Contents

Quick reference diary 2

Editorial 3

News

APL2000 5

Dyalog 7

NARS2000 10

TryAPL 12

Taylor reports from Cambridge 13

BAA Chairman's report (pre AGM) 22

General

Thomson and Camacho about infinity 25

Hui ponders over arrays 29

Smillie surveys computing 34

APL

Taylor gets multi-dimensional 43

Last co-operates 51

Last writes functions 67

Jaeger shares code and grows an APLTree 79

Jaeger documents automatically 87

J

Thomson rules the edge 103

Thomson creates combination lists 105

Reiter interpolates bicubicly 112

Others

Ulmann explains 5 119

Quick reference diary

16+17 April	Düsseldorf, Germany	APL Germany & GSE meeting
22-24 April	Jersey City, USA	APL2000 User Conference
27 April	Cheshunt / UK	BAA AGM
27-29 April	Cheshunt / UK	APL Moot
23-25 May	Ireland	Kx International User Conference
23-24 July	Toronto, Canada	Jsoftware Conference
14-18 October	Elsinore, Denmark	Dyalog Conference

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.

EDITORIAL

This is probably the first Editorial ever that was not written by the Editor; my friend Stephen Taylor has stepped down as Vector's Editor with immediate effect. He tried to get away from the job at the last AGM in 2011 but we didn't let him. Bad move. He had to force us.

When Stephen asked for help back in 2010 because he could not stem the work-load any more a number of people including me offered help. Some learned how to mark up an article written by a contributor. I learned from him how to collect the articles already published on the Web and convert them into a PDF that can go to the print shop. I think it's fair to say that all of us were surprised by the amount of work it takes simply to set an article for the web, in particular when it is lengthy or contains plenty of code in the text or plenty of images or math formulae, let alone an article that combines all these obstacles.

I remember vividly the lengthy email I got from him after I'd produced the first print issue of Vector back in 2010. Although I had spent countless hours on this issue I'd missed many details and I'd made plenty of mistakes. We then met on a weekend and went through this email, fixing one problem after the other; it took us a whole Sunday. I had no idea how much work it takes in the background to get this done.

And that's just the technical bit. Of course the editor also has to negotiate with the authors, and some of them have very strong opinions on how their stuff should look. Others must be encouraged to come forward. Articles written by authors whose first language is not English add to the workload. Occasionally an article was practically rewritten by Stephen.

Finally there's the website which was completely redesigned by Stephen; a big project in its own rights.

We will miss Stephen's passion, his skills and his eye for details. He has done an amazing job for many years. At the moment I cannot imagine how we shall manage without him but somehow we have to.

Kai Jaeger

N E W S

Industry news

APL2000

We are pleased to announce the release of APL+WIN Version 11.1

Enhancements in this release include:

Virtual APLKeyboard

There is now a Virtual APL Keyboard GUI control in the session manager. The keyboard can be displayed through the "View" menu or by pressing Ctrl+B.

APL Idioms Manager

There is now an APL Idioms Manager in the session manager. It can be accessed through the "View" menu or by pressing Ctrl+1.

APL+Win Highlights

- V10: System speed and memory capacity dramatically increased
- V10: Support for larger dimension arrays in the workspace
- V10: Separate debug and release execution modes
- V10: Capture execution history with minimal performance effect
- V10: New crash recovery mechanism
- V10: New/enhanced control structures
- V10: Enhanced exception handling
- V10: Enhanced debugger
- V10: Enhanced Unicode support
- V10: New system commands
- V10: APL Supervisor for multi-thread/multi-processor access in APL+Win

- V11: Idioms manager to maintain and deploy idioms into application systems
- V11: Fully-interactive, scalable, glyph position correct, virtual APL keyboard

A .Net string-key dictionary is now available to current APL+Win Subscription Licensees. Dictionary data type with string keys and values of any data type is implemented using the APL+Win interface to Microsoft .Net.

VisualAPL Enhancements

- The VisualAPL 'Lightweight Array Engine' is now based upon .Net Framework 3.5 so that end users can implement custom .Net extension methods to any LAE operator or method
- The VisualAPL data representation (xml-serialization of VisualAPL objects) and the wrapl system function now use UTF-8 encoding
- An automated converter from APL+Win component files to enhanced VisualAPL component files has been implemented
- Performance enhancements for the grade up/down functions implemented
- VisualAPL documentation now uses the pdf format with extensive bookmarks
- VisualAPL demonstration/evaluation versions are now available for production and Express versions of Visual Studio

2012 APL2000 User Conference

Join us for the 2012 APL2000 User Conference to be held Sunday – Tuesday, April 22-24 at the Hyatt Regency Jersey City on the Hudson. This is a beautiful location, conveniently located near New York City and easily accessible to Newark Liberty Airport. APL2000 is offering a comprehensive training class on Sunday, taught by Joe Blaze, giving attendees the opportunity to learn how to create a Windows Presentation Foundation (WPF) Graphical User Interface (GUI) for their APL+Win applications. The agenda for Monday and Tuesday includes informative presentations and stimulating discussions on a wide range of topics. Attend the conference and enjoy the camaraderie of spending a few days with fellow APL enthusiasts. For more information and to register, visit our website, www.apl2000.com.

Dyalog Ltd

TryAPL.org

Since 2006, Dyalog has made a free educational version of APL available to anyone enrolled in a full-time education program. The number of downloads has gradually increased from a trickle to an average of one per day, with a total of twelve hundred licences downloaded to date (plus a few hundred non-commercial licences). However, we know that many people are reluctant to enter personal information in order to be able to test our product. In order to make APL more easily accessible to anyone who is interested, we have decided to make a version of Dyalog APL available as a "no-questions-asked" download for anyone who would like to play with APL. The unregistered version will show a pop up every 20-30 minutes to let you know that it is unregistered - and that you need to sign up for a free educational or a low cost (£50/$75) non-commercial licence to have access to a version without the pop-up.

Initially the unregistered version will only be available as 32-bit Unicode Edition of Dyalog APL for Windows; other platforms are planned to be added during 2012. The unregistered version will be available from http://www.tryapl.org/ and from the download zone of www.dyalog.com; the only field to be filled in is a checkbox acknowledging the terms and conditions of Dyalog's non-commercial licence.

From January 1st 2012, the tryapl.org site will also host a simple interactive "APL Timesharing System", which will allow anyone to enter APL expressions interactively while learning the language. This will complement Gary Bergquist's on-line APL Tutorial, which is already available at http://tutorial.dyalog.com.

MiServer

Observant users of the APL Wiki will have noticed a project which was started under the name "MildServer", which has been an experimental framework developed with the goal of "making it possible for anyone who is able to write an APL expression to expose it as a web page".

Thanks to the APL Tools Group at Dyalog, which is headed by Brian Becker and includes Dan Baronet, Nicolas Delcros, Morten Kromberg and intern Brian McCormick, this project is now rapidly gaining momentum. In addition to acquiring a catchier name, the "MiServer" now contains a number of tools for creating very nice-looking interfaces using the "JQuery UI" library, a handful of sample web applications and an extensive User Guide.

The MiServer is designed to be an "open source" project; all of the source code is stored in Unicode text files, and a modular architecture is intended to allow straightforward participation by users who are keen to help add support for more JQuery widgets, database interface tools, security mechanisms and other more sophisticated versions of the simple mechanisms which are provided as samples.

We hope that with the trial APL system and MiServer, students and other developers will find it much easier to take a first look at APL – and that the availability of a simple but effective web server framework will make APL an attractive platform for quickly delivering technical applications on the internet.

New versions of MiServer will be released constantly in the future, and will also be available on the APL Wiki and the dyalog.com/library page.

Dyalog'12 in Elsinore

Reserve October 14th-18th 2012 in your calendar – these are the dates for the next Dyalog conference, which will be held at LO-Skolen in Elsinore, Denmark! The conference is experiencing therefore decided to add one more day to the conference, bringing it to five days including training sessions before and after the main conference. We have also decided to add more training sessions during the main conference days, focusing on introductions to features added to Dyalog APL in the current millennium (but not necessarily in the last one or two releases). If you plan to attend, please send suggestions for course topics to conference@dyalog.com. See http://www.konventum.dk/arkitekturen for more images of the venue.

The 2011 Dyalog conference was held on October 2nd-5th, at the John Hancock Conference Center in Boston, Massachusetts. As in with previous Dyalog conferences, we recorded about half of the presentations and they will start to appear on the http://video.dyalog.com website in December, alongside the existing recordings from Dyalog '08, '09 and APL2010 in Berlin.

Dyalog Programming Contest

One of the highlights of the Dyalog conference in Boston was the presentation of the grand prize to the winner of this year's Dyalog Programming Contest (as it was in '09 and '10). This year, the puzzles had been set by last year's winner, Ryan Tarpine from Brown University – and were well received by the contestants – 130 of whom downloaded Educational licenses. This year's winner, Joel Hough from the University of Utah, came third last year and decided to try again – and this time he made off with the grand prize of USD 2,500 and a round trip with all expenses paid from Salt Lake City to Boston.

Despite his age, Joel is already a veteran programmer, with (literally) dozens of programming languages under his belt – and he ranks learning APL as one of the more enjoyable experiences in his career.

He will be speaking at the QCON conference in San Francisco on November 18th, together with Morten Kromberg, on "Why APL is Still Cool". More details at http://qconsf.com/sf2011/presentation/Why+APL+is+Still+Cool. For more about Joel and the other prize winners, see http://www.dyalog.com/news.htm.

We have prepared the 2012 Programming Contest and unveiled the next set of problems. As usual, anyone who introduces one of the winners will win the same amount of prize money as the winners themselves, so start thinking about which of your student friends you will put to work for you.

Introductory APL Courses

As this is being written, Bernard Legrand is conducting another of his highly rated introductory APL courses, based on his book "Mastering Dyalog APL" (which is available from Amazon or as a free PDF download from http://www.dyalog.com/intro). The frequency with which these courses are being conducted is slowly increasing, and we expect to hold another course in 2012. If you or any of your colleagues could benefit from APL training, write to sales@dyalog.com for details.

Version 13.1

The target release date for Version 13.1 of Dyalog APL is 13 April 2012. Tune in again in the next issue to read more about that, the new "Dyalog File Server", and a number of additional tools and products that Dyalog will be making available in 2012.

NARS 2000

The Next Major Release

Bob Smith, Sudley Place Software, 24 March 2012

Begun in September 2006, NARS2000 remains the only full-featured Extended APL Standard compliant free open source APL interpreter still in active development. As an experimental APL interpreter, it contains a host of old, new, and borrowed ideas including these features from the past year:

- Language bar: All APL chars in easy view – no more fumbling with Ctrl- or Alt-key combinations.
- User-defined keyboard layouts: Change the keyboard to exactly where you expect the APL and non-APL characters to be.
- Multiset operator: `f⍦ R` and `L f⍦ R` applies various primitive and user defined functions to vectors treating them as Multisets – sets with repeated elements.
- Root primitive: `√R` and `N√R` – square root and Nth root.
- Number factoring and number-theoretic fns: `∏R` and `L∏R` – factors numbers, tests for prime, finds Nth prime, number of `primes≤N`, etc.
- Support for Rational and VFP numbers:
  `123x`, `2r3`, and `2v3` – Find the low-order ten digits of the sum of N to the Nth for N from
  1 to 1000 – `¯10↑⍎+/*⍨⍳1000x ⍝ 9110846700`
- Sequence function:
  `L..R – ¯2..8 ⍝ ¯2 ¯1 0 1 2 3 4 5 6 7 8`
- Native File functions: `⎕NCREATE`, `⎕NTIE`, etc.

NARS2000 runs on several platforms: 32- or 64-bit Windows XP and later, along with any OS that can run Wine (a translation – not emulation – layer) available on almost all Linux systems.

Download either the 32- or 64-bit version from the website mentioned below. It's free – no forms to fill in, no permissions needed, and it comes with its own APL font or you can use an existing APL Unicode font.

If you like the idea of having an open source interpreter, please look for ways to help:

- Promote it: tell your friends and demo it to them.
- Test it: migrate your favourite code via .atf files or Copy & Paste Unicode characters; write test cases for various primitive functions.
- Document it: add to the Wiki.
- Develop it: write APL or C code to add new features or improve old ones – some new and old primitives have been implemented in APL using the language as a rapid prototyping tool and can be recoded in C if and when performance is a concern.

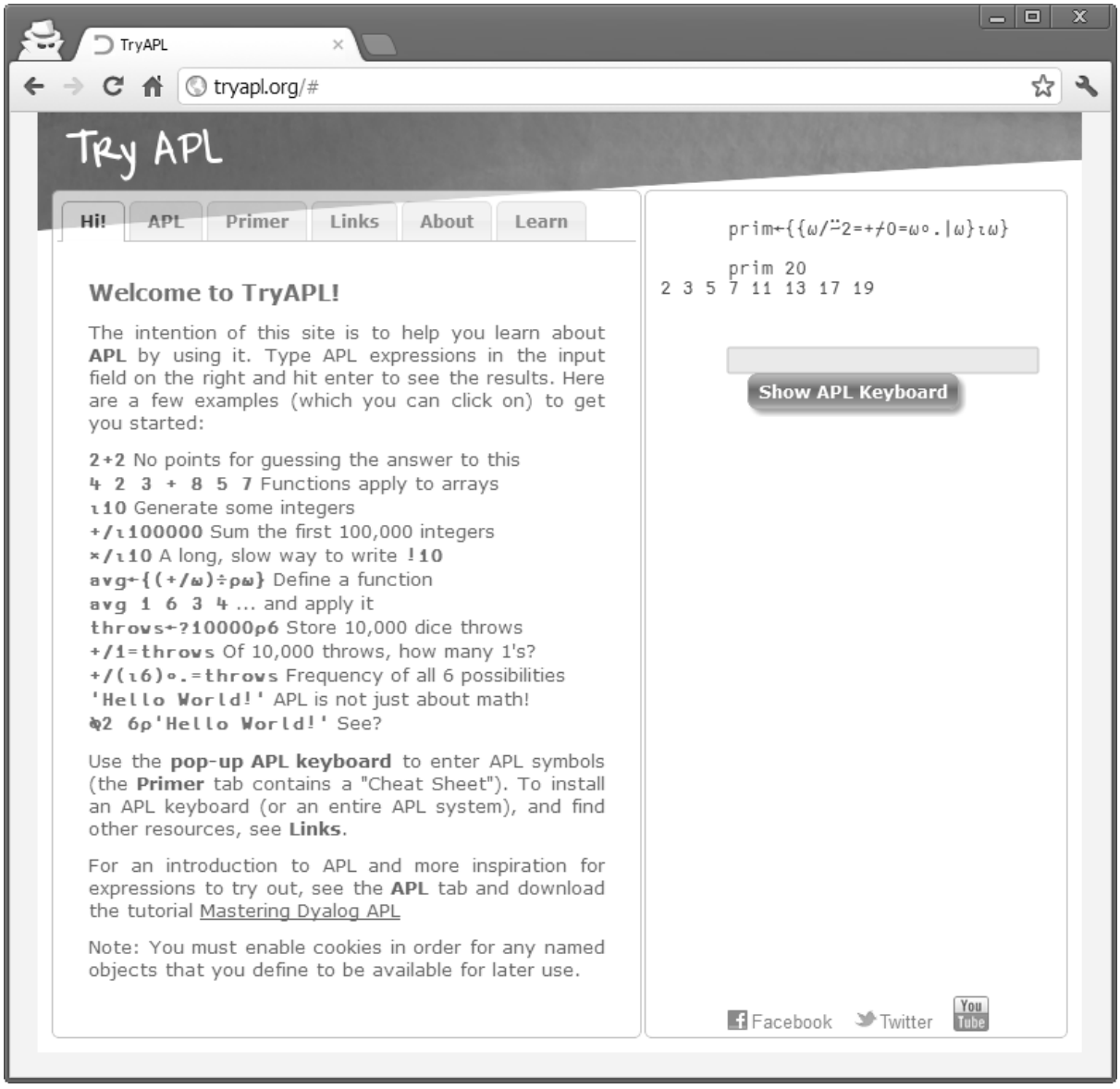
Documentation: http://wiki.nars2000.org
Downloads: http://www.nars2000.org/download/
Home Page: http://www.nars2000.org

http://tryapl.org

During his speech at the Dyalog 2012 Conference in Boston Joel Hough, the winner of the 2012 Dyalog APL Programming Contest, explained how he normally approaches a new programming language. He mentioned a couple of things; one was: “Then I try the language online.”

There is a simple reason why you can try Perl, Python or Ruby online: interpreted languages are perfect for this. However, not with APL.

Until 2012 - now there is http://tryapl.org



The right panel is a sort of session manager. One can try APL expressions and even define functions. Great!

MEETING

Iverson College, Cambridge, Aug 2011

by Stephen Taylor (sjt@5jt.com)

A score of APL programmers gathered in Cambridge this summer to spend a week living and working together. It proved popular and drew attention from luminaries at Microsoft Research.

The FlipDB Team: Kai Jaeger, Phil Last & Paul Mansour

We were, almost, eighteen. Two of us had been unable at the last minute to escape our offices. The other sixteen made it to Trinity Hall in Cambridge for a 'working week' – some kind of hybrid of a computer conference and a monastic retreat. We dubbed it *Iverson College*[1].

Actually, we were just working away from home. Not a big deal with a laptop and the Net.

Some of us remembered working in offices decades ago, for companies such as the legendary I.P. Sharp Associates[2]. Even when not on the same project, there

was so much then to learn from each other, over lunch, over coffee, over drinks, or even just in a heart-felt cry from the keyboard: “How on earth do I…?”

Taking meals together

So we had reserved study bedrooms in ancient Trinity Hall[3] for the last week of August. The college gave us the Leslie Stephen Room as a workroom, some fast Net pipes, and a private dining room for our lunches and dinners. Breakfasts we took in the cavernous dining hall, beneath the disapproving expressions of generations of former college masters.

A q portal

Who were we? Kx Systems assembled a crew from around the world: Arthur Whitney from California, Simon Garland and Charlie Skelton from Switzerland, Chris Burke from China and Arthur's young collaborators from St Petersburg, Oleg Finkelshteyn and Pierre Kovalev. Kx customer Merrill Lynch in London took another four places, but at the last minute only James Garrett and Phil Beasley-Harling were able to leave London.

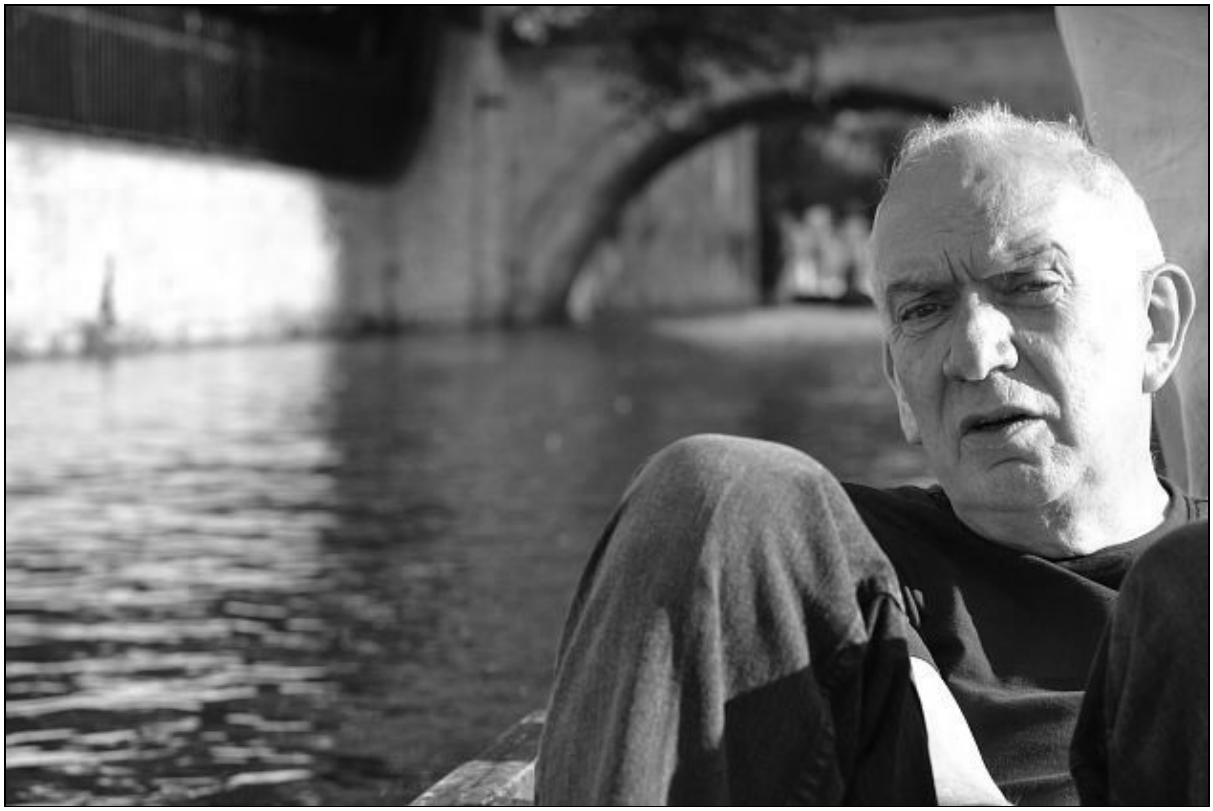
Morten Kromberg brought two implementors from Dyalog: John Scholes and Jay Foad. (John introduced himself as a C programmer who had only ever written one program, which he thought might be nearly finished.) Dyalog customer Joakim Hårsman joined us from Stockholm.

Paul Mansour of The Carlisle Group, host of invited conferences in Tuscany and Cephalonia, came from Pennsylvania and met his England-based collaborators Phil Last and Kai Jaeger. Dyalog customers to a man, all three of them.

And there was me. I booked the rooms.

Apart from meals, we had no programme. Sometimes one worked in one's room to avoid disturbance; sometimes in the common workroom for the chance of one. The workroom was usually silent but for key taps and a murmur of conversation. At other times a lively conversation would start and draw everyone in.

Scholes with pole



Scholes without pole

When we gathered on the Sunday afternoon, we possessed ourselves of the two college punts and took to the Cam to explore the Backs of the Colleges. Jay Foad has a degree from Cambridge. This apparently includes handling a loaded punt. Others, like Morten Kromberg, learned fast. A wet August was ending in a warm and largely dry week. Late afternoon sun angled down between the colleges, through the trees and spread buttery-gold light over the river and lawns.

Dinners were delicious, gusting to sumptuous, with generous portions. We had earmarked some bottles from the college cellars and made steady progress through them on our first few evenings, sitting and talking until late. Phil Last provided a range of good English ales in support.

We had no programme. Or rather: we were the programme. At our first meal I welcomed everyone and announced my job was now done. People looked a bit bewildered. What was the plan? I insisted: we had no schedule of talks or activities. Only meal times were fixed. If we wanted talks, we could arrange some as we went along, perhaps at tea time.

Andrew Kennedy

Andrew Kennedy[4], of the F# team from Microsoft Research in Cambridge, dropped by to see what we were doing. (Don Syme[5] was out of the country at the time, but sent his regrets and his delight that a flock of functional programmers was nesting in his old college.) Andrew gave us a talk on his work implementing units-of-measure in F#, then sat next to me while Arthur Whitney presented his work in progress on k – a research version of the kdb+ interpreter.

Arthur spent some time constructing and analysing versions of his famous expression that generates all solutions to a Sudoku puzzle, kicking off explorations that lasted the rest of the week, writing equivalent expressions in J and Dyalog.

Interesting expressions

The k interpreter occasionally gave wrong answers. It would have been easy to suppose we were looking at a graduate project, did we not remember how Kx makes millions of dollars selling its programming language to a world in which programming languages are largely free.

John Scholes ventured a question: “How do you do garbage collection?” — “No need to do garbage collection. I know where everything is.”

The k binary weighs in at about 50Kb. Someone asked about the interpreter source code. A frown flickered across the face of our visitor from Microsoft: what could be interesting about that? “The source is currently 264 lines of C,” said Arthur. I thought I heard a *sotto voce* “that’s not possible.” Arthur showed us how he had arranged his source code in five files so that he could edit any one of them without scrolling. “Hate scrolling,” he mumbled.

Simon Peyton-Jones

The following day brought us Haskell pioneer Simon Peyton-Jones[6], who graciously autographed a well-thumbed textbook proffered by the blushing John Scholes. “We had,” he said, “no idea what you guys have been doing. You are completely off our radar. You have to come to some of our conferences.”

In the event it all seemed to work astonishingly well. Our first two evenings were spent socialising and drinking. Then the focus sharpened. People were using the late afternoon to exercise, so we reserved the time after dinner for talks in the Lesley Stephen Room. Simon Garland bowed to popular pressure and gave the current version of his always hilarious Kx technical presentation, showing kdb+ woofing down gargantuan tic volumes from the financial markets, as always, illustrated from his vast collection of cartoons.

Days at the keyboard demand exercise. There was the Cam to run beside, a swimming pool at a nearby sports centre, and Cambridgeshire roads for biking. (I managed to stay with Morten for 20km before retiring hurt to celebrate my achievement.)

At Byron's Pool

At the Orchard Tea Gardens

We took one day as a play day, hiring punts and poling them upriver to Grantchester. Most punters were new to the business, but competitive. (Testosterone-fuelled punting: did you ever see anything so silly?) Two of the boats made it past Grantchester, turning back only on reaching the weir at Byron's Pool. Returning to Grantchester, we partied over tea and scones with the wasps at the Orchard Tea Gardens, attended by the shades of Russell and Wittgenstein.

The programmers were the programme. It was agreed the week was fruitful and inspiring, the format a success, and the venue a delight. (Nothing came of a scheme to scale a tower of neighbouring Caius & Gonville College and label its sun-dialled faces "London", "Tokyo" and "New York".) I have promised to mount another 'working week' next year.

Many thanks to Arthur Whitney of Kx Systems, Mark Sykes of Merrill Lynch and Morten Kromberg of Dyalog for encouragement and advice. Many thanks to the generous donors of the Wine Committee. Most of all, thanks to everyone who came: I had one of the most rewarding working weeks of my life.

References

1. 'Iverson College' website: sites.google.com/site/iversoncollege
2. http://en.wikipedia.org/wiki/I._P._Sharp_Associates
3. http://www.trinhall.cam.ac.uk/
4. http://research.microsoft.com/en-us/um/people/akenn/
5. http://research.microsoft.com/en-us/people/dsyme/
6. http://research.microsoft.com/en-us/people/simonpj/

BAA Chairman's Report (pre AGM)

by Paul Grosvenor

Time continues to pass at an ever increasing rate and another AGM is just round the corner. This time we look forward to seeing you all between 27th and 29th April at the Lee Valley YHA for both the AGM and a weekend of all things APL. I do hope that some of you will be able to make it (and Geek with the rest of us). Lively conversation, a barbie, a few libations and who knows even some programming will be the order of the day(s).

I wanted to get some feedback from the membership on a few issues and so have included this report before the AGM is held. Hopefully my questions / comments will make some sense.

1. As Stephen Taylor is no longer Editor of Vector we are looking to fill this post and continue to look for a suitable candidate. One of the suggestions made is for Optima Systems Ltd to put some resource in to assist the other members produce the journal. Whilst this is fine in some ways it would mean that the Chairman, Secretary and Editor roles of the committee would all come from the same organisation. My question therefore is;

   Would the members be happy for so many posts to be filled from a single organisation and would this compromise impartiality?

2. Our issue with the BCS regarding our funds continues but their stance has not changed which is what we suspected would be the case from the outset. Circa £14,000 remains under their control and has been absorbed into their organisation. There was a vote of no-confidence in the BCS management 2 years ago now and an EGM was called as a result. We submitted some information to that meeting on behalf of the BAA. Rather then me repeating all that was said and done please follow the following link where all the information can be found; **www.bcsegm.blogspot.com.**

Since the BCS has not changed its views regarding the membership position of the BAA (and indeed other groups within the BCS) and there seems to be no documentary evidence to suggest that our membership was in any way different we now face a very hard uphill climb to pursue our claims. My question to you all is;

Should we now formally accept that we will not get our funds back from the BCS and direct our efforts into more beneficial directions?

3. In recent years we have had an Outstanding Achievement Award which last year went to Stephen Taylor. This time I would like to do things slightly differently and ask for nominations from the wider community with a view to awarding the trophy next year. My question;

   Are we happy to open up the nominations to such a community which may include APL, J, K, Q etc?

4. Our membership year is tightly bound into the production of 4 issues of Vector and this can cause us problems when material or time is short. The question;

   Would the membership be happy to continue with their subscriptions if the number of issues of Vector was not necessarily 4 per year?

I look forward to hearing any views you may have regarding the above (or indeed any other points). If you would like to send an email with your thoughts to;

chairman@vector.org.uk

or

paul@optima-systems.co.uk

Thank you and see you on the 27th

G E N E R A L

LETTER

About infinity -1-

by Norman Thomson

Norman Thomson wrote this letter in response to Sylvia's article "What is it about infinity?" in Vector 25-1

Dear Sylv,

Following your article “What is it about infinity” in Vector vol. 25 No. 1, may I suggest that taking a slightly different viewpoint may help to clarify and simplify your thoughts concerning rearrangements of terms in the infinite series `ln2 = 1-1/2+1/3-1/4+?` to infinity. First the term absolute convergence means that the sum of the absolute values (i.e. moduli) of all the terms in a series converges to a finite value. The `ln2` series is not absolutely convergent since the sum of the reciprocals of the natural numbers is divergent. The issue of different sums arising from different arrangements of the terms is only a consideration for non-absolutely convergent series, in which case some of the terms must be positive and some negative. The sum for any finite number of terms will be the sum of say `p` positive terms and `n` negative terms. Rearrangement of terms within either of these groups will make no difference to the overall sum once `p` and `n` are chosen. The `ln2` series in standard ordering up to `y` terms is defined in J by

```
   ra=.(_1&^ * %@>:)@i.     NB. reciprocals alternating in sign
   ra 6
1 _0.5 0.333333 _0.25 0.2 _0.166667
```

and so two typical consecutive such sums in this somewhat slowly converging series are

```
   (+/ra 1000),+/ra 1001
0.692647 0.693646
```

The positive and negative terms can be separated by

```
   odds=.#~$&1 0@#
   evens=.#~$&0 1@#
```

Any possible rearrangement of terms of the series can be accomplished by taking `p` of its positive terms and `n` of its negative terms.

For the unrearranged series `p = n`:

```
   s0=.odds ar 10000        NB. positives
   s1=.evens ar 10000       NB. negatives
   (+/s0)+(+/s1)            NB. sum of first 10,000 terma
0.693097
```

If `p` and `n` are chosen differently, a different result is obtained, for example :

```
   s0=.odds ra 10000        NB. p=10000
   s1=.evens ra 5000        NB. n=5000
   (+/s0)+(+/s1)            NB. new sum
1.03962
```

Riemann proved that the sum of a non-absolutely convergent series can be made to have any arbitrary value by a suitable rearrangement of its terms; it can even be made oscillatory or divergent. Further it can be shown that in the case of the `ln2` series this sum is `½ln(4p/n)`, so that a rearrangement can always be made to achieve any given sum `S` by choosing the ratio `p/n` to be `¼exp(2S)`. For example, suppose `S` is required to be 1.5, `¼exp(3) = 5.02`, so take about 5 times as many positive terms as negative:

```
   s0=.odds ra 5020
   s1=.evens ra 1000
   (+/s0)+(+/s1)
1.49936
```

Choosing `S=0` requires `p/n` to be `¼`:

```
   s0=.odds ra 10000
   s1=.evens ra 40000
   (+/s0,s1),(+/s0),+/s1
_1.24991e_5 5.24035 _5.24036
```

which helps in resolving the algebraic paradox :

```
      ln2  = 1-1/2+1/3-1/4+? to infinity
       = 1+1/2+1/3+1/4+?,-2(1/2+1/4+?)
              =  1+1/2+1/3+1/4+?,-(1+1/2+?)
       = X - X    where X= the sum of the reciprocals
                =  0
```

Riemann's theorem may seem extraordinary at first sight, but should be considered in the light of the defining property of a divergent series, namely that the modulus of its sum can be made to exceed any arbitrary value simply by taking enough terms. In an informal sense the Riemann theorem describes a 'half-way house' property between absolutely convergence and divergence.

Yours sincerely,

About infinity -2-

by Sylvia Camacho (sylviac@blueyonder.co.uk)

Sylvia Camacho's answer to Thomson's letter on Infinity.

Dear Norman,

I read Cornelius Lanczos on classes of the number field some years before I was introduced to APL, so I was immediately receptive to Iverson's emphasis on improving the rigour of mathematical notation by distinguishing between the subtraction operator and the notation for a negative number. This made me hyper-sensitive to the ambiguity of the conventional expression for the infinite alternative of reciprocals. Your comment on the definition of 'convergent series' shows that the reciprocal series cannot converge as a simple sum but it does if every alternate value is subtracted. This, in conventional maths implies an order of evaluation:

```
   (((((((9-8)+7)-6)+5)-4)+3)-2) NB.conv.maths evaluates lft to rt
4
```

and is insensitive to a change of sequence providing each term is assumed to include its preceding operator and that no term is omitted, which was how I read Derbyshire's "but only if you add up the terms in this order.":

```
   (((((((9+7)-8)-6)+5)-4)+3)-2) NB.conv.maths: lft to rt
4
```

but, of course, an infinite series implies an infinite procedure and so any actual evaluation will be an approximation equivalent to a truncated series which will be sensitive to omissions. Graham, being a mathematician, appreciated this: hence his final table. Your reference to the need to balance the p and n of any re-arrangement is a way of expressing the same restriction.

If like Iverson we follow Lanczos and use notation which distinguishes the class of positive from that of negative numbers, the conventional expression contains only positive numbers and preceding addition and subtraction operators and will be sensitive to bracketing which can split an operator from the following number. Lanczos explains that the subtraction operator came to be accepted as the indicator of a negative number by giving it an implied left argument of zero. This is also the way that J interprets it:

```
   -3 NB. enter minus 3
_3    NB. J returns negative 3
```

Do you think that conventional mathematicians believe that making this distinction between operator and negative sign is to make a mountain out of a molehill? I have to accept that Lanczos did not use any special notation in his regular mathematical publications.

Your explanation of Riemann's Theorem is intuitively clear, although I could not begin to prove it. The infinity of reciprocals of integers comprises all possible fractions of the unit, from which any number could be composed just as any number can be represented in base-2 (I think).

Regards from sylviac@blueyonder.co.uk

References

1. Sylvia Camacho & Graham Parkhouse, “What is it about infinity?”, *Vector* 25:1, archive.vector.org.uk/art10500640

What is an array?

by Roger Hui (rhui000@shaw.ca)

In a recent e-mail [1], John Scholes reminded me of his last encounter with Ken Iverson, originally described as follows [2]:

In Scranton in 1999 during one of the sessions I was sitting next to Ken, and he leaned over and said to me – in his impish way – John, what is an array? Now I knew better than to rush into an answer to Ken. I guess I'm still working on my answers to that question.

Fools rush in where angels fear to tread…

What is an array?

An *array* is a function from a set of indices to numbers, characters, ... A rank-n array is one whose function *f* applies to n-tuples of non-negative integers. A rank-n array is *rectangular* if there exist non-negative integer maxima $s = (s_0, s_1, ..., s_{n-1})$ such that $f(i_0, i_1, ..., i_{n-1})$ is defined (has a value) for all integer i_j such that $(0 \le i_j)^{\wedge}(i_j < s_j)$. s is called the *shape* of the array. Etc.

This definition accommodates:

- APL/J rectangular arrays
- J sparse arrays
- infinite arrays
- dictionaries (associative arrays)

APL/J rectangular arrays

A typical APL/J rectangular array:

```
   2 2 3 ⍴ 'ABCDEFGHIJKL'
ABC
DEF

GHI
JKL
```

Listing the indices with the corresponding array elements makes the index function more apparent:

```
0 0 0   A
0 0 1   B
```

```
0 0 2   C
0 1 0   D
0 1 1   E
0 1 2   F
1 0 0   G
1 0 1   H
1 0 2   I
1 1 0   J
1 1 1   K
1 1 2   L
```

APL rectangular arrays to-date have been implemented by enumerating the array elements in row-major order (and employ the 'implementation trick' of not storing the indices). But there are ways to represent a function other than enumerating the domain and/or range of the function.

J sparse arrays

Sparse arrays were introduced in J in 1999 [3], [4]. In the sparse representation, the indices and values of only the non-'zero' elements are stored.

```
   ] d=: (?. 3 5 $ 2) * ?. 3 5 $ 100
 0 55 79 0  0
39  0 57 0  0
 0  0 13 0 51
   ] s=: $. d   NB. convert from dense to sparse
0 1 | 55
0 2 | 79
1 0 | 39
1 2 | 57
2 2 | 13
2 4 | 51
   3 + s
0 1 | 58
0 2 | 82
1 0 | 42
1 2 | 60
2 2 | 16
2 4 | 54
```

Reference [3] has an example of solving a 1e5-by-1e5 tridiagonal sparse matrix in 0.28 seconds.

Infinite arrays

Infinite arrays were described by McDonnell and Shallit [5] and Shallit [6]. Having infinite arrays facilitates working with infinite series and limits of sequences.

```
    ⍳4
0 1 2 3

    ⍳∞
0 1 2 3 4 5 …

    - ⍳∞
0 ¯1 ¯2 ¯3 ¯4 ¯5 …

    3 * - ⍳∞
1 0.333333 0.111111 0.037037 …

    +/ 3 * - ⍳∞
1.5

    ⌽ ⍳∞
DOMAIN ERROR
    ⌽⍳∞
  ^
```

Infinite arrays can be implemented by specifying the index function as a function. For example, the index function for `⍳∞` is the identity function, `⊢` or `{⍵}`.

Let `x` and `y` be infinite vectors with index functions `fx` and `fy`. If `s1` is a scalar monadic function, then `s1 x` is an infinite vector and its index function is `s1∘fx`, `s1` composed with `fx`. If `s2` is a scalar dyadic function, then `x s2 y` is an infinite vector and its index function is the *fork* `fx s2 fy`, or the dynamic function `{(fx ⍵) s2 (fy ⍵)}`.

In the following examples, the infinite vectors are listed with the index function, both as an operator expression (tacit function) and as a dynamic function.

```
    ⍳∞                      │   ⊢
0 1 2 3 4 5 6 7 …           │   {⍵}
                            │
    ∞ ⍴ 2                   │   ⊢∘2
2 2 2 2 2 2 2 2 …           │   {2}
                            │
    - ⍳∞                    │   -∘⊢
0 ¯1 ¯2 ¯3 ¯4 ¯5 …          │   {-⍵}
                            │
    3 * - ⍳∞                │   (3∘*)∘-∘⊢
1 0.333333 0.111111 …       │   {3*-⍵}
                            │
    ⎕←x←3*⍳∞                │   3∘*∘⊢
1 3 9 27 81 243 729 …       │   {3*⍵}
                            │
```

```
   □←y←(⍳∞)*2            │  *∘2∘⊢
0 1 4 9 16 25 36 …       │  {⍵*2}
                         │
   x+y                   │  3∘×∘⊢ + *∘2∘⊢
1 4 13 36 97 268 765 …   │  {(3*⍵)+(⍵*2)}
```

Dictionaries (associative arrays)

The proposed *string scalars* are suitable for use as indices in dictionaries. For example:

```
   ⍴⍴Caps
1
   Caps["UK" "China" "France"]←'London' 'Beijing' 'Paris'
   Caps
"UK"     │ London
"China"  │ Beijing
"France" │ Paris

   Caps["China"]
 Beijing

   Caps["USA"]
INDEX ERROR
   Caps["USA"]
  ^

   Caps ⍳ 'Paris' 'Tokyo' 'London'
"France" ⍳ "UK"

   ⍞ Caps
DOMAIN ERROR
   ⍞Caps
  ^
```

References

1. Scholes, J.M., e-mail on 2010-10-11 11:41.
2. Christensen, G., Ken Iverson in Denmark, *Vector*, Volume 22, No. 3, 2006-08.http://archive.vector.org.uk/art10002270
3. Hui, R.K.W., Sparse Arrays in J, APL99 Conference Proceedings, *APL Quote Quad*, Volume 29, Number 2, 1999-08-10 to -14.
4. Hui, R.K.W., and K.E. Iverson, J Introduction and Dictionaryhttp://www.jsoftware.com/help/dictionary/d211.htm, 2010.

5. McDonnell, E.E., and J.O. Shallit, Extending APL to Infinityhttp://www.jsoftware.com/papers/eem/infinity.htm, APL80 Conference Proceedings, 1980.

6. Shallit, J.O., Infinite Arrays and Diagonalizaton, APL81 Conference Proceedings, *APL Quote Quad*, Volume 12, No. 1, 1981-09.

Introducing "Some Topics in Computing"

by Keith Smillie (smillie@cs.ualberta.ca)

> History … interprets the past to understand the present and confront the future… . P. D. James, *The Children of Men.*

Introduction

On my eightieth birthday – which is now longer ago than I care to admit - a physicist friend gave me quite unexpectedly a copy of Great Ideas in Physics by Alan Lightman[1]. The two aims of the book according to the author who holds appointments in both Humanities and Physics at MIT are "to provide a grasp of the nature of science, and to explore the connection between science and the humanities".

Using only elementary mathematics not including calculus, Lightman discusses the conservation of energy, the second law of thermodynamics, the theory of relativity, and quantum mechanics, and introduces illustrative excerpts from the writings of Newton, Kelvin and Einstein as well as many other scientists, poets, novelists and philosophers.

I read the book with great pleasure and finished it feeling envious of the students, most of whom were in their first year, who had taken the course from which the book was developed. I particularly remembered the first page of the Introduction which began with a brief account of a visit Lightman had made to the Font-de-Gaume prehistoric cave where the walls are covered with 15,000-year-old Cro-Magnon paintings. After describing one painting of two reindeer, he continues:

> ... The light was dim, and the colours had faded, but I was spellbound. Likewise, I am spellbound by the plays of Shakespeare. And I am spellbound by the second law of thermodynamics. The great ideas in science, like the Cro-Magnon paintings and the plays of Shakespeare, are part of our cultural heritage.

Unfortunately most of our university science courses regardless of the discipline are intended for students majoring or intending to major in the subject or who wish to use the subject as a tool in their own discipline. Very little time is spent on the historical, literary or other cultural aspects of the subjects.

These matters are often treated briefly in panels inserted in the text. There appear to be relatively few introductory courses intended for those persons with modest scientific and mathematical backgrounds wishing some knowledge of a

scientific discipline as part of a liberal education. There are certainly very few such courses or books written by specialists which introduce computers and computing as part of our culture to the general reader.

Therefore, encouraged by Lightman's book I decided to assemble a few chapters dealing with what I consider some of the more important ideas in computing as seen in historical perspective. To avoid having to define computer or computing science and to emphasize the undoubted incompleteness of the work I used the title "Some Topics in Computing". As with Lightman's book there is very little mathematics, and certainly no calculus. However, in addition to conventional mathematical notation J is used both as an executable mathematical notation and as a programming language. The first chapter gives a brief introduction to J and further aspects of J are introduced in subsequent chapters as they are needed and in an unobtrusive manner as possible. The Table of Contents is given as an Appendix to this paper, and the chapters are available as Web pages at http://goo.gl/39jq2 [2].

The purpose of this short article is to give a very brief description of the topics presented on the Web in the hope that some readers will be encouraged to make use of some of the material in their own work and may extend it in directions and with emphases which they consider necessary or desirable.

Topics

The first chapter introduces Kenneth Iverson and APL and then J as a "modern dialect" of APL. J is then illustrated by two short "dialogues", annotated sessions with the computer, in which some of the J primitive verbs are introduced by means of simple examples, one of which finds the number of grocery items purchased in a shopping trip and the total cost of the items. This allows the introduction of both functional and explicit definition of verbs and a comparison with a BASIC program for the same problem. A final dice-rolling example allows the introduction of the very useful verb table /, and is the first of several dicing examples.

Number systems are the subject of Chapter 0 which begins with the additive number systems of ancient Egypt and Greece and the multiplicative systems of China and Japan. This is followed by a discussion of our positional number system which is illustrated with several recreational examples including the game of Nim and an example from Alice in Wonderland to be mentioned later in this paper. The chapter ends with an extension of the dice-rolling example of the first chapter.

The first two chapters give a somewhat leisurely introduction to J which it is hoped will prepare a reader new to the language for the following chapters on the development of computing. In brief, these chapters discuss the work of John Napier, Charles Babbage, George Boole, Alan Turing, and some of the pioneers in early computing. J is used to provide illustrative examples such as a Windows form for a Difference Engine simulator and a simulator for a Turing machine, and a brief introduction to Boolean algebra. The dice-rolling example appears again in the discussion of programming languages where dice-rolling programs are given in twelve different languages including a machine-language simulator written in J. The final chapter makes some remarks on solvable and unsolvable computational problems and ends with a brief discussion of the on-going work on polynomial- and exponential-time problems and the P vs NP problem.

Computing and literature

Although we have taken an historical approach to the growth of computers and computing, we must not neglect other branches of the humanities as a source of examples which may illuminate and enrich computing topics. Indeed some of these examples may illustrate that we have been introduced to important concepts in computing and mathematics through books and stories we have known since childhood. In this section we shall give just a few examples from English literature.

We shall begin with number systems, as we did in Chapter 1 of Topics, with the following from Thomas Hardy's The Trumpet Major which takes place during the Napoleanic Wars as an illustration of the problems presented by the decimal number system to those with a limited formal education:

> Behind the wall door were chalked addition and subtraction sums, many of them originally done wrong, and the figures half rubbed out and corrected, noughts being turned into nines, and ones into twos. These were the miller's private calculations. There were also chalked in the same place rows and rows of strokes and open palings, representing the calculations of the grinder, who in his youthful ciphering studies had not gotten as far as Arabic figures.

Number systems to a varying base are illustrated at the beginning of Alice in Wonderland as Alice repeats the multiplication table shortly after having fallen down the rabbit hole:

> Let me see; four times five is twelve, and four times six is thirteen, and four times seven is – oh dear! I'll never reach twenty at that rate!

A marginal note in Martin Gardner's Annotated Alice[3] refers to an explanation which justifies Alice's arithmetic since four times five is twelve to base eighteen,

four times six is thirteen to base twenty-one, … and four times twelve is nineteen to base thirty-nine. However, four times thirteen is not twenty to base forty-two so Alice does not reach twenty after all.

A character much loved by children is A. A. Milne's Winnie-the-Pooh who early in the stories illustrates the difference between the exclusive- and inclusive-or:

> Pooh always like a little something at eleven o'clock in the morning, and he was very glad to see Rabbit getting out the plates and jugs; and when Rabbit said, 'Honey or condensed milk with your bread?' he was so excited that he said 'Both,' and then, so as not to seem greedy, he added, 'But don't bother about the bread, please.'

An excellent illustration of the indirect method of proof is given when Pooh is looking for companions to join himself and Christopher Robin in an Expedition to the North Pole. He meets Rabbit first and says:

> 'Hallo, Rabbit,' he said, 'is that you?'
>
> 'Let's pretend it isn't,' said Rabbit, 'and see what happens.'
>
> (Hello, Rabbit, is the number of primes infinite? ...)

Charles Dickens, who knew Babbage well, got some of his ideas in Little Dorrit for the Circumlocution Office, the government office that did nothing, from Babbage's experiences with the government and modelled Daniel Doyce in part on Babbage.

Arrays with one or more dimensions of length zero occur in Douglas Adams's The Hitch Hikers Guide to the Galaxy: "A hole had just appeared in the galaxy. It was exactly a nothingth of a second long, a nothingth of an inch wide, and quite a lot of millions of light years from end to end."

The problems associated with recursive processes and their termination are illustrated in Arthur C. Clarke's "The Longest Science-Fiction Story Ever Told" which tells of an editor who writes a letter which ends by referring to itself and thus starts over again and again … .

Many more examples could be cited but it is hoped that these few may suffice to show how fiction can provide a source of examples to illustrates important ideas in computing, and indeed in other fields of science. Moreover such examples in themselves may encourage the science student to put aside, if only briefly, the prescribed science texts and turn to literature for relaxation and renewal. The rewards of such actions are beautifully expressed in the Introduction to one of the several editions of Mrs Gaskell's Cranford:

… in that pleasantest hour of all where the toils of the day are over and business cares are forgotten, the last hour of the day before retiring, one reads again one chapter; or forgetful of the morrow one reads on along the peaceful river of gently flowing prose, of effortless charm, and tranquil truth.

References

7. Lightman, Alan, 2000. Great Ideas in Physics. McGraw-Hill, New York.

8. https://webdocs.cs.ualberta.ca/~smillie/RevisedTopics/TopicsRev.html

9. Gardner, Martin, 1960. The Annotated Alice. Bramhall House, New York.

10. Smillie, Keith, 1986. “Language, literature and the computer.” The Creating Word, P. Demers (ed.). The University of Alberta Press, Edmonton, Alberta.

Appendix: Table of Contents

1. Introduction
2. Kenneth Iverson, APL and J
    - Development of APL and J
    - A very short dialogue with J
    - A longer dialogue with J
    - Grocery shopping again; Summary
    - Throwing dice
3. Positional Number Systems
    - Introduction
    - Additive systems
    - Multiplicative systems
    - Arithmetic tables
    - Positional systems
    - Guessing numbers
    - The binary clock
    - Down the rabbit-hole
    - The game of Nim
    - A genealogical problem
    - A closer look at multiplication
    - Rolling more dice
4. John Napier and Logarithms
    - Introduction
    - John Napier of Merchiston
    - Napier's logarithms
    - Further development of logarithm tables
    - Napier's rods
    - Slide rules
5. Charles Babbage and his Engines
    - Charles Babbage
    - The method of differences
    - The Difference Engine
    - A simulator for the Difference Engine
    - The Analytical Engine
    - Other Difference and Analytical Engines
    - Prime numbers and coffee tables

6. George Boole and Logical Design
    - Aristotelian logic
    - George Boole
    - Boolean algebra
    - Truth tables
    - Binary addition
7. Alan Turing and Computability
    - Mathematical foundations
    - Early life and education
    - Bletchley Park
    - National Physical Laboratory and Manchester
    - Turing Machines
    - Computability
    - A play and a novel
    - A Turing Machine simulator
8. Early Computers
    - Introduction
    - Electromechanical computers I. Konrad Zuse
    - Electromechanical computers II Howard Aiken
    - Eletromechanical computers III. The Bell Telephone computers
    - Electronic computers I. ENIAC and EDVAC
    - Electronic computers II. NPL, Manchester and FERUT
    - Electronic computers III. Cambridge and EDSAC
    - Nim-playing computers; A machine-language simulator
9. FORTRAN and Some Other Languages
    - Introduction
    - Before FORTRAN; FORTRAN; BASIC; ALGOL; Pascal; C, Java and Perl; MATLAB
    - Spreadsheets
    - Backus-Naur Form
    - Acknowledgements
    - Appendix - Example programs for dice frequencies
10. Some Overwhelming Numbers
    - Introduction
    - Bubble sort
    - A couple of legends
    - Some problems from the real world

- A few notes on cryptography
- Public key encryption
- P vs NP
- J programs

11. Appendix. J4.06 Script File

A P L

Eight-way street

Handling high-ranking arrays

by Stephen Taylor (sjt@5jt.com)

Describes techniques for handling arrays of rank much higher than three; also introduces a mnemonic for the left argument of dyadic transpose.

When you sup with the devil, use a long spoon. (Proverb)

One of the features of APL attractive to the learner is how functions work on conformable arrays without loops and indexes. When `A×B` can refer to arrays of arbitrary rank it is clear one has stepped into a world of new abstraction and generality.

Later it can be surprising to find how little use one has made of higher-rank arrays. Interpreter writers track their usage to learn where optimisation would be valuable. Only a tiny fraction of arrays have ranks higher than 3. (Let us call such arrays ‘noble’.)

Somehow this seems disappointing. When I started to write APL, I foresaw working in higher realms of abstraction and generality, free from the mental clutter of loops and indexes. The loops are mostly gone, but noble arrays remain tantalisingly rare. I conducted an informal survey of career APL programmers I know. The highest-rank array reported used in commercial software was 5.

The dreams of youth are notoriously prone to disappointment. But is useful to understand the sources of disappointment.

In contrast to youthful dreams, much of life is quotidian, spent brushing one’s teeth and washing dishes. So with programming. Even with much looping swept away there is much to be done that resists abstraction. Life and programming both are less tractable than our dreams.

Another source I can own to is a combination of my own laziness and others’ pressure for results. It takes time and effort to master techniques of higher abstraction and generality. Suppose one is faced, say, with multiplying a table through a rank-4 array. One has perhaps a hazy idea of a short expression that would do it. But writing two loops solves it faster than working out the unfamiliar expression. Repetition of the problem might prompt one to stop and

study. Peer pressure from more experienced colleagues might also do it. Otherwise one is pressed always toward the road more travelled.

Visual Basic is famously easy to start writing in. It is also notoriously quick to mire a writer's ambition in repetitive code. We might call this the 'VB effect'. But APL writers are also vulnerable to it, and for similar reasons. Both languages favour the self-taught, enabling domain experts, untrained in the orthodoxies of software development, to produce usable code. Without support from writers more experienced in abstractions, autodidacts are liable to miss available coding abstractions.

Sometimes this has profound consequences. I was recently asked to rewrite a core section of an application developed and maintained in just this way over twenty years. The chief developer, a domain expert, was soon to retire. Faced with changes required by regulatory changes he advised that he could not patch in further changes with any confidence. The algorithms should be rewritten as they would have been had the eventual requirements been known twenty years earlier. After years of 'cut and paste' patching, the code base had, as some developers put it, "gone sour". It could no longer be amended with confidence.

Going sour is terminal. Code goes sour when its developers can no longer navigate its exceptions, tricks and redundant parts. Sour code has to be rewritten.

"Software is a constant battle against complexity."[1] Developers aim to defer as long as possible the souring of code. Abstraction is a weapon in this battle. When short, abstract code has to be revised, it encourages 'clean' reformulation that preserves the rigour of the original. It is less susceptible to the accumulation of 'cut-&-paste' patching that sours code.

Reading the soured application code I was able to see the traces of cut–&-paste work. Pages of code that looked something like this:

```
NRRUBBLE ← RATER × NRUBBLE
NSRUBBLE ← RATES × NRUBBLE
NTRUBBLE ← RATET × NRUBBLE
NURUBBLE ← RATEU × NRUBBLE
MRRUBBLE ← RATER × MRUBBLE
MSRUBBLE ← RATES × MRUBBLE
MTRUBBLE ← RATET × MRUBBLE
MURUBBLE ← RATEU × NRUBBLE
```

Tracing execution revealed the `RUBBLE` arrays to be vectors. One could see within the verbose solution a terse, higher-rank solution struggling to emerge.

Factor arrays

Analysis revealed the bulk of this code allocated a sum of money. The vectors were time series, created by applying interest rates, inflation rates and other time-related factors. The money was split other ways as well. In some cases the money was allocated to one of a choice of categories. In others, it was apportioned between categories.

All of these allocations could be represented by multiplication. To apply an interest rate R across valuation years Y:

```
fTime ← (1+R)*⍳⍴Y
```

To allocate to one of a list of categories `CATS` according to category `C`:

```
fCat ← CATS=C
```

To apportion across periods according to dates `D1 D2 D3`:

```
fPeriod ← apportionPeriod(D1 D2 D3)
```

Since these 'business rules' are defined orthogonally, we can use Cartesian (outer) products to construct a rank-3 array that allocates, apportions and revalues:

```
VAL ← AMT × fTime ∘.× fCat ∘.× fPeriod
```

Or, to remove the repetition:

```
VAL ← AMT × ⊃ ∘.×/fTime fCat fPeriod
```

The result `VAL` has 3 axes: valuation years, category and period.

Sadly, the problem does not admit of so satisfyingly simple a solution. The outer-product reduction works only because the factors are mutually independent. But some business rules work across multiple axes.

For example, a set of rates apply across time, but differ between categories. The rate table `RT` has `⍴Y` rows, and columns corresponding to the categories of `CATS`. That is to say, `RT` has the time and category axes. We can conveniently insert it into our combination of factors:

```
VAL ← AMT × (RT × fTime ∘.× fCat) ∘.× fPeriod
```

Where the old code had a plethora of similar names, the new has everything stacked in a rank-3 array, from which we can select what we need by indexing. There turns out to be a good deal of this ahead. It is inconvenient for the reader to remember that, for example, the first index on the period axis denotes pre-1988 and the second post-1988. We shall do better with enumerators:

```
(iM iN iO iP) ← ,¨⍳4 ⍝ enumerate category axis
(iPr8 iPo8)   ← ,¨⍳2 ⍝ enumerate period axis
```

Thus we can select the pre-1988 value in 2013 for category N a `VAL[Y⍳2013;iN;iPr8]` or tabulate the post-1988 values for categories O and P as `VAL[;iO,iP;iPo8]`. Note that the index enumerators are vectors, so indexing in this way preserves rank: the result too has rank 3. This is not very valuable for a rank-3 array, but will turn out to be helpful as we add axes to `VAL`.

`VAL` still has three axes. We suspect it will need more before we're done.

It turns out we need `VAL` calculated both with and without the rates in `RT`. We can accommodate this with another axis. First we extend the rate table to apply and not apply:

```
fRates ← RT ∘.* 1 0
```

`fRates` has 3 axes: valuation years, categories and a new axis: with and without rates. We want some more enumerators:

```
(iWth iWto)   ← ,¨⍳2 ⍝ enumerate: with & without rates
```

Folding `fRates` into the expression for `VAL` requires a little care now. We could jam the extra axis onto the `fTime fCat` outer product:

```
VAL ← AMT × (fRates × (fTime ∘.× fCat) ∘.× 1 1) ∘.× fPeriod
```

Or we could enclose rank-3 `RT` into a table of 2-element vectors. This can then be multiplied by the `fTime fCat` outer product: scalar extension multiplies each element of `fTime∘.×fCat` by a 2-element vector:

```
VAL ← AMT × (⊃ (fTime ∘.× fCat) × ⊂[3]fRates) ∘.× fPeriod
```

We can think of the enclose on the third axis as 'hiding' it, leaving only the time and category axes exposed. Disclosing the result of the multiplication 'restores' enclosed axes at the end of the shape – in this case exactly where the enclosed axis had been before.

VAL now has four axes: valuation year, category, with/without rates, and period. Still we think there will be more axes before we're done.

Switching axes

APL developers often write to explore the problem domain. In working with a noble array one might find oneself adding and removing axes as one's view of the domain improves. Having index enumerators and axis enumerators (see below) are a great help in revising the code as the shape of VAL changes.

There is a particular problem with the approach used above to generate the factor arrays. While exploring the domain, their shapes change. (In the example above, we saw the rate table shifted from two axes to three.) Some of the axes of the factor arrays match some others, some will not.

Combine factor arrays into a single array of factors by successively enclosing along selected axes, multiplying and disclosing – as above. Each time a factor array changes shape, this sequence has to be revised. (Sadly, axis enumerators are no help here, as they refer only to the axes of VAL.)

Worse, as the combination of factor arrays gets revised, so the shape of the result alters. Every axis will be there, but not necessarily in the order you want. Each revision of the combination is liable to change the sequence of axes. We know dyadic transpose will re-order the axes of an array. Now is the time to get a grip on its left argument.

Suppose that VAL is to be a rank-8 axis. Let's define eight enumerators for its axes.

```
(aTIM aCAT aPRD aSEX aRAT aBNS aYRS aPLS)←⍳⍴⍴VAL ⍝enumerate axes
```

We have combined the various factor arrays we defined from the business rules. The resulting factor array has all eight axes, but not in the order above. How do we get them into the right order? Start by identifying the axes in the combined factor array. Suppose they have emerged from the combination in this order:

```
aYRS aRAT aTIM aPRD aSEX aBNS aCAT aPLS
```

Very good. Then we use this array as the left argument of our dyadic transpose:

```
aYRS aRAT aTIM aPRD aSEX aBNS aCAT aPLS⍉
```

puts the axes in the order they are to be in VAL.

From using axis enumerators in this way we see that the left argument of dyadic transpose functions as an address list – where the corresponding axes of the right argument are to end up.

Summing and selecting

In reading values from `VAL` the business logic sometimes needs categories and periods distinct, sometimes summed.

Tabulating with-rate values for all years and categories, summed across period:

```
+/VAL[;;↑iWth;]
```

The ↑ is required to collapse the third (with/without rates) axis, as the enumerators are all length-1 vectors. (We shall see shortly why this is so.) The sum defaults to the last axis and removes it. The result is a table of valuation years by category.

Tabulating with-rate values for all years and periods:

```
+/[2]VAL[;;↑iWth;]
```

Here the axis operator applies the summing to the category axis. Reading the expression requires us to remember that the second axis is for categories. Again, we will do better with enumerators:

```
(aTIM aCAT aRAT aPRD)←⍳4 ⍝ enumerate axes
```

This enables us to write the previous tabulations as

```
+/[aPRD] VAL[;;iWth;]
+/[aCAT] VAL[;;iWth;]
```

with some gain in legibility. Now we see the value of using vector index enumerators so that indexing does not reduce rank. It allows us to use the axis enumerators for reduction. (A drawback above is that the rates axis remains, with length 1.)

Nor is this is an approach that scales well to nobler arrays. Suppose our array has grown to 8 axes. We wish to index some of them and sum others. The reader's primary interest is to see what has been selected and what axes are in the result.

We can use successive summations providing we sum axes in descending order. For example, for axes

```
(aTIM aCAT aPRD aSEX aRAT aBNS aYRS aPLS)←⍳⍴⍴VAL ⍝ enumerate axes
```

we could get a table with rows and columns corresponding respectively to aSEX and aCAT something like:

```
sel ← VAL[iThis;;;;;iThat;;iTher]
⍉+/[aTIM]+/[aPRD]+/[aRAT]+/[aBNS]+/[aYRS]+/[aPLS] sel
```

The indexing makes the selection clear enough, but it still takes some work to see that the axes not summed are aCAT and aSEX and thus the result is a table with rows and columns corresponding to those axes, then transposed. For maintenance one would worry that changes in the order of axes will change the result of the successive summations without necessarily breaking execution.

Better if we could write something like:

```
aSEX aCAT SUMSOF VAL[iThis;;;;;iThat;;iTher]
```

This it turns out we can do quite neatly:

```
     ∇ Z←resultaxes SUMSOF array
[1]   ⍝ array summed across all axes except resultaxes
[2]   ⍝ result has axes in order of resultaxes
[3]   Z←(⍳⍴,resultaxes)⍉+/⍤¯1,⍤¯1⊂[(⍳⍴⍴array)~resultaxes] array
     ∇
```

The expression in line 3 encloses all the axes that are to be 'hidden', producing an array whose shape contains the axes in resultaxes, but in ascending order. Each cell of this is then ravelled and summed. The dyadic transpose then shuffles the axes into the desired order.

While any change to the shape of VAL would still require a change to the indexing, provided the aSEX and aCAT axes persist, they may remain as the left argument of SUMSOF.

Conclusion

Using index enumerators considerably improves readability when selecting from noble arrays. Using axis enumerators makes it easier to put axes of intermediate arrays into a desired order. The left argument of dyadic transpose may be thought of as an 'address list' of where you want the corresponding axes of the right argument to be 'sent'.

The dyadic function SUMSOF improves the readability and robustness of what would otherwise be a sequence of sum functions vulnerable to shape changes in VAL.

References

1. Eric Evans, *Domain-Driven Design*, Addison Wesley, 2003

Co-operators

by Phil Last (phil.last@ntlworld.com)

Based on a presentation given at Dyalog'11 in Boston on 5 Oct 2011, Phil Last develops some demonstrative operators to solve some common programming problems.

Preamble

I've been developing user-defined operators since a time when it was only possible to simulate them by executing prepared expressions that include a function name passed as left argument to another function. The following allowed first-generation APLs to use reduction with any dyad:

```
     ∇ R←F RED W
[1] ⍝ R←F⌿W
[2]   R←''⍴⍴W
[3]   R←⍎(⍴,F)⍎,((R,⍴,F)⍴F),' ','W','[',(⍕(⍳R)∘.+,0),((R,-1-
⍴⍴W)⍴';'),']'
     ∇
    'f' RED ⍳6
( 0 f( 1 f( 2 f( 3 f( 4 f 5 )))))
    'ϵ' RED 2 5⍴'aeiou','vowel'
0 1 0 1 0
```

So when IBM announced a mechanism for users to define their own operators in about 1983 I had a queue of them awaiting proper implementation.

When Dyalog announced D-fns with lexical scope in 1997 I unashamedly switched to using that as my standard notation for all new functions and operators and embedding D-fns for most amendments to old ones. So without apology I'll warn you that almost everything you'll see here is in D-notation. Also that some coding techniques might be new to you.

In the original paper to the Boston presentation I included a short note about those techniques. This time I refer you to a companion article that appears in the same edition of *Vector* as this: "A way to write functions". If you see anything here that is confusing or appears to make no sense at all you'll probably find it explained there.

Application-specific operators tend to be rare, often restricted to two or three in a large application while I currently have a collection of over forty more-or-less

general purpose ones. It's always mystified me that so few APLers seem to write or even use them at all.

A simple piece of code that maps or operates on one or two input arrays of known domain and structure can be extracted and encapsulated into a function that can be called upon whenever needed and this perhaps should almost always be done. An extension of that where different but syntactically congruent operations were to be applied would encourage us to write an operator and pass the operation in as a parameter along with the arrays.

I should mention here that strictly speaking when we write an operator what we are writing is the definition of the derived function of that operator. When we say an operator returns a function the operator does nothing. The parser merely binds it with its one or two operands. All the work specific to that operator is still to be done by the derived function and that after it has one or two arguments to operate on. In what follows I'll almost certainly forget this distinction between the derived function and the operator and refer to the operator's doing things which are actually the province of the derivation. I doubt that this will confuse.

Co-operators

A class of operator that I've become interested in comprises those that are designed to be called multiple times within the same expression:

```
[a] f op g op h op j op k op l w
```

where `a` and `w` are arrays; `f, g, h, j, k` and `l` are functions [see Appendix A]; and `opp` is the operator in question. The first and most obvious thing to notice about this is that the operator is dyadic – it takes two operands – making the number of operands in the expression one more than the number of calls to the operator. I'll mention here that I'll use Iverson's names for monadic and dyadic operators – adverb and conjunction – if I need to distinguish them. Although without an awful lot of messy code an operator can't examine its operands – it just runs them on trust – in this context we need to know something about them. I'm calling these co-operators because in order to be useful they must take account of the likelihood that they are not operating alone.

An examination of the calling sequence of the above expression will help. Redundant parentheses show how it's parsed:

```
[a]     f op g op h op j op k op l w
[a](((((f op g)op h)op j)op k)op l)w
```

In first generation APL this was explained as operators' having 'long left scope' as opposed to functions' having 'long right scope'. Along with strand-notation and the generalisation of operators came the idea of 'binding strength'. The above parsing is now explained on the basis of right-operand binding's being stronger than left – so an operand between two conjunctions is bound by the one whose right operand it is; that to its left.

In the case above `g` is bound by `op` to its left, `h` by that to its left and so on to the rightmost `op` that has the whole function expression to its left `f...k` as its left operand – 'long left scope'. A corollary to this is that all but the leftmost call to `op` have left operands derived from `op` itself, that exception being the greatest complication in all that follows; we need to identify what is that 'leftmost call' because we'll have to treat it differently. We also can be sure that the right operand of every instance of `op` is one of the original operands of the unparenthesised expression. The arguments to the expression become the arguments to the first call, the rightmost.

Function Selection

Here's a simple operator to start us off. I never got into writing case statements like:

```
:Select whatever
:Case this ◇ ...
:Case that ◇ ...
...
```

because I'd already started using operators to control program flow before control structures were implemented in APL and D-fns don't permit them even if I wanted to. Instead consider an operator – call it `or` – to be used as:

```
0 0 1 0 f or g or h or j w ↔ 0 0 1 0(((f or g)or h)or j)w
```

I hope you can guess what this is intended to do. In the specific case above I want the result to be the result of `h w`, the boolean vector being used as a mask to select from the four operand functions which to apply to argument `w`.

We could imagine this as being implemented something like:

```
(0 0 1 0 / (f g h j))w
```

where a boolean compression is applied to an isolated list of functions but such a list is not recognised by APL as something it can deal with in any way let alone being able to apply compression to it. More about this later.

I'll allow only the first, leftmost, one to select the function, effectively treating the boolean as if it's its own less-than scan `<\`. If the mask is all zeros we shouldn't actually want to run any function at all; we should merely return the argument. So here are the first two lines:

```
or←{
      ~∨/⍺:⍵
```

Looking at the parenthesised version above we can see that the rightmost `or` has:

```
⍺⍺ ↔ f or g or h
⍵⍵ ↔ j
```

So if only the final element of `⍺` is a one we can run `j`:

```
</⍺:⍵⍵ ⍵
```

If this isn't the case then we need to do something with `⍺⍺`. We need to call it without the final element of `⍺` that applied to `j`. So:

```
or←{
      ~∨/⍺:⍵
      </⍺:⍵⍵ ⍵
      (¯1↓⍺)⍺⍺ ⍵
}
0 0 0 0 f or g or h or j 23
23
0 0 0 1 f or g or h or j 23
(j 23 )
0 0 1 1 f or g or h or j 23
(h 23 )
0 1 0 0 f or g or h or j 23
(g 23 )
1 0 0 1 f or g or h or j 23
( 1 f 23 )
```

These all look right except for `f`. I mentioned the complication of the leftmost call in the eponymous section above. In this we've called `f` as dyad `⍺⍺` in the final line of `or`. Another statement is needed to call `⍺⍺` when it's the leftmost of the supplied operands. We know that by then our expression is `f or g` so `⍺` must have a length of 2. If it were 0 0 we should have returned `⍵` unchanged on line:

```
[1] ~∨/⍺:⍵
```

If it were 0 1 we should have run `g` on line:

```
      [2] </⍺:⍵⍵ ⍵
```

So it must be 1 0 or 1 1 and in either case we should run `f`:

```
      [2.1] 2=⍴⍺:⍺⍺ ⍵
```

So:

```
    or←{
          ~∨/⍺:⍵
          </⍺:⍵⍵ ⍵
          2=⍴⍺:⍺⍺ ⍵
          (¯1↓⍺)⍺⍺ ⍵
    }
    1 0 0 0 f or g or h or j 23
 (f 23 )
```

We can make an alternative version of this that, rather than selecting the function indicated by only the first 1, will select all corresponding functions:

```
    0 1 0 1 f or g or h or j 23
 (g(j 23 ))
```

We need to look at the last item of `⍺` whether it's the only one or not and use the power operator to run `⍺⍺` and `⍵⍵` conditionally:

```
    or←{
          ~∨/⍺:⍵
          g←⍵⍵⍣(⊢/⍺)
          2=⍴⍺:⍺⍺⍣(⊣/⍺)g ⍵
          (¯1↓⍺)⍺⍺ g ⍵
    }
    1 0 1 0 1 1 f or g or h or j or k or l 45
 (f(h(k(l 45 ))))
```

Function Sequence

If we look at the reduction operator we see:

```
    f/12 23 34 45
  ( 12 f( 23 f( 34 f 45 )))
```

the function is inserted between the items of the argument. If the function could be a list then perhaps we could do:

```
     (f g h j)/12 34 56 78 90
  ( 12 f( 34 g( 56 h( 78 j 90 ))))
```

Again we can't do this because `f g h j` is illegal in APL. J implements this concept with the `tie` conjunction `` ` `` to bind the functions in a list of 'gerunds' that it applies with `/`:

```
    +`-`*`% / 1 2 3 4 5
0.6
```

But we can interpose a defined operator both to bind the functions and insert them:

```
    f seq g seq h seq j 12 34 56 78 90
          ↔ ((f seq g)seq h)seq j 12 34 56 78 90
          ↔ 12 f 34 g 56 h 78 j 90
```

Notice we have one more item in the argument than there are operands in the expression and one more of them than there are calls to the operator. Also that the operands are dyads while the derived function is a monad. The first (right-most) call will have:

```
    αα  ↔  f seq g seq h
    ωω  ↔  j
    ω   ↔  entire right argument
```

so we can run `j` between the last two items using reduction and catenate its result to the rest running `αα` on that result:

```
    seq←{
          αα(¯2↓ω),ωω/¯2↑ω
    }
    f seq g seq h seq j 12 34 56 78 90
(f 12  ( 34 g( 56 h( 78 j 90 ))) )
```

Visually scanning from the right looks good until we get to `f`. Again it's the left-most function that adds the complication. When `f` is the left operand, `f seq g` must be the sub-expression, there must be three items in `ω` so we do almost the same but we apply `αα` between rather than to the remaining pair after doing the same with `ωω`:

```
    seq←{
          3=ρω:αα/(¯2↓ω),ωω/¯2↑ω
               αα (¯2↓ω),ωω/¯2↑ω
    }
    f seq g seq h seq j 12 34 56 78 90
  ( 12 f( 34 g( 56 h( 78 j 90 ))))
    + seq - seq × seq ÷ 1 2 3 4 5
 0.6
```

and get our expected result.

Function Arrays

Another meaning that could be attributed to an isolated list of functions was in fact my first accidental encounter with them after about three months of APL. I had to solve the simple problem of the dimensions of a matrix formed from three others; one above the juxtaposition of the other two:

```
.-----------------------------.
|               A             |
|-------------.---------------------.
|             |         C           |
|      B      |                     |
|             |---------------------'
'-------------'
```

APL seemed so good at doing whatever I hoped it would that what I wrote was:

```
      (ρA)(+⌈)(ρB)(⌈+)(ρC)
```

expecting the pairs of functions to be distributed between the pairs of dimensions of the three matrices:

```
           .----.-------.----.-------.
           |    |       |    |       |
      (ρA)[0 1](+⌈)(ρB)[0 1](⌈+)(ρC)[0 1]
             |   |       |   |       |
             '---'-------'---'-------'
```

Of course it didn't work for the same reason that the other constructs above didn't work. Although in this case there are only two functions in each list we can extrapolate this to a list of functions corresponding to a vector of any length. If we define conjunction `fv` (function vector) we'll want:

```
a b c d (f fv g fv h fv j) w x y z      A [0]
a b c d (f fv g fv h fv j) ⊂w          A [1]
(⊂a)    (f fv g fv h fv j) w x y x      A [2]
         (f fv g fv h fv j) w x y z     A [3]
```

Our perennial problem of identifying the leftmost operand will depend on the length of the argument(s) as they must conform to the number of functions. We can check for a monad and for the two-item final call. But in cases [1] and [2] above `⍺` or `⍵` is a scalar so before the length check we'll resolve scalar extension with laminate and split creating `A` as 0 if `⍺` isn't supplied. If that's the case we won't use it anyway but it's easier to do it redundantly than to skip it:

```
fv←{
      ⍺←⊢
      m←1≡⍺ 1
      (A W)←↓(⍺⍪0),[¯0.1]⍵
      t←2=⍴W
```

For the two-item monad we want the two functions `⍺⍺` and `⍵⍵` applied to the two items of `W`:

```
      t∧m:(⍺⍺ 0⊃W)(⍵⍵ 1⊃W)
```

If it's a monad and this doesn't run it must have more than two items. We apply `⍵⍵` to the last item of `W` and `⍺⍺` to the rest:

```
      m:(⍺⍺ ¯1↓W),⍵⍵¨⊢/W
```

If it's still two items it must be a dyad so we do similar to the `t∧m:` case but using `A` also:

```
      t:((0⊃A)⍺⍺ 0⊃W)((1⊃A)⍵⍵ 1⊃W)
```

We must be left with the dyad with more than two items so we do similar to the `m:` case but with `A` also:

```
      ((¯1↓A)⍺⍺ ¯1↓W),(⊢/A)⍵⍵¨⊢/W
}
```

Putting this all together:

```
    fv←{
          α←⊢
          m←1≡α 1
          (A W)←↓(α⍪0),[-0.1]⍵
          t←2=⍴W
          t⍪m:(αα 0⊃W)(⍵⍵ 1⊃W)
          m:(αα ¯1↓W),⍵⍵¨⊢/W
          t:((0⊃A)αα 0⊃W)((1⊃A)⍵⍵ 1⊃W)
          ((¯1↓A)αα ¯1↓W),(⊢/A)⍵⍵¨⊢/W
    }

    1 2 3 4 f fv g fv h fv j 5 6 7 8
( 1 f 5 )  ( 2 g 6 )  ( 3 h 7 )  ( 4 j 8 )
    1 2 3 4 f fv g fv h fv j 5
( 1 f 5 )  ( 2 g 5 )  ( 3 h 5 )  ( 4 j 5 )
    1       f fv g fv h fv j 5 6 7 8
( 1 f 5 )  ( 1 g 6 )  ( 1 h 7 )  ( 1 j 8 )
          f fv g fv h fv j 5 6 7 8
(f 5 )  (g 6 )  (h 7 )  (j 8 )
    1 2 3 4 + fv - fv × fv ÷ 5 6 7 8
6 ¯4 21 0.5
```

Conditionals

```
    :If f w
    :AndIf g w
        r←h w
    :Else
        r←j w
    :End
```

As I mentioned above I really can't do with having to do stuff like that. It was always easy to write operators that did:

```
    f then g w              ⍝ if f w then r←g w else r←w
    t g else h w            ⍝ if t then r←g w else r←h w
                            ⍝ see *else* below
```

But putting them together to form something like:

```
    f then g else h w       ⍝ if f w then r←g w else r←h w
```

was never going to be so easy. The two conjunctions `then` and `else` need to co-operate, `f then g` being an operand of `else`, and introducing interdependencies should be avoided whenever possible. So how about a single co-operator

that fulfils both tasks? The biggest problem is what to call it. I believe at least one language uses ? for this – not a bad choice:

```
antecedent ? consequent ? alternative arg
```

I'll use cond which I also believe is used elsewhere so that in:

```
f cond g cond h w  ↔  ((f cond g)cond h) w
```

we apply f w first then g w if f w proves true or h w otherwise. The length of the argument isn't going to help as it could be anything at all. One thing we do know is that there's no left argument. It wouldn't make any sense if there was. The right call has:

```
αα  ↔  (f cond g)
ωω  ↔  h
```

We can't do anything with h yet as that's the alternative we want to run if f returns false so we need to run αα and we need to do it in such a way that it runs one or other of f and g. We can use that missing left argument as an internal flag to tell the left call what to do. Arbitrarily we'll give it a one to tell it to run f and a zero for g if f returns true:

```
cond←{
      α←⊢
      1 αα ω:0 αα ω
```

but this is no good because as soon as we call it it'll just call itself again unconditionally. We actually need to test α for being a 1 or 0 before we use it! Please see the companion article if lines [2] and [3] below confuse:

```
cond←{
      α←⊢
      1=α⍳0:αα ω      ⍝ if α is 1 run f ω
      0=α⍳1:αα ω      ⍝ if α is 0 run g ω
```

If α is ever going to be 1 or 0 we need to make it so, so now we run the line higher up with the two calls to αα. If 1 αα ω which calls f ω is true we run 0 αα ω that calls g ω

```
      1 αα ω:0 αα ω
```

but if f w isn't true we run h ω instead.

```
        ⍵⍵ ⍵
    }

    cond←{
        ⍺←⊢
        1=⍺⌷0:⍺⍺ ⍵      ⍝ if ⍺ is 1 run f ⍵
        0=⍺⌷1:⍵⍵ ⍵      ⍝ if ⍺ is 0 run g ⍵
        1 ⍺⍺ ⍵:0 ⍺⍺ ⍵   ⍝ if f ⍵ then g ⍵
        ⍵⍵ ⍵             ⍝ else h ⍵
    }

    >∘0 cond j cond k 23
(j 23 )
    >∘0 cond j cond k ¯45
(k ¯45 )
```

else: now if we look at the `else` expression that we saw a couple of pages back; I've copied it here for convenience:

```
    t g else h w            ⍝ if t then r←g w else r←h w
```

and just out of interest we try it with `cond` we find:

```
    1 g cond h 67
(g 67 )
    0 g cond h 67
(h 67 )
```

that cond is indeed a working else and we can see why on those lines [2] and [3] above. This isn't the way we should have written else because lines [4] and [5] won't run in either case but it's quite nice that the co-operator also works as a stand-alone:

```
    f cond g cond h w       ⍝ if f w then r←g w else r←h w
         t g cond h w       ⍝ if t then r←g w else r←h w
```

It turns out that that arbitrary 1 and 0 were not so arbitrary after all.

And if we want the usual semantics we can define:

```
    then←{⍺←⊢ ⋄ ⍺ ⍺⍺ cond ⍵⍵ ⍵}
    else←{⍺←⊢ ⋄ ⍺ ⍺⍺ cond ⍵⍵ ⍵}
```

More conditionals

I started the previous section with an `:If` construct that included `:AndIf`. It turns out that conjunctions `and` and `or` are very easy to write. They aren't co-operators. They just happen to work together:

```
and←{⍺⍺ ⍵:⍵⍵ ⍵ ⋄ 0}      ⍝ if ⍺⍺ ⍵ then r←⍵⍵ ⍵ else r←0
or←{⍺⍺ ⍵:1 ⋄ ⍵⍵ ⍵}       ⍝ if ⍺⍺ ⍵ then r←1 else r←⍵⍵ ⍵
f and g or h ↔ (f and g)or h
```

This is *not* the same as `(f ⍵)∧(g ⍵)∨(h ⍵)` that would run all functions, `f`, `g` and `h` and determine the result in the usual APL right to left mode. Operands here are run in left to right order and then only if they are capable of changing the result exactly as a series of `:AndIf` or `:OrIf` clauses in an `:If` clause.

We can now construct the statement at the very top of the "Conditionals" section:

```
f and g then h else j w ↔
      (((f and g)then h)else j)w
      ⍝ if f w and g w then r←h w else r←j w
```

but unlike the exclusivity of `:AndIf` and `:OrIf` control clauses and remembering that those operands that run will do so in a strictly left to right order we can write:

```
f or g and h and j then k else l w ↔
      (((((f or g)and h)and j)then k)else l)w
```

Forks

J refers to lists of functions as trains and parses them in a very particular way:

A train of 2 functions (2-train) is called a hook, which counts as a function in its own right; an APL equivalent would be:

```
(f g)       ↔ f⍥g⍨⍨
```

so that:

```
a(f g)w     ↔ a f(g w)
(f g)w      ↔ w f(g w)
```

A 3-train constitutes a fork (another function) where:

```
a(f g h)w   ↔ (a f w)g(a h w)
 (f g h)w   ↔ (  f w)g(  h w)
```

Beyond the 3-train a fork peels off from the right and becomes the right tine of the next construct, hook or fork, recursively:

```
(g f h j)          ↔ (g(f h j))
(g f h j k)        ↔ (g f(h j k))
(g f h j k l)      ↔ (f(g h(j k l)))
```

Even-trains above 2 that simulate a hook whose right tine is a fork aren't possible with a single defined operator but we can simulate an odd-train with a fork operator `fk`. The only difference will be that due to operand binding as mentioned above forks will be bound from the left:

```
f fk g fk h fk j fk k   ↔ (f fk g fk h)fk j fk k
```

We'll only consider a 'simple' fork of three functions here:

```
a(f fk g fk h)w   (a f w)g(a h w)
 (f fk g fk h)w   (  f w)g(  h w)
```

Again at the first call we have:

```
αα ↔ f fk g
ωω ↔ h
```

so can do something with `h`, but we need to pass its result along with the original argument(s) to the next call where `f` and `g` will be available:

```
fk←{
      αα(α ω)(α ωω ω)
}
```

But at that next call we don't want to do the same thing at all.

```
αα ↔ f
ωω ↔ g
```

We want to apply `g` between the results of `f` and `h`. This is the same problem as ever identifying the subsequent call but this time we can't rely on the length of the argument(s) to tell us when we are there. We need to pass in a flag to the second call to tell it – as with `cond` above - and we might as well have two; one for the monad and one for the dyad. In the first monadic call we present the flag as left argument to the next call. We have to hope no-one ever calls our fork with these flags as supplied arguments:

```
    fk←{
          α←⊢
          (m d)←'left monadic fork' 'left dyadic fork'
```

if it's a monad, `α` will be assigned `⊢` so for all `x – x ≡ α x –` so we pass `m`, `ω` and the result of `h` to the left call:

```
          ω≡α ω:m αα(ω)(ωω ω)
```

If we get past here we know `α` exists so if it's our monadic flag we run `g` between results of `f` applied to the origin `ω` and that of `h` which we computed above as `(ωω ω)`:

```
    fk←{
          α←⊢
          (m d)←'left monadic fork' 'left dyadic fork'
          ω≡α ω:m αα(ω)(ωω ω)
          α≡m:(αα 0⊃ω)ωω 1⊃ω
    }
    f fk g fk h 23
((f 23 )g(h 23 ))
```

So far so good. The dyad should be very similar. If `α` is `d` – the dyadic flag – we run `f` between our original arguments and `g` between that result and `h`:

```
          α≡d:(⊃αα/0⊃ω)ωω 1⊃ω
```

If not we'd better make it so passing `d`, `α ω`, and `α h ω` in the right dyad:

```
          d αα(α ω)(α ωω ω)
    }

    fk←{
          α←⊢
          (m d)←'left monadic fork' 'left dyadic fork'
          ω≡α ω:m αα(ω)(ωω ω)
          α≡m:( αα 0⊃ω)ωω 1⊃ω
          α≡d:(⊃αα/0⊃ω)ωω 1⊃ω
          d αα(α ω)(α ωω ω)
    }
    f fk g fk h 23
((f 23 )g(h 23 ))
    +/fk÷fkρ 12 23 34 45 56 67 78
45
    12 f fk g fk h 23
(( 12 f 23 )g( 12 h 23 ))
    12 34 56 78 90 >fk∨fk= 98 76 54 32 10
0 0 1 1 1
```

Afterword

If it were decided to allow APL to acknowledge an isolated list of functions as a valid syntactic construct it would remain to be decided how it should be applied and how the individual members should interact.

The designers of J decided to go for hook and fork.

There is currently discussion that Dyalog might make the same decision.

It isn't necessary to go for a single solution except as a default. The selection of hook and fork as default behaviour wouldn't need to preclude the introduction of operators that apply the list in some other way. The co-operators described here hint at some of those other ways. Instead of building up a derived function by alternating with their operands, they could take the list as a single operand and apply its members appropriately.

Appendix A

Functions used to demonstrate calling sequences:

```
f←{α←⊢ ⋄ '(',α,'f',ω,')'}
g←{α←⊢ ⋄ '(',α,'g',ω,')'}
h←{α←⊢ ⋄ '(',α,'h',ω,')'}
j←{α←⊢ ⋄ '(',α,'j',ω,')'}
k←{α←⊢ ⋄ '(',α,'k',ω,')'}
l←{α←⊢ ⋄ '(',α,'l',ω,')'}
```

A way to write functions

by Phil Last (phil.last@ntlworld.com)

A companion article to Phil Last's Co-operators paper to explain some of his more unusual or obscure programming techniques.

Someone recently stopped to look at my Dyalog development session. But only for a moment. He made a gesture to indicate his bewilderment and moved on. This is a man who writes most of his code in the direct (formerly known as "dynamic") functional subset of Dyalog APL (D-fns or D:).

As do I.

So what was the problem?

Having bewildered many more in a presentation at Dyalog'11 at Boston and having been rebuked by a number of them since, I've come to the conclusion that it's because I've discovered and use various techniques for doing common tasks in APL that have more verbose and perhaps more familiar counterparts.

As I've come to value all of these idioms and now find them highly recognisable I'd come to assume their self-explanatory nature but now accept that they present a relative impenetrability to many. This is hopefully to counter the latter and realise the former. As mentioned above almost all of my code is in D notation about which you can read in John Scholes' paper, "Dynamic Functions in Dyalog APL" [1] so I'll only mention here the parts whose possible unfamiliarity is the reason for this article.

Ambivalence

I probably should start where most of my functions do. All D-fns are potentially ambivalent: they can be called with or without a left argument. Whether the code actually refers to that argument is another matter. The simplest D-fn is: `{}` that doesn't refer to either of its arguments or anything else for that matter. Nevertheless most functions need to do something and if they expect a left argument the avoidance of a value error is something to be considered.

Traditional ways of doing this mostly check the name-class `⎕NC` of the argument name for zero and conditionally assign an array value.

When I started using IBM's APL2, which was the first time the problem arose for me, an earlier observation made me realise that a more generally useful course would be to attempt to fix a right identity function with the argument name:

```
    ∇ RES←LARG FOO RARG
[1]   0 0⍴⎕FX,⊂'R←LARG R'
...
    ∇
```

If `LARG` is supplied, `⎕FX` fails gracefully returning `⎕IO` because it won't fix a function in place of an array. But if `LARG` has been elided then `⎕FX` creates it as an identity function and any use of the name within `FOO` as a left argument behaves as if it isn't there. In `[2]` below if `LARG` is an identity function, `FO1` has no left argument and its result is passed to `RES`. So for elided `LARG`:

```
[2] RES←LARG FO1 RARG
```

is exactly the same as:

```
[2] RES←FO1 RARG
```

I went so far as to create a utility function whose argument was the left argument-name of the function that called it:

```
    ∇ AMBIVALENT AMBIVALENT
[1]   AMBIVALENT←⎕FX,⊂'R←',AMBIVALENT,' R'
    ∇
```

It reassigns its eponymous argument with the result of `⎕FX` possibly having created a semi-global identity function with the name contained in that argument.

```
    ∇ R←LEFT FO2 RIGHT
[1]   AMBIVALENT'LEFT'
[2]   R←LEFT FO3 …
…
    ∇
```

It was published in Vector 6.2 as a tongue-in-cheek response to something much worse in Vector 5.4 but didn't go down well with the other correspondents. I thought it was nicely declarative and self-documenting.

I have to admit there are some who are surprised by the fact that what they expect to be an array could be a function instead. Given the flexibility of infix notation what surprises me is that most will immediately assume the expression

`e t y`, which constitutes three letters randomly chosen from the line above, to represent a three item vector rather than any of the other eleven valid three-fold array or function expressions that can be formed from permutations of arrays, functions or operators.

Ambivalence in D-fns

The D method to accommodate an elided left argument `⍺` is a single assignment that is executed conditionally on the argument's absence.

I believe John Scholes was surprised to find that the equivalent of the above:

```
[1] ⍺←{⍵}
```

actually worked. It has exactly the same effect creating the identity function as `⎕FX` did. I used this until Dyalog Version 13.0 finally introduced the right function `⊢` that implements the right identity as a primitive. Thus many of my functions now start with the even more terse but equally useful and recognisable:

```
[1] ⍺←⊢
```

In twenty-five years I've found no situation where this technique fails to work and none where any other excels it.

Assigning a default

Some might object here that it's easier to assign a known value to the argument as default.

```
[2] ⍺←default
```

This can still be done after `[1]` above if we are content to use another name:

```
[2] larg←⍺⊣default
```

This uses the other new identity function of Dyalog 13.0, left `⊣` that returns its left argument if it has one or its right otherwise – following the J implementation of left `[` that I believe was Ken Iverson's later preference, rather than that of SAPL and APLX that don't return a result from the monad. Thus in our D-fn if `⍺` is supplied then `⊣` in the above expression gets its value as left argument and returns it to be assigned as `larg`; if not then `⍺` is `⊢` from `[1]`; `⊣` is called as a monad and returns its right argument default instead, to be passed by `⊢` and assigned to `larg`.

Very often though, assigning a default to be passed to another function is not the best thing to do. If a called function `f04` is designed to be ambivalent we can

assume it must be able to take care of its own defaults. Were an ambivalent caller `f03` to provide another default it would override whatever was the intention of the author of `f04`. This must be decided case by case.

All this becomes much more important in the case of operators. If a derived function, an operator with its operand(s), is called without a left argument it very reasonably can be inferred that, all else being equal, the function operand is also expected to be called as a monad. Each ⍨ is a primitive case in point.

More ambivalence

We usually apply the word *ambivalent* to functions permitting the optional elision of left argument `⍺:` argument ambivalence. The dichotomy above where ⍺ can be an array or a function (supplied or assigned) also occurs in operators' accepting either an array or a function as operand ⍺⍺ and/or ⍵⍵. Hence: operand ambivalence. The left operand of the /operator of APL2 and the right of the power ⍣ operator of Dyalog both have this property.

In:

```
[4] ... ⍺⍺⊣⍵
```

the use of left ⊣ allows us to supply left operand ⍺⍺ as either a function or an array without our having to code separate cases for either of them. If ⍺⍺ is an array then ⍺⍺⊣⍵ is just ⍺⍺, the left argument of ⊣. If it's a function then it's applied to the result of monadic ⊣ which is ⍵.

Differential processing

Perhaps unusually, different processing is sometimes required for the monadic case of a function than the dyadic. Unusually in defined functions, that is; we should be surprised if primitives didn't exhibit this behaviour. In a D-fn this would probably entail the using a guard. What should be the condition?

In:

```
[2] ⍺←⊢
[3] 1≡⍺ 1: expression
```

`1≡⍺ 1` returns true and invokes `expression` if the value 1 matches the value of `⍺ 1`. This might seem unlikely. How can a scalar 1 match a two item vector? Obviously it can't but if ⍺ is ⊢ then `⍺ 1` is `⊢ 1` which is just 1. In fact this works equally well for any array replacing the 1 on both sides of the equality so, for instance, to make it more explicit:

```
[3] 'monad'≡α'monad': expression
```

Scripting languages from very early on have permitted similar constructs:

```
If . = . %1 Then expression-for-missing-%1
```

which behaved identically after parameter substitution for optional `%1` though the explanation would be somewhat different.

Conditions and multi-line expressions

It's possible to break any D-fn inside its enclosing left or right brace. The next two D-fns are equivalent:

```
{single expression}

{
      single expression
}
```

Not only this but long lines embedding anonymous D-fns can also be broken:

```
{this long line{embeds another D-fn}within it}

{this long line{
      embeds another D-fn
}within it}
```

Guards sometimes present a problem in that a quite long line is required for the terminating expression to run when the condition is true. By embedding the expression in an anonymous D-fn we can extend the statement onto more than one line:

```
conditional expression: α{
      long terminating expression to be triggered when true
}ω
```

Sometimes we want to run an expression conditionally or perhaps one expression or another to extract the value within, rather than terminating, the current function. In this case embedding the entire guard with its conditional, consequent and alternative expressions in an anonymous D-fn serves very well:

```
value←α{condition: expression if true
      expression if false
}ω
```

In both these cases it's worth supplying the original arguments so that all names and tokens inside the inner D-fn reflect those outside.

Commented one-liners

The following doesn't define a D-fn:

```
    f03←{expression ⍝ comment}
SYNTAX ERROR
    f03←{expression ⍝ comment}
                                ^
```

The reason is that the final brace is a part of the comment and doesn't actually close the D-fn. This has unfortunately and unjustifiably caused some to claim that D promotes uncommented code. The ability to create functions – D-fns or otherwise – without comments; believing them to be unnecessary; or that one doesn't have time to write them promotes uncommented code.

Fortunately the left ⊣ function comes to the rescue yet again. We can use:

```
    f03←{expression⊣'comment'}
or the prettier:
    f03←{expression⊣'⍝ comment'}
or even:
    nb←⊣
    f03←{expression nb' comment'}
```

Naming operands

This is something I've relearned to do only very recently having gone perhaps too far down the road of designing algorithms that required no assignments whatsoever.

It's always useful and very often necessary or desirable to be able to define anonymous D-fns embedded within the coding of a D-fn or D-op. This entails another level of parameter substitution so for instance, operator `where`, that needn't be understood, contains two levels of embedded operations:

```
    where←{
          (⍴⍵)⍴⍺⍺{⍵ ⍺⍺{⍵⍷⍺,[-0.1]⍵\⍺⍺ ⍵/⍺},⍵⍵⊣⍵}⍵⍵,⍵
    }
                 ↑                                ↑
                      ↑                    ↑
```

In order that the operands `⍺⍺` and `⍵⍵` continue to refer to the same thing they must be passed in as operands at each level making the inner operations

operators also. It is also quite difficult to see that the innermost operation is a monadic operator deriving a dyadic function while the next out is a dyadic operator deriving a monadic function as is the outer, named, D-op. The following version appears much simpler merely by naming the operands:

```
where←{f←⍺⍺ ⋄ g←⍵⍵
       (⍴⍵)⍴{⍵{⍵∊⍺,[-0.1]⍵\f ⍵/⍺},g⍤⍵},⍵
}
```

Lexical scope rules mean that named items are visible to any D-fn or D-op defined within the one wherein they are named. Thus they don't have to be respecified for the inner levels so we actually reduce the number of tokens in the line as well as avoiding the added complication of the embedded operations' being promoted to D-ops.

Separating operands and arguments

Operator operands can be either functions or arrays. The power ⍣ operator takes either a terminating function or an iteration count as its right operand. In the latter case we have an array (scalar integer) operand and an array right argument. This brings up the question of the relative strengths of operand- versus vector-binding. Implementors chose differently in this regard, vector binding being the strongest of all in Dyalog APL and NARS2000, while both bracket- and right-operand binding exceed it in APL2 and APLX.

Consequently we have the following dichotomy. The expression:

```
f dop a b c
```

where `f` is a function, `dop` is a dyadic operator and `a`, `b` and `c` are arrays, is parsed as follows:

APL2, APLX &c.:

```
f dop a b c ↔ (f dop a)(b c)
```

an array expression; function with argument; `a` is the operand and `b c` is the argument.

Dyalog, NARS2000 &c.:

```
f dop a b c ↔ f dop(a b c)
```

a function expression without an argument; `a b c` is the right operand.

This difference adds an extra complication to Dyalog which is well worth the cost considering the alternative:

APL2:

```
f dop 1 2 3 ↔ (f dop 1) 2 3
```

what appears to be a numeric vector is treated as a scalar separated from the remainder.

So in Dyalog we need to separate the array right operand from the argument. The surest way to do it is to parenthesise the entire function expression:

```
(f dop operand)argument
```

But a neater way is to insert an identity function between the operand and argument. Right-operand binding is stronger than left-argument so the identity is parsed as monadic:

```
f dop operand+argument
f dop operand,vector
```

Until recently plus `+` as above was common as a separator because its monadic definition was to return the right argument unchanged. But Dyalog 13.0 has complex numbers and monadic plus is now redefined as conjugate for that domain:

```
+ 12J34 'zxc'  →  12J¯34 'zxc'
```

so it is no longer a reliable identity in all cases. Fortunately and in part consequently Dyalog 13.0 also includes the two new identities mentioned above: `⊣` and `⊢`; either of which suffices for our purpose:

```
f op operand⊢argument
f op operand⊣argument
```

Boolean equivalents

We learn that there are ten scalar dyadic boolean functions in APL: `< ≤ = ≥ > ≠ ∨ ∧ ⍱ ⍲`. Each one maps each of the four pairs (0 0)(0 1)(1 0)(1 1) onto one of the pair 0 1 in a different way. So there should be exactly sixteen such functions; there are six others missing from APL. And given that there is that mapping it must be possible to enumerate them in a standard way.

Applying `r ← ,0 1 ∘.f 0 1` gives us the four results. Applying `2⊥r` gives us a number between 0 and 15. Here they are put into the order dictated by that result:

```
.---.--------------.---------.----.
| ∧ | ,0 1 ∘.∧ 0 1 | 0 0 0 1 |  1 |
| > | ,0 1 ∘.> 0 1 | 0 0 1 0 |  2 |
| < | ,0 1 ∘.< 0 1 | 0 1 0 0 |  4 |
| ≠ | ,0 1 ∘.≠ 0 1 | 0 1 1 0 |  6 |
| ∨ | ,0 1 ∘.∨ 0 1 | 0 1 1 1 |  7 |
| ⍱ | ,0 1 ∘.⍱ 0 1 | 1 0 0 0 |  8 |
| = | ,0 1 ∘.= 0 1 | 1 0 0 1 |  9 |
| ≥ | ,0 1 ∘.≥ 0 1 | 1 0 1 1 | 11 |
| ≤ | ,0 1 ∘.≤ 0 1 | 1 1 0 1 | 13 |
| ⍲ | ,0 1 ∘.⍲ 0 1 | 1 1 1 0 | 14 |
'---'--------------'---------'----'
```

Alternatively we can define the operator or adverb:

```
    fnum←{2⊥,0 1 ∘.⍺⍺ 0 1}
    ∧fnum 0
 1
    ∨fnum 0
 7
       ...
```

The six missing would be scalar pervasive equivalents of: 0: `{0}`; 3: `⊣`; 5: `⊢`; 10: `{~⍵}`; 12: `{~⍺}`; and 15: `{1}`.

Given so many boolean dyads it seems odd that much of the time only two of them get used. We've all seen such as:

```
:If p∨~q ⋄ ...
:If (~p)∧q ⋄ ...
```

Applying `fnum` to each of these:

```
    {⍺∨~⍵}fnum''
 11
    {(~⍺)∧⍵}fnum''
 4
```

we find that `{⍺∨~⍵}` is function 11, `≥` while `{(~⍺)∧⍵}` is function 4, `<` so why don't we see:

```
:If p≥q ⋄ ...
:If p<q ⋄ ...
```

for boolean `p` and `q`? I've been told that they're unfamiliar in this context. But so was most of APL to each one of us when we started. Nevertheless many such

equivalences exist. And the reason we don't automatically associate them with the logical domain is because of their common names.

Most people refer to ≥ as *greater than or equal* and some as *not less than*; just two obvious ways to say the same thing. Very few if any call it *or not* or *implied by* but if the comparands are propositions or logicals with boolean values 1 or 0 then it makes complete sense.

Below are some of the more common boolean equivalents many of which I've seen in use. This is not just a matter of terseness versus verbosity. If p and q are large then applying two, three or even four functions to them rather than one is significant.

expression	D-fn	fnum	f	exp
~p^q	{~α^ω}	14	⊼	p⊼q
~p^q	{~α^ω}	8	⊽	p⊽q
p^~q	{α^~ω}	2	>	p>q
p^~q	{α^~ω}	11	≥	p≥q
~p^~q	{~α^~ω}	13	≤	p≤q
~p^~q	{~α^~ω}	4	<	p<q
(~p)^q	{(~α)^ω}	4	<	p<q
(~p)^q	{(~α)^ω}	13	≤	p≤q
(~p)^~q	{(~α)^~ω}	8	⊽	p⊽q
(~p)^~q	{(~α)^~ω}	14	⊼	p⊼q
~(~p)^~q	{~(~α)^~ω}	7	^	p^q
~(~p)^~q	{~(~α)^~ω}	1	^	p^q
(p^q)^~p^q	{(α^ω)^~α^ω}	6	≠	p≠q
(p^q)^~p^q	{(α^ω)^~α^ω}	9	=	p=q

Iteration

It's often necessary to be able to repeat a function sequentially with one argument different but each time using the result of a previous iteration as the other argument. Say we have prototype p and vector v of items to be applied in order a b c d e. Depending which way round our function requires its arguments this will be:

```
((((p flr a)flr b)flr c)flr d)flr e
```

or:

```
e frl(d frl(c frl(b frl(a frl p))))
```

where: `frl ↔ flr⍨`

Given the likely variable length of our vector most will code this as:

```
:For i :In v
      p←p flr i   ⍝ or: p←i frl p
:End
```

but look again at the expression with repeated `frl`. Not everyone will recognise it immediately but this is a precise definition of reduction:

```
⊃frl/e d c b a p
```

The right argument of the reduction is merely `v` reversed and with `p` stuck on the end:

```
⊃frl/(⌽v),⊂p
```

Applying a monad to one of a pair

Given a 2-vector it's required occasionally to keep the first item intact while running a function between the two preferably without having to prise the two apart beforehand. The description might not immediately suggest scan but that's exactly what it requires:

```
f\x y ↔ x(x f y)
f\two ↔ (0⌷two),f/two
```

At least as often I find that what I really want to do is to keep the first while applying a monad to the second. All D-fns are ambivalent. But in this case I want the monadic definition of the function to apply. If the function ignores its left argument then the above is all we need:

```
{2×⍵}\two ↔ (0⌷two),{2×⍵}/two
          ↔ (0⌷two),{2×⍵}1⌷two ⍝ for a de facto monad
```

But if our function is properly ambivalent and works differently in the two cases this will always use the dyad. We need a way to force an ambivalent function to act as a *de facto* monad. Composing it with right `⊢` as `⊢∘{…}` pushes the left argument out to become that of `⊢` which it will ignore. Our function never sees the argument at all:

```
   minus←{⍺←⊢ ⋄ ⍺-⍵}
   minus 5           ⍝ monad: negate
¯5
   8 minus 5         ⍝ dyad: subtract
3
   8 ⊢∘minus 5       ⍝ de facto monad: negate
¯5
   minus\8 5
8 3
   ⊢∘minus\8 5
8 ¯5
```

Afterword

We all have our own favourite tricks and tips. The first version of this was a preamble to a presentation I gave at Dyalog'11 in Boston and this lengthier version is designed to accompany a re-presentation of that paper that appears in the same edition of *Vector* as this. I hope it's been of some use.

References

1. http://dyalog.com/download/dfns.pdf

Sharing code - the APLTree project

by Kai Jaeger (kai@aplteam.com)

The APLTree project is an attempt to make general tools available to the Dyalog programmer. This article discusses the motivation of the project and its details.

Overview

Sharing code is something that never gained popularity in the APL community, and for good reasons: without having a proper tool for modularising code (classes or at the very least namespaces) sharing code is virtually impossible in APL due to the danger of name clashes.

With the introduction of namespaces in Dyalog APL things got better but there were still some obstacles. Namespaces don't allow hiding implementation details: dotting into a namespace means you see everything that's contained in that namespace. In a complex namespace that could mean you see a hundred functions although only, say, 10 of them are the actual "public interface". So in order to use the functionality provided you need to worry about just 10 functions while looking at a hundred.

There is also the problem of how to prevent your dear colleagues from calling functions they are not supposed to call. If like me, you worked on teams who were supposed to follow such rules then you know how difficult a task that is.

Classes to the rescue: one advantage of the object-oriented paradigm is that you can see only the public interface when you dot into a class or an instance of that class. In other words, while namespaces allow you to modularize code and therefore to structure it, classes offer true encapsulation: a programmer only interested in using a class will see just the fields, properties and methods that constitute the public interface.

That does not mean that all the rest which is called implementation details cannot be investigated: You can still edit the class script, which allows you to investigate everything, not only the public interface. The code editor's new tree structure emphasizes whether fields, properties and methods are public or not, and it allows fast navigation, too. Last but not least you still can trace into a class or even an instance, and you can look at everything you want.

Of course it is a big advantage when the implementation details are hidden: it makes using a class much easier. That one cannot call functions which are not supposed to be called from outside is very important, too.

Since the introduction of classes in Dyalog APL Version 11 there is no technical reason why developers should not share tools and utilities. However, this has not become popular so far. One reason is that many APLers still think along the lines: "If I can write it myself in a couple of hours why should I bother to learn the public interface of somebody else's code?" Or in other words: if the amount of time it takes to master somebody else's class is equal to the amount of time it takes to implement it yourself there is no point in sharing code, right?

Even if we ignore the simple fact that programmers tend to underestimate the time it needs to implement something by at least a factor of two this is also a short-sighted view: in total it needs much more time than simply implementing it: you also need to spend time on documentation and, even more important, on test cases. In fact these two tasks quite often take longer than writing the code in the first place.

It is also an important question who is providing the tools. Are they maintained regularly? Are bugs published openly? Can you buy support? These are all questions one might well ask when considering using third-party tools. But when all these questions are answered satisfactorily then you should remind yourself that there is no point in reinventing the wheel.

Documentation

If you are a one-man-band you might be able to abandon the idea of spending time on documentation. You know how to use your tools, don't you? However, things are different in a team. Since software development is normally done in teams these days you can't escape spending time on documentation.

In practice old APL hands sometimes find themselves mixed into a team with other old hands at the start of a project. All APLers have their own set of tools. None is actually documented; they all wrote them for their own purposes. So from the start you have as many toolsets as there are team members, none of them documented. Not a good idea of course.

Test cases

Test cases are considered mandatory nowadays in the IT industry despite the fact that the amount of resources consumed by the implementation of test cases

is significant. Occasionally it exceeds the time spent on the implementation of the software.

Software projects become more and more complex. We know from experience that changing a complex system has all sorts of unforeseen and unwelcomed consequences. All of us have said more than once "I don't understand why it's crashing: I haven't changed anything near this code."

There is only one way to escape this problem: after a change, before making it available to the other team members, let alone the customer, run test cases.

Improved quality

The most important reason for having test cases is that they are designed to improve the quality of any piece of software. This is always an important issue of course, but it is particularly important in projects which evolve dynamically, with a large number of software releases. By changing code you will inevitably break code but if you have a full-blown test suite ready to run against your changes you have a much better chance of discovering these problems.

That alone is a very good reason to spend the resources on implementing test cases. In the long run this will actually save time and money for that very reason.

Test cases as examples and documentation

An implicit advantage of well-written and documented test cases is that they provide examples of how to use the software. That is a real time-saver for anybody supposed to understand and use a class.

This is more important for APLers than for other people. The reason is readability.

As a matter of fact APL is harder to read than code written in other languages. This is mainly owing to the fact that in one line of APL there is much more going on than in one page of COBOL, but it's also due to the usually higher abstraction level.

Now APLers often argue that one can always work out easily what a function is doing by using a debugger and simply watch what the lines are doing with the data. Due to the outstanding debugging facilities of modern APLs this is certainly true. However, by definition it can be true only if correct data is provided to the function in question.

Now there's a problem: when you get a bug report on your plate it's quite likely to be caused by unexpected data, ill-formed data or data the application was

never intended to deal with. With improper data you might have a hard time to find out what the code is supposed to do. It might be very costly or even virtually impossible.

By definition test cases provide valid data, so it is not a big deal to find out what the code is actually doing.

The big problem with written documentation is that nobody debugs it. Implicit documentation gathered from test cases is actually both, debugged and up-to-date. What an asset that is!

The idea of test cases has been around for a long time but only with the Agile Software Development paradigm did the idea really take off. The obvious reason is that in agile projects software is constantly updated in production, often every week or so. Whenever a developer has finished a task she is supposed to check in the updated code into the code repository. Before she is allowed to do so she must run the test cases. Only then does she stand a chance not to break the code.

Conclusion

So, in total, when considering writing a utility which is already available somehow, one has to compare the time it needs to implement the software, write proper documentation, implement appropriate test cases and the time spent on making the software stable and reliable with the time it takes to master the interface of an already existing piece of software.

In total it's fair to say that software coming without a proper documentation and/or without test cases should not even be considered a potential source.

There are no technical reasons any more which prevent Dyalog programmers from sharing code. It makes sense to provide solutions for typical every-day problems APL programmers are likely to face. With the APL Wiki the APL community has the platform to host such tools and utilities.

The APLTree project

The APLTree project was introduced on the APL Wiki in 2009 without any effort to "sell" it: the idea was to get some flesh on the bones first.

APLTree is a pure Dyalog project: The OO features implemented by the different APL vendors are incompatible: something that was written for, say, Visual APL cannot be used in either Dyalog APL or APLX.

In July 2011 the project consisted of 10 utility classes and 4 tools. Note that all members of the APLTree project…

- come without any sort of copyright
- are documented at length
- come with test cases

These are in fact preconditions for becoming a member of the APLTree project. There is a page listing all of them: http://aplwiki.com/CategoryAplTree

There are two different types of tools available:

- Those supporting an APL programmer in the development process
- Those designed to be used within an application

All members of the APLTree project belong to one of these two groups. The page is reflecting this by dividing them into Tools and Utilities.

I. The Tools

The tools are designed to support a programmer during her work somehow, so I am going to give a short overview about what the tools are actually doing.

a) ADOC

ADOC is a self-contained class which extracts information from scripts, compiling an HTML page with all the pieces of information found.

b) APLCode2HTML

APLCode2HTML is a simple tool which takes either a line of APL or the name of an APL function or operator or script and compiles an HTML page in order to display that code.

c) Compare

Compare offers methods designed to compare functions, operators, namespaces and scripts or even workspaces with other functions, operators, namespaces and scripts located either in the workspace or on file. It can also be used as a merge utility.

It makes use of the brilliant third-party-software CompareIt! [1] if available.

d) ScriptManager

ScriptManager deals with scripts only. It allows loading, saving, comparing, editing and updating scripts. ScriptManager can deal with SALTed scripts and supports SubVersion [2] as well as acre [3].

ScriptManager's GUI shows ...

- which scripts are different from their file version

- which scripts point to an invalid location
- which scripts differ from their SubVersion base version
- which scripts have SALT backup files, and how many

The most important actions you can perform from within ScriptManager are:

- Compare a script with the file version with CompareIt!
- Compare a script with any SALT backup files
- Compare a SubVersioned script with its SubVersion base file
- Compare a script managed by acre with the workspace version or an older version in the repository.
- Update all or selected scripts in the workspace with their file version
- Loading and saving scripts
- Delete a script from the workspace and/or from file
- Edit a script
- Manage scripts with the built-in Favorite Manager.

II. The Utilities

All utilities in the APLTree project solve every-day problems almost all APLers come across during their daily work, at least potentially:

- Deal with the Windows Registry with `#.WinReg`
- Deal with files and directories with `#.WinFiles`
- Deal with INI files with `#.IniFiles`
- Get information closely related to Windows with `#.WinSys`
- Start programs with `#.Execute`
- Write log files with `#.Logger`
- Manage key-value pairs with `#.Hash`
- Display compiled help files (*.CHM) with `#.ShowChmHelp`
- Write to the Windows Event Log with `#.WindowsEventLog`
- Create CHM files from APL with `#.APL2XML`; requires the third-party-software "HelpAndManual"[4].

I was asked many times why #.WinFiles was implemented at all; after all we have .NET at our finger tips. Well, create a folder with 90,000 files in it and then try to get a directory listing first with #.WinFiles and then with .NET and you know the answer.

Documentation can be created by the means of ADOC; that makes ADOC the most important of the tools. That's the reason why ADOC has its own article in this issue of Vector.

The code repository

All APLTree projects are saved in two or maybe three different locations. The URL is: http://download.aplwiki.com/APLTree/

This page offers links to three sub pages:

Latest stable version

This is the latest production release. It only contains the script(s).

History

Every version which was once a "Latest stable version" is supposed to be represented here. However, because some members of the APLTree project are older than the project itself they are not represented with all their versions in the "History" folder.

Note that here the full project is saved, including all test cases.

If you want to get hold of the full project, in particular the test cases, this is what you want to download.

Development

As soon as any development is carried out on a certain project it's supposed to be saved in the development folder. Again the full project is saved. Note that only code that has passed the test cases successfully is allowed to be saved. That also means that no matter what you have changed you are supposed to execute the full test suite first.

As soon as a development branch is promoted to the "Latest stable version" folder all files belonging to that project are deleted from the "Development" folder.

Contributions

Any kind of contribution is welcome, even if it's just fixing a typo. However, for the time being it is only possible for a restricted number of administrators to save anything directly in the folders mentioned above. If you want to contribute please download the current branch or, if there is no such branch, the latest file from the "History" folder. When you've made a change somewhere and the test cases still execute fine than send your changes via email to kai@aplteam.com

I would love to get overworked by dealing with a wealth of contributions but right now I can handle it.

References

1. CompareIt! is an outstanding comparison utility. Seehttp://www.grigsoft.com/wincmp3.htm for details

2. SubVersion is a widely used version control system. Seehttp://en.wikipedia.org/wiki/Apache_Subversion for details

3. *acre* is a version control system written in Dyalog for Dyalog by Phil Last (phil.last@ntlworld.com)

4. HelpAndManual is an excellent help and documentation tool. See http://www.ec-software.com/ for details.

Generating documentation with ADOC

by Kai Jaeger (kai@aplteam.com)

One of the important advantages of the object-oriented (OO) paradigm is that implementation details are hidden from the user of a class. All one can see is the public interface. There is a problem as well: without having created an instance you cannot see anything of the public interface that is not shared. For creating the instance you might have to look at the documentation. With ADOC this can be achieved effortlessly.

Dyalog's APL Tools Group recommends the use of ADOC for documenting classes and intends to do so itself as appropriate. Therefore you are likely to find ADOCable information in classes delivered by Dyalog in the future.

Introduction to ADOC

ADOC offers two services:

- Gather all the built-in information available in any class by definition: fields, properties and methods of both types, shared and instance.
- Collect information added by the programmer.

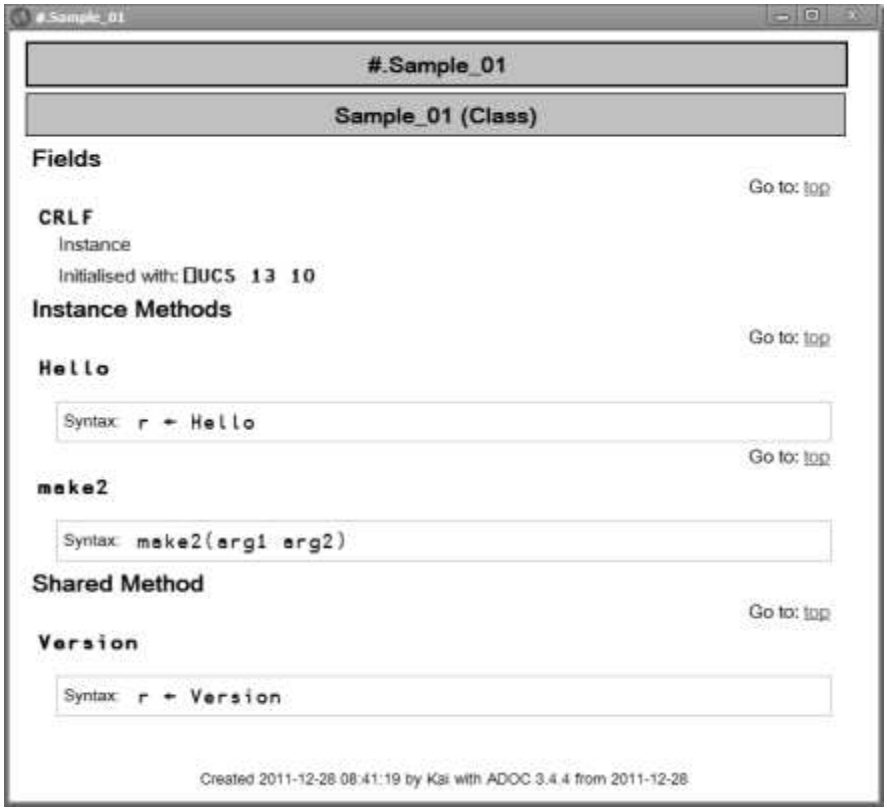
It then compiles an HTML file which not only displays all these pieces of information; it also allows you to print it. Let's take a look at an example:

```
:Class Sample_01
:Field Public Instance CRLF←⎕UCS 13 10
∇ r←Version
  :Access Public Shared
  r←'1.0.0' '2011-09-25'
∇
∇ make2(arg1 arg2)
  :Implements Constructor
  :Access Public Instance
∇
∇ r←Hello
  :Access Public Instance
  r←'World'
∇
Prim←{{⍵/⍨2=+⌿0=⍵∘.|⍵}⍳⍵
:EndClass
```

The script `Sample_01` has one shared and two instance methods. It also has a field of type Instance and one private method. If ADOC is installed as a User Command (discussed in a second) then this statement:

```
]ADOC.Browse Sample_01
```

shows this in your default browser:



As you can see ADOC has gathered all the information provided by the public interface of the class Sample_01. That is certainly useful but ADOC can do much more than that. Let us introduce a namespace script `NS_01`:

```
:Namespace NS_01
∇ r←Hi
  :Access Public Instance
  r←'There'
∇
∇ r←Sum vector
  :Access Public Shared
  r←+/vector
∇
∇ r←a Times b
  r←a×b
∇
:EndNamespace
```

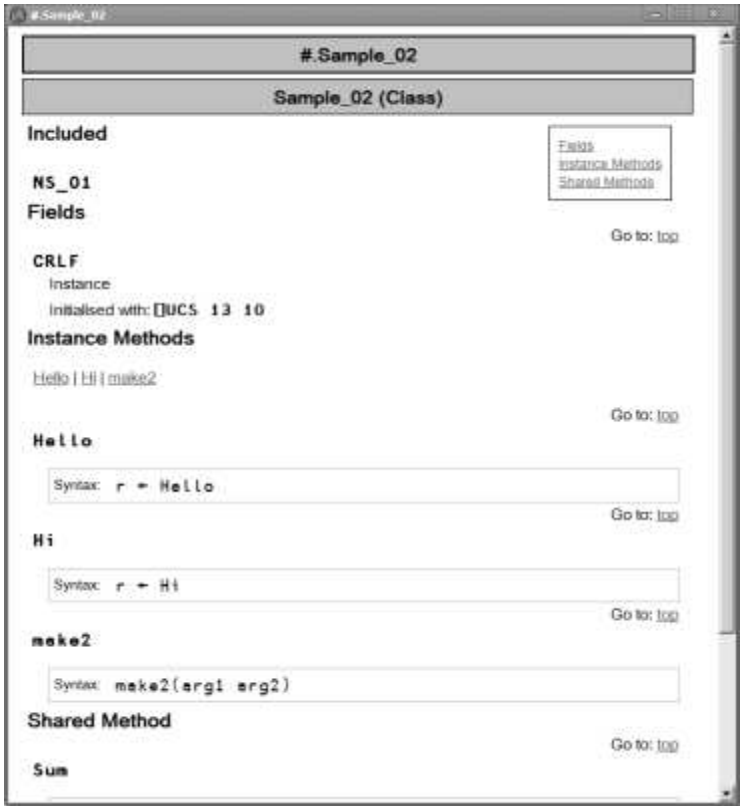
Note that the namespace script comes with one private method, one shared method and one instance method. Now let's introduce a class Sample_02 which is a copy of Sample_01 plus one more statement right after the `:Class` line: it includes the namespace script `NS_01`:

```
:Class Sample_02
:Include NS_01
:Field Public Instance CRLF←⎕UCS 13 10
…
```

Now let's execute:

```
]ADOC.Browse Sample_02
```

Note that ADOC has included the two public methods that come from the included namespace:

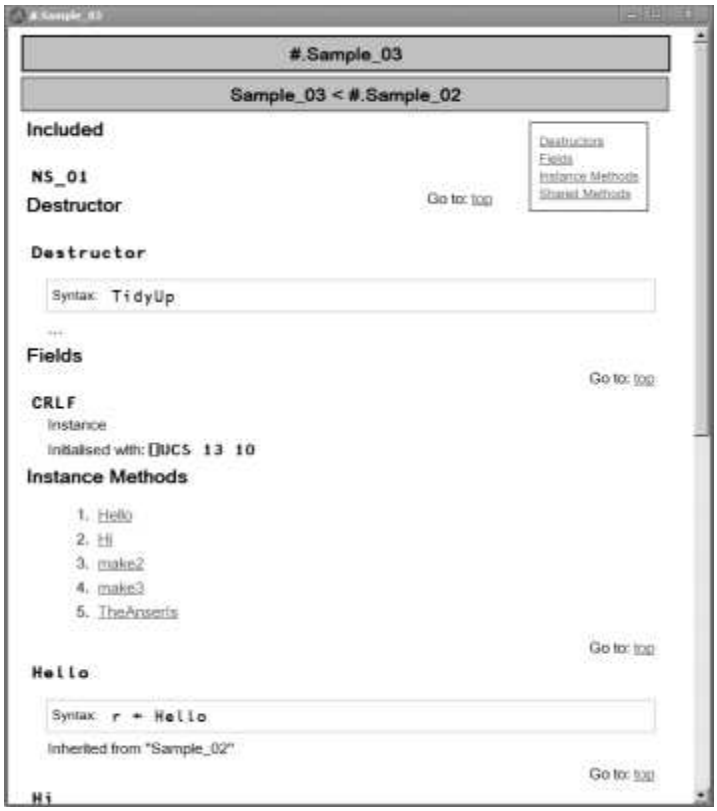


ADOC has also added a table-of-contents (toc) with links to fields, instance methods and shared methods. ADOC not only deals with included namespaces, it can also handle inheritance. In order to prove that let's introduce a class `Sample_03` which inherits from `Sample_02`:

```
:Class Sample_03 : Sample_02
:Include NS_01
:Field Public Instance CRLF←⎕UCS 13 10
∇ r←TheAnswerIs
```

```
    :Access Public Instance
    r←42
  ∇
  ∇ TidyUp
  :Implements Destructor
  ∇
  :EndClass
```

After executing `]ADOC.Browse Sample_03`, this is what the browser shows:



ADOC has added a kind of sub-toc underneath the *Instance methods* heading, providing links to the three methods. Now this may look a little bit over the top right now, but with more methods these links will prove to be useful.

Note that the header declares that `Sample_03` is inheriting from Sample_02. Note also that the method `Hello` is listed as an instance method, together with the information that it was inherited from `Sample_02`.

This is all well and good, but to become really useful it needs more information than just the method signatures.

Adding content (documentation)

Information regarding the type and structure of the arguments a method is expected, and what is returned as a result is naturally something that needs to be

added by a human being. By following a set of simple rules you can make ADOC insert such pieces of information into the HTML page generated by ADOC.

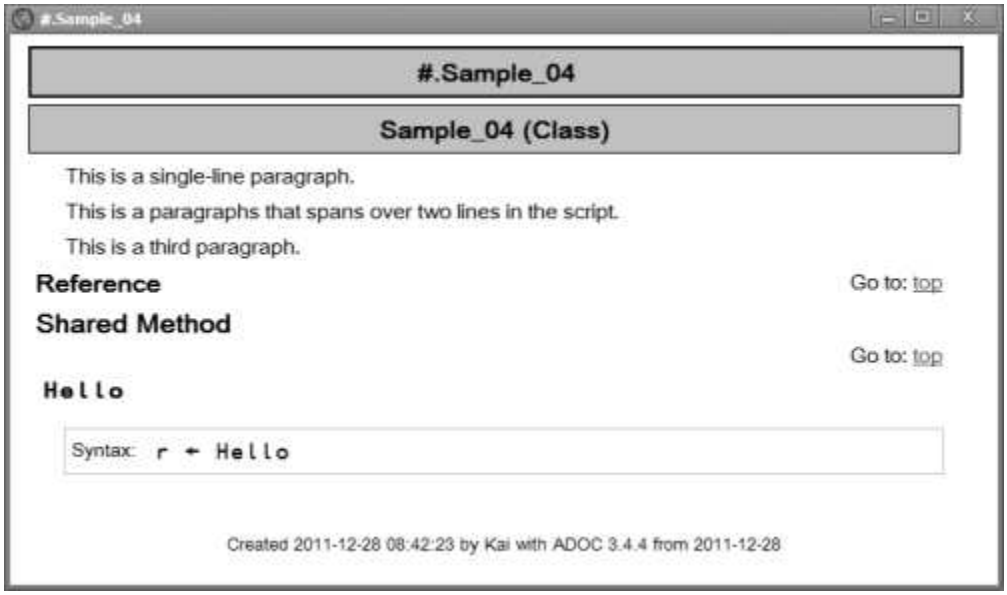
Adding public comments

Let's introduce a class Sample_04 which has just one shared method but a couple of paragraphs:

```
:Class Sample_04
⍝ This is a single-line paragraph.
⍝ This is a paragraphs that spans _
⍝ over two lines in the script.

⍝ This is a third paragraph.
  ∇ r←Hello
  :Access Public Shared
  r←'World'
∇
:EndClass
```

This is what ADOC makes of this:

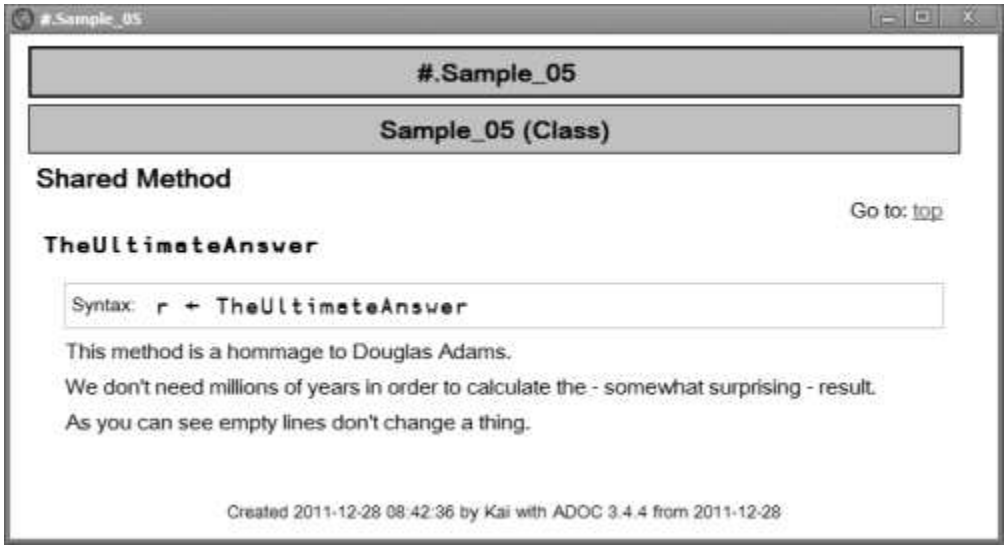


As you can see, a blank followed by an underscore at the end of a line is treated by ADOC as *glue this line together with the next one.* But how does ADOC determine what it should take into the HTML page and what it shouldn't? To find out we process `Sample_05`:

```
:Class Sample_05
∇ r←TheUltimateAnswer
⍝ This method is a homage to Douglas Adams.
  :Access Public Shared
⍝ We don't need millions of years in order to calculate _
⍝ the - somewhat surprising - result.

⍝ As you can see, empty lines don't change a thing.
  r←42
⍝ We simply assign it.
∇
:EndClass
```

And this is what ADOC is making of that:



Note that the `:Access Public` line is ignored, and so are any blank lines. All comment lines until the very first APL statement are processed by ADOC. In ADOC terms they are *public comments*.

Adding Lists

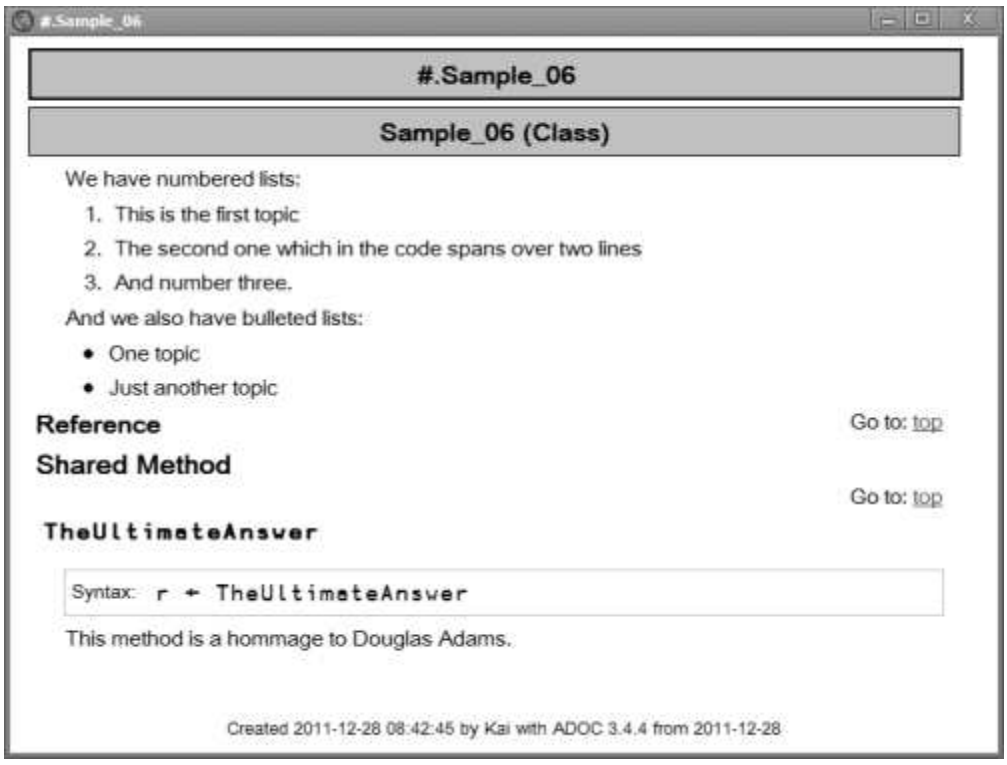
`#` can be used for marking up numbered lists, and `*` for marking up bulleted lists:

```
:Class Sample_06
⍝ We have numbered lists:
⍝ # This is the first topic
⍝ # The second one which in the code spans _
⍝    over two lines
⍝ # And number three.

⍝ And we also have bulleted lists:
⍝ * One topic
⍝ * Just another topic

∇ r←TheUltimateAnswer
⍝ This method is a homage to Douglas Adams.
  :Access Public Shared
  r←42
∇
:EndClass
```

And this is the result:



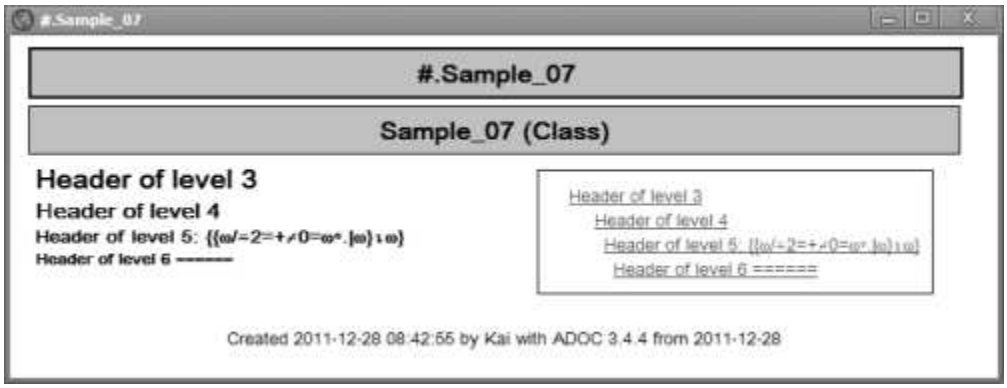
Note that the underscore at the end of a list element is treated the same way as it is within paragraphs.

Adding headers

There are headers available as well:

```
:Class Sample_07
⍝ === Header of level 3
⍝ ==== Header of level 4
⍝ ===== Header of level 5: {{⍵/⍨2=+⌿0=⍵∘.|⍵}⍳⍵}
⍝ ====== Header of level 6 ======
:EndClass
```

Note that the headers only need to be marked up to their left – see level. Level 1 header shouldn't be used: they are reserved for the main header of the document as such. All the entries make it into the table of contents:



Last but not least, investigate the level-5 header. Although it is not set in a monospaced font, it is actually a special version of APL385 Unicode that allows us to display APL characters in a header.

Showing APL characters

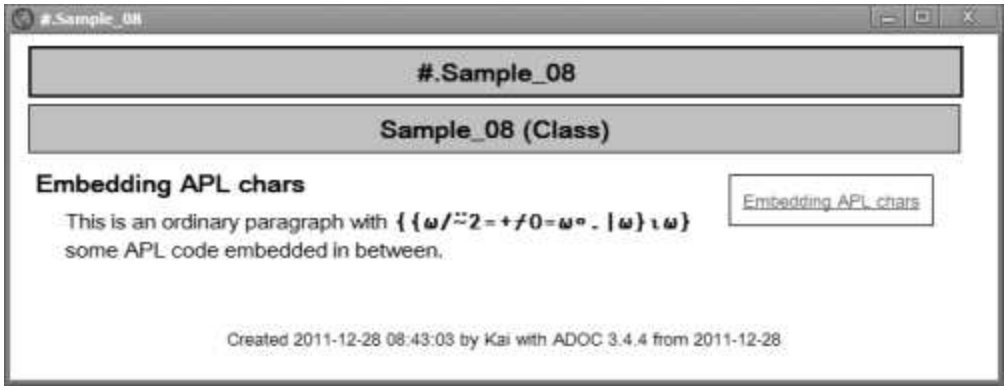
There are two different ways to show APL characters: you either embed APL code in an ordinary paragraph or you create stand-alone APL code, also called a code block.

Embedded APL characters

In order to embed APL characters within an ordinary paragraph, the APL code needs to be enclosed by two double-quotes:

```
:Class Sample_08
⍝ == Embedding APL chars
⍝ This is an ordinary paragraph with ""{{⍵/⍨2=+⌿0=⍵∘.|⍵}⍳⍵}"" _
⍝ some APL code embedded in between.
:EndClass
```

And this is how the result looks like:

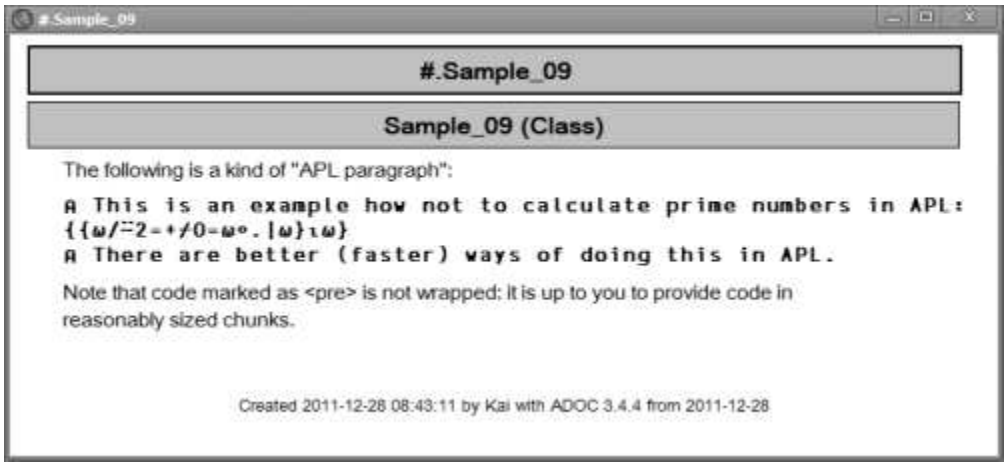


A block of APL code

Not surprisingly, the HTML `pre` tag is used in order to insert a block of APL code:

```
:Class Sample_09
⍝ The following is a kind of "APL paragraph":
⍝ <pre>
⍝ ⍝ This is an example how not to calculate prime numbers in APL:
⍝ {{ω/⍨2=+⌿0=ω∘.|ω}⍳ω}
⍝ ⍝ There are better (faster) ways of doing this in APL.
⍝ </pre>
⍝ Note that code marked as <pre> is not wrapped: it is up to _
⍝ you to provide code in reasonably sized chunks.
:EndClass
```

The result:



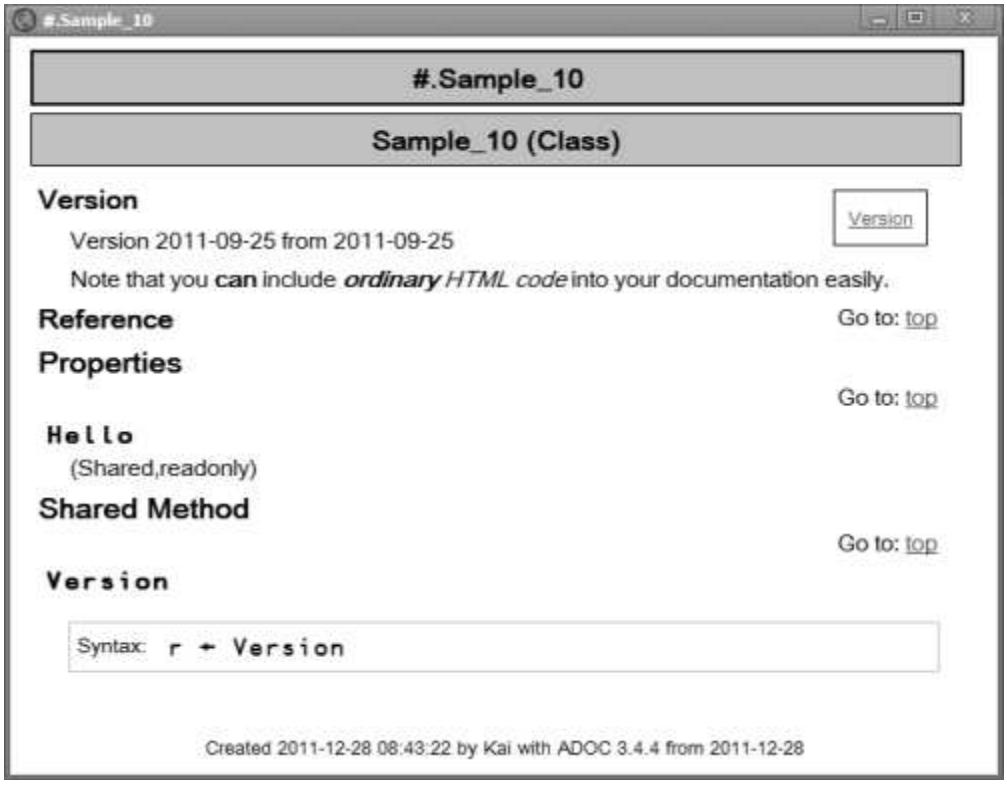
Note that code blocks are not wrapped: it is up to the author to take care of reasonably long lines.

Embedding HTML

Within paragraphs you can embed HTML tags into your documentation:

```
:Class Sample_10
⍝ Note that you can include _
⍝ <i><b>ordinary</b> HTML code</i> _
⍝ into your documentation easily.
:Property Hello
:Access Public Shared
      ∇ r←get
        r←'World'
      ∇
:EndProperty
∇ r←Version
  :Access Public Shared
  r←'1.0.0' '2011-09-25'
∇
:EndClass
```

This has actually the desired effect:



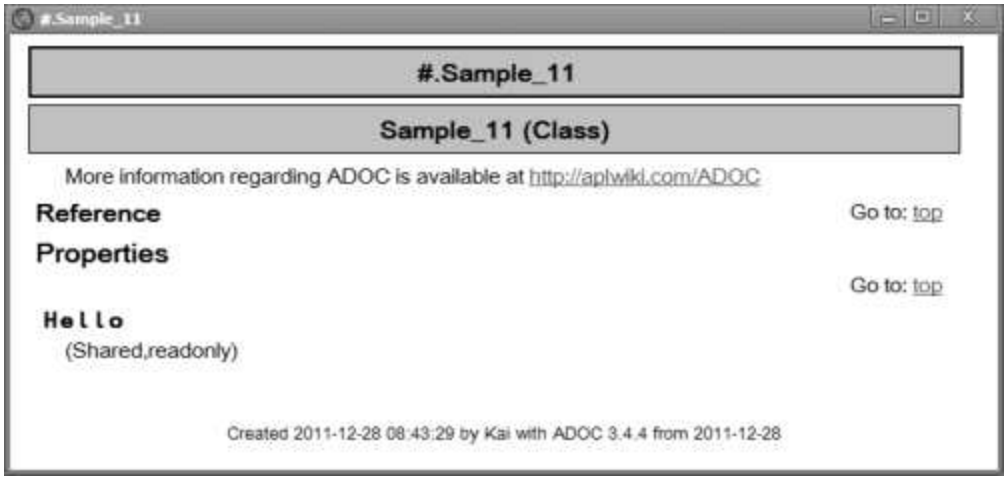
However, this has a drawback if you actually *want* any of the special HTML chars: `<`, `>` or `&`. In other words if you would like them to appear in your documentation you must include them as HTML entities[1] rather than as the characters themselves. Note that this is not true within a block of APL code.

The number of tags you can make use of is naturally limited because most HTML tags are created by ADOC automatically anyway. That leaves `<b>` and `<i>` an `<em>`.

One speciality needs to be mentioned in this context. Look at this class:

```
:Class Sample_11
⍝ More information regarding ADOC is available at _
⍝ http://aplwiki.com/ADOC
:Property Hello
:Access Public Shared
      ∇ r←get
        r←'World'
      ∇
:EndProperty
:EndClass
```

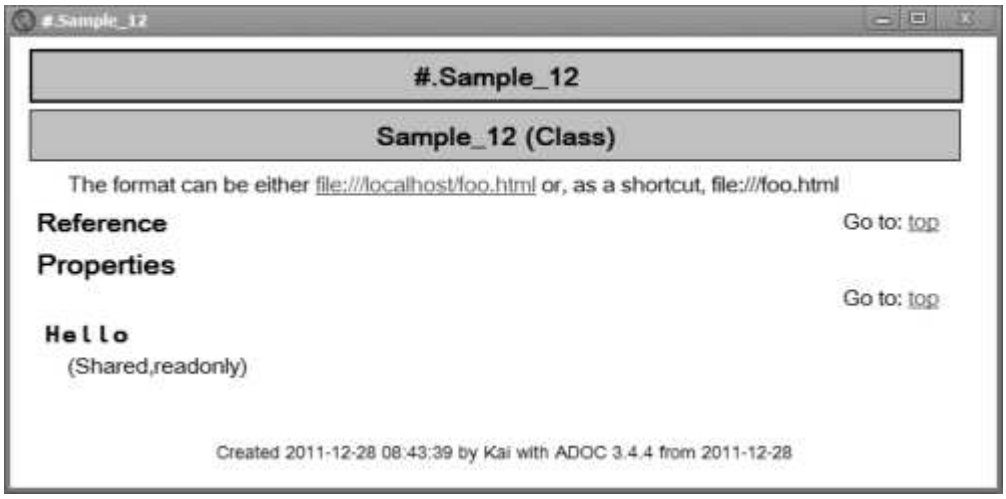
It contains an external link. You don't need to worry about this because this is handled for you:



Note that this is not done by ADOC – it's actually the browser which is adding the link. Sometimes however you don't want a particular string to appear as a hyperlink at all. Look at the following example: the text tries to explain what the `file://` entry means. In this case you don't want this string to be a hyperlink because the link would get you nowhere anyway. The only reasonable way to get around this is to specify the two slashes trailing the word `file:` as `&amp;#47;`.

```
:Class Sample_12
⍝ The format can be either file:///localhost/foo.html or, as a _
⍝ shortcut, file:&amp;#47;&amp;#47;/foo.html
:Property Hello
:Access Public Shared
      ∇ r←get
        r←'World'
      ∇
:EndProperty
:EndClass:EndClass
```

This is the result:

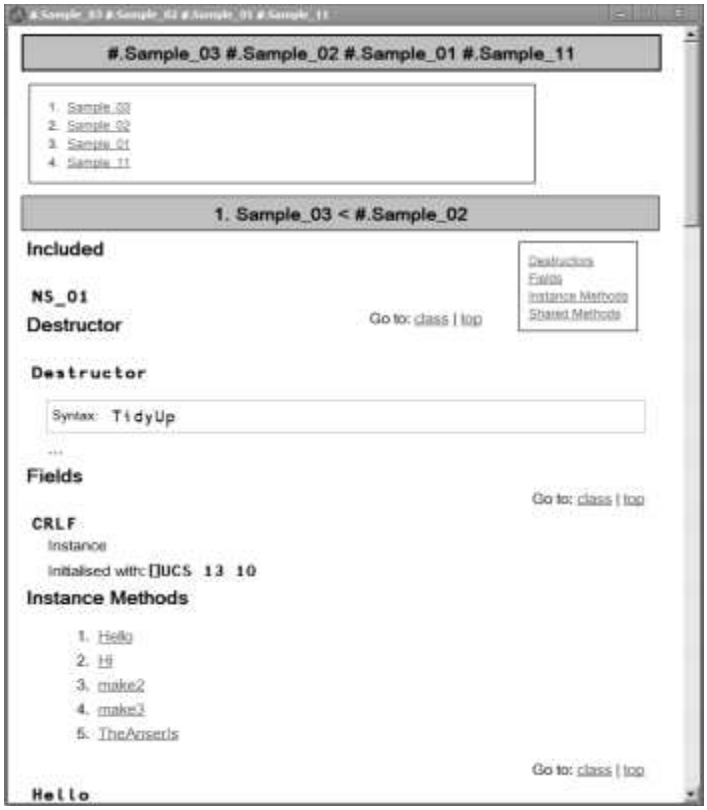


A bunch of classes

If it happens that the complexity of a given project forces you into writing a bunch of classes rather than a single one then most likely we want to generate a document that contains all these classes. You can achieve this by specifying more than one class to the `Browse` method:

```
ref←Sample_03 Sample_02 Sample_01 Sample_11
cs←#.ADOC.CreateBrowseDefaults
cs.Caption←'Complex example'
cs #.ADOC.Browse ref
```

This is the result:



As you can see ADOC has compiled a kind of main table-of-contents listing all the classes involved. There is still a problem: when dealing with a bunch of classes having a reference of some sort for every class involved is not enough: in order to get something done you need to know the workflow. For example, quite often you start with an instance of a certain class, and then you add instances of other classes to properties of this main instance.

To create all-singing, all-dancing documentation you need to add information explaining the workflow, which I like to call 'the big picture'. This can be achieved by adding ordinary functions to the list provided to ADOC via the right argument which contain nothing but comments. If it is just one function at the start of the list that's fine, but you can add such functions pretty much everywhere.

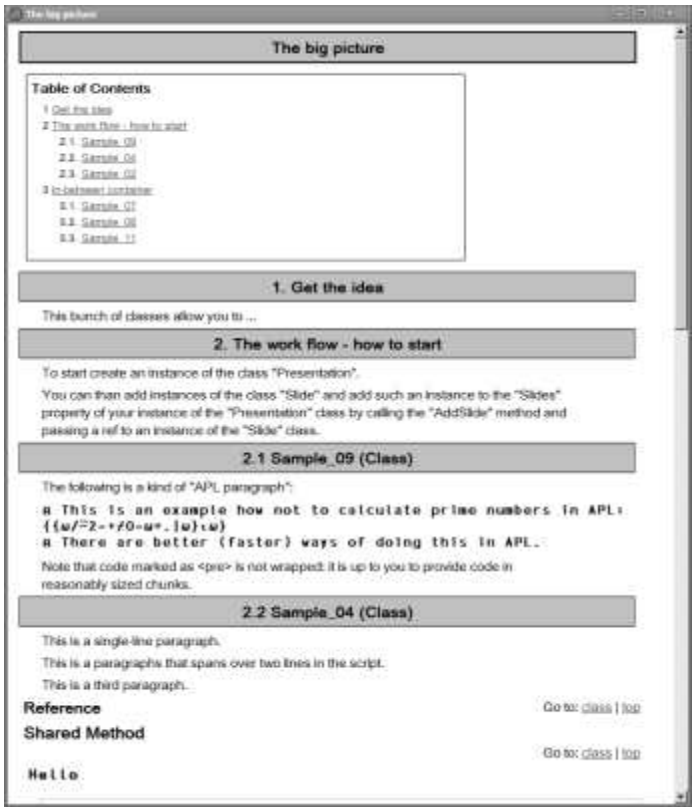
ADOC considers all of them but the first one as containers: all references after such a function name pointing to classes are going to become children of the function in the hierarchy built up by ADOC. That is reflected in the table-of-contents inserted at the top. See this example:

```
l←''
l,←⊂'#.BigPicture'
l,←⊂'#.Workflow'
l,←⊂Sample_09
l,←⊂Sample_04
l,←⊂Sample_02
l,←⊂'#.Container'
l,←⊂Sample_07
l,←⊂Sample_08
l,←⊂Sample_11
cs←#.ADOC.CreateBrowseDefaults
cs.Caption←'The big picture'
cs.withColor←0
cs #.ADOC.Browse l
```

These are the three functions `BigPicture`, `Workflow` and `Container`:

```
∇ BigPicture
⍝ Get the idea
⍝ This bunch of classes allow you to ...
∇
∇ Workflow
⍝ The work flow - how to start
⍝ To start create an instance of the class "Presentation".
⍝ You can than add instances of the class "Slide" and _
⍝ add such an instance to the "Slides" property of your _
⍝ instance of the "Presentation" class by calling the _
⍝ "AddSlide" method and passing a ref to an instance of _
⍝ the "Slide" class.
∇
∇ Container
⍝ In-between container
⍝ This is a kind of 'container'; Sample_07, Sample_08 & _
⍝ Sample_11 are all ...
∇
```

This is the result:



Note that the first line of the three functions is converted into a header. Note also that ADOC restricts you to just one level of nesting: a container 'contains' all the refs until the next container arrives.

The first function (`BigPicture`) stands on its own: by definition it never has children. The second one (`Workflow`) has three children. The remaining scripts then by definition all become children of `Container`.

ADOC as a User Command

ADOC can be made available as a User Command. There are two options available when calling `ADOC.Browse` as a User Command:

```
]ADOC.Browse {refToScript} -caption='My caption'
]ADOC.Browse {refToScript} -browser='C:\Programs\Opera\opera.exe'
```

Of course they also can be specified together.

In order to define a list of scripts note that they must be comma-separated with no blanks in between:

```
]ADOC.Browse ADOC,WinFile
```

This is because a blank is treated as an argument separator.

Conclusion

ADOC is a powerful tool that allows you to create proper documentation on scripts. By following a set of simple rules one can add comments to a script which are extracted and prepared by ADOC appropriately.

The further away documentation is from the actual code the less likely it is to be up-to-date when you look at it, so I strongly recommend keeping it as close to the code as possible. ADOC allows you to do just that.

ADOC is available on the APL Wiki [2].

References

1. www.w3schools.com/html/html_entities.asp
2. aplwiki.com/ADOC

J

Letter

The ruler's edge

by Norman Thomson

It is a pleasure to be able to respond my old partner in education Ray Polivka [1] on the subject of programming rulers. Readers may like to compare the different style of approach which might be taken in J to this problem, which, to remind readers, is to write a program for a ruler of length `y.` with numbers and tick marks at intervals of `x.` .

First pre-define an adverb index

```
   index=.1 : '(i.@$*x.)@]'
```

which identifies elements where a criterion verb such as 'equals zero' is met, for example:

```
   t
0 1 0 1
0 0 1 1
1 0 1 1
   (=&0)index t                    NB. give indices of 0s
0 0 2 0
4 5 0 0
0 9 0 0
```

This can be used in conjunction with the adverb *amend* (`}`) to provide a general purpose replacement facility, for example:

```
   99(=&0)index}t                  NB. replace 0s with 99
99  1 99 1
99 99  1 1
 1 99  1 1
```

This is used in the final line of the verb `numbers`, which supplies the formatted numeric parts of the ruler:

```
   numbers=.4 : 0
t=.x.*>:i.>.y.%x.                  NB. multiples of x.up to y.
t=.|:10 10 #:t                     NB. digitise and transpose
t=.y.{."(1)x.":t                   NB. put in spaced char form
t=.(' '(=&'0')index}{.t),:{:t      NB. blank ldng 0s
)
```

```
   3 numbers 17
           1  1
  3  6  9  2  5
```

Adding the tick marks, both, horizontally or vertically, is now straightforward:

```
   rulera=.4 : 0
t=.x. numbers y.
t,(y.$((<:x.)#0),1){'-^'          NB. add tick marks
)
   rulerc=.4 : 0
t=.x. numbers y.
(|:t),.(0=x.|>:i.y.){'|+'         NB. add tick marks
)
   3 rulera 17
           1  1
  3  6  9  2  5
--^--^--^--^--^--
   3 rulerc 11
  |
  |
 3+
  |
  |
 6+
  |
  |
 9+
  |
  |
```

References

1. “The ruler’s edge revisited”, Ray Polivka, *Vector* Vol.23, No.4
   http://archive.vector.org.uk/art10012030

J-ottings 55

Combination Lists

by Norman Thomson

I enjoyed R.E.Boss's article on lists of combinations [1] because, like him, I have been fascinated both by their patterns and by algorithms which generate them. In both APL and J systematic permutation lists are easier to generate than those for combinations. In J the former are available directly through the primitive `A.` so that

```
   A. 1 2 0
3
```

says that `1 2 0` is permutation `3` (in index origin 0) of `i.3` in lexical order, a process which is reversed by dyadic `A.` :

```
   3 A. 0 1 2
1 2 0
```

A full list of such permutations is given by e.g.

```
   (i.@! A. i.)3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

Analogously with monadic `A.` , any combination of `r` integers from `i.n` can be put into one-to-one correspondence with the counting integers by adding appropriate values of `kCr` ,where the `k`'s are the integers in the combination and `r=1,2,..`, e.g. for the combination `1 3 4`, `1C1 + 3C2 + 4C3 = 8`. Unlike permutations, the value of n need not appear in a combination, and so its number is independent of `n` . The following verb gives unique combination numbers:

```
   ctoi=:monad : '+/(>:i.#y)!y'  NB. combination to integer
   ctoi 1 3 4
8
```

A challenge is to produce itoc such that 8 `itoc 3` is `1 3 4` .

The emphasis in [1] is on the pragmatic matter of how to generate combinations more efficiently using lines such as `[:(,.&.><@:\.)/ >:@-~ [\i.@]` . This on

deconstruction shows that orderly lists of combinations are a little closer to primitives in J than might at first sight be imagined. The key is the combination of box, stitch, infix and suffix, for which a preliminary note on the "fix" family of adverbs is in order, namely prefix (\ monadic), suffix (\. monadic), infix (\ dyadic) and outfix (\. dyadic). These have the general effect of making objects larger either by increasing rank as in

```
   (,\i.5);(,\.i.5)          NB. ravel prefix;suffix
┌──────────────┬──────────┐
│0 0 0 0 0│0 1 2 3 4│
│0 1 0 0 0│1 2 3 4 0│
│0 1 2 0 0│2 3 4 0 0│
│0 1 2 3 0│3 4 0 0 0│
│0 1 2 3 4│4 0 0 0 0│
└──────────┴──────────┘
```

or by repetition with amendment :

```
    <\.i.5                    NB. box suffix
┌─────────┬───────┬─────┬───┬─┐
│0 1 2 3 4│1 2 3 4│2 3 4│3 4│4│
└─────────┴───────┴─────┴───┴─┘
```

The dyadic form infix delivers overlapping x-lists and outfix delivers the result of progressively removing them:

```
   (2,\i.5);(2,\.i.5)
┌─────┬─────┐
│0 1│2 3 4│
│1 2│0 3 4│
│2 3│0 1 4│
│3 4│0 1 2│
└─────┴─────┘
```

An informal rule is that without dots (that is prefix and infix) things proceed from the left, with dots they do so from the right.

Combinations

A deconstruction of comb2 in [1] for the orderly listing of combinations begins with

```
   |:3 ,\i. 6          NB. dyadic infix
0 1 2 3
1 2 3 4
2 3 4 5
```

Apply box-ravel suffix to the final row above:

```
   <@,\.2 3 4 5
┌─────────┬─────┬───┬─┐
│2 3 4 5│3 4 5│4 5│5│
└─────────┴─────┴───┴─┘
```

and then stitch the numbers in the penultimate row on an item by item basis:

```
   Each=.&.>
   1 2 3 4 ,.Each<@,\.2 3 4 5
┌─────┬─────┬─────┬─────┐
│1 2│2 3│3 4│4 5│
│1 3│2 4│3 5│   │
│1 4│2 5│   │   │
│1 5│   │   │   │
└─────┴─────┴─────┴─────┘
```

Call this form of stitching `Stitch` with an upper case `S` to distinguish it from the name of the primitive `stitch` :

```
   Stitch=.,.Each <@;\.
```

so that the preceding display is obtained as

```
   1 2 3 4 Stitch 2 3 4 5
```

By applying this successively to the columns of `3 ,\i.6` , everything is in place to generalize this to a verb for generating combinations of `x` from `y` . (The name `comb2` is chosen because this is the technique described by that name in [1].)

```
   comb2=:dyad : 'z=.Stitch/|:x,\i.y'
   3 comb2 6
┌─────┬─────┬─────┬─────┐
│0 1 2│1 2 3│2 3 4│3 4 5│
│0 1 3│1 2 4│2 3 5│     │
│0 1 4│1 2 5│2 4 5│     │
│0 1 5│1 3 4│     │     │
│0 2 3│1 3 5│     │     │
│0 2 4│1 4 5│     │     │
│0 2 5│     │     │     │
│0 3 4│     │     │     │
│0 3 5│     │     │     │
│0 4 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

A final `;` (raze) could be used to transform the above into normal unboxed lists – retaining the boxes both helps appreciation of the structure, and also reduces the number of print lines required.

The integer representations of the combination list `3 comb2 6` in the order given above are

```
   ;ctoi each<"1 ;3 comb2 6
0 1 4 10 2 5 11 7 13 16 3 6 12 8 14 17 9 15 18 19
```

which shows that combinations generated by `comb2` are not in their 'natural' order.

The boxed result of `comb2` suggests that reduction could equally well have been applied from the right rather than the left by repeated use of the primitive verb reverse:

```
   comb=.dyad : 'z=.|."1 Each Stitch/|.|."1 |: x ,\ i. y'
   3 comb 6
```

3 4 5	2 3 4	1 2 3	0 1 2
2 4 5	1 3 4	0 2 3	
1 4 5	0 3 4	0 1 3	
0 4 5	1 2 4		
2 3 5	0 2 4		
1 3 5	0 1 4		
0 3 5			
1 2 5			
0 2 5			
0 1 5			

As a bonus `comb` delivers the combination list in reverse counting order:

```
    ;ctoi each<"1 ;3 comb 6
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Relationship to the Pascal Triangle

The number of boxes in `r comb n` is one more than `d=.n-r` and the numbers of combinations in the various boxes are directly derivable from the Pascal Triangle whose first few numbers are:

```
   !/~i.10
1 1 1 1 1   1   1   1   1    1
0 1 2 3 4   5   6   7   8    9
0 0 1 3 6  10  15  21  28   36
0 0 0 1 4  10  20  35  56   84
0 0 0 0 1   5  15  35  70  126
0 0 0 0 0   1   6  21  56  126
0 0 0 0 0   0   1   7  28   84
0 0 0 0 0   0   0   1   8   36
```

```
0 0 0 0 0  0  0  0  1   9
0 0 0 0 0  0  0  0  0   1
```

`3 comb2 6` and `3 comb 6` both consist of 20 combinations displayed in 4 boxes, the numbers in which are obtained from the right as the first four non-zero items in row 2 counting in origin 0. More generally `r comb n` has `d+1` boxes in which the numbers of combinations are given by the first `d+1` non-zero integers in row `r-1`. For example for `6 comb 9` read a total of 84 combinations from the Pascal triangle, then go one up along the diagonal and read 1+6+21+56 along the row.

When `r` is large relative to `n` it is more efficient to use complementary combinations as suggested by the symmetry of the Pascal triangle, for example

```
   time=.6!:2
   time Each '2 comb 55';'53 comb2 55'
┌─────────────┬──────────┐
│0.000388038│0.0130343│
└─────────────┴──────────┘
   time '(<i.55)-."1 Each 2 comb 55'
0.00245115
```

Defining an itoc verb

This challenge stated earlier requires the delivery of a unique combination, given an `r` and an integer `i`. If `n` is also given `i` must be in the range `0 .. nCr -1`. To do this, first find the largest `k` such that `kCr` does not exceed `i`. Subtract `kCr` from `i` and carry on repeating this process for `(i- kCr)` and `(r - 1)`. For example, to find combination number 8 of `3 comb n` , `5C3 = 10` which exceeds 8, but `4C3 = 4` does not, so select 4 as rightmost element. `8 - 4 = 4` which exceeds `3C2` , so catenate 3 to the left of 4. `4 - 3 = 1` which is less than `2C1` but equals `1C1` , so that the final digit is 1 and the required combination is 1 3 4. Here is this process expressed in J:

```
   lgstk=.4 :0
i=.<:y                    NB. increase k up to lgst y!k <x
while. x>: y!>:i do. i=.>:i end.
)
   itoc=.4 :0
x=.x-y!r=.x lgstk y
while. y>1 do.
   y=.<:y                 NB. decrement y
   r=.(x lgstk y),r       NB. catenate new value to left of r
   x=.x-y!{.r   end. r    NB. reduce x for next iteration
)
   8 itoc 3
1 3 4
```

Another iterative algorithm

`comb` and `comb2` are not the only methods for constructing combination lists. For example[1] starts by describing another such algorithm, and without worrying too much about the J details, tracing the iterative steps can be used to illustrate how alternative techniques reach the same goal by different routes.

```
   trace=.monad : 'y(1!:2)2'
   combi=.4 : 0                    NB. d is n-r for any given nCr
k=. i.>:d=.y-x                     NB. k is a list of integers
z=.(d$<i.0 0),<i.1 0               NB. initially z is d+1 empty boxes
for_j. i.x do.                     NB. loop thru items of i.x
trace z=. k Stitch >:Each z  end.
)
   3 combi 6
```

```
┌─┬─┬─┬─┐
│0│1│2│3│
└─┴─┴─┴─┘
```

```
┌───┬───┬───┬───┐
│0 1│1 2│2 3│3 4│
│0 2│1 3│2 4│   │
│0 3│1 4│   │   │
│0 4│   │   │   │
└───┴───┴───┴───┘
```

```
┌─────┬─────┬─────┬─────┐
│0 1 2│1 2 3│2 3 4│3 4 5│
│0 1 3│1 2 4│2 3 5│     │
│0 1 4│1 2 5│2 4 5│     │
│0 1 5│1 3 4│     │     │
│0 2 3│1 3 5│     │     │
│0 2 4│1 4 5│     │     │
│0 2 5│     │     │     │
│0 3 4│     │     │     │
│0 3 5│     │     │     │
│0 4 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

```
┌─────┬─────┬─────┬─────┐
│0 1 2│1 2 3│2 3 4│3 4 5│
│0 1 3│1 2 4│2 3 5│     │
│0 1 4│1 2 5│2 4 5│     │
│0 1 5│1 3 4│     │     │
│0 2 3│1 3 5│     │     │
│0 2 4│1 4 5│     │     │
│0 2 5│     │     │     │
│0 3 4│     │     │     │
│0 3 5│     │     │     │
│0 4 5│     │     │     │
└─────┴─────┴─────┴─────┘
```

Other methods are described in [2]. The relative efficiencies of algorithms are parameter dependent. These can be tested informally by e.g.

```
   time Each '9 comb 23';'9 comb2 23';'9 combi 23'
┌────────┬────────┬────────┐
│0.251543│0.173648│0.199808│
└────────┴────────┴────────┘
(n.b.  combi had the trace removed for fair comparison)
```

References

1. R.E.Boss, Vector Vol.24 Nos. 2 &3, pp. 75-88 Generating Combinations in J efficiently.

2. Norman Thomson , Vector Vol. 22 No.4, pp. 99-105, J-ott

Bicubic Interpolation in J

by Cliff Reiter (reiterc@lafayette.edu)

Prolog

In 2008 there was a query on the J Programming forum about whether bicubic interpolation had been implemented in J[4]. This is a standard technique for resizing images. I was doubtful that there were many situations for bicubic interpolation to be noticeably better than resizing by sampling pixels. However, when I created the hiking guide[6] as a PDF, I found the palette based topographic maps were very washed out after conversion until I selected the bicubic interpolation option. Perhaps more is going on in creating the PDF, but I put taking a closer look at bicubic image interpolation onto my "to do" list. Recently I implemented Keys[3] convolution bicubic algorithm. In this note I share the implementation and some experiments. A script with the functions defined here may be found at[7]. Other versions of bicubic interpolation exist[1, 5].

Bicubic Interpolation

Bicubic interpolation uses four-by-four patches of equally spaced discrete data points to obtain cubic polynomials in two variables that can be used to approximate data that falls within the interior two-by-two patch. This is done independently in the two image directions and for colour planes in the case of RGB images. Thus, we begin by considering the one dimensional version.

Keys type interpolation can be described in terms of matrix multiplication on the four data (pixel) values by the matrix `acon` shown below. The entries in the matrix are derived by Keys[3]. The entries do not arise from interpolating all the points. The coefficients arise from interpolating the central pair and using some high order derivative approximations and boundary conditions. We refer the reader interested in the details to Keys[3]. It is convenient to think of the independent points where the data is known as `_1 0 1 2`. Multiplying the values at those four points by `acon` results in the coefficients of Keys cubic polynomial. It is designed to approximate the data on the interval from `0` to `1`. At the end points it gives the appropriate data values.

For example, if we want to use a Keys type interpolation of the data that has values `2 3 5 7` at the points `_1 0 1 2` respectively, we can do that as shown below. We see that it exactly returns correct values at `0` and `1` and a cubic interpolation of `3.9375` at `0.5`.

```
   acon
   0    1    0    0
_0.5    0  0.5    0
   1 _2.5    2 _0.5
_0.5  1.5 _1.5  0.5

   mp=: +/ . *

   acon mp 2 3 5 7
3 1.5 1 _0.5

   (0 0.5 1^/i.4) mp acon mp 2 3 5 7
3 3.9375 5
```

A two dimensional version of this interpolation is given by the verb `bicuev` below. Consider the four-by-four patch, `p`, of primes given below and interpolation in two variables. Note that the input values are between zero and one and matrix oriented coordinates are used with the origin being at the upper left of the two-by-two central patch. Thus evaluating at `0 0` gives `13` and `0.1 0.9`, which is near `0 1`, gives a value near `17`.

```
   bicuev=: 4 : 0
'X Y'=.y^"0 1 i.4
Y mp"2 acon mp X mp"3 acon mp"3 x
)

   ]p=:p: i.4 4
 2  3  5  7
11 13 17 19
23 29 31 37
41 43 47 53

   p bicuev 0 0
13
   p bicuev 1 0
29
   p bicuev 1 1
31
   p bicuev 0.1 0.9
17.9766
```

Resizing Images

Before discussing bicubic interpolation on a larger array, we consider the resizing verb from the image3 add-on shown below. That verb uses subsampling or resampling as necessary to obtain the resized image. The local variable `szi` gives the size (height-width) of the input image while `szo` gives the size of the

output image which is the largest image that fits inside a window specified by the left argument while respecting the aspect ratio of the input image. We cheat a bit displaying `ind` since it is actually a local variable, but we see the indices are repeated (or not) in a smooth way obtaining an array of the desired size by over (or under) sampling the rows and columns of the array as necessary.

```
   resize_image=: 4 : 0
szi=.2{.$y
szo=.<.szi*<./(|.x)%szi
ind=.(<"0 szi%szo) <.@*&.> <@i."0 szo
(< ind){y
)

   6 6 resize_image p
 2  2  3  5  5  7
 2  2  3  5  5  7
11 11 13 17 17 19
23 23 29 31 31 37
23 23 29 31 31 37
41 41 43 47 47 53

   ind
+-----------+-----------+
|0 0 1 2 2 3|0 0 1 2 2 3|
+-----------+-----------+
```

The verb `bicubic_resize_image`, shown below is similar in many regards. However, the floating point arithmetic used in the interpolation uses substantial space, so that is mitigated by updating one row at a time in the array `z`. Also, a local function `get_patch` is defined to facilitate obtaining the four-by-four patches needed for the interpolation. Due to the patch sizes of four-by-four we need to enlarge the input array by increasing the number of rows and columns by `3`. We do that by constant extensions: by two copies on the leading edges and one on the trailing edges. We again cheat and display some local variables. Below is the result of applying the boundary conditions. The last value computed depends on the indices `3.5 3.5`. Thus, the last patch used is determined by `3 3` and that patch is used to obtain the bicubic interpolation at `0.5 0.5`. The result of that interpolation is given. One might not prefer these boundary conditions, but they are fairly simple and appear reasonable in practice for image data. Also, for image data using a scale of `0` to `255`, we will want to clamp (or round) the data into the desired range.

```
   conext0=:{.,{.,],{:

   conext3=:conext0"_1@:conext0

   clamp=:0>.255<.<.

   bicubic_resize_image=:4 : 0
szi=.2{.$y
szo=.<.szi*<./(|.x)%szi
get_patch=.(( [:<((i.4)+{.);(i.4)+{:)) { (conext3 y)"_
'indj indk'=:(>:(<:{.szi)*(i.%[){.szo);>:(<:{:szi)*(i.%[){:szo
z=.(szo,2}.$y)$0
for_j. i.#indj do.
  inds=.(j{indj),.indk
  as=.get_patch"1 <.inds
  t=.clamp as bicuev"_1 1 ]1&| inds
  z=.t j}z
  end.
z
)

   6 6 bicubic_resize_image p
 2  2  3  3  5  6
 5  6  7  8 10 11
11 11 13 15 17 18
16 18 20 22 23 25
23 25 29 30 31 34
32 34 37 38 39 43

   conext3 p
 2  2  2  3  5  7  7
 2  2  2  3  5  7  7
 2  2  2  3  5  7  7
11 11 11 13 17 19 19
23 23 23 29 31 37 37
41 41 41 43 47 53 53
41 41 41 43 47 53 53

   indj
1 1.5 2 2.5 3 3.5

   indk
1 1.5 2 2.5 3 3.5
```

```
   get_patch <.3.5 3.5
13 17 19 19
29 31 37 37
43 47 53 53
43 47 53 53

   (get_patch <.3.5 3.5) bicuev 1|3.5 3.5
43.1797
```

Image Experiments

First we apply bicubic interpolation to a tiny randomly chosen four colour image.

```
   $b=:(?.5 8$4){?.4 3$255
5 8 3

   view_image 720 720 resize_image b
720 450

   view_image 720 720 bicubic_resize_image b
720 450
```

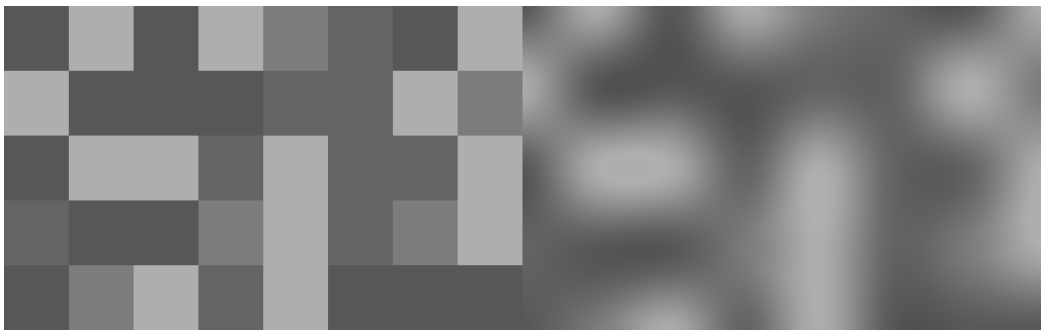


Figure 1. Sampling and Bicubic Interpolation of a Random Image

In the bicubic image we see that the "super" pixels blend into one another in a fairly natural way and that the blending near the boundaries is visually unbiased with respect to which edge is chosen.

As a second example we consider a thumbnail sized zoom into an image of Ken Iverson at Kiln farm from the image3 addon that is expanded to create a web sized image.

```
   B=:read_image jpath,'~addons/media/image3/atkiln.jpg'

   view_image B
468 700

   ken=:100 100{.120 210}.B
```

```
   view_image 720 720 resize_image ken
720 720

   view_image 720 720 bicubic_resize_image ken
720 720
```

The original image quality is poor, but the contrast between the pixilation of sampling and the smoother bicubic interpolation can be observed in Figure 2.

Figure 2. Sampling and Bicubic Interpolation of an Image Piece

References

1. Bicubic Interpolation, Wikipedia, http://en.wkipedia.org/wiki/Bicubic_Interpolation, 2011.
2. Jsoftware, J6.01c, with Image3 addon, http://www.jsoftware.com, 2007.
3. Robert G. Keys, Cubic Convolution Interpolation for Digital Image Processing, *IEEE Transactions on Acoustics, Speech and Signal Processing*, , 29 6 (1981) 1153-1160.
4. David Porter, Jprogramming forum, Bicubic Image Smoothing, January 24, 2008.
5. William H. Press et al, *Numerical recipes: the Art of Scientific Computing*, 3rd edition, Section 3.3.6, 2007.
6. Cliff Reiter, *Witness the Forever Wild, A Guide to Favorite Hikes around the Adirondack High Peaks*,http://webbox.lafayette.edu/~reiterc/wfw/index.html, Lulu.com, 2008.
7. Cliff Reiter, bicubic_keys script, http://webbox.lafayette.edu/~reiterc/j/vector/index.html, 2011.

O T H E R S

5 – a stack-based array language

by Bernd Ulmann (ulmann@vaxman.de)

5 is a portable and extensible stack-oriented array language that combines the features of APL and Forth and some ideas from Perl to yield a highly interactive environment for the professional developer as well as students of computer science. The interpreter is very lightweight (about 650 Kb all in all, including documentation) and runs readily on a variety of operating systems including Windows, LINUX, Mac OS X and even OpenVMS (VAX, Alpha, Itanium). 5 is Open Source and resides on SourceForge[1].

What is 5 and why was it developed?

The development of 5 started in August 2009 during a boring train ride when I programmed my beloved HP48GX pocket calculator to kill some time. As much as I love the stack-oriented approach of HP's calculator-language RPL, I always thought that one could have done better by not combining Forth and LISP but Forth and APL instead. I wondered what such a language inheriting the main ideas of APL and Forth (and some bits from Perl) would look like and started writing a simple interpreter as a proof of concept. Since this new language looked like 'Forth on steroids' the name 5 seemed quite natural (quickly abandoning the idea of 'Fifth').

It turned out that a stack-based array language yielded very concise code and during the next six months the language was steadily extended until it turned out that this first interpreter was indeed too limited in its design to incorporate all of the new ideas that popped up. Thus Mr Thomas Kratz and I decided to restart from scratch and implement a more flexible interpreter. (Most of the interpreter has been written since by Mr Kratz, whom I would like to thank for this truly great work).

First steps

The 5 interpreter is quite mighty and thus the following sections can and will only give a brief overview by showing and explaining a couple of typical 5-programs – much more information can be found in the introductory manual[2].

Getting and installing 5

The 5 interpreter and its accompanying documentation can be downloaded from SourceForge. The only prerequisite for installing 5 is a Perl interpreter, which is already found on most UNIX systems and easily installed on Windows machines (ActiveState Perl works fine). Since the 5 interpreter does not need any special

Perl-packages, its installation does not require any changes on an existing Perl installation. To install 5 all you need to do is to unzip the distribution kit, which is only about 330 Kb in size to any suitable location and make the file `5` executable. On UNIX systems you might want to extend your environment variable PATH by the directory into which you unzipped the distribution kit. On Windows and OpenVMS systems the interpreter can be started most easily from the command line by typing `perl 5` (although one would normally define a foreign command on OpenVMS to make 5 directly callable).

The distribution kit not only contains the interpreter itself but also detailed documentation, as well as many examples which will help one get used to the language and the interpreter.

Using 5 interactively

Let us use the 5 interpreter interactively to simulate dice being thrown 100 times and computing the arithmetic mean of the outcomes:

```
6 100 reshape ? int 1 + '+ reduce 100 / .
```

There is not much to say about this simple program which would not be obvious to people inclined to work in APL and other array languages. The most noteworthy thing is the stack-oriented operation of the interpreter, so all programs are read strictly from left to right. (As a result there are neither parentheses nor operator precedence rules in 5.)

1. First two scalars, `6` and `100`, are pushed onto the stack.
2. These values are used for the operator `reshape` which removes both from the stack and pushes an array containing 100 elements back onto the stack: `[6 6 6 … 6]`
3. To this array, the unary `?` operator is applied, which generates pseudo-random numbers in a range between 0 and a given maximum value (exclusively). Since this operator is unary it is applied to all elements of the vector automatically and yields a new vector on the stack containing 100 elements between 0 (inclusively) and 6 (exclusively). Applying the unary `int` operator to this vector yields a vector of integer values which are then incremented all by one due to `1 +` .
4. The next step pushes the name of the binary addition operator onto the stack, `'+` , and applies the `reduce` function, which in turn sums all vector elements into a scalar.
5. Dividing this result by 100 and printing it to the console is accomplished by `1 100 / .`

Although one has to get used to the reverse-Polish notation style of 5 it turns out to be very efficient and intuitive after a short time of playing with the interpreter. The main advantage of the stack-based nature is that one can build complex expressions iteratively and 'watch' the results of a computation through the different stages step by step. (Using the word `.s` one can generate a pretty printed view of the stack without destroying its contents, which is quite handy for understanding the actions of a program.)

More complex examples

The following examples introduce some of the more sophisticated features of 5 – due to the complexity of the language many concepts will be mentioned only briefly – a comprehensive description of the language and its many operators and functions can be found in the documentation (see above).

Generating a list of primes

One of the archetypical examples found in nearly every introductory text for APL is the generation of a list of primes without any explicit loops or the like. The following program shows how this is done in 5:

```
: prime_list
1 - iota 2 + dup dup dup
'* outer swap in not select
;
100 prime_list .
```

This example is much more complex than the one before and could be run from a file using 5 in batch mode. To accomplish this just call the 5 interpreter with the name of the source code file as a command line parameter. (5 also supports quite a lot of qualifiers to get statistical information about program runs and the like, which are described in the documentation.) Assuming that there is a file named `prime.5` containing the code shown above, it can be run by typing `5 prime.5` or `perl 5 prime.5`.

This example introduces the concept of so called 'user defined words' ('UDW' or just 'word' for short) that effectively extend the language itself and can be used in exactly the same way as built-in functions and unary or binary operators (in that respect they are much more powerful than the traditional words of Forth which have no provisions for acting as unary or binary operators extending their usability to nested data structures).

First a word named `prime_list` is created – the colon starts a word definition that ends with a semicolon. This user-defined word is neither unary nor binary,

so it just sees the stack as it is and operates on it. (In contrast to that, unary and binary words get a local stack with only one or two elements on it to operate on when they are being called. When such a unary or binary word terminates only the top most stack element of its local stack is copied back to the main stack of 5 thus unary or binary operators are side-effect free with regard to the main stack.)

The basic idea of generating a list of primes up to some value `n` is to generate two vectors `[2 3 4 … n]` and generate a matrix by applying an outer product operator to these two vectors. Since this matrix obviously contains only non-primes, it can be used to select all primes from a copy of such a vector. The vector itself is generated by `1 - iota 2 +` which expects a number like `100` on the stack: Subtracting one yields `99`, applying `iota` yields a vector `[0 1 2 … 98]`, adding two to this vector yields the desired vector `[2 3 4 … 100]`. The command sequence `dup dup dup` creates three copies of this vector which will be needed soon.

In the next step the name of the multiplication operator, `'*` , is pushed onto the stack and `outer` is called, which expects an operator's name (`*`) and two vectors on the stack and creates a matrix as the result of an outer product in this case.

`swap` swaps the two topmost stack elements, so now one of the remaining copies of the vector is on top and the matrix is the second element from top. Applying the `in` function generates a vector containing a `1` in every place corresponding to an element of the vector that exists in the matrix and a `0` otherwise. Inverting this vector with `not` yields a selection vector which is then applied to the last copy of the original vector by `select`. This yields a vector containing prime numbers between `2` and `n` only.

The main program only consists of `100 prime_list.` This places the value `100` onto the stack, calls the word `prime_list` and prints the resulting vector using the dot.

Sum of cubes

The following two-liner computes all natural numbers less than 1000 that equal the sum of the cubes of their digits:

```
: cube_sum(*) "" split 3 ** '+ reduce ;
999 iota 1 + dup dup cube_sum == select .
```

The first line again defines a word but this time it is a unary word – denoted by `(*)` following the name of the word. This has the effect that this word will not only work on scalar values but will be automatically applied to all elements of

nested data structures. (So applying this word to a vector like `[1 2 3]` will implicitly and automatically apply it to the three vector elements `1`, `2` and `3` and return another three element vector containing the particular results.) The star denotes that the type of the argument is not relevant (a later example will show the ability of 5 to 'dress' data structures – the 5 way of overloading operators etc.).

What does the word `cube_sum` do? First of all it pushes an empty string onto the stack and calls the `split`-function. This function expects a regular expression on the stack and splits the scalar found below on every place where this expression matches. Since the expression is an empty string in this case, it will perform a split after each character of a value. The nice thing is that this naturally extends to numerical values, too – if there was the value `123` on the top of the stack prior to performing `"" split` the result of this operation would be a vector `[1 2 3]`.

This vector is then cubed element wise by `3 **` and summed (element wise) yielding a scalar value by `'+ reduce`, so the word `cube_sum` expects a value on the stack and returns the sum of the cubes of its digits.

The main program is equally simple: First a vector running from `1` to `999` is generated by `999 iota 1 +` . This vector is then copied two times with `dup dup` before `cube_sum` is applied. Since `cube_sum` is a unary word, it will be applied in an element wise fashion to the elements of this vector and yields another vector with 999 elements which are the sums of the cubes of the digits of the numbers of the original value. This cube-sum-vector is then compared element wise with one of the copies made before, which yields another vector with 999 elements being `1` or `0` reflecting the result of the comparison operator. This vector is in turn used to `select` only those elements from the last copy of the original vector that equal their digit-cube-sum, which is then printed with the dot.

Dressed data structures

If that were about all that 5 can do, it would not be too worthwhile but there is more: 5 allows one to "dress" data structures – i.e. mark some data as being of a certain type like a complex number, a quaternion, a matrix, whatever. The following example shows how to use this feature in the generation of a Mandelbrot set:

```
: d2c(*,*) 2 compress 'c dress ;
: iterate(c) [0 0](c) "dup * over +" steps reshape execute ;
: print_line(*) "#*+-. " "" split swap subscript "" join . "\n" . ;
75 iota 45 - 20 /
29 iota 14 - 10 /
'd2c outer
10 'steps set
iterate abs int 5 min 'print_line apply
```

What is a Mandelbrot set anyhow? It is the result of applying an iterative calculation to points of the complex plane, so first of all we will need a matrix of complex numbers. The 5 interpreter has no idea what a complex number might be but it is easy to extend the language by overloading operators to handle data dressed in a special way. So the basic arithmetic operators are already overloaded in `mathlib.5` to handle complex numbers, which are dressed by the letter `c`. This 'dress code' is just a convention – one could have chosen anything but in order to keep 5 code short and concise, a single letter was chosen to denote complex numbers (`m` denotes a matrix, `v` a vector and `p` a polar coordinate).

Generating a matrix from two vectors by creating an outer product was already shown in the prime number example above. We will use this technique to generate a matrix consisting of complex numbers. Therefore we need a binary word which takes two scalar values and returns a complex number made from these two values. This word is called `d2c` in the code shown above, short for "dupel to complex". Since the name of the word is followed by `(*,*)` it is a binary word which does not care about the type of its arguments. All that it does is to compress the two values found on its local stack by `2 compress` into a simple two-element vector. This vector is then dressed by `'c dress` to form a complex number. Let us assume that `d2c` is called with `1 2 d2c`, so it finds the values `1` and `2` on its local stack. Executing `2 compress` yields the vector `[1 2]` which is then dressed to return `[1 2](c)` – a complex number.

To see how this word is used, let us look at the three lines in the middle of the program: `75 iota 45 - 20 /` generates a vector `[-2.25 -2.2 -2.15 … 1.3 1.35 1.4 1.45]` while `29 iota 14 - 10 /` yields `[-1.4 -1.3 … 1.3 1.4]` respectively. These two vectors are then combined into a matrix by using `d2c` as the binary operator for the `outer`-function. The result of this is a two dimensional matrix of complex numbers – the basis of our Mandelbrot set.

Now that we have a complex matrix, a unary word is needed that operates on complex numbers and performs the necessary iteration for a Mandelbrot set.

This iteration has the form $z_{i+1}=z_i^2+c$, where c is a point of the complex plane with $z_0=0$. If this series is non-divergent, the point c belongs to the Mandelbrot set. In the program shown above this iterative formula is applied 10 times and the resulting value is used to choose a display character for the point c in question.

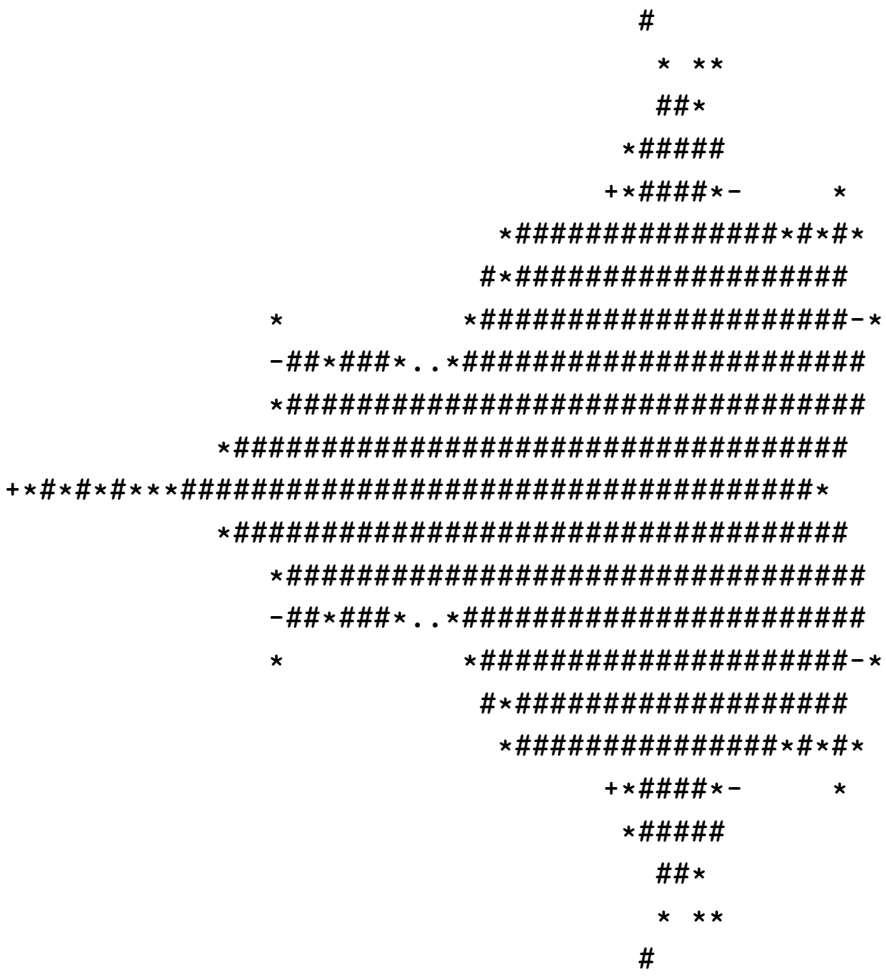
To compute this iterative formula, the word `iterate` is defined, which is a unary operator expecting a complex number which is denoted by the start of the word definition: `: iterate(c)`. In a first step this word pushes the complex number `[0 0](c)` onto the stack which serves as z_0. A single iteration step can now be performed by the instruction sequence `dup * over +`. The function `dup` makes a copy of the value on the top of the stack, `*` multiplies the two topmost stack elements, effectively computing z_i^2 while `over` fetches the element below the top of stack, which is c, so z_i^2+c is computed with `+`.

To perform a given number of iterations a sequence of these steps must be generated. A traditional language like C or Java would need an explicit loop for this, but array languages like APL or 5 have the means to express this much more elegantly. (“Look Ma – no loops!”) The main program sets a variable named `steps` to `10` which is used in this word to control the number of iteration steps being performed. The result of `"dup * over +" steps reshape` in this case yields a one-dimensional vector with 10 elements, looking like this (effectively unrolling the loop): `["dup * over +" … "dup * over +"]`. This vector is then used as an instruction stream by means of the `execute` function, which effectively computes the iterative sequence desired.

The main program now calls `iterate`, which is implicitly applied to all elements of the complex matrix built before. The result of this is another complex matrix, which is transformed into a matrix of simple scalars by applying the `abs`-operator to it (`abs` has been already overloaded in `mathlib.5` to work on complex numbers and returns simple floats). The resulting elements are then capped by `5 min` and then another unary user defined word, `print_line`, is applied in a row-like fashion to the matrix by using it as an argument to the `apply`-function.

`print_line` is now called for every row of the matrix. It first generates a vector `["#" "*" "+" "-" "." " "]` by `split`ting the string `"#*+-. "` on an empty regular expression and then uses the elements of the line vector, which are integers between `0` and `5`, as an index into this character vector. The result is a vector containing as many characters as the line contained integer values. This vector is then concatenated into a simple string b `join`ing it with an empty

string. This resulting string is then printed with a newline character appended. The resulting picture looks like this:



Conclusion

Although there is much more to say about 5, I hope that the few examples given above made you curious about this language and I would like to refer you to the extensive documentation which comes with the installation kit. Why should one use 5 when there are APL, J, K etc. implementations available? Some things that may speak in favour of 5 are listed below:

1. 5 is Open Source and easily portable to any architecture for which a Perl interpreter exists.
2. Since 5 does not need any special characters there is no need to install additional fonts, which makes its installation even simpler.
3. Installing 5 does not require any special rights – even end-users can install it locally in any directory, even `C:\Temp` on Windows systems, which makes the interpreter an ideal tool for ad-hoc analyses and experiments at customer locations.

4. The interpreter is very well structured, which makes extensions to the 5 code quite straight forward, thus facilitating experiments in language design etc. (The complete interpreter consists of only 2895 lines of Perl code and 457 lines of 5 code contained in the standard libraries `stdlib.5` and `mathlib.5`.)
5. 5 is an emerging language where the individual can have a real impact on the directions of future developments. The development of the interpreter is still ongoing and we would love to hear about your suggestions and needs.

Some of the areas which will see future developments are those listed in the following:

- The mathematical library `mathlib.5` needs to be extended to overload more operators for complex numbers, polar coordinates, quaternions and the like. Also the library is still lacking most of the common linear algebraic operators and functions.
- The interpreter currently lacks powerful operators for transposing and rotating matrices etc.
- We need more test cases for the interpreter – although there are a lot of test cases defined, which are run every time changes to the interpreter have been made, these are no longer sufficient and need to be extended heavily.
- We need example programs from the field to learn more about the power of 5 and to enhance the interpreter.

We would love to hear from you and I hope that I could interest you in this new array language. Have fun with 5 and happy array programming.

References

1. 5 at SourceForge: http://lang5.sourceforge.net
2. 5 documentation at SourceForge: http://lang5.sourceforge.net/viewvc/lang5/trunk/doc/introduction/introduction.pdf

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