand summer holidays. Ronald is a wonderful mathematician and C coder; in his first month at Dyalog he re-implemented matrix inverse and the *beta* and *gamma* distribution functions for all our numeric types, resulting in faster and more precise results all round! Over Christmas, Ronald has been playing with a model of an interface to the R programming language. We will be sorry to see him return to University to attack a PhD in February, and hope to see him again in the future!

In addition to new employees, we have been very happy to collaborate with a growing number of external APL consultants who have helped us develop and test code in 2010. Peter-Michael Hager has been applying his combined APL and cryptographic skills to help us design a comprehensive Cryptographic Library for APL, which will provide cross-platform access to fast algorithms for hashing, symmetric and asymmetric encryption using a variety of algorithms. We would also like to thank Kai Jaeger, Michael Hughes and Brian Becker, who presented recent work that they have done in collaboration with Dyalog on tools for Dyalog APL, at APL2010.

By the time this is in print, we hope to have hired one or two full time APL developers in order to be able to accelerate the development of tools and interfaces written in APL. Over the next few years, it is our ambition to augment the enhancements that we have been putting into the interpreter with a comprehensive set of application development tools for APL developers, and to work hard on sharing these with the APL community on our own web page, on the APL Wiki and other online forums.

Keep an eye on the Dyalog web page in 2011! If you have not already done so, subscribe to the new Dyalog forums at <http://forums.dyalog.com> – you do not need to be a user of Dyalog APL to do so. The Forums are getting livelier all the time and have become our primary channel for formal and informal announcements including FAQs. In 2011, we also expect to make increasing use of our Twitter and Facebook feeds as distribution channels for news.

## **Kx Systems**

Kx Systems, the leader in high-performance database and time series analysis, today announced that Guosen Securities Co. Ltd, one of China's top-tier securities firms, has signed an OEM license deal for its kdb+ database. The database platform will power the investment bank's algorithmic trading system, which will be embedded into Guosen's offering to its corporate clients.

kdb+ allows vast amounts of data to be accessed and processed with minimal latency. With a single format for both real-time and historical data, kdb+ provides performance and flexibility for high-volume, data-intensive analytics and applications.

Hanxi Liu, GM of Guosen Securities' IT department commented: "Choosing kdb+ was an important strategic decision for us. The sophistication of Kx's programming language and the addition of its calculation engine to our new algorithmic trading system provide us with a great competitive advantage and the optimal balance of speed and simplicity. Kx has made a serious commitment

to the Asian market, and its industry knowledge and superior solutions have made the company a serious contender in the local arena.”

Chris Burke, Director of Asia Pacific, added: “We are seeing rising data volumes across Asia, growth in algorithmic trading and increasingly stringent, data-intensive risk management processes. At the same time, companies are looking to drive down latency at all levels. The kdb+ platform was designed from the outset with this dual challenge in mind, and its combination of speed and flexibility has proven to be very compelling. We are delighted to be expanding our Chinese client base with such a well-respected name as Guosen Securities.”

This announcement follows the opening of Kx’s Hong Kong office two years ago to support its locally-based international clients, as well as the domestic financial community. Kx’s ‘Growth via Partnership’ strategy with First Derivatives and Symagon has been extremely successful, resulting in several new relationships and the expansion of global clients into the Asian market.

### **Optima Systems**

Seems like a very long time since we last communicated and much has happened.

Firstly our prime customer has, after many years, made us strategic to their business. This makes a huge difference to us and places additional responsibilities at the same time. So why their change of heart? Well, put simply, using APL in the way we all love to use it and by avoiding administration, committee meetings and the usual ‘large organisation drag’ we have been able to show that systems can be delivered on time and on price and that this performance can be relied upon.

Secondly we have started to build a new software model in APL to deliver time series clinical data in new and intuitive ways. Whilst our solution is not pure APL, all of the heavy-duty analysis and grunt work certainly is. We are currently experimenting with new (to us) technologies to deliver the final user interface and hopefully provide the necessary wow factor. I hope to be able to demonstrate some of this work in future APL conferences.

We attended the Berlin APL 2010 conference and enjoyed it very much. It was very refreshing to see so many active APLers in one place and, more importantly, so many young faces. As always the conversations were fast and furious and I feel sure that the world was ‘put to rights’ many times over.

I remain unsure where 2010 has gone but very much look forward to 2011. Much is going on and much more remains to be done. Another busy year beckons.

## IBM

The IBM APL Products and Services group is pleased to announce Service Level 17 for Workstation APL2 for Multiplatforms Version 2.0. This new service level was made available on November 11, 2010 for all Workstation APL2 platforms (AIX, Linux, Sun Solaris, and Windows.)

Along with fixes for customer-reported problems, Service Level 17 contains several enhancements.

In the interpreter, the performance of the *find* primitive is improved for right arguments of rank 2 or higher. When running under the Calls to APL2 facility, the execution stack is now available to the calling program on error. In the DEMOJAVA workspace, the XML2APL function has been updated to support DTD addresses, and a new function, JAVA\_PRINT, prints APL objects using Java print services.

On Windows, a session manager system option has been added to control the workspace size of watch windows, and the line number popup window in the object editor has new choices for 'Trace this line' and 'Stop this line'. New key shortcuts include Ctrl+Double-Click to mark a token, Ctrl+Shift+Double-Click to localise or unlocalise a name, and Alt+PageDown to move the current line to the top of the window.

Further information on the new facilities will be found in the updated APL2 User's Guide and on-line help after installing Service Level 17. A complete list of fixes included in this level will be found in the updated README (Unix systems) or Service Information (Windows systems) file.

The APL2 Service Level 17 download is available to customers with a Software Maintenance contract, through the Support link at the APL2 web site:

<http://www.ibm.com/software/awdtools/apl>

# A personal view of APL2010

by Graeme Robertson

How should APL deal with multicore machines? Should a programmer be able to assign a thread to a CPU? Or should the interpreter automatically distribute a task over a number of CPUs at its own discretion? Or something else? Questions like these permeated APL2010. Apart from the main sessions, there were four parallel streams and since an individual could not distribute himself over all streams, he had to choose. This brief summary of the first day reflects one individual's choice of sessions.

Roger Hui gave an interesting account of how J deals with the concept of infinity and he successfully showed its potential value to APL programmers.

Dr Scholz argued enthusiastically for the need for programmers to tune their applications to make the most efficient use of multicore CPUs and graphics card GPGPUs, and he made a related case for functional programming without side effects by the removal of imperative programs (read niladic functions) and of state data (read global variables). 'Maximise frames and minimise cells' in your programs is the way to think about extracting the most from parallel processing, he suggested.

Helmut Weber pointed to the fundamental limits of faster chip technology with theoretical limits to the number of threads per core and the number of cores per die being nearly reached. With paramount future demands being measured by performance per milliWatt, he urged multicore and hybrid designs with single purpose function elements for high-spec robustness.

David Liebttag described some recent advances in APL2. As well as the goal to produce a workstation with the same functionality as mainframe APL2, he outlined new socket technology, the ability to call APL from other languages, new data type facilities and a new parallel processing each operator. He ended with a comparison of IBM graphics through the years, culminating with JAVA graphics.

Brian Becker afforded hands-on experience of a Stand Alone Web Service, the SAWS framework, written in Dyalog 12.1. It does not require IIS, only an open port on the server. One can then communicate very effectively under APL with web services on a client machine.



Morten Kromberg introduced APL# which is a new functional scripting language intended for deployment with Silverlight on a client machine. The language incorporates strong data typing, including a rank-zero string data type. Functions and operators are written in dynamic programming style with a new type of header line and new symbols for *current namespace* ( $\boxplus$ ), *shallow copy* ( $\ll$ ), *new namespace* ( $\boxtimes$ ) and a *rank* operator ( $\ddot{\circ}$ ). The timeframe for releasing APL# is one or two years.

Bob Smith chaired a panel discussion on the target form of APL by 2020. Issues discussed included: APL's lack of visibility, the evident lack of experimentation in traditional computer science, promotion of arrays to objects, rationalising APL by removing anomalies, and disappearance of the APL character set obstacle.

And this was just the first day of four fascinating, entertaining and instructive days of conference at the Technical University of Berlin, the most fascinating talk for me being the account by Professor Zuse of the unquestionably heroic efforts of his father, Konrad Zuse, to invent, construct and protect through the war, the first automated programmable computer ever. It was a truly amazing story!

I went to APL2010 because I wanted to meet some of the many friends that I have found in the APL world over the years. I also went in order to learn something. Not particularly about parallel processing, which I hope will be incorporated behind the scenes without too much effort on my part by way of some primitive operator such as *spawn* that will allow us APLers to harness multicore hardware effortlessly; not in order to learn some new languages such as APL# and WPF and Silverlight that one day might enable us to empower a client to talk fruitfully to an APL server; not even to learn about the latest APL applications and supportive hardware. None of these. I went with the principal purpose of learning how to write an application immediately on the Internet platform, using raw ASP.NET if needs be but preferably using a thin cover function like  $\boxtimes$ WC provided by an APL vendor. I didn't find anything like that.

The internet browser is now the obvious platform for new applications just as the PC was in the 80s. By the early 90s it was possible to access Microsoft Windows objects such as Forms and Buttons via  $\boxtimes$ NA calls to the Windows API DLLs. This was difficult without significant knowledge of C programming and very few people succeeded until Dyadic and other vendors supplied cover functions such as  $\boxtimes$ WC, whereafter success was manifest and ubiquitous.

I am pleading for a thin cover over Microsoft .Net that will make it easy to program rich APL applications on the cloud, thus making them immediately

accessible to any browser in the world. I know it is possible to do this now using ASP.NET, but it is far too difficult for me to do well.

Ken Iverson invented a powerful notation with very few rules and an extensible collection of profound functions such as `+` and profound operators such as `/` whose combination according to two simple rules led all the way to consolidated reports on the desks of chief executives of many of the largest corporations in the world. This was no coincidence.

Ken rightly strongly objected to the unnecessary new rules in APL2. When I proposed direct function definition at Dyadic in 1995 with constructs such as `3(ω-α)4` or `(ω[⌈ω])1 3 2` it was more or less ignored. No new syntax rules were invoked, only a simple interpretation of  $\alpha$  and  $\omega$  which had been understood for many years in the APL literature. On the other hand, dynamic programs as later introduced by Dyadic introduced lots of new rules.

Every time a new rule is introduced into APL I turn off. APL# introduces new rules all over the place and I don't like what I see. I used to agree with just about everything Ken Iverson said, even when he criticised the irrational parts of APL itself. I was convinced by his simple argument that APL2 introduces new rules unnecessarily and makes APL more difficult. Nowadays so many new rules have been introduced that APL is becoming as difficult as any other language and the pure elegance and simplicity of the original conception with essentially just two or three fundamental rules is being lost.

Windows Presentation Foundation, well presented by Michael Hughes, is interesting but, like APL#, it seems to be a solution for a distant future rather than the solution for today that I am begging for. ASP.NET seems to me to be the current way forward. Dyalog APL can talk to it, with difficulty. I think that an APL vendor could put a cover on this quite easily and give us a way of writing applications on the modern application platform, the browser window, but I heard nothing encouraging at APL2010, from Dyalog, APL2000 or APL2, nothing that gave me confidence that an APL programmer who does not have the time or inclination to learn a new language will be able to write serious applications on the cloud any time soon. This was exactly the situation with Microsoft APIs and Windows before Dyalog introduced the beautiful `⌈WC` solution.

The beauty of APL is its fundamental simplicity. Without that it risks sliding into oblivion with a hundred other corrupted languages that once were used to communicate effectively and profoundly between humans or between humans and machines. How exactly APL manages to add one matrix to another is not of any great interest to most APL programmers because they are thinking at a

higher level and ask most of all that the interpreter will free them to think abstractly.

When I suggested rewriting an APL interpreter based on the algebraic foundation of GiNaC or the like [see *Vector* 20.1 (2003) 132-142], the only response from Dyadic was that too much had been invested in the current interpreter ever to consider starting again from scratch even if it meant getting infinity and many other mathematical goodies for free, and yet now we have a proposal to do exactly that with APL# – but this time adding all sorts of new rules, getting no more (perhaps less) significant functionality, while at the same time abolishing canonical forms and even the concept of the workspace. I can't agree with that.

# Notes from APL2010

by Devon McCormick

This report first appeared on the NYCJUG pages at the J Wiki.

## **Multi- and many-cores: array programming at the heart of the hype!**

Dr habil. Sven-Bodo Scholz (University of Hertfordshire, UK) gave an enthusiastic talk about his work with the functional array-programming language Single Assignment C (SAC)[1].

The current SAC compiler, called `sac2c`, is developed and runs on UNIX systems. Binary distributions are available for the following platforms:

- Solaris (Sparc, x86)
- Linux (x86 32bit, x86 64bit)
- DEC Alpha
- Mac OS X (ppc,x86),
- netBSD (under preparation)

Unfortunately for us in the Windows community, these are the only offerings.

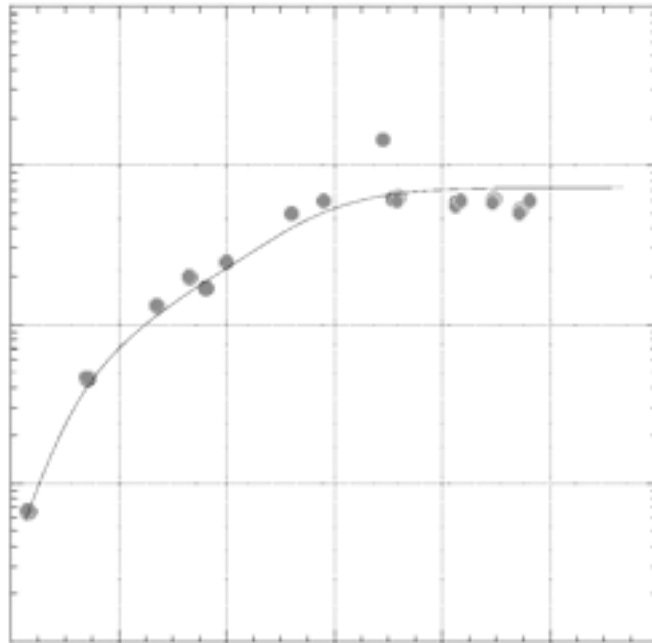
He had a somewhat amusing story to tell about one of his students who had a paper rejected by a parallel-programming conference because it was based on using an array notation and this was said by one reviewer to be too unconventional – a more proper paper would be to have written about discovering parallelism within loops. It was considered a kind of ‘cheating’ to avoid this problem by using a notation that renders this consideration moot.

Sven-Bodo’s work on SAC targets numerically-intensive programs, particularly those involving multi-dimensional arrays. He claims to get performance comparable to that of hand-optimized Fortran. Some of his exciting recent work targets GPGPUs[2].

## **Multi-core and hybrid systems – new computing trends**

IBM’s Dr Helmut Weber gave a very thoughtful, informed, and forward-looking keynote exploring contemporary trends in performance improvements with the current generation of multi-core and hybrid systems. He started by outlining the physical limitations driving the new directions of chip architecture, which are:

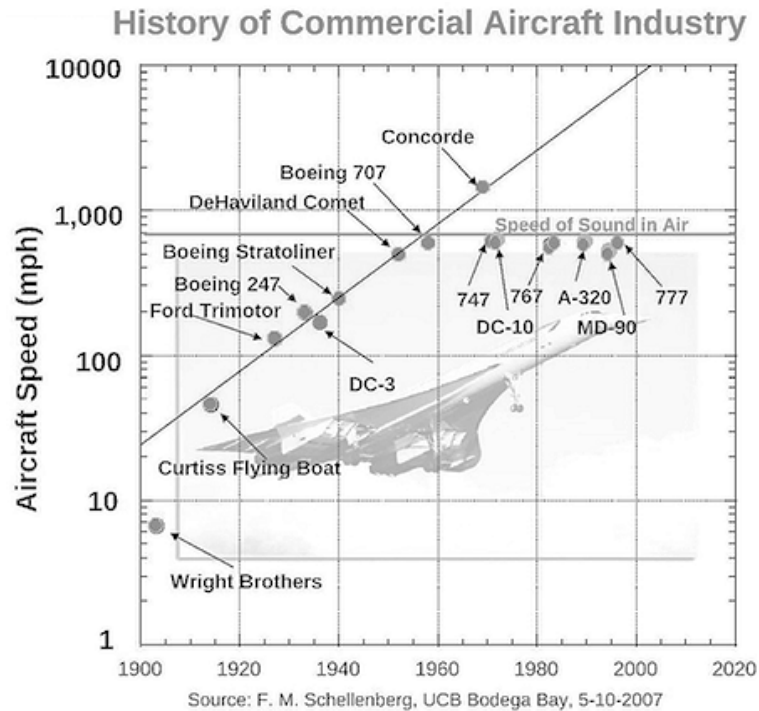
- higher n-way
- more multi-threading
- memory access optimization



*Fig.1 What does this plateauing graph represent?*

He showed a misleading graph without labels and asked us to guess what it represents. Of course, most people thought it might show processor performance over time.

Once the labels were revealed, we saw this was not the case. However, Dr Weber was making a point about the natural evolution of performance of any technology over time – that there will eventually be a plateau after a period of exponential growth for a given technology without some fundamental change.



*Fig.2 What the plateauing graph represents*

Some of the technology trends he sees coming up are:

- increasing CMOS density – from current 22 nm to 15 nm or better
- more instruction-level parallelism
- more cores per die and threads per core
- perhaps leading to many small cores – microservers
- I/O link bandwidth growing to 100 Gb
- optical links becoming a commodity

Other trends include power consumption limiting future DRAM performance growth and the increasing prevalence of solid-state disks. Further in the future, he sees increasing use of “storage-class memory” (large non-volatile memory), which could lead to a new memory tier. Some recommendations on how to compute more efficiently from a perspective of power consumption:

- Run many threads more slowly for constant power and peak performance
- SIMD power benefit proportional to width but diminishes for very wide
- Data movement management using reduced cache and simpler core design with multiple, homogenous processors
- Add hardwired function specialisation to allow bandwidth targeting

He also had some ideas about how software can improve to be better integrated with hardware and work within a power budget. Part of this improvement will make use of hardware function accelerators, perhaps as part of special purpose appliances. A suggested plan for building these appliances includes adding open-source technology to off-the-shelf hardware and using ‘builder tools’ (e.g. rPath) to tailor software stacks to take advantage of this, along with FPGA and ‘IP-cores’ to improve performance for key functionality. Example appliances include Tivo, the IBM SAN volume controller, and the Google Search Appliance. Some suggestions to help programming models take advantage of these trends:

- support efficient parallelism
- be at higher level of abstraction
- match the mental model of the programmer

This leads to the conclusion that the time is ripe for languages to adapt to the multicore future. Dr Weber suggests that most of these efforts aim at C derivatives, which restricts the number of programmers able to exploit parallelism by losing the productivity gains of languages like Java. This prompted IBM to develop X10 which is an adaptation of Java for concurrency.

In addition to the better-known mature parallel models of MPI and OpenMP, he mentions OpenCL, which has support for accelerators like GPGPUs. He also showed off some new IBM processor technology: a 5.2 GHz Quad-core processor with 100 new instructions to provide support for CPU intensive Java and C++ applications. It also includes data compression and cryptographic processors on-chip.

### **Awards to winners of the Dyalog Programming Competition**

The talks by two of the winners were refreshing and it was valuable to see how novices view their first experiences with the language. One thing that puzzled me was that both speakers criticised the lack of type handling in APL – to me, this has always seemed to be a positive feature of the language. I don’t understand how this kind of restriction provides a benefit that outweighs the limitations it imposes. Can anyone explain? Or better yet, provide a short program illustrating the desired behaviour and expound on why this would be helpful?

### **References**

1. [www.sac-home.org](http://www.sac-home.org)
2. [www.sac-home.org/publications/RollSchoJoslStav.pdf](http://www.sac-home.org/publications/RollSchoJoslStav.pdf)

# APL 2010 – Berlin

by Mike Hughes

Berlin is a great city – it was my first visit and I found it light and airy with its wide tree-lined streets. The river meanders through the middle and some of the architecture is stunning. It proved a good backdrop to what I found was an interesting conference, even if the wireless LAN failed to work.

The conference itself was well attended – I think there were about 150 attendees in total. I was very pleased to see more hair colour than grey (or even hair!) than I was expecting. There were many young faces in the crowd and they weren't all German students, who were given free admission. A good idea from the organisers.

I felt the whole conference was generally very forward looking, Dyalog, APL2000 and IBM put on a good show with their various stands in the area where the coffee and food were served. There seemed plenty of interest around each stand.

There were one or two talks on the history but the majority of presentations seemed to be looking forward. I sensed a real feeling of renewal and optimism. The theme of the conference was parallelism and there were many talks on different aspects of this. It really seemed as if APL was poised to take full advantage of the new hardware as it becomes available.

There were presentations from Dyalog that ranged from a new universal IDE to interact with any and all Dyalog sessions on any platform. This will solve problems such as: not being able to interact with a Dyalog session when running it with IIS7; allowing a session to be attached to a remote Dyalog process for remote debugging on any platform supporting Dyalog. At one point I think John Daintree had a Unix, a Mac and a Windows session alongside each other – it looked very exciting from a developer's point of view.

Brian Becker gave a workshop talk on SAWS and Web Services. Morten Kromberg presented a paper on *peach* which attempts to look at parallelism at the operator rather than function level – so trying to emulate *each*, *outer product*, *rank* and namespaces (dot notation) in a cross-processor model.

APL2000 demonstrated version 10 of APL+Win and showed how WPF and .Net worked with APL+Win. I was lucky enough to get a personal demonstration from Joe Blaze on the ease with which APL+Win can work with these.



It was good to catch up with old friends and to discuss issues with like-minded people. I found this, the most useful and enjoyable part of the conference. The accessibility of the various suppliers, various developers and ideas arising from some of their work. I heard many voices trying to get us to try and fit in better with our IT neighbours, perhaps the tide will turn to better cooperation within and outside the APL community.

Paul Grosvenor gave a good insight into the art of selling bacon to a vegetarian, using it as an example of thinking on your feet and being prepared, in an interesting talk on how best to sell APL and yourself.

There were language enhancements discussed such as mask and mesh, regular expression, unifying Dynamic and Traditional functions and a new dialect APL#. Dyalog and APL2000 are clearly both working hard to make APL capable of living in the new .Net Object Orientated world without losing the core benefits of APL.

Overall I found it a good conference, one that gave me hope that APL is not only going to survive in the future but that it will also be a great and better recognised language in 2020.

# APL2010 – a brief introspection

by Brian Becker

Reprinted from Catherine Lathwell's blog *Chasing Men Who Stare At Arrays* [1]

Having been on the periphery of the APL community for the last several years, I found it refreshing to attend an APL conference once again. It was good to see old friends and associates and meet new people as well.

## Prominent theme

Parallel computing... perhaps computing hardware is finally catching up with APL. There were at least six sessions and many more informal discussions about how APL is natural fit for multi-core, parallel execution and about work and research that is actively taking place. It seems there is great potential, but not without challenge – for instance, while multiple fast cores are available to execute code they are still dependent on relatively slow memory for data.

## Interesting thing in the works

Dyalog's APL# – a new, functional, array-oriented scripting language which aims to compete with Python, Ruby and JavaScript for technical and computational applications, and make it possible to deploy APL code in places where Dyalog APL cannot go. (I stole this from Morten Kromberg's slide because I couldn't say it any better myself.)

## Something cool I plan to use

WPF (Windows Presentation Foundation) – want a slick user interface? WPF seems the way to go. It gives you virtually unlimited control over the look, feel, and functionality of your user interface. That's the good news... the bad? WPF books are *very* thick. Thank you Michael Hughes and Joe Blaze for your WPF presentations.

## Something cool I want sooner than later

Dyalog's RIDE (Remote Integrated Development Environment)... imagine being able to connect to an APL session and debug it from your iPhone. RIDE allows you to connect to a Dyalog session from a web browser on almost any client platform. Very cool stuff.

**Most entertaining session**

Dyalog's Jay Foad's presentation on juggling patterns augmented by Jay himself juggling up to 5 balls. This raises the bar for future presenters... what's next? Unicycles?

**Something I miss from past APL conferences**

Interprocess Systems buttons... they had the most amusing captions like "Greek Looks Like APL To Me" and "Another Brilliant Mind Ruined by APL". Though, Dan Baronet's "Ich Bin Ein APLer" tee-shirt was a welcome sight.

**Interesting banquet**

The banquet was held at Mercedes World at Salzufer. Imagine eating a buffet dinner surrounded by every imaginable model of Mercedes Benz. That, plus a 4-storey climbing wall. Now if they only gave out cars as door prizes...

**Most encouraging aspect**

The first and second place winners of the 2010 International APL Programming Contest, Ryan Tarpine and Mstislav Elagin... Both gave thoughtful and candid presentations about their experiences and impressions with APL, pointing out the aspects that they liked and those they didn't. They shared a true enthusiasm and appreciation for the expressive power of APL. I found it poignant when Ryan spoke about the "beauty" of an APL expression, something I felt when I first encountered APL 35 years ago. Ryan and Mstislav represent APL's future and we need to reach out to others like them.

**Final impressions**

There is active use and interesting development of APL and array-processing technology.

We need to expand beyond our own community – there were several interesting papers and presentations that would play equally well, perhaps even better, at non-APL conferences. We need to get the word out that APL is a viable and vibrant technology.

**References**

1. <http://lathwellproductions.ca/wordpress/2010/09/22/apl2010-a-brief-introspection-guest-brian/>

# Best APL2010 conference presentations

by Vibeke Ulmann

Reprinted from Catherine Lathwell's blog *Chasing Men Who Stare At Arrays* [1]

Based on those I managed to attend.

## **Most laughs**

Prof Dr Ing Horst Zuse, "The origins of the computer"

## **Best entertainment value**

Jay Foad (Dyalog), "An Interpreter for Vanilla Siteswap"

## **Presentation with highest sex-appeal (I want one of those!):**

John Daintree (Dyalog), "Taking APL for a RIDE"

## **Most "wake up we're moving ahead" presentation**

Morten Kromberg, John Scholes & Jonathan Manktelow (Dyalog), "APL#"

## **Most fantastic application presentation**

Lars Wenzel (Fujitsu Sweden), "Volvo application"

## **Best up-coming APLers presentation**

Mstislav Elagin & Ryan Tarpine, winners of the programming contes.

## **Most intriguing and thought-provoking on parallel/multicore**

Sven-Bodo Scholz (University of Hertfordshire)

## **Best selling proposition for APL**

Paul Grosvenor (Optima Systems), "Making Money with APL"

## **Most progressive and 'on the ball' APL vendor with resources for R&D**

Dyalog Ltd

Berlin is a nice city and I would like to go back sometime and play tourist. Great restaurants and good music venues. Went to A-Trane (Larry Goldings trio) and Quasimodo (Blues rock with German 5 piece blues band called 'five' & bought their CD *Five in the Kitchen*. Great food at Ottenthal – best pudding I've ever had: Poppyseed & Lavender Sabayon).

[http://lathwellproductions.ca/wordpress/2010/09/22/apl2010-a-brief-introspection-guest-brian/Ready to post](http://lathwellproductions.ca/wordpress/2010/09/22/apl2010-a-brief-introspection-guest-brian/Ready%20to%20post)

# BAA Annual General Meeting 2010

by Anthony Camacho  
*secretary@vector.org.uk*

Minutes of the British APL Association AGM 2010 held at The Albion, 3 New Bridge Street, London EC4 on Friday 21 May 2010

## **Minutes of AGM 2009**

The minutes had been published on the web site and were taken as read without correction.

## **Report from the Chairman (Paul Grosvenor)**

Paul said that plans were being made for a possible conference next year.

The London meetings were going well. An away day is to be held on 23 July.

We need more people to join the committee and willing to do something to help. Three members are standing down from the committee – Anthony Camacho (we need a secretary), Ray Cannon (we need an activities officer) and Stephen Taylor. Stephen will continue as webmaster but we need someone to edit *Vector*. Paul thanked the three for their work over the years.

Stephen Taylor is unable to be with us (Paul thinks he is working in Belgrade) but there is plenty of material in hand and most of the work is done on the next issue of *Vector*.

Phil Last later volunteered to take over as Activities Officer.

## **Report from the Treasurer (Nicholas Small)**

(Including report from membership secretary.) Nicholas said there is not much to report since last year. One issue of *Vector* has been produced and we have money for two more before we need to ask for further subscriptions.

We have 256 paid-up members plus some Japanese members who have not yet paid and this compares with about 290 at this stage of 2008 (this comparison was not made last year).

### **Committee for 2010-2011**

Paul suggested the following should be next year's committee and Auditor:

	<b>2009-2010</b>	<b>2010-2011</b>
Chairman	Paul Grosvenor	Paul Grosvenor
Secretary	Anthony Camacho	(vacant)
Treasurer	Nicholas Small	Nicholas Small
Vector editor	Stephen Taylor	(vacant)
Vector Webmaster	Stephen Taylor	Stephen Taylor
Activities	Ray Cannon	Phil Last
Education	Alan Mayer	Alan Mayer
Projects	Ian Clark	Ian Clark
Auditor (not on committee)	Chris Hogan	Chris Hogan

This was proposed by Ray Cannon and seconded by Mike Hughes and approved without objection.

### **Questions**

There were no questions to officers.

### **Other business**

#### **Outstanding achievement award**

The outstanding achievement award was made to Kai Jaeger for his promotion of APL. Kai thanked us and said that the most effective aid to promoting APL he had seen in the last 20 years was the video made by John Scholes.

#### **Catherine Lathwell's film project**

Paul felt that we couldn't justify spending the money we had control over on Catherine Lathwell's project, much as we applaud it and wish it well. This was clearly agreed by those present. Anyone who can help in any way is urged to get in touch with her.

#### **Our funds with BCS**

Paul circulated a letter from the BCS which was uncompromising. They will pay no more bills incurred by us and quote their rules in justification. They are holding something like £14,000 of our money.

We discussed what could be done to bring pressure on them to return it to us. Paul felt that we could not afford to take the legal approach. It was suggested that sustaining members write to the BCS at the top level.

Paul said that the BCS were unable to find anything in writing about the terms on which we had joined or even a date.

Anthony said that we had joined after APL86 had made a large profit. There had been a spot of bother because our chairman was Philip Goacher who was also an employee in charge of a BCS subsidiary that arranged various events. In that role he had underwritten the APL86 conference. Then we discovered (about three months before the conference) that he had been fired from that role (although he still had a desk at the BCS) and we went to the BCS asking them to confirm the underwriting (of which there were no written records). They did so and after the conference, almost as a 'thank you', the committee was persuaded by Philip to join the BCS as a sub-group.

The rules the BCS quote in their letter were made a good deal later than 1986 so they can hardly apply.

Paul Grosvenor agreed to pursue these suggestions.

Anthony Camacho, Hon Sec 25 May 2010

# AGM addendum by Paul Grosvenor

## 5 October 2010

by Paul Grosvenor

Since the AGM was held a number of people have come forward to offer their services on our Committee. As a result I proposed a new structure to the committee which was seconded by Peter Merritt on 5th October 2010. Apologies to all for this rather strange procedure. I am pleased to be able to announce therefore that we once again have a full committee for 2010/2011 which is made up as follows;

	<b>2009-2010</b>	<b>2010-2011</b>
Chairman	Paul Grosvenor	Paul Grosvenor
Secretary	Anthony Camacho	Peter Merritt
Treasurer	Nicholas Small	Nicholas Small
Vector editor	Stephen Taylor	Stephen Taylor
Vector Webmaster	Stephen Taylor	John Jacob
Vector Production Manager	n/a	Kai Jaeger
Activities	Ray Cannon	Phil Last
Education	Alan Mayer	Alan Mayer
Projects	Ian Clark	Ian Clark
Auditor (not on committee)	Chris Hogan	Chris Hogan

My sincere thanks go out to all those people who have volunteered to help out, old or new. Particular thanks go out once again to Anthony Camacho, Ray Cannon and Stephen Taylor for all of their work over the years and the benefits we have all enjoyed as a result. n.b. Stephen, you might notice, has not quite got off the hook yet!



# BAA AGM 2010 – Chairman's report

by Paul Grosvenor – October 2010



*Paul Grosvenor, BAA's old and new chairman*

As I write I detect a very positive momentum beginning to build up once more. The vendors seem to have a range of exciting initiatives that are ongoing and the market seems to be quite buoyant generally. As always our big challenge is to keep on waving the APL flag and making sure that even more people know we are here.

I am thinking about the possibility of another conference next year along similar lines to BAPLA09 but this time (you've guessed it) BAPLA11. It all rather depends on time and sponsorship as this time I would want to get the costs right down and try to make it three days not two. Anyhow, watch this space and we shall see what comes.

Our AGM this year at the Albion went well and attracted some 25 or so people. Thanks to our sponsors (Dyalog, MicroAPL and Optima Systems) for providing the food and drinks. The venue turned out not to be ideal but nevertheless allowed us to undertake the formalities of an AGM and have talks from two invited speakers afterward.

This year for the first time in many years we had three long standing members of the committee stand down. Anthony Camacho (secretary), Ray Cannon (activities) and Stephen Taylor (Webmaster and Editor) all moved aside to allow for new blood to move in. I personally cannot remember a time when Anthony and Ray were not on the committee and rumour has it that their involvement goes back almost to day one! Stephen was a more recent member of the team but has invested much time into *Vector* and our website; time he no longer has. I

hope you can all join me in offering our thanks for their combined efforts over the (many) years and hope they will keep in touch. Stephen in fact remains on the Committee looking after the *Vector* Editor role. Since our AGM, Peter Merritt, Kai Jaeger and Phil Last have joined the team so now we are once again back up to full strength. Thank you one and all. Full details of the new Committee can be seen elsewhere within this issue of *Vector*.

Very little progress has been made with the BCS regarding our withheld funds. Some £14,000 remains within their grasp and they still consider it their money. As the AGM minutes point out we do not agree with this position and will try over the coming months to get our sustaining members to apply pressure on them. The decision to break away from the BCS was made in the knowledge that this situation was likely to arise but it should be pointed out that even without this money our existence is not threatened.

Since our BAPLA09 conference last year we have been in some lightweight conversations with Catherine Lathwell regarding the documentary film she wants to make about APL. Our responses to her have not been all that satisfactory thus far as without the money from BCS we are not in the position to be able to offer her very much towards this project. Nevertheless I do intend to keep a dialogue with her going and will report back to the membership as matters transpire.

The BAA London group have managed to keep on track with their meeting each month and I hope that this continues on. Phil Last will be announcing each meeting through various forums including `comp.lang.apl` so if you are in the area, please feel free to drop in on them. The group held an 'away-day' meeting on 23 July at Dyalog's new offices in Bramley which was a great success. As always discussions were fast and furious (15 APLers, 15 different solutions).

The *Vector* production team pulled out all the stops to produce Volume 24 Nos 2&3 in time for the Berlin APL conference. We now hope to keep the *Vector* production process under control and outputting regular issues throughout 2011.

This year we decided to award a new Outstanding Achievement Award. The award is designed to acknowledge the efforts of an individual, or organisation, within the world of APL where a particularly high level of achievement has been made. This year I was very pleased to be able to award it to Kai Jaeger for his work promoting APL and in particular for his efforts building the APL Wiki. Kai has spent more than just a few hours putting together the Wiki and has tried to make it as vendor-independent as possible. Today the Wiki contains a wealth of

information and could be an invaluable starting point for the APLer both old and new. As always with these developments they rely upon content and so I know that Kai would very appreciate anyone who would be prepared to add a page or two for him.

It should be noted at this point that on receiving his award Kai was lost for words. An event that I do not think has happened before or since.

Please go on line and take a look or even better, create an account and add to the content at <http://aplwiki.com/>



*Congratulations, Kai!*

Thank you once again to all our sponsors for contributing towards the cost of this award. We hope to be able to make a similar award next year.

The award reads:

***The Outstanding Achievement Award  
For Services to APL  
Presented to Kai Jaeger  
21st May 2010***

Finally, many of us will have just returned from the Berlin APL2010 conference. It was an interesting event with loads of good stuff going on. Almost as importantly it was refreshing to see 150 plus APLers all in the same place. We were reminded that 2016 will be the 50th anniversary of APL so I guess we should start planning for something a little bit special.

# DISCOVER

# Introduction to the PhraseBook project on the APL Wiki <http://aplwiki.com/PhraseBook> by Phil Last

Various collections of APL ‘idioms’ have been published over the years and the possibility of a cross-vendor phrasebook has been mooted a number of times.

About a year ago Dick Bowman suggested to the BAA London group in one of its regular monthly meetings that we might consider taking up the mantle of a pioneering effort that had been made some twenty years before to produce a phrasebook for as many APLs as would cooperate. After gauging initial interest Dick brought along an inch-thick sheaf of papers that were the only tangible evidence of that other project.

Chris Hogan scanned the whole lot and placed them on a website, a wiki, provided by Kai Jaeger.

Ellis Morgan then added cross-links and new pages to the wiki where phrases within a category were incorporated into a table. Provision was made for each phrase to be shown in a different version for a number of different APL implementations.

At a later meeting we made the decision to take a slightly different approach using the inbuilt wiki facilities to link individual pages, each of which centres on a single phrase. The phrase is accompanied by a full description including alternative codings but the main phrase is coded in such a way that it should work on as many platforms as possible including the *de facto* standard of APL2 and as far as possible the ISO Extended APL Standard[1].

A page template assists in the making of a new page. This is complete and flexible enough to allow individual contributors to build a new phrase page within a short time, that will be published immediately.

The template includes headings for examples, how not to do it, conforming variants, specialities, compatibility and test cases.

The author succeeded in creating a new page from scratch well within a half an hour albeit without any test cases. He was also able to leave a preformed “Page under construction” notice on the page temporarily to excuse this shortcoming.

The hope is that the standardised headers will enable someone at some time to write software to link to the wiki as an extra resource for his or her own training material.

Wiki categories are used to group pages that share some arbitrary feature. A page is added to a category simply by naming the category at the bottom of the page. Clicking on a category name brings up that page with a list of all pages that name it. The phrasebook project has many categories but until there are many more phrases the most useful so far is `CategoryListAllPhrases` to which all phrase pages belong.

The phrasebook project differs in several aspects from its predecessors such as the FinnAPL library:

## ⌈IO

The phrasebook uses 0 as the default value of `⌈IO`. The main reason for this decision is that dealing with `.NET` can be quite confusing otherwise. That does not mean that a phrase cannot use 1 but `⌈IO` should then be set explicitly.

## APL2 syntax

APL2 is considered to be the general standard for modern APLs. However, this is of significance only in Dyalog, which is syntactically different by default, at least for the time being. One can force Dyalog closer to the APL2-syntax by setting its `⌈ML` (Migration Level) to the highest possible value, which is three at the time of writing.

## Testcases

Every phrase in the phrasebook is expected to come with proper test cases. The test cases not only ensure a high level of code quality, they also prove useful for compatibility reasons. Imagine someone writing a certain phrase in, say, Dyalog which you would like to use in, say, APLX: by copying and executing the test cases in your interpreter you can make sure that the phrase is doing what it is supposed to do.

Note that there is a button *Execute test cases* available on each phrase page. You may wonder what's happening when you press this button. Well, quite a lot:

1. The link is sent to a particular port on the APL Wiki server where not Apache but MildServer[2], a web server written in Dyalog APL, is listening.
2. MildServer then analyses the page, extracts the code and executes it. The rules are explained on the APL Wiki[3].
3. MildServer then creates an ordinary HTML page, reporting either success or failure, which is then sent back to the client. To some extent this is arguably a dangerous thing to do; bad guys can easily write test cases which could bring MildServer down, or keep it busy forever. But then there are no bad guys in the APL community. (Phew. *Ed.*)

## Contributions

If you are tempted to contribute to the project but feel a little depressed after looking at the template offered: try to take the test cases as far as you can and then save the page. One of the administrators will sooner or later come along and fix any problems. By comparing your version with the newer one you can easily figure out what was changed.

The last thing to emphasise is that the project exists and is open for business. So far it contains only sixteen phrase-pages, but we hope that will increase rapidly with more than the few current contributors.

We hope many of you will have a go and 'get stuck in'. You need only an APL wiki account and a bit of time. If you have your own favourite phrase that you want to leave to posterity, or a whole library of them, all well and good. If not, there is a To-Do list that links back to sources for phrases if you just want to help.

Remember, it is a wiki and so no-one should be too precious about having his or her efforts 'improved' by others though the "Page under construction" notice will hold other contributors off for a while if you have temporarily to leave a page unfinished, while a similar "Please will someone improve this page" notice invites the vultures immediately.

## References

1. <http://preview.tinyurl.com/62qs9f>
2. <http://aplwiki.com/MildServer>
3. <http://aplwiki.com/PhraseBook/TestCasesGuidelines>

# Bring something beautiful

by Roger Hui

The following e-mail exchange took place on 2010-06-24 during a discussion on numeric representation.

**Morten Kromberg:** And... I shall fight against adding any form of NaN / Infinity – to the death! They will horribly complicate our implementation and don't help users do *anything* useful.

**Roger Hui:** In the year 2033 Earth was discovered by Survey Fleet MCXII of the Galactic Empire. The Emperor ordered Earth to send a representative to the court, with strict instructions to “bring something beautiful”.

*Emperor*

What beautiful things does Earth have?

*Earth Representative*

Your excellency, our mathematician Euler proved in our year 1737 that  

$$+/(1+\imath\infty)*-s \leftrightarrow x/\div 1-(\div\imath\infty)*-s$$

*Emperor*

What is the  $\div$  symbol?

*Earth Rep.*

$\div i$  is the  $i$ -th prime.

*Emperor*

And what is  $\infty$ ? Does it do *anything* useful?

*Earth Rep.*

It denotes infinity, your excellency.

*Emperor*

(ponders equation for a minute or two) Respect!

*Emperor*

Neat notation you have there. Tell me more.

*Earth Rep.*

Your excellency, it's called APL. It was invented by the Canadian Kenneth E. Iverson...



# LEARN

# Treetable: a case-study in q

by Stevan Apter

This article is the first in an occasional column, *No Stinking Loops*. Stevan Apter is one of the programmers Jeffry Borror referred to as “the q gods” in his textbook *q for Mortals*. The world of q programming has so far been largely hidden behind corporate non-disclosure contracts. *Vector* is glad to see it opening and proud to be publishing this. *Ed*.

## 0. Introduction

A treetable is a table with four additional properties.

Firstly, the records of the table are related hierarchically. Thus, a record may have one or more child-records, which may in turn have children. If a record has a parent, it has exactly one. A record without a parent is called a root record. A record without any children is called a leaf record. A record with children is called a node record.

Secondly, it is possible to *drill down* into a treetable. If a record is a parent, then some of its columns may be rollups of its child-records. By drilling down into a parent-record, it is possible to inspect the elements which are aggregated in the parent. All rollups are performed on the leaves of the tree rather than on the immediate children. This means that tree-construction can be ‘lazy’: not all intermediate rollups from parent to leaves need exist.

Thirdly, treetables have *state*. If the user drills down into the tree along a particular path, then closes a node along that path, the records on that path become invisible. If the user re-opens that parent, then the nodes along that path will become visible if they were visible before the parent was closed. In other words, closing an open parent does not destroy the visibility state of its children.

Fourthly, a treetable is naturally sorted in a way that is an extension of ordinary table sort. Intuitively, the sort of a treetable is a structure-preserving sort of the ‘blocks’ out of which it is composed. The sort is *structure-preserving* because the parent-child relation between records is preserved even though record-order is not. I’ve included an explanation of how such a multi-column sort works.

The treetable is a natural candidate for a control in a non-procedural data-driven GUI. K and q have a long tradition of such GUIs, stretching back to A+ and the

native K3 GUI. The examples in this paper are abstracted from an implementation of a GUI recently developed for q. But the case-study is meant to stand alone, as an exercise in pure data design. In a future instalment, I hope to show how such designs are smoothly integrated into a data-driven GUI.

## 1. Lists, dictionaries, tables, keytables

This section contains the necessary background on q's collection data-types.

A list is a collection indexed by position:

```
q)1:10 20 30
```

```
q)1 2 0
30 10
```

```
q)1?30 10
2 0
```

A dictionary is a collection indexed by a q object:

```
q)d:`a`b`c!10 20 30
```

```
q)d`c`a
30 20
```

```
q)d?30 20
`c`a
```

```
q)key d
`a`b`c
```

```
q)value d
10 20 30
```

The q object is usually a symbol (a name) but need not be. For example:

```
q)e:(10 20;30 40 50;70 80)!`a`b`c
q)e(30 40 50;10 20)
`b`a
```

A dictionary is a map from a list of elements (the key) to a list of elements (the value). The key and the value must have the same count. Moreover, the key should not contain duplicates. Although q does not enforce key-uniqueness, dictionaries containing duplicate keys may not behave as you'd expect.

Atomic functions penetrate both lists and dictionaries:

```
q)1+1
11 21 31
```

```
q)d+1
a| 11
b| 21
c| 31
```

A table is a list of dictionaries, or *records*, all of which have the same key. For example:

```
q)t:(d;d+1;d+2;d+3)
q)t
a  b  c
-----
10 20 30
11 21 31
12 22 32
13 23 33
```

A list of key-dissimilar dictionaries is not a table:

```
q)(`a`b!10 20;`b`c`d!30 40 50)
`a`b!10 20
`b`c`d!30 40 50
```

Since tables are lists, we can index them positionally:

```
q)t 1
a| 11
b| 21
c| 31
```

The *transpose* of a table is a dictionary whose values are lists:

```
q)flip t
a| 10 11 12 13
b| 20 21 22 23
c| 30 31 32 33
```

The *flip* of a dictionary of equal-length lists is a table:

```
q)t~flip flip t
1b
```

q has a compact notation for constructing tables:

```
q)u:([]a:10 11 12 13;b:20 21 22 23;c:30 31 32 33)
q)t~u
1b
```

A table can therefore be constructed in three ways:

- as a list of dictionaries
- as the flip of a dictionary of vectors
- using table-notation

A keytable is a dictionary in which the key and the value are both tables. For example:

```
q)k:([]f:`a`a`b;g:1 2 1)
q)v:([]a:10 20 30;b:40 50 60;c:70 80 90)
q)a:k!v
q)a
f g| a  b  c
---| -----
a 1| 10 40 70
a 2| 20 50 80
b 1| 30 60 90
```

Since keytables are dictionaries, they are indexed by key:

```
q)a(`a;1)
a| 10
b| 40
c| 70

q)a((`a;1);(`a;2))
a  b  c
-----
10 40 70
20 50 80
```

Since keytables are dictionaries, they can be split into key and value:

```
q)key a
f g
---
a 1
a 2
b 1
```

```

q)value a
a  b  c
-----
10 40 70
20 50 80
30 60 90

```

A keytable can also be defined using q table-notation:

```

q)a~([f:`a`a`b;g:1 2 1]a:10 20 30;b:40 50 60;c:70 80 90)
1b

```

The data universe of q can be summarised as follows: There are atoms and lists. Dictionaries map lists to lists. Tables are lists of dictionaries and keytables are dictionaries which map tables to tables.

## 2. Trees

We can represent a tree as a list of paths. For each element of the tree there is a corresponding path to that element:

tree	path
----	----
A	A
B	A B
C	A B C
D	A B D
E	A E
F	A E F
G	A E F G
H	A E F H
I	A E F I

From the path of an element e we can easily compute the parent: the parent of e is `enlist e` if the path is a singleton, else the parent is the `drop` of the last element of the path.

While the path-list representation is intuitive, it can be clumsy to work with. For example, to find the children of A we need to search the path-list for all 2-element lists which have A as their first element. To find the children of A B we need to find all 3-element lists which have A B as their first two elements.

We can represent the parent-child relation explicitly, as a table PC:

```
q)PC:([ ]parent:`A`A`B`B`A`E`F`F`F;child:`A`B`C`D`E`F`G`H`I)
```

```
parent      child
```

```
-----
```

```
A      A
```

```
A      B
```

```
B      C
```

```
B      D
```

```
A      E
```

```
E      F
```

```
F      G
```

```
F      H
```

```
F      I
```

Then:

```
q)select child from PC where parent=`A
```

```
child
```

```
-----
```

```
A
```

```
B
```

```
E
```

```
q)select parent from PC where child=`F
```

```
parent
```

```
-----
```

```
E
```

We can represent the relation as a parent-vector:

```
p:0 0 1 1 0 4 5 5 5
```

That is:

i	tree	p
-	----	-
0	A	0
1	B	0
2	C	1
3	D	1
4	E	0
5	F	4
6	G	5
7	H	5
8	I	5

$p[i]$  is the index of the parent of the  $i$ th element of the tree.

To find the children of any element  $e$  in the tree, search  $p$  for occurrences of  $e$ 's index:

```
q)where p=5
6 7 8
```

To find the root of any element  $e$ , repeatedly index  $p$ , starting with  $e$ :

```
q)p 6
5
q)p 5
4
q)p 4
0
q)p 0
0
```

To find the root of  $e$  in one step, reduce  $p$  over  $e$ :

```
q)p over 6
0
```

$p$  is applied repeatedly to the previous result until the result is the same twice in a row. This is why it is convenient to treat the root as self-parenting.

To find the path from  $e$  to the root in one step, scan  $p$  over  $e$ :

```
q)p scan 6
6 5 4 0
```

To find the paths of all elements of the tree, scan  $p$  over each of  $0 \dots \text{count}[p]-1$ :

```
q)i:(p scan)each til count p
q)i
,0
1 0
2 1 0
3 1 0
4 0
5 4 0
6 5 4 0
7 5 4 0
8 5 4 0
```

To find the leaf-elements of the tree, discard every element which is a parent:



```

q)l:til[count p]except p
q)l
2 3 6 7 8

```

We can use `p` to aggregate data associated with the tree. For example, suppose that the five leaf-elements have the following data:

```

q)d:count[p]#0
q)d[1]:1*10
q)d
0 0 20 30 0 0 60 70 80

```

Now to sum `d` to the root, amend a zero-vector with `+` at `i`, the effect of which is to accumulate sums on all paths:

```

q)@[count[d]#0;i;++d]
260 50 20 30 210 210 60 70 80

```

Think of it this way: where `r` is the result, `r k` is `sum d i k`. In other words, the result at each node of the tree is the sum of that node's descendants.

From the parent-vector representation and a list of elements, we can compute the path-list:

```

q)e:`A`B`C`D`E`F`G`H`I
q)n:reverse each e i
q)n
,`A
`A`B
`A`B`C
`A`B`D
`A`E
`A`E`F
`A`E`F`G
`A`E`F`H
`A`E`F`I

```

And from the path-list representation we can compute the parent-vector: drop the last element of each path whose count is greater than 1 and find each truncated path in the path-list:

```

q)n?neg[1<count each n]_'n
0 0 1 1 0 4 5 5 5

```

From `p` and `e` it is also easy to derive the parent-child table:

```
q)PC:([ ]parent:e p;child:e)
```

```
q)PC
```

```
parent child
```

```
-----
```

```
A      A
```

```
A      B
```

```
B      C
```

```
B      D
```

```
A      E
```

```
E      F
```

```
F      G
```

```
F      H
```

```
F      I
```

And vice-versa:

```
q)e?PC.parent
```

```
0 0 1 1 0 4 5 5 5
```

### 3. Treetables

One way to think about the treetable is that it is a keytable whose records are related by the parent-child relation.

A record is either a leaf or a parent. A parent record is the rollup of its children.

A treetable has a single grand-total record, the root of the tree.

The parent-child relation is constructed from an underlying table  $\tau$ . The records of  $\tau$  are precisely the leaf-records of the treetable. Nothing more is required of  $\tau$  except that it be a table, but in practice all suitable candidates for  $\tau$  will conform to the following condition:  $\tau$  will contain one or more columns which are suitable to group by, and one or more columns which are suitable to aggregate.

For example, the following table satisfies that condition:

```
A B C v w
```

```
-----
```

```
a f n 12 x
```

```
a f o 10 y
```

```
a f p 1 z
```

```
a f q 90 w
```

```
a g n 73 x
```

```
a g o 90 y
```

```
...
```

A, B and C are suitable to group by, and v and w are suitable to aggregate. But it is worthwhile emphasising that this distinction is entirely arbitrary, and that nothing in the algorithm requires that columns of either type have any special properties. It would be silly to group on a column most of whose values are different, but the algorithm doesn't preclude that. And in this example, although w is a column of symbols, it can be aggregated as long as its rollup function satisfies the condition that it takes list input and returns an atom.

Let's look at one possible treetable based on T. The grouping columns are A, B and C. Order matters. T grouped by A B C is different from T grouped by B A C. The rollup columns are v and w, and the rollup functions for those columns are sum, nul (explained below), and count.

A single column may be aggregated more than once. In the example below, we aggregate v with sum and count.

A treetable based on that scheme is R1:

n_	A	B	C	counts	v	w
-----	-----	-----	-----	-----	-----	-----
`symbol\$()				1000	52015	
, `a	a			224	12054	
, `b	b			200	11173	
, `c	c			192	10290	
, `d	d			192	8136	
, `e	e			192	10362	

Here we can see that R1 is a keytable. The key of R1 is the column n\_, a path-list. The first record of R1 is the grand-total of T. We can see that T has 1000 records and that column v sums to 52015. The aggregations of column w are null.

We can also see by examining the remaining records that R1 contains a single level of aggregation based on distinct values of column A.

We can also see nulls in R1: the A column of the first record, and all of column w. In the examples used in this paper, null means *cannot aggregate this group*.

Now let's drill down on the record where A=`a, giving us the table R2:

n_	A	B	C	counts	v	w
-----	-----	-----	-----	-----	-----	-----
`symbol\$()				1000	52015	
,`a	a			224	12054	
,`a`f	a	f		28	791	
,`a`g	a	g		28	2072	
,`a`h	a	h		28	2058	
,`a`i	a	i		28	1967	
,`a`j	a	j		28	1078	
,`a`k	a	k		28	1645	
,`a`l	a	l		28	1484	
,`a`m	a	m		28	959	
,`b	b			200	11173	
,`c	c			192	10290	
,`d	d			192	8136	
,`e	e			192	10362	

One way to think about treetables like `R1` and `R2` is that they are constructed out of sub-tables. These ‘blocks’ are computed independently from `T`, then stitched together in the right order. This is the pattern followed by the algorithm described below.

Let’s begin by identifying the parameters.

The first parameter is `T`, the underlying table of unaggregated records.

The second parameter is a list of the grouping columns:

```
q)G:`A`B`C
```

The third parameter is a dictionary of rollup functions:

```
q)A:`counts`v`w!((sum;`n);(sum;`v);(nul;`w))
q)A
counts| count                                `v
v      | sum                                `v
w      | {first$[1=count distinct x,();x;0#x]} `w
```

The key of `A` is a list of names of the aggregated columns in the treetable. The value of `A` is a list of pairs of the form `(f;c)`, where `f` is an aggregator and `c` is a column in `T`.

`count` and `sum` are primitive aggregators of `q`: `sum` is `+/` and `count` returns the number of elements in a list. `nul` serves as our general default aggregator:

given a list  $x$ , return the first element of  $x$  if  $x$  is all duplicates, else return the null of  $x$ . Nulls in treetables are always the result of aggregation by `null`.

The fourth and final parameter is a package of information which represents the “drill-down state” of the treetable to be computed. For  $R1$ , this state is the keytable  $P1$ :

```

n                                | v
-----| -
(`symbol$())!`symbol$()| 1

```

and for  $R2$  it is the keytable  $P2$ :

```

n                                | v
-----| -
(`symbol$())!`symbol$()| 1
(, `A)! , `a                | 1

```

The state is a keytable where the key  $n$  is a list of dictionaries, each of which functions as an instruction to the algorithm to compute a specific sub-table block of the treetable. The meaning of  $v$  is described below in the section on state.

For example, the grand-total block is arbitrarily represented by the unique empty dictionary:

```
(`symbol$())!`symbol$()
```

whose key and value both consist of the empty symbol list. This dictionary will be interpreted as the instruction to select all records from the underlying table  $T$  and aggregate them by distinct values of the first element of  $G$ , which in this example is ``A`.

The  $R2$  block of aggregated  $A = `a$  values consists of the dictionary:

```
A | a
```

This will be interpreted as the instruction to select records from  $T$  where  $A = `a$  and aggregate them by distinct values of the second element  $G$ , which in this example is ``B`.

Finally, let's look at a treetable  $R4$  which has been drilled down to the leaves along one of the paths:

n_	A B C counts v	w
-----	-----	
`symbol\$()	1000 479131	
,`a	a 224 106670	
`a`f	a f 28 13952	
`a`f`n	a f n 7 2867	x
`a`f`n`0	a f n 908 908	x
`a`f`n`1	a f n 256 256	x
`a`f`n`2	a f n 401 401	x
`a`f`n`3	a f n 288 288	x
`a`f`n`4	a f n 543 543	x
`a`f`n`5	a f n 258 258	x
`a`f`n`6	a f n 213 213	x
`a`f`o	a f o 7 3707	y
`a`f`p	a f p 7 3640	z
`a`f`q	a f q 7 3738	w
`a`g	a g 28 14948	
`a`h	a h 28 12190	
`a`i	a i 28 13535	
`a`j	a j 28 13835	
`a`k	a k 28 12945	
`a`l	a l 28 13643	
...		

We append a unique identifier to the key of each leaf. This preserves `n_` as a valid key.

The instruction table for `R4` is `P4`:

n	v
-----	-
(`symbol\$())!`symbol\$()	1
(,`A)!`,`a	1
`A`B!`a`f	1
`A`B`C!`a`f`n	1

`R4` has the following blocks as constituents:

- the grand-total block (root)
- the `A=`a` block
- the `A=`a`, `B=`f` block
- the `A=`a`, `B=`f`, `C=`n` block (leaves)

Concentrating just on the value parts of the blocks, let's see how we would generate those using the native query language of q.

To compute the grand-total block:

```
q)flip enlist each exec nul A,nul B,nul C,count v,sum v,nul w from T
A B C v      v1      w
-----
      1000 479131
```

To compute the first subtotal level:

```
q)0!select nul B,nul C,counts:count v,sum v,nul w by A from T
A B C counts v      w
-----
a      224      106670
b      200      100048
c      192      90541
d      192      92853
e      192      89019
```

To compute the A=`a` block:

```
q)0!select nul C,counts:count v,sum v,nul w by A,B from T where A=`a
A B C counts v      w
-----
a f  28      13952
a g  28      14948
a h  28      12190
a i  28      13535
a j  28      13835
a k  28      12945
a l  28      13643
a m  28      11622
```

To compute the A=`a`, B=`f` block:

```
q)0!select counts:count v,sum v,nul w by A,B,C from T where A=`a,B=`f
A B C counts v      w
-----
a f n 7      2867 x
a f o 7      3707 y
a f p 7      3640 z
a f q 7      3738 w
```

And finally, to compute the block containing the leaves:

```

q)0!select A,B,C,counts:1,v,w from T where A=`a,B=`f,C=`n
A B C counts v    w
-----
a f n 1      908 x
a f n 1      256 x
a f n 1      401 x
a f n 1      288 x
a f n 1      543 x
a f n 1      258 x
a f n 1      213 x

```

#### 4. Construction

Now we know what a treetable is, and have an intuitive grasp of what its parts are, how they are related, and how those parts are computed. The next step is to explain the q code which implements those ideas. My advice is that the reader get a q session, load the associated script[1], and experiment by reading along and executing (and varying!) bits of code. All the examples used in this paper are defined in that script.

Rather than write a line-by-line commentary on the implementation, I've chosen to focus on the concepts which drive that implementation, and on a few of the knottier parts of the code.

An indispensable companion in this (the reader's) task is Jeffry Borror's splendid book, *q for Mortals*. There are a few 'dangerous curves' ahead. In particular, I recommend close study of the chapter in Jeffrey's book on Functional Forms.

The four parameters of treetable construction are:

T the underlying table  
 G a list of group columns  
 P the path table  
 A the rollup dictionary

The construction function takes T, G, P, and A and returns R, the treetable:

```
construct: {[t;g;p;a]!`n_ xasc root[t;g;a]block[t;g;a]/visible p}
```

construct uses three subfunctions:

visible determine which paths are visible  
 root construct the root block  
 block construct non-root blocks



The form of the `construct` function is:

... `r0 f/s`

`r0` is the initial state and `s` is a list of arguments to the dyadic function `f`. Note that the block function takes five arguments, but in this context is applied as a dyad. In `q`, we say that `block` is *projected* on its first three arguments `t`, `g` and `a`. The first three arguments are fixed and the remaining two argument positions are open. So `block[t;g;a]` is a dyad.

Suppose `s` has `n` elements. Then:

```
r1:f[r0;s 0]
r2:f[r1;s 1]
...
rn:f[rn-1;s n-1]
```

In this case, `r0` is the root block of the treetable and `s` is the result of applying `visible` to the path table `p`. For now, all we need to know is that this result is a list of instructions, for example:

```
(`symbol$())!`symbol$()
(,`A)!,`a
`A`B!`a`f
```

So in this example, the block function `f` will be applied three times: first to the root block and the first instruction; then to the result of that application and the second instruction; and finally to the result of that application and the third instruction.

The result is a table with the structure:

```
root block
A=`a block
A=`a, B=`f block
```

The `construct` function then up-sorts the result by `n_` and makes it the key:

```
1!`n_ xasc ...
```

This is necessary because the blocks have to be recursively interleaved. For example, the `A=`a, B=`f` block must appear in the treetable immediately after the record in the `A=`a` block where `B=`f` :

n_	A	B	C	counts	v	w
`symbol\$()				1000	479131	
,`a	a			224	106670	
`a`f	a	f		28	13952	
`a`f`n	a	f	n	7	2867	x
`a`f`o	a	f	o	7	3707	y
`a`f`p	a	f	p	7	3640	z
`a`f`q	a	f	q	7	3738	w
`a`g	a	g		28	14948	
`a`h	a	h		28	12190	

...

Sorting on `n_` is a fast non-recursive method for interleaving records from the different blocks.

`construct` uses `over` to produce a single table, where successive blocks are appended to the initial root table. This method destroys structural information about the blocks. That is, we have a single table as the result rather than a list of blocks.

There is an alternative method which constructs the blocks in parallel using `peach` instead of `over`. Assume `q` is started with slaves (e.g. `q -s 4`). Then:

```
pconstruct: {[t;g;p;a]
  1!`n_ xasc root[t;g;a],
  raze pblock[t;g;a]peach visible p}

pblock: {[t;g;a;p]
  f:[g~key p;leaf;node g(`,g)?last key p];
  (`n_,g)xcols f[t;g;a;p]}
```

We'll now look more closely at the root and block functions. The visible function is discussed in section 5 below. A few auxiliary functions mentioned in the text are not discussed.

The root function is:

```
root: {[t;g;a]
  a[g]:nul,'g;
  (`n_,g)xcols node_[g]flip enlist each?[t;();();a]}
```

Recall from the previous section that the root block is computed with the expression:

```
flip enlist each exec nul A, nul B, nul C, counts:count v, sum v, nul w from T
```

We can use q's native parsing primitive to see the underlying functional form of the query part of this expression:

```
q)parse "exec nul A, nul B, nul C, counts:count v, sum v, nul w from T"
?
`T
()
()
`A`B`C`counts`v`w!((`nul;`A);(`nul;`B);(`nul;`C);(#::`v);(sum;`v);(`nul;`w))
```

The functional form of `exec` is:

```
?[t;();();a]
```

where `a` is the rollup dictionary constructed in the first two lines of the root function. In the case where every element of `a` is a rollup, this expression returns a dictionary of atoms. To get our one-record table, we therefore enlist each atom and `flip` the result. This table is now passed to the `node_` function, which adds the `n_` column which will become the key of our root block. The result is reordered to put `n_` and the grouping columns at the front.

The `block` function doesn't do much: it calls the `leaf` function if the key of the instruction contains all the grouping columns, else it calls the `node` function with by-clause `b =` the next grouping column:

```
block: {[t;g;a;r;p]
  f:[g~key p;leaf;node g(`,g)?last key p];
  r,(`n_,g)xcols f[t;g;a;p]}
```

As the final step, it pushes the key and the grouping columns out to the front of the query result and appends this to the treetable `r` computed to this point.

The `node` function is:

```
node: {[b;t;g;a;p]
  c:constraint p;
  a[h]:first, 'h:(i:g?b)#g;
  a[h]:nul, 'h:(1+i)_g;
  node_[g]0!?[t;c;enlist[b]!enlist b;a]}
```

Recall again from the previous section how we compute a block which is neither a leaf nor a root:

```
select nul C, counts:count v, sum v, nul w by A,B from T where A=`a
```

The functional form of this query is:

```
?
`T
,,(=;`A;;`a)
`A`B!`A`B
`C`counts`v`w!((`nul;`C);(#;`v);(sum;`v);(`nul;`w))
```

In an expression of the form:

```
?[t;c;b;a]
```

*c* is the constraint, or ‘where’ clause (where *A*=`a), *b* is the grouping, or ‘by’ clause (by *A*,*B*), and *a* is the rollup dictionary.

The first line of the function constructs the ‘where’ clause *c* from the instruction *p*:

```
constraint:{[p]flip(=;key p;flip enlist value p)}
```

For example,

```
q)p
A | a
B | f
C | n

q)constraint p
= `A ,`a
= `B ,`f
= `C ,`n
```

The next two lines construct the ‘by’ clause from *A* and the group column vector *g*.

In the last line, the constructed query is evaluated, de-keyed, and passed through the `node_` function, which adds the `n_` column to the table. The columns are re-ordered in the `block` function, which calls `leaf` and `node`.

The `leaf` function has a similar form:

```
leaf: {[t;g;a;p]
  c:constraint p;
  a:last each a;
  a[g]:g;
  leaf_[g]0!?[t;c;0b;a]}
```

Again, the first three lines construct the arguments to the functional form of the leaf query, and the resulting table is de-keyed and passed through the `leaf_` function, which adds the `n_` column to the result.

## 5. State

The treetable is intended for interactive use as the data-structure backing a GUI control. The user of the control clicks on a record to open or close that record. Opening a record `r` in the control reveals the records which are children of `r`. Closing `r` conceals the children of `r`.

If a child `c` of `r` is open, and then `r` is closed and re-opened, then `c`'s state must be restored. Therefore we must keep track of the state of the treetable. We do this by associating the instruction for a block with a boolean value. The value is `1b` if the parent of the block is open, else `0b` if it is closed.

The state of a treetable is contained in the path table `P`. For example, here is the state `P4` of the table `R4`:

n	v
-----	-
(`symbol\$())!`symbol\$()	1
(, `A)! , `a	1
`A`B!`a`f	1
`A`B`C!`a`f`n	1

The `visible` function takes a path table and returns only those instructions which compute blocks which lie along visible paths:

```
q)visible P4
(`symbol$())!`symbol$()
(, `A)! , `a
`A`B!`a`f
`A`B`C!`a`f`n
```

Let's simulate closing R4 at A=`a. The result R5 should match R1, the initial, minimal treetable:

```
q)P5:closeat[P4;G;`a]
q)P5
n                                | v
-----| -
(`symbol$())!`symbol$()| 1
(,`A)! ,`a                    | 0 ← closed at A=`a
`A`B!`a`f                     | 1
`A`B`C!`a`f`n                 | 1

q)visible P5
(`symbol$())!`symbol$()

q)R5:construct[T;G;P5;A]
q)R5~R1
1b
```

Now we'll reopen R5 at A=`a. P6 should match P4 and the resulting treetable R6 should match R4:

```
q)P6:openat[P5;G;`a]
q)P6~P4
1b
q)R6:construct[T;G;P6;A]
q)R6~R4
1b
```

openat and closeat are projections of the underlying function at:

```
at: {[b;p;g;n]p, ([n:enlist(count[n]#g)!n,() ]v:enlist b)}

openat:at 1b
closeat:at 0b
```

at relies on the fact that catenation to a dictionary is *upsert*: append if the key is new, else update. For example:

```
q)d:`a`b`c!10 20 30
q)d,`c`d!40 50
a| 10
b| 20
c| 40
d| 50
```

at is trivial: flip the visibility bit for an instruction in the path table. The heavy lifting is performed by the `visible` function:

```
visible: {[p]
  q:parent exec n from p;
  k:(reverse q scan)each til count q;
  n where all each(exec v from p)k}
```

The first line computes the parent-vector `q` from the key of the path table `p` (see section 2 above). Line 2 computes the list of paths from the root to all nodes. Line 3 performs a running logical-and scan (`q` keyword: `all` down the boolean states of each path).

Here is a transcript of the console for an example run:

```
q)P5
n                                | v
-----| -
(`symbol$())!`symbol$()| 1
(,`A)! ,`a                | 0
`A`B!`a`f                 | 1
`A`B`C!`a`f`n            | 1

q)p:P5

q)q:parent exec n from p
q)q
0 0 1 2
q)k:(reverse q scan)each til count q
q)k
,0
0 1
0 1 2
0 1 2 3
q)exec v from p
1011b

q)(exec v from p)k
,1b
10b
101b
1011b
q)all each(exec v from p)k
1000b
```

```
q)where all each(exec v from p)k
,0

q)n where all each(exec v from p)k
(`symbol$())!`symbol$()
```

There are good reasons for breaking out the state in this way.

In our example, the underlying table  $\tau$  has 1000 records, and the treetable in its fully opened state, in which all leaves of  $\tau$  and all aggregations are constructed, has 1206 records. The state-table therefore has 206 instructions, each of which corresponds to a complete scan of  $\tau$ . (See Q2 and S2 in [1].) Clearly, this can get expensive. For example, where the underlying table contains millions of records and hundreds of aggregated columns, and the tree-structure is deep and bushy. Moreover, we cannot rule out the possibility that the underlying table is the target of frequent updates; for example, if it is connected to a real-time data-source. In that case, we cannot even be confident that the structure of the tree won't change. (See the `valid` function in [1].)

For these reasons, the design is deliberately *lazy*: we compute only as much of the tree as the path-table directs.

## 6. Sort

The APL sorting primitives *grade-up* and *grade-down* appear in `q` as the keywords `iasc` and `idesc`.

We can use `over` to do multi-column sorts:

```
msort:{x y z x}
```

`msort` is a function of three arguments: `x`, an index vector; `y`, a permutation function; and `z`, a list. Thus: `x` permuted by the result of applying `y` to `z` permuted by `x`.

Let `v` be a list of two vectors:

```
q)v
0 2 4 4 3 0 4 3 0 3
0 3 1 4 1 3 1 3 1 2
```

Sort `v 0` descending within `v 1` ascending:

```
q)i:msort/[til count first v;(idesc;iasc);v]
q)i
0 2 6 4 8 9 7 1 5 3
```



```
q)v@\:i
0 4 4 3 0 3 3 2 0 4
0 1 1 1 1 2 3 3 3 4
```

`msort/` repeatedly permutes `x` by the result of applying `y` to `z` permuted by `x`.

`msort` can be used to sort dictionaries of vectors:

```
q)d:`a`b!v
q)d
a| 0 2 4 4 3 0 4 3 0 3
b| 0 3 1 4 1 3 1 3 1 2

q)d@\:msort/[til count first d;(idesc;iasc);d]
a| 0 4 4 3 0 3 3 2 0 4
b| 0 1 1 1 1 2 3 3 3 4
```

and tables:

```
q)t:flip d
q)t
a b
---
0 0
2 3
4 1
4 4
3 1
0 3
4 1
3 3
0 1
3 2

q)t msort/[til count t;(idesc;iasc);flip t]
a b
---
0 0
4 1
4 1
3 1
```

```

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

```

Our problem is to adapt `msort` to apply recursively to the hierarchically-related blocks of a treetable.

In our example, `R4` has 25 records and blocks at four levels:

n_	A B C	counts	v	w
-----	-----	-----	-----	-----
`symbol\$()		1000	479131	
,`a	a	224	106670	
`a`f	a f	28	13952	
`a`f`n	a f n	7	2867	x
`a`f`n`0	a f n	908	908	x
`a`f`n`1	a f n	256	256	x
`a`f`n`2	a f n	401	401	x
`a`f`n`3	a f n	288	288	x
`a`f`n`4	a f n	543	543	x
`a`f`n`5	a f n	258	258	x
`a`f`n`6	a f n	213	213	x
`a`f`o	a f o	7	3707	y
`a`f`p	a f p	7	3640	z
`a`f`q	a f q	7	3738	w
`a`g	a g	28	14948	
`a`h	a h	28	12190	
`a`i	a i	28	13535	
`a`j	a j	28	13835	
`a`k	a k	28	12945	
`a`l	a l	28	13643	
`a`m	a m	28	11622	
,`b	b	200	100048	
,`c	c	192	90541	
,`d	d	192	92853	
,`e	e	192	89019	

`R4` is implicitly hierarchical. The typical approach to operating on such structures is to apply a ‘flat’ algorithm like `msort` recursively. The deprecated function `rsort` in [1] exemplifies this approach.

But there is a better way. We want an ‘array solution’ where the iteration is handled covertly by primitives. And we have one. Our solution will (i) convert the parent-vector into a list of child-vectors; (ii) use the child-list to partition the treetable into child-blocks; (iii) sort each child-block; (iv) use the key of the treetable to reassemble the sorted blocks into a treetable.

Let's step through it using R4.

First, de-key R4:

$$q) t: 0!R^4$$

Next, compute the parent-vector of  $t$  from column  $n$ , the path-list:

q) parent: {[n]n?-1\_ 'n}

q)n:exec n\_ from t

q)p:parent n

$$q \rightarrow p$$

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

Next, compute the child-list from  $p$ :

```
q)children:{[p]@[ (2+max p)#enlist();first[p],1+1_p;,,;til count p]}
```

q)i:children p

q) i

, 0

1 21 22 23 24

2 14 15 16 17 18 19 20

3 11 12 13

4 5 6 7 8 9 10

$i$  is a list of indices into  $t$  such that  $t[i]$  is a list of the subtable blocks of  $t$ :

q)t i

```
+`n_`A`B`C`counts`v`w!((,`a;;`b;;`c;;`d;;`e);`a`b`c`d`e;````;````;224 200 1...
```

```
+`n_`A`B`C`counts`v`w!(((`a`f;`a`g;`a`h;`a`i;`a`j;`a`k;`a`l;`a`m);`a`a`a`a`a`a...
```

```
+`n_`A`B`C`counts`v`w!(((`a`f`n;`a`f`o;`a`f`p;`a`f`q);`a`a`a`a;`f`f`f`f;`n`o`p...
```

```
+`n `A`B`C`counts`v`w!(((`a`f`n`0;`a`f`n`1;`a`f`n`2;`a`f`n`3;`a`f`n`4;`a`f`n`5...
```

For example, block 1 is:

```
q)t i 1
n_ A B C counts v      w
-----
a  a      224    106670
b  b      200    100048
c  c      192    90541
d  d      192    92853
e  e      192    89019
```

Suppose we have a single sorting operation `o` and a single column `c`:

```
q)c:enlist`v
q)o:enlist iasc
```

Sort each block:

```
q)j:msort[t;c;o]each i
q)j
,0
24 22 23 21 1
20 15 18 16 19 17 2 14
3 12 11 13
10 5 9 7 6 8 4
```

We now have the permutations we need to sort each block of `t`:

```
q)t j 1
n_ A B C counts v      w
-----
e  e      192    89019
c  c      192    90541
d  d      192    92853
b  b      200    100048
a  a      224    106670
```

Our adaptation of `msort` for treetables is:

```
msort: {[t;c;o;i]i{x y z x}/[til count i;o;flip[t i]c]}
```

As the last step, we need to mesh the permutations to give us the single permutation vector `v` such that `t v` is `t` in sorted order.

To do that, we first compute the reordered keys of the blocks:

```
q)m:n j
q)m
,`symbol$( )
(,`e;,`c;,`d;,`b;,`a)
(`a`m;`a`h;`a`k;`a`i;`a`l;`a`j;`a`f;`a`g)
(`a`f`n;`a`f`p;`a`f`o;`a`f`q)
(`a`f`n`6;`a`f`n`1;`a`f`n`5;`a`f`n`3;`a`f`n`2;`a`f`n`4;`a`f`n`0)
```

Then, to mesh the keys we insert each path-list into the appropriate slot of the mesh of the previous path-lists. This is our function `pmesh`:

```
pmesh:{i:1+x?-1_first y;(i#x),y,i _ x;() }
```

We apply it over `m` to give us the permuted path-list of the sorted table:

```
q)k:pmesh over m
q)k
`symbol$( )
,`e
,`c
,`d
,`b
,`a
`a`m
`a`h
`a`k
`a`i
`a`l
`a`j
`a`f
`a`f`n
`a`f`n`6
`a`f`n`1
`a`f`n`5
`a`f`n`3
`a`f`n`2
`a`f`n`4
`a`f`n`0
`a`f`p
`a`f`o
`a`f`q
`a`g
```

Finally, we look up  $k$  in  $n$ , which gives us the index-vector  $v$  which permutes  $n$  into  $k$ :

```
q)v:n?k
q)v
0 24 22 23 21 1 20 15 18 16 19 17 2 3 10 5 9 7 6 8 4 12 11 13 14
```

and hence  $t$  into  $t$  upsorted by  $v$ :

```
q)t v
n_      A B C counts v      w
-----
`symbol$()      1000  479131
,`e      e      192   89019
,`c      c      192   90541
,`d      d      192   92853
,`b      b      200  100048
,`a      a      224  106670
`a`m      a m    28   11622
`a`h      a h    28   12190
`a`k      a k    28   12945
`a`i      a i    28   13535
`a`l      a l    28   13643
`a`j      a j    28   13835
`a`f      a f    28   13952
`a`f`n     a f n 7    2867  x
`a`f`n`6   a f n 213  213   x
`a`f`n`1   a f n 256  256   x
`a`f`n`5   a f n 258  258   x
`a`f`n`3   a f n 288  288   x
`a`f`n`2   a f n 401  401   x
`a`f`n`4   a f n 543  543   x
`a`f`n`0   a f n 908  908   x
`a`f`p     a f p 7    3640  z
`a`f`o     a f o 7    3707  y
`a`f`q     a f q 7    3738  w
`a`g      a g    28   14948
```

Assembling the steps:

```
tsort: {[t;c;o]
  n:exec n_ from t;
  i:children[parent n]except enlist();
  j:msort[0!t;c;o]i;
  n?pmesh over n j}
```

## 7. Conclusion

q is a language of lists and dictionaries. By adding tables (lists of dictionaries) and keytables (dictionaries of tables), q inverts the traditional relationship between database and programming language.

In the familiar model, tables live in a database. Programs extract data from tables in the database, and insert data into them. Other programs, usually written in some special database-y language, can be attached to database tables as ‘triggers’. If you’re used to this sort of thing it doesn’t seem so onerous. If you’re not, it feels like sorting rice-grains while wearing mittens.

In q, tables and keytables are first-class entities whose parts are first-class. You assign them, transform them, bust them apart, stick them in lists, and pass them into and out of functions, just the way you do with lists and dictionaries. And that’s because they *are* lists and dictionaries.

In most applications, the built-in SQL-like syntax of q is perfectly adequate:

```
select/exec/update/delete ... by ... from ... where ...
```

But as the treetable example shows, it may be necessary to drop down to the functional level where the SQL keywords give way to the primitives `?` and `!` and the content of the queries is carried as q-object arguments to those primitives.

## Acknowledgements.

Thanks to Attila Vrabecz for corrections and several black-belt one-liners.

## References

1. <http://www.nsl.com/q/treetable.q>

# A commentary on the formulator

by Neville Holmes

*neville.holmes@utas.edu.au*

University of Tasmania

School of Computing and Information Systems

Launceston 7250,

Australia

A design for a commodity device called a *formulator* was outlined in a recently published essay[1]. Just as the calculator extended the abacus beyond basic arithmetic, the formulator is designed to extend commodity computation beyond arithmetic to algebra.

This article examines aspects of the proposed formulator in more detail than given in the essay, and relates its capabilities to those of APL/J interpreters[2].

## The formulator

Calculators are widely familiar devices that are used to do basic arithmetic. They come in various forms but the simplest, the commodity calculator, is very cheap and apparently simple and reliable. The commodity calculator is also emulated on other handheld devices.

One of the potentially most important uses of calculators is in the teaching of arithmetic in early schooling. They are commonly used for simple quantitative problem solving but could also be adapted easily to teaching basic arithmetic skills.

What is there that goes a bit beyond calculator technology, either in daily life or in later education? Well, there are spreadsheets, but they are more a data management tool than a mathematical tool[3].

Interpreters like those for APL varieties are of course genuinely and ardently mathematical tools, but they go way beyond what is needed for everyday or school use. This makes them daunting for beginners and dabblers.



What is needed is an intermediary capability, one that combines versatility and simplicity. The formulator[1] has these properties and also has enough in common with APLs that it would make it easier for users to move up.

### **The exact calculator**

The first step in developing a formulator design is to fix the problems of the commodity calculator. After all, these problems are arguably the reason behind the banning of calculators from primary schooling in countries such as Japan.

There are several significant problems, and these and fixes for them are given in my essay "Truth and Clarity in Arithmetic"[4] in more detail than is appropriate here.

The source of falsehood in the commodity calculator is the adoption of inexact arithmetic pretending to be exact. Exact arithmetic is the cure.

Two measures underpin exactitude: computation with variable length numbers and the use of fractions with a non-decimal denominator.

To make variable length numbers practical, control over the entry and display of such numbers is necessary. For example, scaling is needed on numbers both being keyed in and being displayed. Very long numbers need to be able to be abbreviated, but their inexactitude must be clearly shown when this is done[5].

To make fractions of all kinds practical, numbers with non-decimal denominators must be distinctively represented through use of both a decimal point and a fraction point[6].

To complete the design of an exact calculator, a variety of very simple functions needs to be added to the miserly 'normal' quota. At the very least, quotient and remainder functions are absolutely necessary, both for practical and educational use.

At the same time, inexact values such as multiples of  $\pi$  and inexact functions such as fractional powers need to be excluded, though there are ways that  $\pi$  could be handled symbolically. Simple multiples of  $\pi$  and roots of integers, however, can be approximated by fractions[7].

One important source of obscurity in the commodity calculator is the disappearance of the numbers being used.

Calculators work on more than one number at a time. Typically there are three numbers immediately relevant to the user of a calculator: two arguments and a

result, not to mention the symbol of the function yielding the result. All should be displayed to the user, who should also be able to manipulate them[8].

### **Towards the formulator**

The commodity calculator is used to fiddle with numbers one at a time, constructing them by editing and by applying the basic functions provided.

The commodity formulator is designed to be used to fiddle also with functions, constructing them by editing and substitution. Of course the functions are used for numeric calculation, but the focus is on manipulating the functions, that is, on algebraic calculation.

The formulator is necessarily more complex than the calculator. It needs a richer keyboard and character set and more powerful basic functions.

The exact calculator serves as a very appropriate starting point. The published design assumed a conventional numeric keyboard with a sign symbol on either side of the zero key, and a column of four function keys[4]. The signs were used as prefixes to the function keys to modify the four straight functions.

With the larger keyboard needed for the formulator, the function keys could be completely independent of the numeric keys, so that better symbolism could be used[9].

It is not appropriate here to consider in any detail just what the formulator keyboard should be like, just what basic functions should be provided, and just what symbols should be used. Rather the skeletal structure of the formulator is the focus. However, Ken Iverson's Turing Award essay[10] is an excellent guide for designers of notation, and the original mathematical symbols should be practical by now.

### **Truth and inexact arithmetic**

Overtly putting inexact arithmetic onto the exact calculator has two aspects; the use of numbers and the use of functions.

The numbers are entered, edited, and displayed.

The user needs to be able to key numbers in as inexact and, in case of need, to specify just how inexact. Similarly, the user needs to easily see whether the number is exact as shown, or exact but shown inexactly, or inexact with all of the exact part shown, or inexact with the exact part going beyond the end of the number shown[11].

The functions are selected, modified, and applied.

The basic traditional arithmetic function symbols, such as the saltire  $\times$  and the obelus  $\div$ , are used, as they are in APL[12]. Variations on those functions are denoted by diacritics (though not shown thus here as it seems such diacritics are not provided by HTML/Unicode[13]), for example the tilde signifies argument commutation:  $\sim\div$  for *divide into*. Numbers – positive, zero, or negative – are used as diacritics or superscripts of a function to have it repeatedly applied, or ignored, or repeatedly inverted[14].

With inexact arithmetic come various basic functions, such as the trigonometric ones, that normally yield inexact results. Inversion functions, like the square root, even yield more than one result, and, even though it's popular to ignore all but one, it's more strict and informative, especially in mathematical education, to produce them all as a set of results[15]. The root function also produces complex numbers, so they need to be handled alongside other kinds of value[16].

### **Breadth and list processing**

Adding inexact to exact arithmetic broadens the calculator without changing its basic nature as a calculator that applies functions one at a time to one or two arguments. But it does make it rather lopsided with a richness of calculation that involves producing one value from at most two values. The only argument widening described so far is the ability to use a set of numbers as a single value.

The logical next step is therefore to allow lists of values as arguments and results[17]. Again this step has two aspects: handling list arguments and providing structural functions to derive values from such arguments.

A list argument is simply a sequence of numbers separated by spaces[18]. Each item in the list, and each number in a set item, has its exactitude independent of other items or numbers in the list or set.

Because a numeric display of a list could be lengthy, the list calculator offers a variety of ways to view a list as a graph[19]. List arguments can be produced by editing – selecting, joining, truncating, or otherwise fiddling with lists already in the argument stack.

Dyadic functions handle argument lengths as in APL/J, except that a single item list is treated as a scalar. However, a diacritical dipole ( $<$  or  $>$ ) can be used to point to the argument whose items are to be dribbled in so that items of the result are sets of the result from the entire other argument. Used monadically, a function with a dipole is reflexive so that the result is the same whichever dipole is used.

Much of what is done structurally by functions in APL/J can be done by editing commands in a commodity list calculator[20]. Quite otherwise are those functions that do arithmetic dyadically within the list, collapsing subsequences to single items in various ways. The operators of APL/J are used here as the diacritical modifiers acute ´ and grave ` , corresponding to the two solids / and \ [21].

### Clarity and formulae

The list calculator couples a richness of arithmetic with provision for its application to lists of numbers. The arithmetic functions can be applied by the user to the extent needed and understood, and the lists can be entered, inspected, changed and selected as needed.

The richness has merely been suggested here, with some discussion in the appended notes. Users of APL/J will not be at a loss to fill in the detail in various ways.

However, the design is still of a calculator. The functions are applied one at a time. The next two stages are very simple, but they are the essence of the formulator and the basis for algebraic clarity.

The formulator has a function stack in addition to, and separate from, the argument stack. Just as for the calculator, one or two arguments are set up for the next function to use. The next function is either keyed into the bottom slot in the stack or chosen from the function stack and possibly edited before being used.

A function in the stack can be either a basic function, with or without modifiers, or a composite of such functions, or a train of functions.

A simple composite function is like an integer – notationally juxtaposed functions that build their results up progressively from right to left, from least significant to most significant. Thus only the rightmost function has a choice of being used monadically or dyadically; the others are all monadic since they get their argument from the function to their right. So  $\div -$  is the reciprocal of either the negation (monadic) or the difference (dyadic) of the argument(s).

However, a composite function can also have a dyadic point just like the decimal point of a number. Dyadically, the basic functions to the right of the dyadic point are applied monadically to each argument and the two results become the arguments for the functions to the left. Monadically, the basic functions to the right are applied similarly to produce the second argument at the dyadic point, while the original argument becomes the first argument at the point. So  $-\Delta\div$  is

the difference of either the reciprocal of two arguments or of the single argument and its reciprocal[22].

A train of functions is like a list of numbers – a sequence of basic or composite functions separated by spaces. A train serves to repeatedly use its argument(s) while passing results from right to left along the train and combining those results arithmetically with the original argument(s) on the way.

The first, third, and so on functions from the right in a train of odd length are fed the argument(s) of the train as a whole and their results are fed to the intervening dyadic function. A train of even length is in effect made of odd length by prefixing an identity function, +0 say[23].

Parentheses may be used within formulae for various effects. For example, modifiers can be applied to an enclosing parenthesis to affect the enclosed function, and a train within parentheses can be used within a composite function.

### **Depth and algebra**

Composing formulae is the basic level of algebra. The ability to work with patterns of formulae is the next level, and is needed to give depth to the formulator.

A formula pattern or metaformula or template is a functional expression with place holders for substitution of meta-arguments. Templates are keyed in or edited in a template stack alongside the argument and formula stacks[24].

Once a template is selected for use, functions are selected in the function stack or are keyed in and they replace corresponding place holders in the template wherever they occur and as a single function[25]. When replacement is complete the resulting function is placed in the function stack whence it can be used on selected arguments.

### **Rationale**

The formulator is not intended to compete with programming systems such the APL/J family. Indeed it is purposefully and drastically simpler, and not only because of the absence of name assignment and the restriction to list arguments[26].

Rather, the formulator is intended for everyday general use in work and in education – even in the home. In education it opens a wide field beyond simple arithmetic, the arithmetic that the commodity calculator has so badly betrayed,

so that it has the potential to support teachers in greatly improving the development of numeracy in their students.

The design of the formulator embodies a contrast between computation and programming, a contrast I pointed out in a published essay on APL of more than thirty years ago[27]. The formulator uses the symbolism of APL in rejecting use of the Roman alphabet, but ideas like trains come from J.

Advanced programming systems like the APL/J family present a forbidding prospect to anyone simply wanting to do some particular ad-hoc calculation. Arguably, this prospect explains why spreadsheets became so much more popular than APL.

Adoption of the formulator would provide an intermediate step between school arithmetic and APL/J systems that would make it easier for people to shift up[28]. Implementation of the formulator should be a relatively easy adaptation of an APL/J interpreter, and it could find prompt use on various handheld digital devices[29].

The APL/J community would be well placed to produce for teachers and students the material needed for teaching and learning the use of the formulator[30]. Properly done, and with conversion packages that bring formulator stacks into APL/J systems, these could lead to the much wider use of the programming systems favoured by the readers of *Vector*.

## Notes and references

1. Holmes, W.N. (2009) "Truth and Breadth, Clarity and Depth in Algebra", *Computer*, 42(11), 112, 110-111 (PDF in archive: [eprints.utas.edu.au/9474](http://eprints.utas.edu.au/9474))
2. This article is named a commentary. Its body is meant to be a fairly straightforward description of the design of the formulator slanted to readers familiar with the APL family of systems. Citations and further comments are relegated to this trailing section so that a first reading need not be distracted by comments, speculations and other kinds of rant. The hyperlinks are also in this section for convenience of use in the online version of this article. Any reader who thinks my use of footnotes excessive should for perspective read Anthony Grafton's delightful book, *The Footnote: A Curious History*, (Harvard University Press, 1997, ISBN 13: 978-0-674-90215-2; see [www.hup.harvard.edu/catalog/grafoo.html](http://www.hup.harvard.edu/catalog/grafoo.html)).
3. A spreadsheet program I used through an IBM 2741 typewriter terminal in the early 70s was written in APL\360. The author told me that he did it for an insurance firm in New England (USA, not NSW) and that spreadsheets were very widely used in the insurance industry, though many actuaries in Australia used APL bare. Interestingly, this was well before VisiCalc ([bricklin.com/history/saiidea.htm](http://bricklin.com/history/saiidea.htm)).

4. Holmes, W.N. (2003) "Truth and Clarity in Arithmetic", *Computer*, 36(2), 108, 106-107 (PDF in archive: [eprints.utas.edu.au/1573](http://eprints.utas.edu.au/1573))
5. Numbers need to be shown in their shortest form, for example by choosing between a decimal and a vulgar fraction. Long numbers can be shortened on display by scaling, using an italic scale to show that an exact value is available to the user. Also, it might be worthwhile to use the traditional integer superscript to enter and display exact powers, at least as an option.
6. The so-called decimal point separating the integral part of a number from its fractional part needs to be keyed in even for vulgar fractional parts like two thirds. However, simple fractional parts could be displayed with a solidus and without the decimal point, for example:  $2\frac{3}{4}$ .
7. Better still, exact functions that do for exponentiation what quotient and remainder do for division would be a very interesting capability that would allow exact approximations for inexact roots and powers to be investigated. Two pairs of functions would be needed. With, say, 2 and 12 as arguments, one pair would yield 3 and  $1\frac{1}{2}$  as results, the other 2 and  $11/3$ . (Why, oh why, doesn't HTML/Unicode provide `&frac13`;)?)
8. The three numbers on display straight after tapping a function key would best be displayed like
 
$$\begin{array}{r} 12 \\ \times 34 \\ --- \\ 408 \\ === \end{array}$$
9. which is the traditional format used in teaching elementary arithmetic. This is in effect a stack and the user would key in the next number as the second argument for a dyadic function. The stack could be extensive so that the user can select numbers previously stacked for editing and use as a new argument.
10. Using the two special point symbols as basic function modifiers in the exact commodity calculator was motivated by the idea of imitating the ordinary commodity calculator. Otherwise, clarity is greatest if the numeric symbols are all distinct from the function symbols. A shift key to switch between the two sets, for instance with a touch pad on a screen, would keep the keyboard simple.
11. Iverson, K.E. (1980) "Notation as a Tool of Thought", *CACM*, 23(8), 444-465 (HTML in the Web: [jsoftware.com/papers/tot.htm](http://jsoftware.com/papers/tot.htm))
12. While the distinctions can be made typographically, the user needs control over display of the number so that the entirety of a long number is accessible.
13. For clarity, all simple or primitive functions should have a distinct non-alphabetic symbol in the spirit of "Notation as a Tool of Thought", though Greek letters are used in mathematics and engineering as though they aren't alphabetic.

14. Tragically, present day typographical encoding doesn't allow the freedom to put diacritics over any character of choice, a trivial matter in Donald Knuth's old-time T<sub>E</sub>X (see [tug.org](http://tug.org)). For rants on this topic, see my archived essays on general (eprints.utas.edu.au/2005) and sortemic (eprints.utas.edu.au/1564) text encoding, on Unicode (eprints.utas.edu.au/1527), and more recently on cultural theft (eprints.utas.edu.au/1806). In the original essay on the formulator, getting the production artist to put the diacritics above the function symbols was so protracted that changes I would have liked to make to the draft essay went by the board. For example, I was interested in the effect of using special encoding for the non-decimal values of dates, times, and angles (a date can be added to a time but not to anything else) though the examples were both distracting and inconsistent.
15. In the calculator with both exact and inexact arithmetic, literal numbers can be used as superscripts also to numbers, typically as a short way of representing a longish number. However, since in neither the calculator nor the formulator are arguments specified symbolically, such superscripts aren't seen as specifying calculation, merely representation. In function modification, an interesting notational simplification is to use a numeric subscript to mean the same as its negated superscript. A subscript of one then signifies inversion. A zero superscript can be for the identity function for its left or only argument, and a zero subscript for its right or only subscript.
16. A set of values is clearly and conveniently represented as such by enclosing them, systematically sequenced and without duplication, within parentheses. The empty set ( ) is particularly useful for missing values in a sequence. In respect of inexact values, this idea raises the question of whether intervals, which are sets and are also what inexact values strictly are, should be the way inexact values are handled within a formulator. See [wikipedia.org/wiki/interval\\_arithmetic](http://wikipedia.org/wiki/interval_arithmetic).
17. The representation of complex values presents quite a challenge, especially since each component can independently have the different kinds of exactitude and inexactitude. The traditional mathematical convention of expressing a complex value as a sum is clearly flawed and confusing to learners. An imaginary point like the decimal and fraction points is needed, but using *j* as in *J* is inconsistent. The best prefix point for the imaginary part of a complex number is surely the \$ symbol. Another aspect of roots in inexact arithmetic is the problem of representation of multiple determinate or even indeterminate angles but determinate magnitude, which implies that, at least as an alternative, a polar representation using the traditional  $\angle$  symbol should be provided.
18. It might be argued that tables of values and value arrays of higher rank should be introduced in the same step. However, for ad-hoc calculation or algebration this goes much too far. Apart from the problem of displaying and editing such arrays on a small screen, the extra basic functions needed for dealing with them are a severe complication. In any case, much of what is needed of tables can be handled by using a list of sets.
19. A notational abbreviation for simple sequences would be useful for both entry and display of arithmetic progressions. For example, 5[6]78 would go up from 5 in steps of 6 but not go beyond 78.



20. The graph of a list would usually show the value or values of each item vertically in item sequence along the horizontal axis. If complex values are involved, options are given to graph the real or imaginary part, the magnitude or the angle, or the values plotted on the complex plane and connected by lines showing their sequence.
21. For an ad-hoc or learner user, it's easier and clearer for joins, merging, sequencing, rotations, reflections, substitutions, and the like to be done by editing within the argument stack because they don't involve any arithmetic. Keeping such functions separate also simplifies the function set by keeping it for those that do work on values.
22. Clearly the formulator needs a keyboard that provides directly and systematically for the kind of typographical versatility that traditional mathematical notation has exploited to great effect. The American Mathematical Society has built its own mathematical text formatter, AMS-LaTeX ([ams.org/tex/amslatex.html](http://ams.org/tex/amslatex.html)), on top of Donald Knuth's T<sub>E</sub>X, but the formulator doesn't need to be so sophisticated. Nonetheless, the shortcomings of HTML/Unicode present a most unfortunate barrier.
23. This monadic use of a pointed composite function was not described in the original formulator essay, but it adds a very simple and useful capability, in particular that of using more than one dyadic point in a composite function.
24. Functionally, this is just the same as the trains of J, but in the formulator the items of the train must be spaced out with blanks.
25. In the original formulator essay[1] the templates were described as sharing the formula stack. Second thoughts suggest that it would be simpler in operation to keep templates separated from formulae.
26. Having two distinct place holders is simple and consistent. However, it is appropriate not to handle templates in quite the same manner as functions in their stack, because replacement needs to be done either from the function stack or directly from the keyboard. Note also that a place holder in a template stands for a single function, so that selecting a train for placing into a template will in effect enclose it in parentheses. Furthermore, a number could be used as a function within a template, or for giving to a place holder, by using a diacritic to convert it to a function that returns the value itself regardless of any arguments it might be given during evaluation of the resulting function.
27. Of course, it might be useful to be able to name a trio of stacks when saving them for later restoration or for use in documents, but it would be better to use a generated name that could be changed outside the host formulator. This would keep the alphabet out of the way.
28. Holmes, W.N. (1978) "Is APL a Programming Language?", *The Computer Journal*, 21(2), 128-131 (PDF in the Web: [comjnl.oxfordjournals.org/cgi/content/abstract/21/2/128](http://comjnl.oxfordjournals.org/cgi/content/abstract/21/2/128))
29. An important aspect of the formulator, in particular in easing transition to the APL/J family, is the restriction of formulae to compositions and trains. This inculcates the idea of results passing from right to left between functions, and postpones any questions

about the traditional function precedence as seen in expressions like  $2 \times 3 + 4$ , but given up in APL/J.

30. In a tacit programming thread on a J programming forum it was suggested that the 'WolframAlpha computational knowledge engine' ([wolframalpha.com](http://wolframalpha.com)) would provide a formulator. However, I had a look at it and it seemed to be very complex and in a completely contrasting style to the formulator described here.
31. J users will have noticed that I use the traditional APL terminology, viz., function/argument and operator/operand, rather than J's noun, verb, &c. I find it simpler. However, for teaching simplicity there is in the formulator a J-like possibility that would call basic function symbols letters, compositions words, and trains phrases. This linking of numeracy and literacy could help teachers.

# Understanding font embedding

by Kai Jaeger

*kai@aplteam.com*

Explains how to embed the APL385 Unicode font into a web page. Adapted from an article in the APL Wiki.

At the BAA London meeting in February 2010 my friend Stephen Taylor showed me his latest gadget, a brand-new iPhone. It's a fascinating piece of hardware. I tried this and that and finally told it to go to the FinnApl Idiom Library on the APL Wiki. [1]

I got so excited about what I saw that a fellow APLer asked me what was wrong. I told him that nothing was wrong at all: I was simply thrilled that the iPhone not only showed the web site properly with APL characters, it actually used the APL385 Unicode [2] font for this. "Well, it's Unicode, so what?" was all my colleague had to say.

True, the APL Wiki is a Unicode web site, but that does not explain why the iPhone used the APL385 Unicode font. One thing is certain: that font is not installed on an iPhone by default. The magic that makes this work is called *Font Embedding*, and that's what this article is about. I expected all modern browsers to support font embedding – but not an iPhone!

Font embedding means that when a web site is in need of a particular font to display a particular piece of text then this font is downloaded and installed temporarily by the browser in the background. This technology was demanded by web designers who want full control over their page layouts, but as APLers we are benefitting from this as well. It's a chance to display APL code correctly on any machine even if that machine does not have an APL Unicode font installed. This article provides background information about this exciting technique as well as hints and tips regarding the usual obstacles. You need a basic understanding of HTML [3] and CSS3 font rules [4] in order to take advantage of this.

## History

Font embedding was introduced by Microsoft with Internet Explorer 4 a long time ago. This was never adapted by other browsers, and there was a reason for that: IE accepts only *Embedded Open Type* (EOT) fonts.

Now this format, although it has its merits, was owned by Microsoft, and Microsoft never even tried to make it a standard. For that reason other browser vendors kept away from this format. They had to.

In 2009, two things changed. Firstly, almost all other browsers started to implement font embedding, although in a different way. Secondly, Microsoft made the EOT-format public and at the same time started to make it a W3C-accepted standard.

That's good news, although for the time being it means that one has to write different CSS rules for IE and all the other browsers.

## IE and Embedded Open Type fonts

EOT addresses two important issues:

- The font file can be compressed, meaning that it contains not all but only those characters of a particular font used on a particular web site.
- It can be bound to a particular URL. In other words, only when the font is downloaded from a URL contained in the EOT file itself will IE make use of it. This approach was taken to address licensing issues.

The way Microsoft has implemented it in terms of CSS rules is unfortunately different from all the other browsers. You cannot use `local` to avoid a download when a suitable font is installed locally, nor the `format` hint to restrict a rule to a specific font format.

## TrueType fonts

TrueType fonts (TTF) are used by the rest of the world for embedding fonts. Unfortunately there is no way to address licensing issues. This is bad news because it means that, strictly speaking, one may embed only free TTFs. The good news is that thanks to Adrian Smith, the owner and originator of APL385 Unicode, everybody may distribute the APL385 Unicode TTF freely, so we don't have licence issues.

## Converting a TTF font into EOT

In order to use the APL385 Unicode font for font embedding with IE we need to convert it into the EOT format.

If you are interested in how to do this, details are available on the APL Wiki. [5]

But you don't have to. Instead you can download a zipped version of the EOT file[6].

## The CSS

For the time being one needs write special CSS rules in order to make font embedding work with all modern browsers. Since this is likely to change over time, it is not covered in this article. For details see the APL Wiki [5]. In this article I am going to discuss the CSS on a general level.

Note that the link <http://misc.aplteam.com/apl385.eot> used in the examples is a real one but of course you should not use it in your own CSS files.

```
@font-face {  
    font-family: "APLFont";  
    src: url("http://misc.aplteam.com/apl385.ttf");  
}
```

Now APLFont can be used in the same way as any other font-family in CSS. An example:

```
pre {  
    font-family: APLFont, monospace;  
}
```

This tells the browser to use APLFont for any `pre` tags. If that fails for any reason (eg if the URL produces a 404 Not Found error), then the browser will use a local monospaced font instead.

A second example:

```
@font-face {  
  font-family: "APLFont2";  
  src:  
    local("APL385 Unicode"),  
    local("APLX Upright"),  
    local("Courier APL2 Unicode"),  
    local("SiMPL"),  
    local("SiMPL medium"),  
    url("http://misc.aplteam.com/apl385.ttf") format("truetype");  
}
```

This example uses `local` and `format` as well. These are defined by the CSS3 standard, but not implemented by IE. (IE9 is expected to implement it fully.)

Note that in this example the browser is told to use a locally installed font, APL385 Unicode. Only if this fails will the browser continue to the next statement. If none of the specified fonts is actually available on the current system, it finally will download the APL385 Unicode font and try to use it as a TrueType font according to the `format` specification.

The advantage is that the font is downloaded only when no local font is found. It simply saves time.

## Who will take advantage?

Under Windows, these browsers are fine:

Chrome 4.0 and better

Firefox 3.4 and better

Internet Explorer 6, 7 & 8

Opera 10.0 and better

Safari 4.0 and better

Note that using NoScript[7] with Firefox stops font embedding from working. Reason is that downloading anything is potentially a dangerous operation, so by default NoScript blocks the CSS font-face rule. This can be changed on the *Embeddings* tab in the *Options* dialogue of NoScript: untick the 'Forbid @font-face' box and it will work.

However, if you declare, say, `http://aplwiki.com` as to be trusted, which is probably a good idea anyway, then not only JavaScript but also font-embedding will work.

This is new stuff, so make sure that you always use the latest version of your preferred browser.

### **Strange effects**

Browsers use different strategies to display a page which comes with embedded fonts. Safari, for example, does not show anything until it is absolutely clear how the page is going to look. As a result some people complain that the browser is slow. That's not exactly true, because it's a strategic decision made by Apple.

Firefox on the other hand tries to display something as soon as possible, even if the final layout might be different. Therefore Firefox comes up with something quite fast. However, for an embedded font Firefox uses one that is locally available. As a result the page changes more or less dramatically when the embedded font finally becomes available. Some people criticise this as distracting. Again this is a strategic decision.

Since APL glyphs might be completely unreadable with any font chosen by chance I prefer Safari's approach, although Firefox is my preferred browser.

### **Help! It still does not work for me!!**

The way forward is to install a piece of software on your preferred browser that allows you to check the HTTP headers. Chrome, IE and Firefox all have tools available. They will tell you exactly what the browser is requesting and what the server is delivering.

It might be helpful to see a working example [8] – an HTML page with some background information. It makes use of font embedding by referring to this style sheet file[9].

To test font embedding you need either a machine which never saw an installed APL font or a Virtual Machine with the same property. De-installing all APL fonts might not work: sometimes strange caching effects take place by both the underlying Operating System and the browser(s).

## The future

We can expect that

- Internet Explorer 9 will process both local and format.
- EOT will become a W3C-recognized standard and sooner or later all other browsers will accept EOT fonts as well.

Mozilla has proposed WOFF (Web Open Font Format) which will be supported by Firefox 3.6. This new format addresses licensing issues and compresses fonts heavily. For details see [hacks.mozilla.org/2009/10/woff/](http://hacks.mozilla.org/2009/10/woff/) [10]

The future is bright.

Latest updates, working CSS files and technical details are to be found on the APL wiki[11] .

## References

1. Example page on the APL wiki with APL chars  
<http://aplwiki.com/FinnApplIdiomLibrary>
2. APL385 Unicode font file  
<http://www.vector.org.uk/resource/apl385.ttf>
3. HTML <http://www.w3.org/html/>
4. CSS3 font rules <http://www.w3.org/TR/css3-fonts/>
5. Converting a TTF font into EOT  
[http://aplwiki.com/UnderstandingFontEmbedding#Converting\\_a\\_TTF\\_font\\_into\\_EOT](http://aplwiki.com/UnderstandingFontEmbedding#Converting_a_TTF_font_into_EOT)
6. APL385 Unicode in EOT format  
<http://aplwiki.com/UnderstandingFontEmbedding?action=AttachFile&do=view&target=apl385.eot.zip>
7. FireFox Add-on NoScript: <http://noscript.net/>
8. Sample UTF-8 web site  
<http://misc.aplteam.com/APLCharTestUnicode.html>
9. CSS file illustrating font embedding  
<http://misc.aplteam.com/fontface.css>
10. Web Open Font Format  
<http://hacks.mozilla.org/2009/10/woff/>
11. Latest updates, working CSS files and technical details:  
<http://aplwiki.com/UnderstandingFontEmbedding>



## 2: The price of bonds or: APL is not for programmers

by Jan Karman  
*jkarman@planet.nl*

Valuation of bond prices is a big thing at Wall Street and at investment departments of every institutional investor, like insurance companies, pension funds, etc. Buying bonds entitles one to receiving the principal at a given date, plus the contractual interest payments, i.e. coupons. This, being called the *cashflow*, has a value that depends on several parameters. These include the principal, the contractual interest rate, frequency of coupons (usually 2), redemption scheme (at once, linear, annuity, etc), date of closure, date of (starting) redemption and maturity date. Hundreds, even thousands, of institutions deal in bonds and equities: national governments states and provinces; even communities, larger corporations, polder-boards, utility companies, hospitals, schools and many others. Most of those institutions are served by their banks for technical details, but even banks fall back on estimates rather than mathematically exact calculations, because of lack of the relevant knowledge.

### Introduction

Financial transactions invariably involve numerical calculations, and, depending on their complexity, may require detailed mathematical formulations. It is therefore important to establish fundamental principles upon which these numerical calculations and mathematical formulations are based. (Samuel A. Broverman, *Mathematics of Investment*)

We will focus for a while on those mathematics.

Given:

- x the required price, i.e. the exchange rate
- i the nominal interest rate
- r the current revenue on the capital market (yield)
- the present value of the redemptions per unit of capital
- L the present value of the amounts on which interest is to be paid

then we may formulate the price of any loan as follows: (1)

$$x = H + rL$$

If the nominal interest equals the current interest then the exchange rate is at par, thus

$$1 = H + rL$$

and it follows, by definition, that in that case

$$H = 1 - rL$$

We could substitute this value of H in (1) getting (2)

$$x = 1 - (r-i).L$$

It follows immediately that, when  $r = i$ , par shows up.

From the same expression, L is defined by

$$L = \frac{1}{r} (1 - H)$$

and this value substituted in (2) produces (3)

$$x = \frac{i}{r} + \left(1 - \frac{i}{r}\right).H$$

(1), (2) and (3) may be considered as the three general formulas for determining the price of bonds. This trinity forms a 'closed algebra' for the entire theory of mathematics of finance. The reasoning of those formulas requires some not too difficult considerations. Further we may choose the appropriate formula for a particular type of a loan, the one which suits best, with one apparent exception: the annuity loan. The price of this type of loan is simply:

$$x = a_{(r)} / a_{(i)}$$

or, the quotient of an annuity (a) based on r and the same one based on i. Although the reasoning of this form of loan is obvious, expressing this one in any of the three types is an exercise for the final exam, and far beyond the purpose of

this article (just note the development of the redemption components of an annuity loan).

From (2) and (3) it is clear, that, in general, the price of a (financial) commodity is decreasing when the price of money is increasing, and the other way around. Finally, from formula (3) it is easy to see that the price of a non-redeemable loan equals to

$$i / r$$

since  $H = 0$ .

The general formulas can be considered from different view points:

- sum of present values of redemption and interest (1)
- price as deviation from par, i.e. the present value of the difference between yield and interest rate (2)
- price related to a non-repayable loan; here the formula will always show up as:  $i/r + \dots$ , since  $i/r$  is the price of a non-repayable loan (3)

Type (3) is the most practical one for calculations, since it only needs flat annuities, rather than incremental or decremental ones, which are needed in Type (1) and (2). Therefore I used Type (3) for the functions in the programming.

**Justification of the subtitle** At this point the average programmer turns off, drops out, gets his coat and leaves the building, heading for mom and the kids. We, on the other hand, will continue.

**The Portfolio** Let us take for an example the bond portfolio of government or private bonds, being hold by a pension fund. The data is neatly being stored in a computer file like this:

No	Principal	Interest %	Nr. of coupons	Redemp scheme	Date of closure	Redemp date	Maturity date
1234	20000000	5.00	2	1	19990901	20090901	20090901
1235	10000000	4.25	2	2	19990401	20040401	20100401

Note that the first loan is redeemable at once at maturity date and the second one in 10 equal parts starting at 1 April 2004. Also keep in mind that there may be more characteristics, like changing the interest rate halfway, or changing the

number of coupons, and that it can become quite complicated. I hope this example will do for our purpose.

Suppose that for the yearly report (or for other reasons, e.g. a quarterly pooling of the data with a statistical institute like WM-Company) we need to calculate the value of the entire portfolio, then estimates are of no harm and may be allowed.

If we, on the other hand, want to watch the performance of the portfolio, entirely or parts of it, under volatile movements of the yields on the capital market, or that another institution is interested in buying a particular fund or bundle of funds, or that the debtor wants to convert his loan or to redeem it at once, life is not that easy. In those cases the price needs to be established exactly, according to the principles as described in the textbooks in the References. But even in those cases banks often make no bones about estimating (the famous 'wet finger').

Only two categories of experts are at our disposal: actuaries and, to some extent, econometrists.

### The K-script

It is impossible to expose the entire script here. A few topics will be selected. First, there will be a frequent use for basic functions like present value, annuity, etc. So, we might design special auxiliary functions for those (decreasing annuity just for curiosity).

```
v: {[r;n](1+r)^-n}          / present value
an: {[r;n](1-v[r;n])%r}     / annuity
Dan: {[r;n](n-an[r;n])%r}   / decreasing annuity
```

Also we may design functions beforehand for the different loan types:

```
a: {i%x}                    / non-repayable
b: {[i;x;n](i%x)+(1-i%x)*v[x;n]} / repayable after n years at
once
c: {[i;x;n](i%x)+(1-i%x)*an[x;n]} / repayable yearly 1/nth
d: {[i;x;n]an[x;n]%an[i;n]}      / annuity
```

Date conversion:

```
aymds: {+-1_ ,/(0 4 6 _+$x), '"-"} / "yyyymmdd" to "yyyy-mm-dd"
```

The data on the file is in flat ASCII, delimited by HT, so we need a splitter:

```
frd: {pos: &(*q: 0:x) = _ci 9
+: '(0,pos-!#pos) _ +q _dv' "\t"}
```

We will need a function for the appropriate handling of the data (making known the variable names, reading the data and formatting them):

```
conv: {[x;y;z]          / x y z being var nms, data, formats
      t:.(x;0$y;.,(`e;0))
      .[t;(~x;`f);::z]}
```

Finally we come to the main function, in which all the work is done:

```
p: {[cr;cp;y;id;md;rd;sc]
    j:-1+(1+cr%100000*cp)^cp
    m:(%360)*0|(ds'aymds rd@&~sc)-__t%86400
    n0:(%360)*(-/(ds'aymds' (,md),,rd))@&~sc
    n1:(%360)*(ds'aymds md@&sc)-__t%86400
    a:.(`e`f;(0;8.5$))
    t:+(`_n;&#cr;a)          / courtesy Arthur
    x0:(j@&~sc)*an[y;m]      / deferral of red.
    x1:v[y;m]*c[j@&~sc;y;n0] / type c
    x2:b[j@&sc;y;n1]         / type b
    .[t;();::((x0+x1),x2)[<<sc]]}
```

```
/ watch that little gem at the end of the bottom line!
/ it's meshing catenated strings when final order is known
```

The meshing function, `x[<<sc]`, is used because the loans of types b and c were split and processed separately. Afterwards they need to be catenated in the order we started in and we saved. The original function comes from Stephen Jaffe, *Topics for a Second Course in APL* (Manchester 1986).

There is only one control: that for the current yield.

```
\d .k.Y
yld:0.06;yld..l:"";yld..f:8.4$100*
yldi:"yld+:%800";yldd:"yld-:%800"
yldi..c:yldd..c:`button
yldi..l:"+";yldd..l:"- "
.k.Y..l:"Yield %"
.k.Y..a:(`yld;`yldi`yldd)
```

## Picture

Investments								
Portfolio: Bonds								
fnum	amount	rate	cpns	scheme	date	redd	matd	price
3219	5000000	7.300	2	0	1987-10-29	1999-01-18	2003-01-18	1.03228
3780	10000000	8.420	1	1	1992-05-11	2002-05-27	2002-05-27	0.81619
370	750000	8.500	2	0	1970-02-02	1971-02-01	2010-02-01	1.27735
458	1000000	8.500	2	0	1971-01-15	1972-01-15	2011-01-15	1.27735
3153	10000000	7.000	1	0	1987-12-30	1998-12-30	2002-12-30	1.02252
3197	5000000	7.250	1	0	1988-01-18	1998-10-01	2002-10-01	1.02815
3880	10000000	6.770	1	0	1993-07-19	1999-09-01	2003-09-01	1.01734
3659	5000000	9.440	1	1	1991-03-04	2000-12-18	2000-12-18	0.66469
3747	10000000	8.870	1	1	1991-12-11	2002-04-02	2002-04-02	0.77578
3770	10000000	8.330	2	0	1992-03-02	2003-03-05	2007-03-05	1.05639
3648	10000000	9.250	1	0	1990-11-27	2001-02-15	2002-02-15	1.03087
3769	10000000	8.900	1	0	1991-12-09	1998-02-17	2002-02-17	1.06532
4066	50000	7.250	1	0	1994-02-16	1995-03-01	2000-03-01	1.03318
3879	7000000	7.000	1	1	1993-03-09	2001-01-15	2001-01-15	0.90372
3868	5000000	7.000	1	1	1993-02-23	1999-06-15	1999-06-15	0.87784
3857	10000000	7.100	1	0	1993-03-02	1999-03-15	2006-03-15	1.03754
4022	14000000	6.150	1	0	1994-01-20	1995-03-15	2009-03-15	1.00849
4011	4666670	6.200	1	0	1993-12-16	1995-03-02	2009-03-02	1.01132
4000	7000000	6.080	1	1	1994-02-07	2000-12-15	2000-12-15	0.99219
3990	10000000	6.080	1	0	1993-11-29	2001-01-28	2004-01-28	1.00147
3989	10000000	6.000	1	0	1993-11-29	1998-01-05	2002-01-05	1.00000
2614	8000000	8.750	1	0	1985-01-15	1995-08-10	1999-08-10	1.06194
2625	4000000	8.625	1	0	1985-01-15	1995-08-29	1999-08-29	1.05912
3978	10000000	6.850	1	0	1993-10-12	2005-01-17	2009-01-17	1.01914
3967	10000000	6.650	1	0	1993-09-29	2005-01-17	2009-01-17	1.01464
2658	4000000	8.500	1	0	1985-01-15	1995-09-03	1999-09-03	1.05631
3956	5000000	6.700	1	0	1993-10-12	2005-01-17	2009-01-17	1.01577
2680	9000000	8.375	1	0	1985-01-15	1995-10-01	2004-10-01	1.09774
3945	10000000	6.800	1	1	1993-10-12	2004-05-13	2004-05-13	0.96050
3472	5000000	7.825	1	0	1989-11-07	2001-01-15	2005-01-15	1.04110
2790	5000000	8.375	1	0	1985-10-03	1996-03-20	2000-03-20	1.05349
3791	7500000	8.250	1	1	1992-09-23	2002-10-21	2002-10-21	0.84193
4055	10000000	6.450	2	0	1994-07-04	2000-05-02	2004-05-02	1.01248

Note:

There are two amortization schemes:  
0 = with 1/n of duration as from redemption date  
1 = at once at maturity date

Yield %

6.0000

+ -

Fig. 1 – There is only one control: the current yield; the exactness is 1/8%. The prices of the loans are being displayed in the rightmost column and adjusted timelessly on every click on yield. Note the little scroll button top-right.

## Show

This picture is the result of

```
/ *** The show
\d ^
.k..l:"Investments"
.k..a:(`D;(`Comment;`Y))
/ .k.D.[`y]: 13
`show$`.k
```

## Online

The entire application is online available and can be downloaded freely from my website[4].

## References

1. Broverman, Samuel A. *Mathematics of Investment and Credit* 1991, 1996, ACTEX Publications, Inc. Toronto
2. Haaften, Dr M. van *Leerboek der interestrekening* 1929, P. Noordhoff N.V. Groningen
3. Hage, Joh. *Koersberekening* 1941, P. Noordhoff N.V. Groningen
4. <http://www.ganuenta.com/index-bonds.htm>

# Punctuation and rank

by Norman Thomson

Lynne Trusse created a best-selling book on the art of punctuation, famously drawing its title from the story of the panda which, after visiting a restaurant “eats shoots, and leaves”, a plausible option allowing for a touch of anthropomorphism, and one which might not cause too great a disturbance to other diners. If on the other hand the panda “eats, shoots, and leaves” the effect is likely to be very different.

Much has been made of the manner in which the constructs of J were inspired by and derived from the parts of speech of ordinary language grammar, nouns, verbs and so on, despite which little reference is made to punctuation. While analogies should not be pushed too far – e.g. unlike *shoots*, there can never be any verb/noun ambiguity for primitive J objects – the concepts of explicit punctuation as provided in J by parentheses and space, and implicit punctuation through the two verb forms *hook* and *fork*, can be helpful in interpreting expressions.

It can be tempting to think of conjunctions such as @ (*atop*) as punctuators as in e.g.

```
((i.#)t);(i.@#)t=. 'abcde'
+-+-----+
|5|0 1 2 3 4|
+-+-----+
```

However the role of @ in the above is that of a neologiser, that is, it constructs a new compound verb, call it ‘index-tally’, operating in scalar fashion on the items of the object *t* to its right. The operational details of such verbs lead naturally to consideration of the most subtle of all J concepts, namely *rank*.

Consider three compound verbs which differ only in ‘punctuation’.

```
v1=.>:@i.@#
v2=.>:@(i.@#)
v3=.(>:@i.)@#
```

The *effect* of all three verbs is the same, that is they are *semantically* equivalent as demonstrated by



```

      (v1 t);(v2 t);(v3 t=. 'abcde')
+-----+-----+-----+
| 1 2 3 4 5|1 2 3 4 5|1 2 3 4 5|
+-----+-----+-----+

```

More general questions are:

1. For any verbs *a b c*, to which (if either) of the two forms *a@(b@c)* and *(a@b)@c* is *v1 necessarily* equivalent?
2. Is *v2* equivalent to *v3*, or, in mathematical terms, is the conjunction *@* associative?

Intuitively it should not be, for the same sort of reason that *eats, shoots and leaves* has a different meaning from *eats shoots, and leaves*. The scope rule for conjunctions is that they bind closely on the right, which means that it is *v3* which is equivalent to *v1*, and this of course is guaranteed by the J interpreter. Using conjunctions is equivalent to coining new verb-names from old in English, so that the meaning of *v2* is ‘increment-(index-tally)’ as opposed to *v3* which is ‘(increment-index)-tally’.

In learning J it takes a degree of mental adaptation to grasp the idea of a compound verb such as ‘index-tally’ let alone triple compounds like *v2* and *v3*, and also to appreciate that the meaning of ‘index-tally’ is not *index then tally*. This difference can be demonstrated by:

```

      (|. @ *: ) t = . 1 2 2 3 NB. rotate-square
1 4 4 9
      (|. @ *: ) t NB. rotate-following-square
9 4 4 1

```

which suggests that the J terminologies ‘a atop b’ and ‘a at b’ are rendered more comprehensibly in pseudo-English as ‘a-b’ and ‘a-following-b’. The compound verb ‘rotate-square’ has rank zero because it takes on the rank of its rightmost component, a property which, for obvious reasons, is called *rank inheritance*. However, in ‘rotate-following-square’ the colon in *@:* can be thought of as signalling a pause in which rank is readjusted before the left hand verb is executed.

The nature of the arguments which can be presented to a conjunctionally compounded verb depend on the arguments presented to its rightmost verb, and so returning to *v1*, *v2* and *v3*, all of these require an argument acceptable to *tally*. This can be any J object since all J objects are fundamentally lists and so can be tallied at their topmost level. Also since the three verbs are to be executed in

right to left succession it would seem, superficially at least, to make no difference how they are parenthesised as the transformed data is ‘passed down the line’ from right to left. However, as ‘rotate-square’ shows, rank inheritance has to be taken into account in the general case.

## Rank lists

Verbs can be categorised according to their rank properties in a manner comparable to conjugation in classical language grammar (see the appendix). Every verb has a rank list, viz.

monadic rank   left rank   right rank

which can always be explicitly obtained by applying the *basic characteristics* adverb `b.` and using `0` as right argument of the resulting verb. Rank can be infinite, and most verbs of infinite rank are *structural*, meaning that, like *box*, they are designed to operate on their argument or arguments as a whole, that is, they do not ‘penetrate’ the outer shells of objects. *Grade-up* is a useful illustration of the notion of ‘infinite rank’ because however large the rank of its argument, it orders objects at the next lowest rank level, thus

```
/:i.2 10 20 30 40 NB. gradeup two 4 dimn1 objects
0 1
```

```
/:i.5 10 20 30 40 NB. gradeup five 4 dimn1 objects
0 1 2 3 4
```

*Infinite* is the default verb rank, which is also the rank of all but the simplest user-defined verbs, since the interpreter could potentially be forced to perform exhaustive and unproductive effort to work out the *de-facto* rank, and so it makes the ‘safe’ assumption of *infinite*. However the rank conjunction allows rank to be used flexibly as in

```
mean=.+ / % #
mean0=.(+/%#)"0      NB. scalarised mean
(mean i.5);(mean0 i.5)
+-+-----+
|2|0 1 2 3 4|
+-+-----+
```

Here are the relevant rank lists:

```
(mean b.0);(mean0 b.0)
+-----+-----+
|_ _ _|0 0 0|
+-----+-----+
```

## Rank inheritance

Returning to 'rotate-square' whose rank list is 0 0 0, although *rotate* is a rank-1 verb, rank inheritance forces rotation at rank 0 (that is, equivalent to an explicit "0") and so it inherits a list of rank-0 objects (scalars) each of which has to be treated as a list, with the result that it does nothing. However, if rank is *not* inherited as with |.@:\*, then rotation applies to the list of squares as a single entity of rank 1.

The equivalence of  $a@(b@c)$  and  $(a@b)@c$  (i.e. associativity) depends on the rank of the inheriting verb being no greater than that of the giving verb, something which will certainly take place if  $a$ ,  $b$  and  $c$  are all rank-0 verbs, but which has to be examined in terms of verb rank properties when this is not the case. Rank inheritance from higher to equal or lower creates no problems as in

```
(>:@i.)6      NB. rank 0 inherits rank 1
1 2 3 4 5 6
```

However, compare

```
(>:@>:)7 3 5   NB. rank infinite inherits rank 0
0
0
```

in which each of the three incremented values is upgraded separately, with

```
(>:@/:)7 3 5   NB. rank 0 inherits rank infinite
2 3 1
```

The next two examples involve verbs of equal ranks, again there is no inheritance issue, although changing the order of the verbs gives a different result because the grade-up of a transposed matrix is not the same as the transpose of a grade-up of the original matrix.

```
(>:@|:)i.2 3   NB. both ranks infinite ..
0 1 2
```

```
(|:@/:)i.2 3    NB. .. but the result is different
0 1
```

## Mood, transitivity and commutativity

J verbs are restricted to the imperative mood apart from the verb 'to be' (*copula*). Mood is independent of transitivity, meaning that a verb is either monadic (intransitive) or dyadic (transitive). For transitive verbs the arithmetic commutativity of say + means that  $2 + 3$  is in every respect equal to  $3 + 2$ . However when a computer does addition it is impossible for both arguments to be fetched simultaneously, and so, analogously with transitive verbs in English for which the subject is in some sense 'stronger' than the object, the left argument of dyadic verbs binds more strongly than the right. This becomes apparent when repetition is invoked by the power adverb. Thus  $2 + ^:(2)3$  means *add 2 twice to 3* (answer 7), as opposed to *add 3 twice to 2* (answer 8).

## Conjugations

Returning to the conjugations, scalar verbs which have rank 0 are the 'most penetrating', meaning that unless otherwise modified by the rank conjunction they operate at the lowest cell levels. The ordinary arithmetic verbs are thus all of rank 0. Second-conjugation verbs are all monadic and operate at the level of lists, such as  $i.$  (*index generator*) and  $\#:$  (*base 2*). The conjugations range from the most penetrating in the first conjugation through to those of the 4th and 5th conjugations which handle objects at a macro level. In between at the third conjugation are a set of verbs which are the counterpart of irregular verbs in natural-language grammars.

## Pseudo-punctuation

Returning to punctuation, there are three verbs, all of infinite rank, which can be thought of as providing a 'pseudo-punctuator' role. These are  $,$   $;$  and  $,.$

In each of the examples below the pseudo-punctuator is the middle tine of a fork, and the sum and difference of a list can be 'punctuated' in the following ways:

```
5 4(+,-)2 0      NB. sums, then differences
7 4 3 4
```

```

      5 4(+;-)2 0      NB. boxed sums, boxed differences
+---+---+
| 7 4|3 4|
+---+---+

```

```

      5 4 (+,.-)2 0    NB. sums, diffs as separate dimensions
7 3
4 4

```

The verb [ , also of infinite rank, provides pseudo-punctuation in the form of a statement separator:

```

      a=.2 [ b=.3
      a,b
2 3

```

which works because the above line is effectively

```

      a=.2[3
7

```

Square brackets can sometimes give rise to what looks orthographically like ‘verb parentheses’ as in

```

      2(*:@[+])3
7

```

although arguably the above phrase would have been written with more clarity as

```

      2((*:@[ ) + ] )3

```

## Redundant punctuation

As the above example shows, redundant parentheses can be invaluable in clarifying the meanings of tacit definitions, although, like all good things, it can be overdone, and too many parentheses can sometimes be just as confusing as too few. Once a string of J symbols exceeds about seven characters even an expert reader’s eyes begin to glaze over. Consider for example:

```

      lens=."<1 @: ,.3&":@: i.&' '"1

```

An example of using `lens` might help to clear the fog :

```

      lens >'Florida';'California';'Alaska'

```

```
+-----+-----+-----+
|Florida      7|California 10|Alaska      6|
+-----+-----+-----+
```

Things become clearer still if `lens` is rewritten with some parentheses and an explicit space for the hook :

```
lens1=."<"1 @: (, . ((3&" : )@:(i.&' ' )"1))
```

But following the seven-character rule it would have been even better to articulate some of the bits by giving meaningful names to verbs along the following lines :

```
boxrows=."<"1
format=.3&" :          NB. width = 3 characters
length=.(i.&' ' )"1    NB. gives length of string
lens2=.boxrows@(<,(format@length))
```

Interestingly, although the above four lines appear at first sight to have only a few primitive symbols, all such symbols in `lens` are faithfully reproduced. Arguably it would have been better to write `lens` this way in the first place as this helps to contrast the `@:`s which define compound verbs, with the implicit punctuation in the hook `<,(format@length)`.

Thoughtful punctuation can often help documentation. As a further example, most readers would find that on first sight the following verb definition conveys little of its purpose:

```
verb=.(+/@: *: @: -+/%#)%<:@#
```

With some redundant parenthesising and renaming, and use of space to emphasise the fork, things become a little clearer:

```
sdest=.(+/@: (*:@: (-+/%#))) % (<:@#)
```

and with a little more renaming of the parts

```
sum=.+/
mean=.+/%#
mdev=-mean      NB. mean deviation
nminus1=.<:@#    NB. n minus 1
sdest1=.sum@: (*:@mdev)%nminus1
```

the objective of providing the usual form of standard deviation estimate from a sample should become reasonably apparent.

## Cap

When cap ([ : ]) was introduced it was argued that it allowed indefinitely long *trains* of verbs to be written without parentheses, thereby implying that parentheses were inherently undesirable. The analogy in English is to favour long strings of words without punctuation, which may not be to everyone's reading taste. Forks and hooks work well because the human mind assimilates readily twosomes and threesomes, but thereafter the reverse is true, that is **a b c d e** wrongly suggests *a then b then c then ...* whereas **a b(c d e)** gives a natural visual picture of the correct meaning. Continuing in the vein of the previous example **\*: - + / % #** does not at first sight reveal its meaning whereas **\*: - ( + / % # )** says with reasonable clarity *subtract the mean from the squares*.

## Space

A first step in the parser of most compilers and interpreters is to remove redundant spaces which are often highly desirable at the orthographic level, for example to underline the fact that three primitive verbs form a fork. Successive digraphs can lead the reader through an unnecessary initial step of disentanglement as, for example, in verb above which, even without the suggested parenthesising and breaking down into smaller verbs, would be easier to interpret if written

**verb = . ( + / @ : \* : @ : - + / % # ) % < : @ #**

that is, *(add following square following mean-adjust) divide by decrement-tally*.

On the other hand spaces are probably best omitted between verbs and their objects, e.g. **i . 5** is probably less clear than **i . 5**, although it is best not to be too dogmatic.

## Appendix

### 1st Conjugation, rank vector = 0 0 0

*Logicals (monadic)*

**- . (dyadic) = ~: < <: > >: +: \*:**

*Arithmetics (monadic and dyadic)*

**+ - \* % ^ ^. < . > . | ! %: + . \* .**

*Arithmetics (monadic)*

**- .**

*Algorithmics (monadic)*

p: (ith. prime)

*Algorithmics (dyadic)*

? j. o. r. q:

## 2nd Conjugation, monadic, list oriented

i. { ;: #. ". #: p. (polynomial)

A. (Anagram Index) C. (Cycle)

## 3rd Conjugation, irregular

*monadic*

%. (rank 2)

*dyadic*

#: (1 0) p. (1 0) {(0 \_) A. (0 \_) %. (\_ 2) C. (1 \_)

## 4th Conjugation, dyadics with left rank=1, right rank=infinite

\$ |. |: # {. }. ": {:: (fetch)

## 5th Conjugation, all ranks infinite

*monadic*

= < ~. ~: {: }: #: \$ |. |: # {. }. ":

L. (Level) {:: (match)

*dyadic*

-. -: i. ". E. (Member of Interval)

*monadic and dyadic*

, ,. ,: /: \: ; e. \$. \$: [ ]

s: (symbol) u: (unicode) x: (extended precision)

*constant functions*

that is 9:, \_8:, ..., 0:, 1:, 2:, ... 9:, also \_: (infinity)



# P R O F I T

## 2. Wastage

by Howard A. Peelle

*hapeelle@educ.umass.edu*

J programs are presented as analytical tools for expert backgammon. Part 2 here develops a program to compute the number of pips expected to be wasted while bearing off pieces in one inner board (without contact).

The inner board is represented as a list of six integers. For example, 0 pieces on the 1, 2, and 3 points, 2 on the 4-point, 3 on the 5, and 4 on the 6:

```
board =: 0 0 0 2 3 4
```

To count pips for bearing off, sum the board times a list of the point numbers:

```
,pips =: +/ board * 1 2 3 4 5 6
47
```

Use program N (from [1]) to compute expected number of rolls to bear off:

```
,n =: N board
6.56681
```

Expected number of pips to bear off is the expected number of rolls n times the average number of pips per roll (8 1/6):

```
n * 8+1%6
53.6289
```

Wastage is this expected number of pips minus the pipcount.

```
,wastage =: (n * 8+1%6) - pips
6.62895
```

Define a program to compute wastage for any inner board:

```
Wastage =: (N * 8+1:%6:) - Pips
Pips =: +/ . * Points
Points =: 1: + i.@6:
```

For the example above:

```
Wastage 0 0 0 2 3 4
6.62895
```

Wastage for all possible full distributions (with all 15 pieces) in the inner board follows:

```
all =: (6#16) #: i.16^6
boards =: (15 = +/"1 all) # all
```

(See Appendix for an alternative way to generate these boards using partitions.)

Wastages for all full inner boards are:

```
wastages =: Wastage"1 boards
```

The minimum wastage is:

```
<./wastages
7.06895
```

The board with minimal wastage is:

```
boards #~ wastages = <./wastages
0 0 0 3 5 7
```

Other distributions can be searched similarly:

# Pieces	#Boards	Minimum Wastage	Best board
15	15504	7.06895	0 0 0 3 5 7
14	11628	7.02941	0 0 0 3 5 6
13	8568	6.98028	0 0 0 3 4 6
12	6188	6.90551	0 0 0 2 4 6
11	4368	6.82085	0 0 0 2 4 5
10	3003	6.75322	0 0 0 2 3 5
9	2002	6.62895	0 0 0 2 3 4
8	1287	6.51014	0 0 0 1 3 4
7	792	6.36102	0 0 0 1 3 3
6	462	6.14734	0 0 0 1 2 3
5	252	5.88967	0 0 0 1 2 2
4	126	5.52802	0 0 0 1 1 2
3	56	5.30579	0 0 0 1 1 1
2	21	4.62037	1 0 1 0 0 0
1	6	4.20833	0 0 0 0 0 1

## Reference

1. Peelle, Howard A. "Backgammon Tools in J: 1. Bearoff Expected Rolls", *Vector*, Vol.24, N°2

## Appendix

Program to compute wastage:

```
Wastage =: (N * 8:+1:%6:) - (+/ . * 1: + i.@6:)
```

See [1] for script with definition of N to compute expected number of rolls to bear off.

Co-recursive program to compute all partitions of an integer x into y parts:

```
ELSE =: `
WHEN =: @.
Partitions =: ;@AllParts
AllParts =: <@Parts"0 >:@i.
Parts =: Ps ELSE (,.@[]) WHEN (]=1:)
Ps =: Add1 ELSE (#1:) WHEN =
Add1 =: 1: + ] {."1 - Partitions - <. ]
```

For example:

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

Program to produce a table of permutations:

```
Perms =: i.@! A. i.
```

All full distributions of 15 pieces on 6 points in an inner board:

```
boards =: ~. ,/ (Perms 6) {"1/ 15 Partitions 6
```

Number of full inner boards:

```
#boards  
15504
```

All wastages:

```
wastages =. Wastage"1 boards
```

Average wastage:

```
Average =: +/ % #  
Average wastage  
15.7857
```

Minimal wastage:

```
<./wastages  
7.06895
```

Board with minimal wastage:

```
boards #~ wastages = <./wastages  
0 0 0 3 5 7
```

Top ten minimal boards:

```
10 {. boards /: wastages  
0 0 0 3 5 7  
0 0 1 2 5 7  
0 0 1 3 5 6  
0 0 0 2 5 8  
0 0 1 3 4 7  
0 0 0 2 6 7  
0 0 0 3 6 6  
0 0 0 4 5 6  
0 0 0 3 4 8  
0 0 1 2 6 6
```

## 4: The year 1998

by Neville Holmes

*neville.holmes@utas.edu.au*

Functional calculation does with operations applied to functions and numbers what numerical calculation does with functions applied to numbers. In preceding articles an introduction was given to what could be done with one commonly available tool for functional calculation, using a notation called J, then details were given of simple numerical calculation, and then simple structural calculation. This article is intended to allow the reader to consider how simple structural calculation can be done in J by showing numeric expressions to produce whole numbers below 100 starting from the enlisted digits of the number 1998.

### Calculation

Preceding articles have been introducing numerical calculation and structural calculation using the interpreter for the J notation. This article reverts to the pattern of the third one, showing how the whole numbers from 0 to 99 can be constructed from the digits of a year – the year 1998. This time, however, digits are to be used in lists or in a list, so that the structural functions reviewed in the previous articles can be used.

Of course, the scalar functions used in the previous exercises can also be used here, but the following structural functions are to be preferred. The first table shows structural functions used for building lists. These are a fairly mixed bunch, and indeed the shape and tally functions are not actually used for building but for reporting what has been built.

\$ shape	reshape				
# tally	copy	#. unbits	undigits	#:	bits digits
? deal					
< box					
		+. cartesian			
				":	format format
> unbox					
		*. polar			
				;;	words
; raze	link				
		i. integers			
				q:	factors

The second table shows structural functions used for extracting or rearranging the values in a list. Again, these functions are a mixed bunch and not all will be found useful in the task of generating numbers.

		. reverse	rotate	:	transpose	transpose
, ravel	append	,. knit	stitch	,:	itemise	lamine
				/:	sort up	
				\:	sort down	
{ from		{. head	take	{:	tail	
		}. behead	drop	}: curtail		
		~. nub				

The third table shows some miscellaneous functions which extract data about their arguments.

	%. invert	project	
= classify			
			-: match
	e. raze in	member	
			~: sieve
	i. index		
			/: grade up
			\: grade down

Finally, there are two operations which are so useful for working with structures that it would be masochistic to ignore them. Both these operations, whose symbols are ~ and /, are monadic, which means that they suffix a function which is the target of their operation. The resulting function may be used monadically or dyadically.

For example,  $2\% \sim 3$  will divide 2 *into* 3 rather than *by* 3 (its arguments are reversed), while  $\wedge \sim 3$  raises 3 to the power 3 (its left argument is copied from its right argument).

Further,  $+/\mathbf{i}.100$  will add up the first hundred integers,  $2\ 3+/\mathbf{4}\ 5\ 6$  will give a table with two rows and three columns containing the sum of each of  $2\ 3$  with each and every  $4\ 5\ 6$ , and  $*/\sim \mathbf{i}.12$  will give a 'times' table for the first twelve integers, or rather  $*/\sim >:\mathbf{i}.12$  will give the more familiar table.

## Making 1998 give 0 to 19

The four digits of 1998 may be used as a single list, that is, as  $1\ 9\ 9\ 8$ ,  $19\ 9\ 8$ ,  $1\ 99\ 8$ ,  $19\ 9\ 8$ ,  $199\ 8$ ,  $19\ 98$ , or  $1\ 998$ . Otherwise two lists may be used, that is, the two lists  $1\ 9$  and  $9\ 8$ , or a scalar and a list, such as  $1$  and the list  $9\ 98$  or as the list  $1\ 9\ 9$  and the scalar 8.

The task is to combine them in as short and simple an expression as possible, to yield each of the numbers between 0 and 19 (here, 99 later), and to yield them as scalars.

As a matter of aesthetics, parentheses are avoided as far as possible. Also as a matter of style, the negative sign is avoided and subtraction or negation is used to similar effect.

To save listing space, in the following it will be assumed that

$x =: 1\ 9\ 9\ 8$

has been issued. Expressions that use  $x$  are preferred as using the simplest but most extensive list of integers.

0	$ /x$	$=/19\ 98$	10	$\{.1\ 9+9\ 8$	$\#1\ 9":9\ 8$
1	$1[9\ 9\ 8$	$<./x$	11	$--\sim/x$	$+/*:\mathbf{i}.\sim x$
2	$1\ 9\ \mathbf{i}.98$	$\#1\ 9;9\ 8$	12	$-.--\sim/x$	$+/19\ 9-8$
3	$1\ 9\ 9\ \mathbf{i}.8$	$\#\sim.x$	13	$<.+/-:x$	$>.>./\wedge.!\mathbf{x}$
4	$\#x$	$+/*x$	14	$>.*/\wedge.19\ 9\ 8$	$+/>:19\ 9-8$
5	$+/\mathbf{i}.\sim x$	$>.\{:\mathbf{!}\%:x$	15	$\#.*x$	$19 */9\ 8$
6	$+//:x$	$>./!\%:x$	16	$\#./:x$	$+/19\ 9 8$
7	$-/ \mathbf{x}$	$\#" :x$	17	$+/1\ 9]9\ 8$	$>.%/19\ 9\ 8$
8	$\{ :x$	$\%%/x$	18	$+/\sim.x$	$-/19\ 9\ 8$
9	$>./x$	$1\{9\ 9\ 8$	19	$+/\}:x$	$\{.19\ 9\ 8$



Two expressions are given in the table for each number. Note carefully that at least one of the arguments in each expression is a list.

## Making 1998 give 20 to 99

Making numbers beyond 19 follows a similar pattern, and it's convenient here to take them twenty at a time.

20	--~/19 9 8	#.<:-:x	30	#.~.x	+/1,>:9 8
21	19+#9 8	<./ 1 9 j.9 8	31	#.}:x	+/>:x
22	#.\:x	+/,~>:#.x	32	-:*/19 9 8	*/<.+:%1 o.9 9 8
23	+/<:x	p:{:x	33	+/<:19 9 8	#":*:*:,~x
24	#.1 9 9 8	!-:{:x	34	<.!%:+>/x	+/,~}.19 9 8
25	--~/ .x	<.*%:x	35	+/,q:19 98	#.-:x
26	<.#.%:19 9 8	1#.9 9 8	36	+/19 9 8	</*/%:19 9 8
27	+/x	19+{:9 8	37	>.*%:19 9 8	#.1 9+9 8
28	{.+ /1 9+ /9 8	+//:,~x	38	{.+ :19 9 8	+ /19>>.9 8
29	>./p:x	+ /1+9 9 8	39	+ />:19 9 8	-:->:- /19 98

Once into the twenties, the integer 19 becomes very useful, though it was also handy in the teens.

40	<.*/-:x	+/-<:p:x	50	<.+:%:*/x	#.>:-:x
41	<.!%:{.19 9 8	- /1 9,-:98	51	#.-:19 9 8	-:1+99+#,~8
42	+ /#.  :+:#:x	+/-:p:x	52	+ /+:}.x	<.+ /!%:19 9 8
43	<: + /+: + /q:>:x	+ /1 9 9+8	53	>./>.!-:x	1{+ /+ /~9 9 8
44	+ /1 9,~,~9 8	+ /19 9+8	54	<.^#x	+ /+:x
45	#.#.=x	*/>.%:19 9 8	55	+ /19+9 8	#.<:x
46	+:p:{:x	+ /}:~,~x	56	<.o.- /19 9 8	+ /,~}:19 9 8
47	<.* /+: %1 o.9 9 8	p: +: #":x	57	1+*./<:9 9 8	+ /19+>:9 8
48	* /#.=x	#.<.-:19 9 8	58	+ />+:x	<: #.}: .x
49	-: { :1 9 98	- /*:1-9 9 8	59	-: >: + /19 98	+ /1 9,-:98

From the forties on, the integer 98 comes into play more often.

60	-: #.1 9 98	>.o.{19 9 8	70	#.x	<:p:>:- /19 9 8
61	#.1 C.9 9 8	+ /p:}:x	71	+ /1 9#<:9 8	p:{.19 9 8
62	1*#.9 9 8	#.}.x	72	*./x	>./1 9*9 8
63	- /*: .x	- /p:<:19 9 8	73	1+*./9 9 8	p:19+##9 8
64	/*:19 9 8	-.- /*:x	74	+ /#.  :+:>: #:~.x	
65	+ /*:1 9-9 8	- /p:19 9 8	75	19+* /<:9 8	#.,~1 9--~9 8
66	+ /+:<:19 9 8	#.19 _9 8	76	+ /1 9 9#-:8	+ /1 9 9*-:8

67	p:-/19 9 8	#.->: .x	77	-/*:>:x	+/,1 9\$9 8
68	-:#.19 98	+/,1 9\$<:9 8	78	-.-//*:>:x	+:+/>:19 9 8
69	#. .19 9 8	#.1+9 9 8	79	~-/19 98	1 9#. <:9 8

In the sixties and seventies, squaring (\*:) becomes useful.

80	*/<:+/>: :=x	-.-/19 98	90	-/1 9 98	*./>:x
81	1 9#. .9 8	+/1 9#9 8	91	-/1+99 8	19+*/9 8
82		+/1 9,*/9 8	92	-/>:-:1 99 8	+/<:p:>:x
83	+/<:#.>:#:x	#.<:#.+:x	93	-.~/1 99 8	*/p:1 9!>:9 8
84	%:*/+:1 9 98	+/p:x	94		gt;.-/%-.1 o.9 9 8
85	#.>:x	-:+/ ,1 9*/9 8	95	<:*/>.%:>:x	.<:19 9 8
86		/,1 9\$>:9 8	96	-:>:-/199 8	*./#.*+:x
87		/#.>:#:x	97	<:1 9]98	#.>:#.+:x
88	<.+/>:o.x	+/>:p:x	98	{:1 9 98	+/,1 9*./9 8
89	1 9#.9 8	-/1-9 98	99	1{.99 8	1 9#.>:9 8

Once in eighties, simpler expressions become possible because integers like 98, 99, and 199 can be brought into play.

## Further examples

The examples given above can only suggest how arithmetic functions can be used in a simple way to produce a variety of numbers. The reader is urged to consider the examples above with a J interpreter to hand, to try the examples out, to check them, and to try to find expressions that are better or in some way more interesting than those given here. When generating these numbers begins to pall, the reader perhaps should go on to consider how to generate the three digit numbers using the same rules. This could start +/}:1 99 8 then +/1 99,\*8.

Alternatively, expressions might be sought for other years. Some years will present special challenges. The following table gives a start for the year 2000, in which a choice is made between 0=0, 0!0 and 0^0 to give a 1 largely on aesthetic grounds.

In the following table it has been assumed that

x =: 2 0 0 0

has been issued.

0	<./x	*/x	10	*/*:>:~.x	20%#0 0
1	./x	2 0 0 i.0	11	>./^x	#.2 0 0>.-.0
2	+/x	-/x	12	+/*:>:x	+/,>:=x
3	>:{.x	*/>:x	13	>./o.>:x	20-<.^+/>:0 0
4	#x	+/*:x	14	#.2+0 0 0	20-<.o.+/>:0 0
5	2+#0 0 0	2++/>:0 0 0	15	+/<.o.>:}:x	#.=~x
6	+/>:x	+/2+0 0 0	16	#.x	^/2>.0 0 0
7	2#.-.0 0 0	+/*:2,>:0 0 0	17	<.-/^>:~.x	>.2*+/^>:0 0 0
8	#.,#::~~.x	-/*:>>x	18	20-#0 0	+/<.o.2+0 0 0
9	*/*:>:x	+/2+>:0 0 0	19	20-=/0 0	>./o.2+0 0 0

Another amusing possibility, though ultimately monotonous because expressions are restricted to monadic functions, is to try to develop all the numbers from four zeroes in at least one list and at most one scalar.

In the following table it has been assumed that

z =: 0 0 0 0

has been issued.

0	~.z	0{0 0 0	10	+/!/:z	>./o.0=0 0 0
1	=/z	0 e.0 0 0	11	#./:z	>./^-.z
2	--//:z	+/0 0=0 0	12	+/*:/:z	*/!//:z
3	{:/:z	+/0=0 0 0	13	+/>+:#}:/:z	>./o.-.z
4	#z	+/<.^0 0=0 0	14	+/*:/:z	#.>:0=0 0 0
5	-.-/+:/:z	<.//^0 0=0 0	15	#.-.z	+/>.!^0=0 0 0
6	+//:z	+/<.o.0 0=0 0	16	+/<.!^-..z	>.^/^0 0=0 0
7	#.0=0 0 0	-.-/*:/:z	17	+/{.}.o./:z	<.*/>:o.0 0=0 0
8	>.-.-/o./:z	+/*:-.z	18	+*:{./:z	+/<.o./:z
9	*:{:/:z	*:+/0=0 0 0	19	>./o./:z	<:+/ ,~!/:z

## Postscript

This series of instructional essays were the course material used by the author over several years for teaching tacit use of J, as described in “Tacit J and I”[1]).

The particular nature of the essay above was to illustrate the functions explained in the preceding essay “Structural Ingredients”[2] and to explain the students’ assigned task for the following week.

This weekly task, which counted substantially towards a student’s overall mark, was to do for their student identification numbers what the text of this essay shows for the year 1998. They were expected to provide 200 expressions, and

their mark depended on the richness and brevity of the code in their work, which was submitted as a J file and analysed by J code.

### References

1. Holmes, N., "Tacit J and I", Vector, Vol.23, No.3, pp.52-56  
<http://vector.org.uk/?vol=23&no=3&art=holmes>
2. Holmes, N., "Structural Ingredients", Vector, Vol.24, Nos.2&3, pp.114-121  
<http://vector.org.uk/?vol=24&no=2&art=holmes>

# Simulating the Enigma

by Keith Smillie

*smillie@cs.ualberta.ca*

The Enigma was the electromechanical cipher machine used by all branches of the German armed forces during the Second World War. Enigma messages which were intercepted by the British and then deciphered at the Government Code and Cypher School at Bletchley Park provided intelligence that has been estimated to have shortened the war by one or two years. The present paper, which begins with some introductory remarks on cryptography, describes the Enigma in general terms and then gives a more detailed discussion of a somewhat simplified version and its implementation in J.

## Cryptography

Cryptography is the science and art of both concealing the meaning of a message by enciphering it according to some given algorithm and also deciphering an enciphered message to recover its original meaning. The two main ciphering methods are the *substitution cipher* in which each letter of a message is replaced by another character but retains its position in the message and the *transposition cipher* in which each letter remains unchanged but changes its position. For example the phrase The University of Alberta may be enciphered either by substitution to UIFVOJWFSTJUZPGBMCFSUB in which each letter is replaced by the letter one place ahead of it in the alphabet or by transposition to NHTEUEISVRTOFYILRABEEAHTT in which each group of five letters is randomly permuted. The original message is referred to as the *plaintext* and the enciphered message as the *ciphertext*. In the following we shall consider only substitution ciphers.

One of the earliest examples of a substitution cipher is attributed to Julius Caesar and is therefore known as the Caesar cipher. In this cipher each letter in a message is replaced by the letter which occurs three places ahead of it in the alphabet. Therefore the cipher may be summarized in the following table:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

The plaintext phrase The University of Alberta of the previous paragraph when enciphered in this manner becomes WKHXQLYHUVLWBRIDOEHUWD. The shift

need not be limited to 3 places, and may be any value between 1 and 26 where a shift of 26 leaves the message unchanged. The example of the substitution cipher in the previous paragraph is a Caesar cipher with a shift of 1.

Caesar ciphers are relatively easy to break since there are only 26 of them and each may be tried in turn on a portion of the enciphered message until the correct one is found and then applied to the entire message. If there is a sufficiently long message a frequency analysis of the text may assist in the decipherment making use of the relative frequencies of the letters in English text with *e* being the most common letter, followed by *t*, then *a*, and so on. (Also an analysis of the frequency of the  $26 \times 26$  pairs of letters can be very helpful since, for example, certain pairs such as *qq* and *zz* never occur and other pairs occur usually in one order but not the other.) A considerably more complicated substitution cipher is obtained by using a random permutation of the cipher alphabet so that, for example, each letter of the plaintext in the first row would be enciphered by the letter below it in the second row in the following table:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
I	O	H	C	F	P	T	V	M	G	D	A	Z	J	Y	U	W	K	X	R	L	N	Q	S	B	E

Even though the total number of such ciphers is  $26!$  which is equal to

403,291,461,126,605,635,584,000,000

such ciphers can still be broken by frequency analysis.

The Caesar ciphers of the previous paragraph are known as *monoalphabetic* substitution ciphers since only one cipher alphabet is used in a given message. However it is possible to define *polyalphabetic* substitution ciphers in which more than one cipher alphabet is used in a message according to a rule called the *key* which indicates which cipher alphabet is applied to each letter. The best known of these polyalphabetic ciphers is the Vigenère cipher named after the 16th century French diplomat Blaise de Vigenère who further developed and promoted a method of encipherment which had been suggested during the previous century by the Italian architect Leon Alberti. In the Vigenère cipher there are 26 cipher alphabets corresponding to Caesar shift alphabets with shifts of 1, 2, ... 26 letters. These may be conveniently arranged as shown in part in the table labelled "Vigenère Square" below.

As an example of the use of the Vigenère cipher let us again encipher the phrase The University of Alberta using as a key the word VECTOR. Write the plaintext with the keyword above it and repeated sufficiently often so that each letter of the plaintext has an associated keyword letter:

V E C T O R V E C T O R V E C T O R V E C T  
 t h e u n i v e r s i t y o f a l b e r t a

To encipher the first letter *t* of the plaintext find the letter *v* of the keyword appearing above it and then select from the column headed *t* the letter *o* which appears below it in the row labelled with a *v* on the left. The second letter *h* of the plaintext is enciphered with *L* which appears below it in the row which begins with *E*. Similarly the third letter *e* of the plaintext is enciphered with *G* which appears below it in the row which begins with *C*. Continuing in this manner – if we had the complete Vigenère square – we may determine that the ciphertext is OLGNBZQITLWKTSHTZSZVVT

*Vigenère square*

	abcdefghijklmnopqrstuvwxyz
A	ABCDEFGHIJKLMNOPQRSTUVWXYZ
B	BCDEFGHIJKLMNOPQRSTUVWXYZA
C	CDEFGHIJKLMNOPQRSTUVWXYZAB
D	DEFGHIJKLMNOPQRSTUVWXYZABC
E	EFGHIJKLMNOPQRSTUVWXYZABCD
F	FGHIJKLMNOPQRSTUVWXYZABCDE
G	GHIJKLMNOPQRSTUVWXYZABCDEF
H	HJKLMNOPQRSTUVWXYZABCDEFG
...	
T	TUVWXYZABCDEFGHIJKLMNQPQRS
U	UVWXYZABCDEFGHIJKLMNQPQRST
V	VWXYZABCDEFGHIJKLMNQPQRSTU
W	WXYZABCDEFGHIJKLMNQPQRSTUV
X	XYZABCDEFGHIJKLMNQPQRSTUVW
Y	YZABCDEFGHIJKLMNQPQRSTUVWX
Z	ZABCDEFGHIJKLMNQPQRSTUVWXY

The Vigenère cipher is equivalent to enciphering text with a several Caesar ciphers with different shifts applied in an orderly manner by use of a given keyword which may be selected from an almost infinite number of keywords. For many years the Vigenère cipher was considered unbreakable, and indeed was known as *le chiffre indéchiffrable*. However, a method of breaking it was found in the middle of the 19th century independently by Charles Babbage, honoured in the history of computing as the ‘father of computing’ for his Difference and Analytical Engines, and by Friedrich Wilhelm Kasiski, a retired Prussian army officer. At the end of the First World War an American Army officer proposed a method of making the Vigenère cipher unbreakable by having for each message a random key as long as the message. However, these ‘onetime pad’ ciphers, as

they were called, never became popular due to the problems of generating and distributing in a secure manner the random keys.

The first attempts to mechanize enciphering and deciphering were made in the fifteenth century by Leon Alberti who was mentioned above as the originator of the Vigenère cipher. He had two copper disks, one slightly larger than the other and mounted concentrically, each with the alphabet inscribed about its circumference. By rotating the disks so that, for example, A on the outer disc was opposite D on the inner disk, the alignment of the letters would facilitate the enciphering or deciphering of a message with a Caesar shift of 3. These cipher disks remained in use as late as the American Civil War, and indeed well into the 20th century as a children's toy. The next development in the mechanization of cryptography is the Enigma which we will discuss in the next section.

## The Enigma

The Enigma was designed and built by the German engineer Arthur Scherbius and was intended as a simple method of mechanically enciphering and deciphering messages with a very large number of polyalphabetic substitution ciphers. He took out his first patent in 1918 for a machine about the size and weight of a typewriter which cost approximately \$30,000 at today's prices. Although it was promoted widely – for example, it was shown at the 1923 Congress of the International Postal Union in Bern, Switzerland – it initially attracted little interest because of the cost and the belief that the degree of security it offered was unnecessary. However in the mid-1920s the German military became interested and Scherbius started mass producing Enigmas and eventually supplied over 30,000 machines to the German armed services.

The Enigma consisted of three parts: a typewriter keyboard for input, a 'scrambler' mechanism for applying a continually changing polyalphabetic cipher to each letter of the message, and for output a set of 26 lamps marked with the letters of the alphabet indicating the encipherment of each letter as it was input. The enciphered message was written down on paper and then transmitted either by telegraph or wireless to the intended recipient who would then use his Enigma which would be configured in the same way as was the sender's Enigma to decipher the message.

The main part of the scrambler was a set of three rotating discs or *rotors* about 4 inches in diameter and mounted in the machine on a removable axle. Each rotor had a circle of 26 electrical contacts near the edge of each face with the contacts on one face being connected to those on the other face in a random order.



Different random connections were used for each of the rotors. Depressing any key on the input typewriter would send an electrical current first to one of 26 contacts on a fixed input disc to the right of the three rotors. (The keyboard was connected to the fixed disc in simple alphabetical order with A on the keyboard connected to the first position on the disc, B to the second position, and so on.) The current would then pass to a contact on the right face of the right-hand rotor and then to the randomly connected contact on the left face of the rotor, through contacts on the right and left faces of the middle rotor, and then through the right and left faces of the left-hand rotor. From the left-hand face of the left rotor the current would pass through a fixed 'reflector' disc with contacts on its right face only which would effect a fixed interchange of pairs of letters. The current would then flow back through the three rotors in a left-to-right order, again through the fixed disc on the right, and finally to one of the 26 lamps which would display the encipherment of the letter which had been input.

The encipherment of each letter would involve a total of seven distinct polyalphabetic substitutions. However the Enigma was designed to change this encipherment from letter to letter by having the right-hand rotor advance to the next of its 26 possible positions at the input of each letter and before the electrical contact was made to the fixed input disc. When the input of 26 letters of a message had caused the right-hand rotor to make a complete revolution, a carry mechanism would cause the middle rotor to advance one position.. After another complete revolution of the right-hand rotor the middle rotor would be advanced another position, and so on until it would have made a complete revolution at which time the left rotor would be advanced by one position. Thus the three rotors may be considered to function as a base-26 odometer. The introduction of the reflector had two important consequences: A letter could never be enciphered into itself, and an enciphered message could be deciphered by entering it into an Enigma with the same initial settings as the machine which enciphered the message.

Each of the rotors had a ring attached to the outside and labelled around the circumference, usually with the letters A to Z, which could be turned to any one of 26 positions. Furthermore each ring had a small carry notch which would engage the rotor to the left and make it advance one position. Although the ring setting of a rotor did not affect the total number of scramblings, it did affect the letter-by-letter substitutions and the point in the revolution of a rotor when the rotor to its left was advanced.

In addition to the rotors a further scrambling was introduced by a stationary plugboard which through a set of cables allowed pairs of letters to be

interchanged. Usually there were 10 sets of cables which thus allowed the swapping of 10 pairs of letters. The plugboard was connected both to the keyboard and to the output lamps so that the pairwise swapping of the letters took place both before it entered the fixed input disc and after it had been scrambled by passing through the rotors, the reflector disc and through the rotors a second time.

The Enigma initially had only three rotors which of course could be inserted in any one of six possible orders. In the late 1930s the Enigma versions used by the German Army and Air Force were issued with five rotors, labelled I, II, III, IV and V, three of which were selected for a given Enigma setting. The Naval version had three additional rotors labelled VI, VII and VIII, and some versions could accommodate four of these eight rotors.

It is of interest to calculate the total number of possible substitution ciphers made possible by the scrambling mechanism. If we assume that there are five rotors, then the three rotors may be selected and installed in  $6 \times 5 \times 4$  or 60 possible ways. For each of these rotor orders the total number of substitutions is  $26 \times 26 \times 26$  or 17,576. To calculate the number of substitutions provided by the plugboard we note that the number of ways of choosing  $m$  pairs out of  $n$  objects is  $n!/((n-2m)!m!2^m)$ . If we assume there were 10 cables that could be connected, then  $m=10$  and since  $n=26$ , this expression becomes  $26!/(6!10!2^{10})$ , which may be calculated in J in extended precision as

$$x: (!26) \% (!6) * (!10) * 2^10$$

and has the value 150,738,274,937,250. Therefore the total number of substitutions available with the Enigma is given by

$$60 * 17576 * x: (!26) \% (!6) * (!10) * 2^10$$

which is equal to 158,962,555,217,826,360,000, or approximately 159 million million million. Obviously some form of frequency analysis together with any knowledge of the possible content of the message or the writing style of the sender is the only possible method of breaking such a cipher.

## Using the Enigma

When using the Enigma, the Germans drew up monthly setting sheets which specified for each 24-hour period all of the Enigma settings except the initial rotor positions, i.e., the selection of three rotors out of the five, their ring settings and order in the machine, and the 10 pairs of plugboard connections. A person

sending a message would first set up his machine according to the settings for that day and then proceed as follows:

1. Select a three-letter *indicator* giving the rotor positions to be used to encipher the initial rotor positions or *message key* to be used when enciphering the message.
2. Turn the rotors to the indicator position and type the message key twice writing down the enciphered letters shown on the lamps.
3. Turn the rotors to the message key letters and type the message writing down the encipherment of each letter.
4. Give the enciphered message together with the indicator and the twice-enciphered message key to the radio operator for transmission.

The recipient of the message would proceed as follows:

1. Set up his Enigma according to the daily settings.
2. Turn the rotors to the indicator position which he would have received unenciphered.
3. Type in the next six letters of the message which would give the repeated message key.
4. Turn the rotors to the message key and type and decipher the message.

### **An Enigma simulator**

In this section we shall give a simulation in J of the simplified Enigma machine given in Simon Singh's *The Code Book* with the following properties: The alphabet is limited to the letters A, B, C, D, E and F; the plugboard allows the interchange of only one pair of letters; the rotatable ring with the carry notch determining when the rotor adjacent to it would rotate one position is omitted; and the rotor or rotors advance one position after (not before) a letter has been entered at the keyboard and has been enciphered. However, whereas Singh's example has the current passing through the rotors first from left to right and then through the reflector and back through the rotors from right to left, our simulation will adhere to the conventional order given in most accounts of the Enigma. The substitutions may be defined as follows:

Rotor 1:     a→B; b→A; c→D; d→F; e→E; f→C  
 Rotor 2:     a→C; b→A; c→D; d→B; e→F; f→E  
 Rotor 3:     a→F; b→C; c→E; d→D; e→B; f→A  
 Reflector:   a→F; b→C; c→B; d→E; e→D; f→A  
 Plugboard:   a→B; b→A; c→C; d→D; e→E; f→F

We may note in passing that the number of substitution ciphers for this simplified Enigma is the product of the 3! or 6 permutations of the rotors, the 6×6×6 or 216 relative positions of the rotors for each of these permutations, and the 15 substitutions for the plugboard, or in total 6×216×15 or 19,440 substitutions. If we follow and extend Singh's discussion of the substitutions given by the first rotor as it rotates for each keystroke, we may draw up the following table giving the cipher alphabet for each position of the rotor:

	0	1	2	3	4	5	6	7	...
a	B	D	A	C	B	F	B	D	...
b	A	C	E	B	D	C	A	C	...
c	D	B	D	F	C	E	D	B	...
d	F	E	C	E	A	D	F	E	...
e	E	A	F	D	F	B	E	A	...
f	C	F	B	A	E	A	C	F	...

This simplified Enigma may be defined in J by the following variables:

```

Alphabet      ALPHA=: 'ABCDEF'
Rotors        I=: 'BADFEC'; II=: 'CADBFE'; III=: 'FCEDBA'
Reflector     Reflector=: 'FCBEDA'
Plugboard     Plugboard=: 'BACDEF'
  
```

If we keep track of the typewriter keystrokes by the variable `Counter`, then the amount of rotation of each of the rotors is given by

```
'S2 S1 S0'=: 6 6 6#Counter
```

and, for example, if `Counter=: 20` then we have that `S0` is 2, `S1` is 3 and `S2` is 0 indicating that the right rotor has moved 2 positions, the middle rotor 3 positions, and the left rotor has remained stationary. The values of `S0`, `S1` and `S2`, which are the digits in the base-6 representation of the decimal value of `Counter`, may be used to determine the substitutions effected by the various positions of the rotors. For example, the third column in the above table giving the substitutions for rotor I is given by

```
((_2|.ALPHA) i. _2|.I) { ALPHA
```

which is equal to AEDCFB representing the substitutions

a→A; b→E; c→D; d→C; e→F; f→B

All of the above substitutions are those given by the current passing through the rotors in a right-to-left direction before passing through the reflector. The second set of substitutions with the current passing through the rotors from left to right may be found in a similar manner, and using the example for rotor I we have

((\_2|.I) i. \_2|.ALPHA) { ALPHA

which is equal to AFDCBE, which represents the substitutions

a→A; b→F; c→D; d→C; e→B; f→E

The Appendix gives a J program in Version 4.06b for a simulation of the simplified Enigma machine based on these considerations. The following is a simple example of its use:

```
InstallRotors 3 2 1
SetRotors 'CFD'
m=:enigma 'ABCDEFABVC'
m
BEAAAABC_F
SetRotors 'CFD'
enigma m
ABCDEFAB_C
```

## References

1. Battle of Wits. *The Complete Story of Codebreaking in World War II*. The Free Press, New York.
2. Hodges, Andrew, 1985. *Alan Turing: The Enigma of Intelligence*. Unwin Paperbacks, London.
3. Kahn, David, 2009. *Seizing the Enigma: The Race to Break the German U-Boat Codes 1939-1943*. Barnes & Noble, Inc., New York.
4. Sale, Tony, 2009. The Enigma Cipher Machine.  
<http://www.codesandciphers.org.uk/enigma/>
5. Singh, Simon, 2000. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, New York.

## Appendix. Enigma simulator

```

NB.    Enigma Simulator
NB.    Keith Smillie
NB.    Department of Computing Science
NB.    University of Alberta
NB.    Edmonton, Alberta T6G 2E8
NB.    January 2010
NB.
NB. This program gives a simulation of a simplified model of the
NB. Enigma cipher machine used by the German armed forces during
NB. World War II. The version is very similar to that given in "The
NB. Code Book" by Simon Singh (Anchor Books, New York, 2000).
NB. The allowable characters are given in the list ALPHA. Blanks
NB. may be used and are not enciphered, and all other characters
NB. are enciphered as underscores "_".
NB.
NB. The following dialogue gives a simple example of its use:
NB.    NB. Install rotors in specified order
NB.        InstallRotors 3 2 1
NB.    NB. Set rotors to given key
NB.        SetRotors 'BCA'
NB.    NB. Encipher message
NB.        x=: enigma 'ABCDEFAC'
NB.    NB. Display message
NB.        X
NB.    BCABBAE_E
NB.    NB. Reset rotors
NB.        SetRotors 'BCA'
NB.    NB. Decipher message
NB.        enigma x
NB.    ABCDEFA_C

ALPHA=: 'ABCDEF'
I=: 'BADFEC'
II=: 'CADBFE'
III=: 'FCEDBA'
Discs=: I;II;III
Reflector=: 'FCBEDA'
Plugboard=: 'BACDEF'

InstallRotors=: 3 : 0
Rotors=: (<:y.) { Discs
Counter=: 0
empty ''
)

SetRotors=: 3 : 0
empty
Counter=: (3$#ALPHA)#.|.ALPHA&i. y.
)

```

```

rotate1=: 3 : 0
:
((x.|.ALPHA) i. x.|.y.) { ALPHA
)

rotate2=: 3 : 0
:
((x.|.y.) i. x.|.ALPHA) { ALPHA
)

Encipher=: 3 : 0
:
(ALPHA i. y.) { x.
)

enigma=: 3 : 0
'Left Middle Right'=: Rotors
PlainText=: y.
CipherText=: i. 0
while. 0 < $PlainText do.
'S2 S1 S0'=: -(3$#ALPHA)#Counter
c=:
{. PlainText
if.
-. c e. ALPHA, ' ' do.
c=:
' '
elseif.
c e. ALPHA do.
c=: Plugboard Encipher c
c=: (S0 rotate1 Right) Encipher c
c=: (S1 rotate1 Middle) Encipher c
c=: (S2 rotate1 Left) Encipher c
c=: Reflector Encipher c
c=: (S2 rotate2 Left) Encipher c
c=: (S1 rotate2 Middle) Encipher c
c=: (S0 rotate2 Right) Encipher c
c=: Plugboard Encipher c
end.
CipherText=: CipherText, c
PlainText=: }. PlainText
Counter=: >Counter
end.
CipherText <
)

```

## Subscribing to Vector

Your *Vector* subscription includes membership of the British APL Association, which is open to anyone interested in APL or related languages. The membership year runs from 1 May to 30 April.

Name \_\_\_\_\_

Address \_\_\_\_\_

Postcode/Zip and country \_\_\_\_\_

Telephone number \_\_\_\_\_

Email address \_\_\_\_\_

UK private membership	£20	___
Overseas private membership	£22	___
+ airmail supplement outside Europe	£4	___
UK corporate membership	£100	___
Overseas corporate membership	£110	___
Non-voting UK member (student/OAP/unemployed)	£10	___

### Payment methods (*Sterling only*)

1. A Sterling cheque, payable to *British APL Association*, drawn on a UK bank.

2. By American Express, MasterCard or Visa:

I authorize you to debit my American Express/MasterCard/Visa account

Number: \_\_\_\_\_ Expires: \_\_\_\_/\_\_\_\_

for the membership category indicated above.

Signature: \_\_\_\_\_ Date: \_\_\_\_\_

#### Privacy Policy

*Your personal information will be stored on computer but not disclosed to third parties. Card data will not be stored on computer.*

3. By electronic transfer.

Our account details are: Barclay's Bank; Cambridge, Chesterton Branch; Sort code: 20-17-35; Account number: 63955591; Account name: British APL Association; SWIFTBIC: BARCGB22; IBAN: GB86 BARC 2017 3563 9555 91.

4. Use PayPal to credit account treasurer@vector.org.uk (no account needed – ask for details).

If you pay by cheque or credit card, please send the completed form to:

BAA, c/o Nicholas Small, 12 Cambridge Road, Waterbeach, Cambridge CB25 9NJ