

Contents

| | | |
|--|-----------------------------|-----|
| Editorial | | 2 |
| News | | |
| Dyalog | | 4 |
| Optima | | 8 |
| APL2000 | | 9 |
| General | | |
| BAA: Chairman's Report 2013 | Paul Grosvenor | 14 |
| BAA: AGM Minutes | James Greeley | 15 |
| Reflections on a long life | Sam Sexton | 18 |
| 2 ⁶⁴ | Roger K.W. Hui | 25 |
| APL | | |
| A question of character | Brian Becker | 34 |
| My favourite APL symbol | Roger K.W. Hui | 36 |
| Using email services from APL | Chris Hogan | 38 |
| Semantic arrays | Stephen Taylor | 46 |
| Compiling APL to JavaScript | Nick Nickolov | 54 |
| Boolean reductions | Phil Last | 62 |
| APLUnit - An APL unit test library | Gianfranco Alongi | 67 |
| NFL passer rating | Brian Becker | 75 |
| Dyalog's parser - a new parser in town | Dan Baronet | 79 |
| ELI: a simple system for array programming | Hanfeng Chen, Wai-Mee Ching | 94 |
| J | | |
| J-ottings 56, Trig Time | Norman Thomson | 104 |
| L-systems in J | R.E. Boss | 109 |
| Backgammon tools in J, 4: Ace-point Bearoffs | Howard A. Peelle | 115 |
| Fibonacci and golden spirals | Cliff Reiter | 118 |

Editorial

Well its taken a while longer than I would have liked but the latest issue of *Vector* has arrived. A glance to the right shows the journal providing much needed support in someone's daily toil.



This issue brings a few changes. Colour for one. We have negotiated a new printing agreement, a little extra cost, that will allow us to include a limited number of colour images and we will use this resource for photographs, graphs, screen images and diagrams in that priority order should we use up our allotment.

*Vector: As a sustaining member
for a sustaining member.*

Also we have automated the process of producing the PDF document for the print shop directly from the on line material which removes the need for a tedious manual task hitherto performed with great fortitude by Kai Jaeger. A consequence of this is that there is no need for a second proof reading of the collated PDF document.

We have experimented with using MathML in the sub-editing submitted papers to produce mathematical equations. This has always been a problem in the past where we have had to rely on using graphic image files. These in turn need more intervention as we moved from an on line to a printed medium.

The next improvement in the production process will be to try and make articles being prepared available for to authors and reviewers in an on line edit facility to help them collaborate in the exercise.

Going forward we are looking to make some changes to our *Vector* web site. One such change will be to post the latest edition of *Vector* as a PDF for members. This be available for download once it is ready for use in a variety of eReaders or similar devices. Six to twelve months on we would make it available to anyone else. Another possibility will be to make downloadable PDFs of the individual articles available. And beyond that perhaps collections of articles could be collated with a view to a print on demand facility. We would however be very interested to know what our members think about the whole issue of on-line publishing. Please note that we are not thinking about stopping the printed version of *Vector*.

John Jacob

News

Industry News

Dyalog Ltd

In accordance with tradition, the preparations for the annual Dyalog User Conference (Dyalog '13, to be held at Deerfield Beach on October 20th-24th) coincide with the final stages of planning of the next release of Dyalog APL (version 14.0, due out in Q2 of 2014). With about 15 developers beavering away on the APL interpreter and surrounding interfaces (including a full-time Technical Writer and Documentation Manager, Fiona Smith, who joined us in April), 2014 is going to be a very busy year for any users of Dyalog APL wishing to take advantage of all the new functionality and read all the new documentation that we are planning to roll out!

Going Cross-Platform

Most computer users will have noticed that a major shake-up of computing hardware and software platforms is in progress. At the server end, Linux is growing in importance as a platform for virtualised server and cloud computing due to its relatively small footprint. At the client end, Android and other tablet and phone operating systems are also growing rapidly. Multi-core hardware is appearing on clients and servers. While Windows Desktop will probably remain the *bread and butter* platform for most of our customers for many years yet, we want to allow people to take advantage of new platforms when they are ready to do so.

To simultaneously make APL more portable and improve the user experience for those wanting to develop on multiple platforms, we are separating the APL Engine from the development environment. 2014 will see the introduction of a new Remote IDE, which will be able to run as a native application on most popular client platforms, providing rich user interfaces with the same functionality across on all platforms. The *RIDE* will be able to connect from any client platform to APL engines running on any platform, locally or across a network. We expect to announce official support for at least two additional engine platforms at the conference, in addition to the four that are currently supported: Microsoft Windows (Intel), IBM AIX (Power) and Linux (Intel and ARM).

Having APL on many platforms is only truly valuable if users can also develop applications that are easy to relocate. To facilitate this, there will be new releases of tools for cross-platform application development, such as our StandAlone Web Service framework (SAWS), the MiServer web server toolkit and the Dyalog File

Server.

Going Parallel

Making it possible for APL users to harness parallel hardware is another extremely important goal. We are working on three different approaches to this problem:

Futures and Isolates:

Version 14.0 will provide new language constructs to make it easy to use multiple processors without the use of locks or semaphores for synchronisation:

- Isolates are namespaces that are managed by separate processes.
- Futures are array items that can pass through structural functions and calls to defined functions without blocking but block automatically when passed to a primitive function that requires an actual value until the value is produced.

Expressions that are executed within isolates immediately return futures.

A new parser to reduce interpreter overhead:

Also scheduled for delivery as part of version 14.0 is a new parser which performs static analysis and optimisation, significantly reducing interpreter overhead. Initially, the parser will only target functions that have no side-effects, but work on the parser will continue for the next several releases. For functions with small arguments, speed increases of a factor of 2 have already been observed.

An experimental APL compiler:

Dyalog is funding research at the University of Indiana into a highly-parallel compiler for “functional” APL code (initially only targeting dfns). Futures and isolates provide the ability to parallelise algorithms with coarse-grained parallel components - if the user indicates where the parallel sections are by explicitly using isolates. The compiler aims to automatically parallelise APL expressions. The goal is to seamlessly integrate the compiler into the interpreter, offering users the option of compiling code that is identified as suitable, but this is still very experimental, it is unknown when this technology might make it into our product line-up.

Going Functional

Functional programming focuses on writing expressions without loops and without mutation of global state (or other side-effects). APL has always supported programming in a functional style, and in Dyalog APL the addition of the lexically-

scoped dfn style of programming has made it easier to write more strictly functional application components. Apart from having advantages in terms of maintainability, the lack of side-effects means that it is easy to compile and automatically parallelise programs that are written in a functional style. The work that we are doing in the direction of compiling and parallelising the execution of user-defined functions is all significantly easier when the code is already functional: if you have not yet looked at using dfns for the computational components of your application, then this would be a good time to do so.

In addition to continuing to enhance the tooling for dfns, we are also adding a number of language features to version 14.0 that promote a more functional and parallel style of programming:

Function Trains:

Version 14.0 recognises trains of three functions as *forks* (as in the J programming language) and trains of two functions as *atop*. For example:

```
(+/ ÷ ρ) 1 2 3 4 ⍝ Average is sum divided by count
2.5
'abcdef' (~ε) 'aeiou' ⍝ α (~ε) ω ↔ ~αεω
0 1 1 1 0 1
```

Function trains allow the construction of functional units of code that are obvious optimisation targets. For example, version 14.0 will optimise uses of *atop* such as (`⊃ 0 >`), making an early exit from the execution of the relational function on the right when it knows it will be searching for a 0 or 1 in the result.

Rank operator (∘):

provides a more general mechanism than the existing axis modifier, for the repeated application of primitive or user-defined functions to sub-arrays of specific ranks.

Key operator (⊞):

similar to an SQL “group by” statement, the key operator repeatedly applies functions to subsets of one array, corresponding to the distinct values in an array of keys. For example:

```
('red' 'green' 'blue' 'green') {α,+/ω}⊞ 10 20 30 40
red    10
green  60
blue   30
```

Many common cases of key and rank, such as $\{+/ω\}⍱$ which computes the sum of values corresponding to each key, are highly optimised and significantly faster than the best APL expression than can be written to compute the same values before v14.0. Also note that, in order to pave the way for parallel implementations, the order of the invocations of the operand function is undefined for rank and key, unlike existing operators such as each or outer product, which always process elements in “ravel order”.

Other Version 14.0 Features

Version 14.0 will also include a number of speed-ups and an extension to dyadic iota to directly support *rowfind* operations in the primitive. We are looking at embedding compression algorithms into the interpreter to support compression of component files and the source of namespaces and classes, and many other improvements.

We are making a number of Technical Preview releases available – TP3 of v14.0 was released at the end of September. Contact sales@dyalog.com if you would like to help us evaluate the new language features.

If you can't make it to Dyalog'13, check our web page around Christmas for recordings of conference sessions.

Optima Systems Ltd – Industry News

Ok so we are 75% through 2013 and I am not sure where most of it has gone. The year has certainly been busy for us and the future is looking very buoyant right now. Over the past 12 months we have taken on 5 new staff;

Since August 2012 we have been training our three apprentices in the delights of APL. Some of you may have been following their blog but if not please take a look at <http://thethreeblindmiceapl.wordpress.com>. They have had a great year visiting the Dyalog offices, taking the APL course, attending their first APL conference in Denmark and of course building their APL controlled robots. Our apprentices are now full time members of staff and employed as Trainee APL Programmers. Their next challenge will be the Dyalog conference in Miami where they will show us all their robots and tell us how they made them. Watch this space (or follow the blog) and see how they get on.

Kevin Wallis has joined us as our new Business Analyst. Apart from his analysis skills he brings considerable Pensions knowledge with him which we plan to make good use of over the coming months. Kevin has also attended the beginners APL course at Dyalog so now not only do we know what he is doing, he knows what we are doing.

And last but not least, Kim Kennington, who is our most recent member of staff and takes on the Q&A role. She is going to make sure that what we say should happen with our code actually does! We can run but no longer hide.

Our COSMOS data visualisation product moves on from strength to strength with a number of small contracts now being signed and right now a significantly larger one is about to start. Most of the product activity has been in America thus far but we hope to start a sales pipeline in Europe and UK very shortly. To assist in the sales and marketing of this product we now own roughly half of Galileo Analytics based in Washington DC and who market COSMOS in the United States.

Although not fully confirmed at this time we are about to set up another company in Sweden (Optima Systems Sweden AB) which will perform much of our R&D work.

As a result of our growth and internal structure we now offer a large multi-disciplined APL team plus all the back-up and ancillary services to be expected of a larger software development company.

We are looking forward to the next twelve months being as exciting as the last.

APL2000

APL2000's leads the way with significant new enhancements while maintaining complete compatibility with existing APL+Win-based application systems:

- Significant performance gains in core APL functions
- Access the Microsoft.Net Framework directly
- Enable Multi-thread Processing
- Expose APL+Win applications to web-based users

APL+Win Versions 10 - 12

Significant overall performance improvements in the APL+Win interpreter were made available to APL2000 customers. Many customers experienced doubling of overall interpreter performance while maintaining complete compatibility with existing application system source code.

APL+Win Version 13.1

APL+Win v13.1 includes optimized catenation for significant performance gains, a new primitive function, and a new operator along with other improvements. Enhancements in this release include:

- Significant Performance Gains with Optimized Catenation
- A New Mismatch Primitive Function
- New Commute Operator
- New Euro Glyph Shortcut Key
- New Log File Options

Optimized Catenation Yields Breakthrough Speed

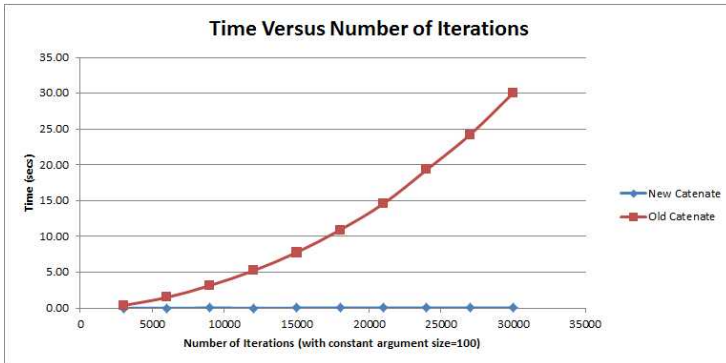
APL+Win expert Brent Hildebrand says it best:

“I want to say just one word about the new catenation - WOW. three orders of magnitude faster. 1825 times faster +/- in one test.”

Optimized Catenation in APL+Win 13.1 provides significant performance gains, particularly in cases that involve repetitive catenations with large arrays such as inside an iterative control structure.

Traditional catenation runs exponentially slower as the number of iterations

increases, making it impractical to use in some real-world applications. The Optimized Catenation function has nearly linear performance characteristics as a function of the number of iterations and volume of data making it an elegant solution that is blindingly fast.



This graph represents APL2000 testing results.

Note the brevity of the new optimized syntax:

```
A←A,B (Traditional syntax)
A,←B (Optimized syntax)
```

Extraordinary speed increases are possible if your application uses catenation for:

- Building Long Lists
- Building Documents such as Web Pages
- Accumulating Results from Stochastic Models

APL+Win Version 13.2

APL+Win v13.2 includes the APL+Win C# Script Engine (CSE). With the CSE enhancement an APL+Win programmer can merge the benefits of .Net technology with the power of APL+Win. Direct access to the .Net Framework from APL+Win is now available while maintaining complete compatibility with existing APL+Win-based application systems.

The C# Script Engine for APL+Win

A few examples of the power the C# Script Engine in APL+Win:

- Create utilities using .Net features for use in APL+Win application systems.

- Create and present Windows Forms or Windows Presentation Foundation GUI's.
- Use the .Net web tools to access and capture web-based information.
- Access Microsoft Office using .Net tools.
- Use third Party .Net tools in native C# format from APL+Win.
- Use ADO.Net to access databases such as Microsoft (SQL Server, Office), Oracle, IBM (DB2, Informix, U2), SAP (Sybase), MySQL and SQLite.

APL2000 developed the `□cse` system function as an interface to the APLNext C# Script Engine for APL+Win. APL+Win application system programmers can now access the features of the Microsoft.Net Framework directly from APL+Win.

The CSE enables an APL programmer to create multiple instances of the CSE object in memory space separate from that of the initiating APL+Win instance. With the CSE, APL+Win data can be passed to a C# object, where it can be manipulated using the extensive Microsoft .Net Framework and the result can be returned as APL+Win data.

The APLNext Supervisor for Multi-threaded Processing in APL+Win Applications

If an APL+Win application performs repeated in-memory processing of similar data elements, multi-core processing can provide significant performance improvements. Typical applications which benefit include:

- Time series analysis
- Population models
- Stochastic process models
- Simulations

The APLNext Supervisor exploits available multiple processors on a workstation by automatically scheduling and controlling multiple instances of the APL+Win ActiveX calculation engine to significantly improve APL+Win application system performance.

The APLNext Supervisor causes application programmer designated APL+Win functions to be executed in a concurrent, multi-threaded manner. The object model of the APLNext Supervisor provides for extensive control and monitoring of multiple threads directly from APL+Win.

The APLNext Supervisor is available to all current APL+Win Subscription licensees at no additional cost. APL2000 consultants can assist APL2000 customers to determine if the APLNext Supervisor technology will produce significant benefits in an APL+Winbased application system.

Expose Your APL+Win Applications to Web Clients with the APLNext Application Server

APL+Win algorithms, business rules, data stores and application output can be accessed by web-based clients using public or private Internet communications for real time or scheduled batch processing. With the APLNext Application Server an APL+Win application can satisfy requests from:

- Browser-based clients
- Server clients under program control
- Local workstation, tablet or smart phone clients under program control

Using the APLNext Application Server an APL+Win application can provide web-based clients with processing capabilities that are:

- Responsive - The APLNext Application Server establishes a programmer-controlled pool of APL+Win instances with workspaces already loaded containing APL+Win functions ready to calculate
- Easily Scalable - The APLNext Application Server can be scaled to multiple servers so that client requests are always satisfied promptly.
- Capable of handling client request which are independent and stateless or dependent and stateful.
- Easily tracked and, if desired, monetized on a transaction request level.
- Deploy the APLNext Application Server using Microsoft Internet Information Server (IIS) or as an independent web server.

APL2000 Conference March 23-25, 2014 in Ft. Lauderdale Florida



Please save the dates and mark your calendar now!

The conference will be held at the GALLERYone Doubletree Suites by Hilton, conveniently located on the Intracoastal Waterway, 8 miles from the Fort Lauderdale/Hollywood International Airport (FLL) and 3 short blocks from the Atlantic Ocean beaches.

We look forward to seeing you.

For more information on APL2000 products and services contact sales@apl2000.com.

General

BAA: Chairman's Report 2013

Paul Grosvenor (paul@optima-systems.co.uk)

Hello to you all and welcome back to Vector. Our production team has been working hard to produce this edition which now comes with a few pages in colour ! We would very much like your input and views as to whether or not this adds or subtracts from our journal. There is a small increase in costs as a result but significantly less than if we printed the whole thing in colour. I think it adds but what about you?



Our AGM was held in May, as last year, during the BAA Moot at the Lee Valley youth hostel [<http://moot.aplwiki.com/>]. Thank you to Phil Last for arranging and generally running around after us. The AGM & Moot saw about 20 keen APLers do what APLers seem to do best – cut code, talk code and drink beer but not necessarily in that order. If anyone is interested, the BAA meets in London monthly to continue these activities so contact Phil to find out more [phil.last@ntlworld.com].

Coming up over the next few months we have the Minnowbrook seminar and Dyalog 2013 conference in Miami and we look forward to including a write up of these in our next edition. If there are any other meetings / gatherings / seminars / that we have not mentioned please let us know so we can include them. Of course if you would like to send us a few words or even photographs of your event we would love to include it into future editions.

I hope you enjoy our journal and look forward to seeing some of you at the forthcoming conferences and just to finish off, a quote from Eugene;

Some of my children and nephews started APL on a 5100. Some started on the two-huge-suitcases version of “portable” APL. Some started on the 2741 time-sharing terminal. Luckily, none of them had to start on a 1050.

I started on the blackboard version.

— Eugene McDonnell

Well we all have to start somewhere ...

BAA AGM Minutes

Start 2.47pm

Chairman - Paul Grosvenor

Secretaries - James Greeley, Shaquil Sidiki, Sam Gutsell (The Three Blind Mice)

Been a relatively quiet year. Most activity in and around *Vector*, Phil Last hosted BAA London, a speech was given about using K to control a GPU.

There will be a meeting in two weeks' time - no agenda as of yet. Andy Shiers asked Phil Last if they were having a meeting as Brian Becker will be over at some stage or another. Open agenda in two weeks' time.

After the AGM John Scholes will be doing a talk.

Vector is running behind but John Jacob has a plan.

2.55pm Paul Grosvenor hands over to the treasurer Nicholas Small

There was an anonymous donation of £200

There was one edition of *Vector* this year and costs were £250 higher than the year before due to postal costs going up by a third.

Membership numbers have fallen 'quite a lot' about half the decline in membership from volume 24-25 he will chase them up shortly to encourage them to join. He asks if anyone knows about the Japan APL association as the editions of *Vector* were returned. Stephen Taylor said that the Japanese association was essentially defunct.

3.00 pm John Jacob

The next edition of *Vector* is about half way done. Optima have provided additional support from their new personal assistant Donna Scozzafava. *Vector's* hosting has been changed and they now have control of the domain name.

John Jacob went on to talk about automating the process of converting the articles to pdf's to be ready for print as well as making the markup easier. This should cut down on the amount of time it takes to put *Vector* together.

The meeting involved a discussion about having *Vector* produced in colour.

It surprised John Jacob how long it actually took to produce *Vector* (this came up at Christmas where a problem with the printers delayed the whole process by 3 weeks)

On the technical front John Jacob aims to iron out the whole process with automation to speed up the whole process so they will be able to produce *Vector* more regularly which will hopefully stimulate demand.

Paul Grosvenor - we've spoke about producing a electronic version of *Vector* for a long time. However many members prefer a physical copy.

Currently there are no restrictions to download from the archive however one member of the audience preferred it be for subscribers only.

3.07pm Back to Paul Grosvenor

Nicholas Small - votes that we keep the committee the same

Kai Jaeger - objected that Peter has not been here for 2 years!

Ray Cannon - Peter sent his apologies

Paul - will ask peter if he wants to carry on

Ray Cannon - He is happy to carry on but is happy to stand down if anyone else wants to start.

Paul Grosvenor - does anyone have any objections to the editor being part of committee

John Jacob was voted as part of the committee

Kai Jaeger - suggests the secretary should be removed entirely

Paul Grosvenor - It's been agreed that the committee will stay the same (with the addition of Editor John Jacob)

AOB

- Memberships are declining as a whole - This is a general trend where people are reaching out to the internet and grabbing it from there, people are retiring but there is young blood (Three Blind Mice/Stephen Taylors apprentice)
- Morten - maybe we should put more pressure on some of Dyalog's major customers - this could increase membership
- There are currently 3 corporate members of *Vector*; Sim Corp being one of them, subscribes to 10 copies, Paul remarked that it maybe worth asking as that is not many for such a large company.

Paul- calls the meeting to an end 3.20PM much applause.

Reflections on a long life

Sam Sexton

Authors Note [This article will appear in various places, so apologies to those of you who don't know many of the names, but I wanted to put on record the history of what is regarded by many as two ground-breaking applications. I've included the Newsflash crew, but there were too many (all around the world) involved in FX Order Management to list you all, but you know who you are! If I've omitted any of the early developers, the fault is mine, so please accept my apologies.]

A long time ago in a galaxy far, far away... well, perhaps it's not quite that distant in time and space, but the life of the products I'm about to tell you about may be among the longest in the commercial world.

To start at the end, on October 15th 2012, Israel Discount Bank (IDB) informed us that they had gone live on RET-LOM. Coincidentally on that day an article appeared on The Hub commemorating the end of System 77, an editorial system which dated from the mid 1980s and had shut down in September. Whilst this is impressive longevity, it didn't go back as far as the technology used in FX Orderwatch (FXOW - the first of various names of the product over the years), which is what IDB had migrated from to RET-LOM.

This represented the end of several years of migrating FXOW customers to the Sybase and Oracle-based successors and the end of an era that possibly started in the 70s. I say possibly, as it's been difficult finding anyone who can remember that far back!

However, with that caveat, back to the beginning... it all started with a Canadian software company called I P Sharp Associates (IPSA). Back in the 1970s they provided a timesharing service with email and several databases with software to manipulate them on their Toronto mainframe, which ran a modified version of IBM's DOS, known as Sharp DOS. This was accessed from around the world using their proprietary packetswitched network, IPSANET, with nodes in around 60 locations.

Around the end of the decade, BP's Oil Trading International division (OTI) wanted to be able to communicate between their traders more efficiently than using phones and later PCs and spreadsheets. They came to IPSA and the product Newsflash was created for them, by Paul Odho, who sadly died of a brain tumour in 1987. BP used

the system to ensure that they had the correct stocks of fuel at airports and docks and it provided them with a significant commercial advantage. This system was written in the wonderful (some may say 'esoteric') APL, with the database implemented in IPSA's component file system.



Left to right: Sam Sexton, Kuldeep Karnan, Hayden Bird, Shaun Doyle (at the back, spoiling the photo!), Gary Smock, Ed Nelson and Andy Bunce.

Newsflash was effectively smart email with bulletin boards, but it was extremely useful to BP for communicating between six of their traders in New York and a similar number in London and it became known as the Sharps system. The number of users grew and eventually exceeded the capacity of Newsflash to cope with the load, so Paul, along with Peter Airs, addressed this with Newsflash 2 in 1986. This used the magical capabilities of Shared Variables, which allowed a user to be notified immediately when they had an incoming message. This capability is part and parcel of everyday life now, but back then it really was cutting edge technology!

News of the effectiveness of this system spread within BP and another system was created for the Marine division. Karl Mabert and Bob Davison moved from developing the OTI system and worked with Greg Prowse, Mike Elbourne and Hazel O'Hare on this new system.

The screenshot shows a DOS-style window titled "FX ORDERWATCH DEMO SYSTEM". The window has a menu bar (File, Edit, Create, Process, View, Signon, Setup, Window, Help) and a toolbar with buttons like NEW, CONFIRM, ALL, MINE, NEAR, DONE, REPEAT, ACCEPT, PASS, FILL, and CANCEL. Below the toolbar is a "Orders All" section with a table of orders. The table has columns for No., V M S, Currency, Qty, u/s, Units, Rate, Bond, Type, Good Unit, Customer, and Comment. The data is as follows:

| No. | V M S | Currency | Qty | u/s | Units | Rate | Bond | Type | Good Unit | Customer | Comment |
|-----|-----------------|----------|------|-----|--------|------|------|-----------------------|-----------|----------|-----------------|
| 110 | | MIO | 10 | | 5.5400 | 0 | | T P (Take IGTC) | | Test | |
| 111 | | MIO | 8 | | 1.8400 | 0 | | | | eur Euro | test - sent dir |
| 112 | Cancel | MIO | 2 | | 3.0200 | 0 | | DL (Drop) (TYP) (GTC) | | Test | |
| 113 | Fill | MIO | 2 | | 3.0200 | 0 | | T P | | Test | |
| 114 | Fill | MIO | 10 | | 1.8100 | 0 | | T P | | GTC | Test |
| 115 | History | MIO | 2 | | 1.8050 | 0 | | T P | | GTC | Test |
| 116 | Reset | MIO | 1 | | 1.8000 | 0 | | T P | | GTC | Test |
| 117 | Withdraw | MIO | 1.5 | | 0.3975 | 0 | | T P | | GTC | Test |
| 118 | Consent to Euro | MIO | 5.25 | | 3.0000 | 0 | | T P (Take IGTC) | | Test | |
| 119 | Create Set | MIO | 1.33 | | 1.6255 | 0 | | T P (Take IGTC) | | Test | |
| 120 | Forward | MIO | 1.33 | | 1.6255 | 0 | | T P (Take IGTC) | | Test | |
| 121 | Fill Now | MIO | 1.33 | | 1.6255 | 0 | | T P (Take IGTC) | | Test | |
| 122 | Note | MIO | 1.33 | | 1.6255 | 0 | | T P (Take IGTC) | | Test | |

At the bottom of the window, the status bar shows "EUR/USD 1.0194 GBP/USD" and "gmock | FX ORDERWATCH DEMO SYSTEM".

In time, the Sharp DOS host in Toronto migrated to IBM's MVS and BP decided to move the service inhouse and onto their own network. Iain Hunneybell and Paul Odho worked on the application code issues and I had the pleasure of transferring the service to BP's offices in Finsbury Square. It later moved to Hemel Hempstead and a few years later I worked with Steve Moles of BP to migrate it to Solaris, using SAX (Sharp APL for uniX) rather than SAM (...MVS). This worked very well and only the code that interfaced with the operating system needed adjustment.

Shell also came looking for this technology and Peter Airs, along with Mike Elbourne and Peter Biddlecombe came up with another Newsflash system called BEANO, for marine bunkering - communicating between different companies within the Shell group to supply marine fuel. Such was the effectiveness of this that in due course, two more systems - LEANO (for lubricants) and ACE (for Aviation fuel) followed.

Eventually, Shell dropped their use of the system, but BP continued, although the Marine division migrated away later on, leaving OTI as the last Newsflash user - but a big one. In March 2008, they had moved from the original handful of users to 3000 around the world.

In 1987, Reuters had bought IPSA and in 1993, after having absorbed the database part of the company, sold the APL development and support groups back to the employees, causing the formation of Soliton Associates. At this point, many of the Newsflash team soon left Reuters, but others who had just joined IPSA at the time of the takeover took their place.

Over the years, attempts were made to make the Newsflash interface more sexy by giving it a GUI, but despite the best efforts of Andy Bunce, Bob Davison, Mike Elbourne and Peter Biddlecombe, most traders were happy with the original line mode interface, as it was quick and efficient. Meanwhile Dave Ziemann joined us as a contractor and reworked the database access functions (a function is an APL program or building block) to provide greater efficiency.

As Reuters was down to one customer for the Newsflash technology, we started to look around for other areas where it could be applied. We were lucky in that the remaining ex-IPSA staff were assigned to City West (later International) division, headed up by Claude Green, with his deputy Ed Nelson, who unlike many of their peers and seniors, had the nerve to take on what Ed recently referred to as "an eccentric but very talented team" (there's no argument about the first part of that description!). Claude soon came to realise that he'd made the right choice and said that he "liked dealing with the ex-IPSA staff as you could ask them to do something, then go away and forget about it and they'd come back on time with the goods". Claude and Ed were invaluable in keeping the team together - despite encouragement to do otherwise - and spurred us on to find alternative uses for the Newsflash technology, given that we only had BP as our customer.

Some weird and wonderful suggestions arose such as management of racing bloodstock, radio advertising and agency nursing and I seem to recall suggesting a bed management system for the NHS. Radio advertising was considered, but when Reuters bought LBC, we were no longer considered a neutral player. Consideration was given to using Newsflash to enhance the Shipping service, but there was an opinion that Reuters Mail was better - but that was shut down shortly afterwards! It wasn't until Paul Jackson, who'd managed the BP Newsflash account, overheard a conversation in a pub (a rare visit to such an establishment for him!) that the idea of managing FX orders came up. This was pursued with Sumitomo and a prototype system produced, but in the end, Sumitomo decided not to proceed. Luckily, as Alan Clarke (known as 'Shoulders'), then at HSBC, recalls:

'I remember we had persuaded IT and management to buy a competitor's product as we didn't know you had one at the time, and then our account manager sent in Super Jacko who suddenly turned up with one. We didn't like that and the following week Super J had found another that was something to do with an oil platform that Reuters had acquired. We liked him and his entrepreneurial approach a lot and decided to eat humble pie with the management and IT and do a U-turn and switch to the Reuters product. Jon [Healey] and I then spent a long long time working with Jacko to turn it into an FX product.'

'The other interesting thing was that at the launch party where Reuters were showing it off to the community, rumours circulated that management at Reuters were going to kill off the product, virtually at its birth. Fortunately we made it clear that HSBC would not be amused and the decision never came to kill it... the rest as they say, is history.'

Indeed, there was much debate at senior management level as to whether it was appropriate for such a complex product developed by such a small group, to be launched on a global basis. As Shoulders states, it was the pressure from HSBC and other clients that resulted in a positive decision being made.

The competitor's product referred to was Telerate Passbook. FXOW proved to be the better product and gained a significantly higher market share. When Telerate was acquired by Reuters, all the Passbook clients migrated to FXOW.

At HSBC, FXOW replaced whiteboards, spreadsheets and faxes with the new system and it used a variant of the Visual Basic GUI which Andy Bunce had provided to BP.

Shoulders' colleague at HSBC, Jon Healey, also commented recently, "To this day, I have not yet found another orders system that fulfilled the single main criteria of an orders system as well; it stayed up".

On a personal note, we all enjoyed working with HSBC and both sides considered it as good a relationship as you could expect between supplier and customer – there were many happy memories, not all of which were associated with licensed premises!

Other banks soon followed suit and with the help of our agent TMT in Helsinki (Timo, Tuija, Tapsa and others whose names didn't start with T!), we had about 30 customers around the world. I was the lucky one who flew around the globe, installed many of those systems and trained internal and customer technical staff to look after it – an onerous task, but someone had to do it! Once it was installed, it didn't really need that much looking after. As an example of this, I'd installed a demonstration system at a bank in London on my way to Heathrow in January 2002. A couple of months later, we attempted to retrieve our hardware, but the customer was loath to let it go, as the dealers loved the system and we suspected were using it for more than just evaluation purposes. In the end, the bank got their own hardware and we got the box back eighteen months after it had been installed, during which time no backups had been taken and no (required) regular housekeeping performed, but the system just kept purring along.

no. 439949 from stew 23:01 26 may 2011 to all
subj Last message from Paul Reed

This may be the last message sent on the 'Sharps' system – and it marks something of a milestone. I have had the 'pjr' sharp code since I first traded Rotterdam barges in our Dutch office in 1985. Originally it was a groundbreaking Instant Messaging system used by BP traders to connect 6 traders in New York with 6 in London. By the time I joined the Crude bench two years later it had evolved into

also keeping records of all crude bids and offers known from the market and was accessible to all our refinery buyers, it was used as our deal entry system and even did real time exposure for our OTC Brent and Dubai at one stage. Within a few years it had grown to over 600 users in 40 countries and for many years was a great competitive advantage when most other firms were still using Telex to communicate and were just implementing Fax technologies and then subsequently mobile phones and later personal computers.

Clearly today everyone has instant messaging and we are not only the first users on the planet to use the system widely but also now the last.

Today's milestone passes whilst recognising that for over two and a half decades this system was at the heart of our collaborative and joined up culture.

PJR (aka Paul J Reed, Chief Executive,
Integrated Supply and Trading)
Regards,

Despite this success, time was catching up with the product and it wasn't a viable proposition to commission a Solaris 10 version of SAX for Solaris 10 from Soliton and so FXOW customers were encouraged to migrate to an Automated Dealing system, based on TIBCO technology with Sybase. This attracted some customers, but wasn't a roaring success. However, in December 2002, Reuters acquired AVT Microsystems in Nottingham as they were looking for a company to help them spread the use of their trading software called GenIdeal. This was much better and the FXOW customers were encouraged to migrate to RET-OM, based on GenIdeal with the database in Oracle and (eventually) a Java GUI. Kevin Clarke had the honour of being the last APL and VB programmer to work on the code, after having left Reuters for Soliton when the Global Limits Control System (that's another story!) that he worked on was closed down, but he returned to look after FXOW.

The system didn't need too much support, but a couple of occasions when it was needed stand out. One year Kevin and I were called on Christmas Day and then again on New Year's Day – but our marriages managed to survive the experience! At Easter 2003 I was attending my niece's 21st birthday party and when I returned to the car, I found I'd left my mobile there and had missed about 18 calls. Our favourite customer had had a hardware crash and needed some help, so as soon as I arrived home, I was off again on the train to London. It was a long and stressful night and the three of us (myself, Teech and a Sun engineer) eventually got the system up and running, but after ten years of therapy, we can look back fondly on the experience!

The BP story ended at close of business in New York on 27 May 2011, as recorded by Paul Reed in his message (see left), which is reproduced with permission.

Between them, these two products, which were seen as too risky in various quarters (without justification, as time was to prove) and only kept going by tenacious and committed managers together with responsive support and development staff, have provided a service to BP and many banks for around 30 years and they outlasted many of those who were not convinced of the potential of both the technology and the team that developed it. They also provided a significant revenue stream.

I am sure that I speak on behalf of all involved in saying that we are proud and honoured to have played our roles in this small episode of history and that we recognise the debt we owe to the late Paul Odho and also Paul Jackson, who sadly also died - in June 2003 - without whom our working lives would have been very different and much less fun!

2^{64}

Roger K.W. Hui

How big is 2^{64} ?

Basics

```
2^64
1.84467e19

2^64x
18446744073709551616

'c0.0' 8!:2 ]2^64
18,446,744,073,709,551,616
```

And, using the verb us from [1],

```
us 2^64x
eighteen quintillion, four hundred forty-six quadrillion,
seven hundred forty-four trillion,
seventy-three billion, seven hundred nine million,
five hundred fifty-one thousand, six hundred sixteen
```

Grains on a Chessboard

One grain of rice is placed on the first square of an 8 by 8 chessboard, two grains on the next square, four grains on the next, and so on, doubling on each square. The total is of course $(2^{64}) - 1$ grains. How deep would that amount of rice cover the earth?

Answer in [2].

Particles in the Universe

Is 2^{64} larger than the number of particles in the universe? Not even close [3, 4].

Avogadro Constant

The Avogadro constant[5] has value $6.022141e23$.

```
6.022141e23 % 2^64
32646.1
```

That the Avogadro constant is the number of atoms in twelve grams of carbon-12 makes evident the enormity of the error of estimation in the previous section.

Age of the Universe

The age of the universe[6] is estimated to be about 14 billion years; its age in milliseconds is:

```
*/ 14e9 365.2425 24 60 60 1000
4.41797e20
```

CPU Cycles

Assume the average modern PC is rated at 2 GHz. The number of CPU cycles in a year is therefore:

```
*/ 2e9 365.2425 24 60 60
6.31139e16
```

The total of CPU cycles in a year for the computers found in a residential neighborhood would exceed 2^{64} . (The required number of computers is $292.277 = (2^{64}) \% 6.31e16$).

Supertanker Bytes

The largest tanker ever built, the Knock Nevis[7], has a deadweight of 564,763 tonnes (tonne = 1000 kg) and measures 1504 feet by 226 feet with a draft of 81 feet. A run-of-the-mill disk drive has a capacity of 200 GB, and $9e7$ drives would have a total capacity of 2^{64} bytes. Unless each drive exceeds 6 kg the tanker would be able to carry them.

Might the tanker be constrained by volume? Its volume exceeds $27532224 = */ 1504 226 81$ cubic feet which would readily accommodate $9e7$ disk drives (0.3 cubic foot per drive).

We used to play a parlour game of wondering, "What's the fastest way to send data across the Atlantic?" Adapted for the current paper and for current technology, the question we may ask is, "What is the fastest time to send 2^{64} bytes across the Atlantic?" (A supertanker full of disks/flash drives/DRAMs? An A380 full of same? Transmission by a 100 Gbps submarine cable?) A rule of this game is that you must do the calculations in your head.

Leaves on Trees

You stand on a mountain top in the North American Pacific Northwest with trees in every direction. Are there 2^{64} leaves on the trees within your sight? Estimate as follows:

- you can see 100 miles[8]in every direction
- there is a tree every 5 feet

Therefore, the number of trees within your sight is:

```
NB. square feet within your sight
o. *: 100 * 5280
8.75826e11

NB. # trees within your sight
(*:5) %~ o. *: 100 * 5280
3.5033e10

NB. required # leaves on a tree
(2^64) % (*:5) %~ o. *: 100 * 5280
5.26553e8
```

Is it plausible for there to be 5.27e8 leaves on a tree? There probably aren't that many leaves on an average deciduous tree. However, trees in the Pacific Northwest are evergreen. 5.27e8 needles on an evergreen tree seem possible (22956.5 = %: 5.27e8 ; 23 thousand branches each having 23 thousand needles).

Compound Interest

How many years does it take to reach 2^64 dollars for \$1 invested at interest rate r ? The equation for semi-annual compounding is:

$$(2^{64}) = (1+r\%2)^{2*y}$$

Taking logarithms on both sides, we get $y = - : (1+r\%2) ^ . 2^{64}$

```
] r=: 0.01 * 1+i.10
0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1

-: (1+r%2) ^ . 2^64
4447.22 2229.14 1489.78 1120.09 898.273 750.393
644.761 565.536 503.914 454.614

r ,. >. -: (1+r%2) ^ . 2^64
0.01 4448
0.02 2230
0.03 1490
0.04 1121
0.05 899
0.06 751
0.07 645
0.08 566
0.09 504
0.1 455
```

Fibonacci’s Rabbits

Fibonacci studied the population growth of (idealized) rabbits where:

- in the first month there is 1 newborn pair of rabbits
- a new-born pair becomes fertile from the 2nd month on
- each month every fertile pair begets a new pair
- rabbits never die

How many months are required for the number of rabbits to reach 2^{64} ?

Let F_n be the number of pairs of rabbits after n months. Only the F_{n-2} rabbits that are alive at $n-2$ months produce a pair, and these are added to the existing population F_{n-1} . Thus $(F_n) = (F_{n-1}) + (F_{n-2})$. F is of course the Fibonacci sequence [9].

It can be shown that $(F_n) = \frac{1}{\sqrt{5}} \left(\phi^n - \left(\frac{\phi-1}{\phi}\right)^n \right)$ where ϕ is the golden ratio $\frac{1+\sqrt{5}}{2}$. The equation to be solved is:

$$(2^{64}) = \frac{1}{\sqrt{5}} \left(\phi^n - \left(\frac{\phi-1}{\phi}\right)^n \right)$$

and the solution is:

```
phi = (1+sqrt(5))/2
phi_inv = (phi-1)/phi
n = log(2^64 * sqrt(5) * (1 - phi_inv^n)) / log(phi)
```

Less than 8 years.

Factorial

The number of ways of arranging n distinct objects is $n!$. What is the smallest n for which this exceeds 2^{64} ?

$$n! \geq 2^{64}$$

20.6671

Partitions

A partition of n is a sorted list x of positive integers such that $n = \sum x$. For example, the following is the sorted list of all the partitions of 5:

| | | | | | | |
|---|-----|-----|-------|-------|---------|-----------|
| 5 | 4 1 | 3 2 | 3 1 1 | 2 2 1 | 2 1 1 1 | 1 1 1 1 1 |
|---|-----|-----|-------|-------|---------|-----------|

What is the smallest n for which the number of partitions of n exceeds 2^{64} ?

The verb `pnv` is from [10] where `pnv n` are the number of partitions for $i . 1+n$.

```
p=: pnv 500
$ p
501
5 10 $ p
1 1 2 3 5 7 11 15 22 30
42 56 77 101 135 176 231 297 385 490
627 792 1002 1255 1575 1958 2436 3010 3718 4565
5604 6842 8349 10143 12310 14883 17977 21637 26015 31185
37338 44583 53174 63261 75175 89134 105558 124754 147273 173525

p (>i.1:) 2^64
417
. . 416 417{p
17873792969689876004
18987964267331664557
2^64x
18446744073709551616
```

Katana

To create a katana[11] (samurai sword) a billet of steel is heated and hammered, split and folded back upon itself many times. If the number of foldings is greater than 64 then the number of layers exceeds 2^{64} .

E=m*c^2

With total conversion, how many kilograms of mass are required to obtain 2^{64} joules of energy?

```
(2^64) % *:3e8
204.964
```

Square Inches

What is the radius in miles of a sphere whose surface area is 2^{64} square inches? The surface area of a sphere with radius r is $0.4**r$. Thus:

```
(* / 12 5280) %~ %: (2^64) % 0.4
19122.3
```

Such a sphere is a little larger than Uranus.

Cubic Inches

What is the radius in miles of a sphere whose volume is 2^{64} cubic inches? The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. Thus:

```
(*/ 12 5280) %~ 3 %: (2^64) % o.4%3
25.8699
```

Hilbert Matrix

The Hilbert matrix [12] is a square matrix whose (i,j) -th entry is $\frac{1}{i+j}$. It is famously ill-conditioned with a very small magnitude determinant.

```
H=: % @: >: @: (+/~) @: i.
H 5x
1 1r2 1r3 1r4 1r5
1r2 1r3 1r4 1r5 1r6
1r3 1r4 1r5 1r6 1r7
1r4 1r5 1r6 1r7 1r8
1r5 1r6 1r7 1r8 1r9
det=: -/ .*
det H 5x
1r266716800000
%. H 5x
25 _300 1050 _1400 630
_300 4800 _18900 26880 _12600
1050 _18900 79380 _117600 56700
_1400 26880 _117600 179200 _88200
630 _12600 56700 _88200 44100
```

The inverse Hilbert matrix has all integer entries, whose (integer) determinant is very large.

```
>./ | , %. H 15x
114708987924290760000
>./ | , %. H 14x
3521767173114190000
% det H 7x
2067909047925770649600000
% det H 6x
186313420339200000
perm=: +/ .*
perm %. H 5x
4855173934730716800000
perm %. H 4x
5314794912000
```

The smallest inverse Hilbert matrix with an entry that exceeds 2^{64} in absolute value is the one of order 15 ; with a determinant that exceeds 2^{64} , order 7 ; with a permanent that exceeds 2^{64} , order 5 .

Making \$\$\$

In U.S. dollars the units in common circulation are:

- bills: 100 50 20 10 5 1
- coins: 0.25 0.10 0.05 0.01

A dollar can be “made” in a number of ways:

| 1.00 | 0.25 | 0.10 | 0.05 | 0.01 |
|------|------|------|------|------|
| 0 | 0 | 0 | 0 | 100 |
| 0 | 0 | 0 | 1 | 95 |
| 0 | 0 | 0 | 2 | 90 |
| ... | | | | |
| 0 | 3 | 2 | 1 | 0 |
| 0 | 4 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |

In fact, a dollar can be made in 243 ways. What is the smallest multiple of \$100 that can be made in greater than 2^{64} ways?

```
h=: 4 : 0
m=. # s=. +/\ y
if. 2.5=x do. (m$5{.1)#m($,)+/\_2]\s else. (m$x{.1)#s end.
)
chm=: 3 : '+/ 2 h 2.5 h 2 h 5 h 4 h 2.5 h (*y)$~1+20*y' " 0
```

If n is a multiple of 100 then chm n is the number of ways of making n dollars.

```
chm 100*>:i.3 5
4.88209e10 4.35246e12 7.62895e13 6.46316e14 3.58401e15
1.50147e16 5.149e16 1.51912e17 3.98556e17 9.51655e17
2.10326e18 4.35756e18 8.54636e18 1.59902e19 2.87178e19
chm 1400 1500
1.59902e19 2.87178e19
```

\$1500 can be made in 2.87e19 ways. The exact number is:

```
chm 1500x
28717791430084742056
```

Suppose the more rarely circulated \$2 bill and 50 cent coin are included. Then:

```
chn=: 3 : '+/ 2 h 2.5 h 2 h 2.5 h 2 h 2 h 2.5 h
      (*y)$~1+20*y' " 0

      chn 100*>:i.3 5
9.82355e12 2.78e15 9.69549e16 1.34924e18 1.10638e19
6.40915e19 2.90001e20 1.09038e21 3.54917e21 1.02915e22
2.71434e22 6.61402e22 1.50698e23 3.24114e23 6.63033e23

      chn 500 600
1.10638e19 6.40915e19

      chn 600x
64091464225604008941
```

\$600 can be made in 6.41e19 ways, and is the smallest multiple of \$100 than can be made in greater than 2^{64} ways.

References

1. Hui, Roger K.W., Number in Words, Jwiki Essay, 2007-07-12.
2. Hui, Roger K.W., Ken Iverson Quotations and Anecdotes, 2005-09-30.
<http://keiapl.org/anec/#rice>
3. Bernecky, Robert, comp.lang.apl post, 1996-03-31. *<https://groups.google.com>*
search for groups or messages: "fewer than 2 power 60 particles in the universe"
4. Hui, Roger K.W., comp.lang.apl post, 1996-04-01.
(as above)
5. Avogadro constant *http://en.wikipedia.org/wiki/Avogadro_constant*
6. Age of the universe *http://en.wikipedia.org/wiki/Age_of_universe*
7. Knock Nevis *http://en.wikipedia.org/wiki/Knock_Nevis*
8. You can see 100 miles *<http://en.wikipedia.org/wiki/Horizon>*
9. Hui, Roger K.W., Fibonacci Sequence, Jwiki Essay, 2005-09-26.
10. Hui, Roger K.W., Partitions, Jwiki Essay, 2005-11-18.
11. Katana *<http://en.wikipedia.org/wiki/Katana>*
12. Hui, Roger K.W., Hilbert Matrix, Jwiki Essay, 2005-09-29.

An earlier version of this paper appeared as a an essay in the Jwiki(www.jsoftware.com/jwiki/Essays/2^64) on 2007-12-06.

APL

A Question of character

Brian Becker

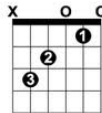
I've used APL for over 35 years and one of the recurring criticisms of APL is its non-standard character set – its use of “funny” symbols. This got me to thinking about symbols in general. We use symbols every day. They're used to convey meaning, to share information, to establish a common format for communication. The letters that form the words on this page are nothing more than symbols.

Symbologies are tailored to optimize communication within a domain.

A musician will see the symbols below and see a “C-major” chord.



Whereas a guitarist may understand C-major as this...



A mathematician sees the expression below and understands it as the sum of the first 100 integers.

$$\sum_{i=1}^{100} i.$$

And an APLer sees it this way `+ / ⍳ 100`

Symbols, until you learn their meaning, are incomprehensible. The argument “I can't read APL” can be applied to any domain that uses a specific symbology – language, music, mathematics, electrical engineering, even cooking – if you don't know that tsp means teaspoon and tbsp means tablespoon, you're in for some potentially unpleasant culinary surprises.

We learn new symbols all the time. Before the EU, the € symbol was unknown. Emoticons :-)) are a new breed of symbols that we've created to compactly express

emotion through typed media.

If we can accept that symbols are a good thing and that we are continually learn new symbols, perhaps we can get past the bias against APL because it uses those “funny characters”.

Downloadable APL fonts and keyboard drivers, on-screen keyboards and language bars have made APL easier to learn than ever. Yes, it does take some effort to learn the symbols and how to enter them – but that's no different than learning a new language, how to read music, or any domain where the symbology is unfamiliar to us. With APL, the payback in productivity far exceeds the investment to learn the symbols.

My favourite APL symbol

⊗ ○ *

by Roger K.W. Hui

One of the distinguishing characteristics of APL is its unique character set, containing 150-200 symbols. My favorite is ⊗, the symbol for logarithm. Originally, the log symbol was formed by 'overstriking' ○ (circle) and * (exponential or power). At present, ⊗ is Unicode[1] code point 0x235F.

Reasons for liking ⊗

- It's kind of cute, possessing a radial symmetry.
- It denotes a function for which conventional mathematical notation [2] does not have a good symbol:

```
⊗y ↔ ln y or log y
x⊗y ↔ logx y
```

- It alludes to $0=1+*○0j1$, the most beautiful equation in all of mathematics [3], relating in one short phrase the fundamental quantities 0, 1, e , π , and $0j1$ and the basic operations plus, times, and exponentiation.
- It is a visual pun – the symbol looks like the cross-section of a felled tree, i.e. a log [4].

Chronology

1962-03

In *A Programming Language* [5], logarithm, exponential, and power were not assigned symbols.

1966-03

In *Elementary Functions*[6], exponential and power were denoted $*y$ and $x*y$, their definitions to this day. Natural logarithm was denoted $*'$ and base- x logarithm was denoted $(x*)'$. (In the book, f' is the inverse of f .)

1966-11-27 15:53:58 (GMT-7)

Initial implementation of APL\360 [7].

1967-10-17

Natural logarithm was denoted by $\otimes y$ no later than the publication of *The APL\360 Terminal System* [8]. The dyadic case $x \otimes y$, base- x log of y , was undefined; instead, it was computed by a defined function in the public library workspace `1 utility` [9].

1968-08

Finally, natural logarithm was denoted $\otimes y$ and the base- x logarithm of y was denoted $x \otimes y$, their definitions to this day, no later than the publication of *APL\360 User's Manual* [10].

References

1. Unicode Consortium, *Unicode Standard 6.2*, 2013
www.unicode.org/charts/PDF/U2300.pdf
2. Abramowitz, Milton, and Irene A. Stegun, *Handbook of Mathematical Functions*, US National Bureau of Standards, 1964; Chapter 4
people.math.sfu.ca/~cbm/aands/page_67.htm,
3. Hui, Roger K.W., *Euler's Identity*, J Wiki Essay, 2010-02-04
www.jsoftware.com/jwiki/Essays/Euler's_Identity
4. McDonnell, Eugene E., *The Story of \circ* , APL Quote-Quad, Volume 8, Number 2, 1977-12 www.jsoftware.com/papers/eem/storyofo.htm
5. Iverson, Kenneth E., *A Programming Language*, Wiley, New York, 1962
www.jsoftware.com/papers/APL.htm
6. Iverson, Kenneth E., *Elementary Functions: An Algorithmic Treatment*, Science Research Associates, Inc., Chicago, 1966-03
www.jsoftware.com/jwiki/Doc/Elementary_Functions_An_Algorithmic_Treatment
7. Hui, Roger K.W., (ed.), *APL Quotations and Anecdotes*, 2010-09-18
www.jsoftware.com/papers/APLQA.htm#APL_birthday
8. Falkoff, Adin D. & Kenneth E. Iverson, *The APL\360 Terminal System*, Report RC-1922, IBM, 1967-10-16
www.jsoftware.com/papers/APL360TerminalSystem.htm
9. Conroy, C.A., Editor, *APL\360 Newsletter Number 1*, IBM, 1967-07
bitsavers.informatik.uni-stuttgart.de/pdf/ibm/apl/APL_Newsletter_1_Jul67.pdf
10. Falkoff, Adin D. & Kenneth E. Iverson, *APL\360 User's Manual*, IBM, 1968-08; Table 3.2 bitsavers.informatik.uni-stuttgart.de/pdf/ibm/apl/APL_360_Users_Manual_Aug68.pdf

this is what actually “sends” the email – and receives it at the far end.

MDA (Mail Delivery Agent): takes incoming mail from the MTA and places it in the addressee’s in-box. Can also be called a LDA (Local Delivery agent) if the email folder and the MTA are on the same server

MAA (Mail Access Agent): manages the folders of an email account and makes the messages available to a MRA.

MRA (Mail Retrieval Agent): accesses the email folders via the MAA and makes the messages available to the MUA

So where does APL fit into this?

The APL code is in effect the MUA :

- it will log into the MSA and send mail.
- it will log into the MAA and act as the MRA

How does all this communication happen?

Through the wonder of TCP/IP, so we use sockets.

My code is all Dyalog APL, but APL2000 also has TCP/IP sockets and I imagine APL2 and APLX can do this too. J can, but that is well beyond the scope of this short article.

What does the traffic consist of? Remember that email really started on Unix machines, so the commands are plain text.

So what’s in a message? This is the real (well, if you know what you are looking for) source of an email. This is the header and unrendered body of a real email from Amazon. I’ve taken a few liberties with the long Amazon addresses in order to get them to fit on this page, but that does not alter the substance of the header.

```
Return-Path: <20113470203e1e8f5b2ae4797a3d6f7453d450d78@bounces.amazon.com>
X-Original-To: sales@4xtra.com
Delivered-To: sales@4xtra.com
Received: from retail-smtp-out-22001.amazon.com
(retail-smtp-out-22001.amazon.com [212.123.28.40])
  by chris.vm.xeriom.net (Postfix) with ESMTP id E74253C5E2
  for <sales@4xtra.com>; Wed, 23 Mar 2011 13:52:12 +0000 (GMT)
DKIM-Signature: v=1; a=rsa-sha256; c=simple/simple;
  d=amazon.co.uk; i=auto-confirm@amazon.co.uk; q=dns/txt;
  s=rte02; t=1300888158; x=1332424158;
h=date:from:to:message-id:subject:mime-version:
  content-type:bounces-to:x-amazon-mail-relay-type:
  x-amazon-rte-version;
z=Date:=20Wed,=2023=20Mar=202011=2013:47:02=20+0000=20(UTC
)|From:=20"auto-confirm@amazon.co.uk"=20<auto-confirm@ama
zon.co.uk>|To:=20"sales@4xtra.com"=20<sales@4xtra.com>
|Message-ID:=20<1003847.1291711300888022380.JavaMail.corr
```

```

eios@rte-svc-eu-12011.dub2.amazon.com>|Subject:=20Your=20
Order=20with=20Amazon.co.uk|MIME-Version:=201.0
|Content-Type:=20multipart/mixed=3B=20=0D=0A=09boundary
=3D"-----3D_Part_93377_21274456.1300888022379"
|Bounces-to:=202011032313470203e1e8f5b2ae4797a3d6f7453d45
0d78@bounces.amazon.com|X-AMAZON-MAIL-RELAY-TYPE:=20notif
ication|X-AMAZON-RTE-VERSION:=202.0;
bh=ctkIU05imPGMCLPT674rciV+cif5MQ3E0lsqcdH+8Us=;
b=NP1SvGK3lKhSCu+FTTL+LoL9+ucTuYPdNjrf2dAh+1Fi14nKsVtuWGGx
bcEjxUT7Ksgrou7u16oP9dQ1gbatTQ=;
Date: Wed, 23 Mar 2011 13:47:02 +0000 (UTC)
From: "auto-confirm@amazon.co.uk" <auto-confirm@amazon.co.uk>
To: "sales@4extra.com" <sales@4extra.com>
Message-ID: <10847.129180.JavaMail.correios@rte-svc-eu-12011.dub2.amazon.com>
Subject: Your Order with Amazon.co.uk
MIME-Version: 1.0
Content-Type: multipart/mixed;
  boundary="-----_Part_93377_21274456.1300888022379"
Bounces-to: 2011032313470203e1e8f5b2ae4797a3d6f7453d450d78@bounces.amazon.com
X-AMAZON-MAIL-RELAY-TYPE: notification
X-AMAZON-RTE-VERSION: 2.0

-----=_Part_93377_21274456.1300888022379
Content-Type: multipart/alternative;
  boundary="-----_Part_93378_3767371.1300888022379"

-----=_Part_93378_3767371.1300888022379
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: quoted-printable
Thanks for ordering from Amazon.co.uk. Your purchase information appears be=
low.

```

The entire header routing information is in plain, human readable text.

The message body consists of one or more sections to hold plain text, HTML and attachments. In fact, because we cannot predict the hardware or software which will handle our message on its journey even the attachment must be plain text.

This is handled by Base64 encoding - the 256 values of a byte are reduced to an encoding of A-Z, a-z, 0-9, "+", "/" and "=" - Note how "=" was used as a continuation character and it is also used as padding. Obviously any attachment is going to be bigger than the original file.

So what does APL have to provide?

The most basic requirement is code to implement the TCP/IP communication. I've used the `APLmail` namespace provided by Konrad Hoesle-Kienzlen, with a few minor changes and improved handling for badly formed messages.

The next major issue is guaranteeing the integrity of any messages. This is to ensure against accidental corruption in transit. Together with encryption (discussed below) a hash can also guard against deliberate tampering with the message.

The simplest way of doing this is to provide a message digest. This is a hash: a fixed length string usually represented in hexadecimal format. It's created using an algorithm which takes the message as input and returns a hash. Two different messages result in different hashes, no matter how small the difference is. One of the most popular algorithms is known as Message Digest 5 or MD5 for short and this is the one which I've used.

We also need an encryption process. MD5 is non-reversible and what is to stop someone with malicious intent from altering not only the message content but the included MD5 hash? Therefore we also need to encrypt the hash and anything else we don't wish to be easily readable. Again there are many algorithms for encryption, but I chose the Tiny Encryption Algorithm (TEA) as it is secure and reasonably simple to implement.

For many of the uses we might want to employ an APL email process for, we will need file attachments. The `aplmail` workspace contains suitable functions which work very well on files up to medium size, read into the workspace using the native file system functions. As stated above, the attachments must also be converted into plain text using Base64 encoding. Naturally we cannot assume that the intended recipient's process is another APL workspace; therefore we must use ASCII translation when reading and writing the files.

If we are sending files of any size we should compress them. While there are several implementations of zipping in APL (for example in Dyalog's `dfns` workspace), for larger files I've found that they are too slow, so I use the freely available 7zip program.

A Message Digest 5 implementation

I wrote my own, but there is one on the APL wiki.

```
r←MD5 n;t;v;F;G;H;I;T
F←{x y z←de ω
  2⊥(x^y)∨(-x)^z
}
G←{x y z←de ω
  2⊥(x^z)∨y^~z
}
H←{x y z←de ω
  2⊥(x≠y)≠z
}
I←{x y z←de ω
  2⊥y≠x∨~z
}
T←{⌈4294967296×10ω}
r←Loop/SetState PaddingBits ASCII n n v
```

```

r←hx indx>,/(21ϕ(8 4ρωτ̃32ρ2)[7 8 5 6 3 4 1 2;]]r
PaddingBits←{
  m←11 ⌈DR ω           A ω is a message, treat it as binary
  p←{ω+512×ω≤0}448-512|pm  A assume ASCII translation has been done
  p←m,11 ⌈DR 1=pt1      A pad to multiple of 512, less reserve
  p←pp̃ϕ8,8±̃pp          A add "1" & as many "0" to pad
  p←,p[,{,ϕ({(4,~ω÷4)ρτω}>ρω)p;] A reshape to byte lengths
  p←p,ϑ2 32ρ(64ρ2)τpm  A reorder to low bits first, double words
  p←ϕpp̃32,~32±̃pp      A add original message length low order 1st
  21p                   A reshape to make 32-bit words
  }                     A returns as floating point

{r}←SetState n
r←hex"67452301" 'efcdab89' '98badcfe' '10325476'
r,←inp̃16,~16±̃pn
r←ϕr

{r}←n Loop state;x;a;b;c;d;t;y;i
A called in this fashion {md5}←Loop/PaddingBits 'message'
A PaddingBits returns 1 row per word, processing 16 words at a time
a b c d←state
x←n indx Order ϑ           A put message block in encoding sequence
i←4ϑ4 16ρ(ω-⌈IO)164
y←4ϑ(64ρ0 3 2 1),x,md5Constants ϑ A md5Constants set up S11, S12 etc.
a b c d←F md5Round/(y indx 1>i),ca b c d      A Round 1
a b c d←G md5Round/(y indx 2>i),ca b c d      A Round 2
a b c d←H md5Round/(y indx 3>i),ca b c d      A Round 3
a b c d←I md5Round/(y indx 4>i),ca b c d      A Round 4
r←a b c d←state

md5Round←{a b c d←α[1]ϕω
  x s ac←14α
  a←b+21sϕ(32ρ2)τa+x+ac+αα b c d A this is an operator αα is F G H I
  (->α)ϕa b c d
}

```

Tiny Encryption Algorithm (TEA)

```

r←k Encrypt v;v0;v1;k0;k1;k2;k3;sum;delta;i;b
A 128-bit key working on 2x4 bytes of data           A Decrypt
A The Tiny Encryption Algorithm (TEA) by David Wheeler and Roger Needham
A TEA is a Feistel cipher with XOR and AND addition
A as the non-linear mixing functions.
A TEA takes 64 bits of data in v0 and v1, ( 2 x 4 bytes -> 8 ascii chars )
A and 128 bits of key in k0 k1 k2 and k3.(4 x 4 bytes->16 bytes)
v0 v1←v           A set up - k is the key
sum←0
A delta is chosen to be the Golden ratio
A ((5/4)1/2 - 1/2 ~ 0.618034) multiplied by 2*32
delta←hex'9e3779b9'           A a key schedule constant
:For i :In i32                 A basic cycle start
  A floor to force back to 32 bit arithmetic
  b←bits sum+k[⌈IO+21-2↑bits sum]
  v0←v0+[signbit 21(bits v1+21(v1 shiftleft 4)≠v1 shiftright 5)≠b
  A we keep on adding - this algorithm treats it as a sign bit...
  sum←+delta
  b←bits sum+k[⌈IO+21-2↑sum shiftright 11]
  v1←[v1+signbit 21(bits v0+21(v0 shiftleft 4)≠v0 shiftright 5)≠b
:EndFor                       A end cycle
r←v0 v1

```

I leave Decrypt as an exercise for the reader.

So what can one use it for?

A mention must be made of *Alissa*: the old Dfns mailing list was built by Konrad Hoesle-Kienzlen on the `aplmail` workspace, conforming to the mailing list conventions.

Mail filtering:

A mass email reader. Analyses the headers to separate email into groups according to sender. Fairly crude textual analysis breaks emails into interesting and less interesting lists. Using Dyalog's OCX class thus, where `ax` is an existing OCX class:

```
{r}←Fetch ax
:While ax.ReadyState≠4
    □DL 0.1
:EndWhile
    A innerText isn't enough, not all text is preserved if frames are used
    A so innerHTML & parse it out - again
r←ax.Document.body.innerText A text of this page, ignoring all HTML tags
    A remember this is just the body, not the complete html
r,←ax.Document.body.innerHTML A HTML of page

ax←Make n;Form;OCX;f;w;o
(f←'Form')□WC'form'('visible' 0)
(o←f,'.OCX')□WC'ocxclass' 'Microsoft Web Browser'
(w←f,'.ActiveX')□WC o
ax←xw

{ax}←http Set ax;count;true;r
r←'http://'
ax.Navigate2 http,~r/~r#http~pr
```

and then analysing these for keywords too.

Remote data entry

One of HMW's clients employs a number of travelling salesmen who must report figures back to head office. They can use a form on a web page, but very often they cannot access the Internet when they wish to enter the data. A simple application on their laptops creates emails which can be sent later and gathered up by an APL process which handles a queue of update requests.

Each email consists of a plain text message with a set of key=value pairs in the mail body: the data isn't particularly sensitive, but it is "signed" with a TEA encrypted MD5 hash to ensure that the information is not corrupt and has not been tampered with.

Delivering updates

In a simpler fashion, another client has the need to distribute code and data updates to clients but cannot use a direct "real time" interface. A GUI is provided to the support team which emails them to select a file from within the system and generate an email to a specified client. This email has a subject line which contains the name of the file and date of the release, a simple message body (typed by the team member), plus an encrypted signature which is used to verify the message and attachment and to which the compressed file is attached.

At the remote client end a "Receive update" process reads the email, decrypts the signature and uses it to check the header and file attachment. The old version of the file is archived and the new copy is unzipped and renamed into its new location, based on the information in the subject line. The email is then deleted.

"Split WS"

Harking back to the days of I.P. Sharp; PC No. 1 is running a workspace with these functions in it:

1. It □SAVES itself.
2. It reads the saved workspace as a native file.
3. Base 64 encodes this, attaches it to an email which is sent to an email account.
4. Finally it "dies", terminating local processing.

Sometime later on a PC far, far away another workspace running these functions receives an email:

1. It detaches the attachment, Base 64 decodes it and writes it out as a native file.
2. It □LOADS the resultant workspace and...
3. The task carries on from where it left off on the other PC.

Conclusion

This article and a sample Dyalog workspace are published on HMW Computing's website.

References

1. 7-zip: Main Project page <http://www.7-zip.org/>
2. APL Wiki MD5: Message Digest on the APL wiki
<http://aplwiki.com/MessageDigestHash>
3. APLmail: The `aplmail` workspace was included as part of a standard Dyalog distribution. It is probably still available upon request.
4. Hash Functions: a Wikipedia article on hash algorithms
http://en.wikipedia.org/wiki/Cryptographic_hash_function

Semantic arrays

Stephen Taylor (sjt@5jt.com)

APL functions are presented to support 'semantic indexing'. These functions are modelled on features of the K language.

An array is a map between its indexes and its values. The indexes of APL arrays are integers and represent position in the array. Where the elements are homegenous, this is congenial. But where elements represent different kinds of things, it is rarely helpful to indicate them by position.

Indeed, Cannon's Canon [1] deprecates using numerical constants to indicate anything but numbers. For example, while the 3 in +3 indicates nothing but a number by which something is to be increased, +customer[3] is deprecated if it requires one to remember that the third element of customer indicates age. If one means *age* then one wants a way to say so.

Call the mapping of numbers to values ordinal mapping and the mapping of character strings to values semantic mapping.

The workspace's global symbol table provides semantic mapping but, by definition, there can be only one global symbol table – and we frequently need multiple maps. Dyalog's namespace provides multiple semantic mappings very nicely through 'local' symbol tables, eg

```
customer.age←38
```

JavaScript and PHP provide semantic mapping through objects, collections of name/value pairs.

```
customer = {name: Jane Doe, age:38}
```

JavaScript also supports character strings as array indexes, allowing eg

```
customer['age'] = 38
```

The K language provides semantic mapping through symbols, which behave much like character strings, eg

```
customer: `name`age!("Jane Doe";38)
customer[ `age]
38
customer@ `age `name
(38;"Jane Doe")
```

Without the use of Dyalog namespaces, I sought convenient representations using more generic APL. I was inspired particularly by K's index function @. Emulating this in APL neatly avoided any need for special syntax. This article presents what I devised, much as it is used in the application on which I work.

Dictionaries and tables

Call a semantic array of rank 1 a dictionary. It is a pairing of names and values. Represent it in APL by a vector of the values, prefixed by the enclosed vector of their names. Thus, ↑dict has the same length as 1↓dict. [2]

```
]display French ← (c'cow' 'dog' 'cat') , 'vache' 'chien' 'chat'
```

Call a semantic array of rank 2 a table. It is a pairing of names and vectors. All the vectors have the same length: the depth of (number of rows in) the table. Represent it in APL by a nested matrix. The first row of the matrix contains the column names.

```
Δ←('cow' 'vache' 'Kuh')('dog' 'chien' 'Hund')('cat' 'chat' 'Katze')
[]←tbl←'English' 'French' 'German' ⍤ Δ
English French German
cow      vache  Kuh
dog      chien Hund
cat      chat  Katze
```

Happily, the default display of a matrix is exactly what one would want.

Clearly, the matrix can be seen as a special case of the dictionary, in which all the values are vectors of the same length. It is trivial to switch between forms:

```
]display dict ← (cTbl[1;]), []SPLIT[1] 1 0↓ tbl
```

The at function

Tables and dictionaries are in the left domain of the `at` function. The right argument is either a single key:

```
French at 'dog'
chien
German at 'dog'
Hund
tbl at 'French'
vache chien chat
```

or an array of keys:

```
French at 'dog' 'cow'
chien vache
German at 'dog' 'cow'
Hund Kuh
tbl at 'English' 'German'
cow dog cat Kuh Hund Katze
dict at 'English' 'German'
cow dog cat Kuh Hund Katze
```

In our application we have large parameter sets known as bases. Their definitions vary slightly from one version of the application to the next, so we need to manage that variation during the upgrade.

Suppose we have a table `basestable` that contains columns `V8` and `V9`. Both columns contain bases, represented as dictionaries. The vector `parameters` lists names defined in the dictionaries. We can tabulate parameter values from the bases:

```
(basestable at 'V8') ◦.at parameters
(basestable at 'V9') ◦.at parameters
```

`at` can also take an array of names as its right argument, permitting, for example:

```
basis+(c'MAGEDIF' 'MAGEDIFT' 'FAGEDIF' 'FAGEDIFT') , '5' ' ' ' ' 'diffs.csv'
]display basis at 'MF' ◦., 'AGEDIF' 'AGEDIFT'
```

```

┌───┐
│ .e. │
│ 5 | | │
│ - | | │
│ .e. → │
│ | | |diffs.csv| │
└───┘
ε
```

The functions pop and push

The functions `pop` and `push` are syntactic sugar for constructing and parsing dictionaries. For example:


```

]display French ← 'cow' 'dog' 'cat' push 'vache' 'chien' 'chat'
┌───┐
├───┬───┬───┬───┬───┬───┐
│   │   │   │   │   │   │
│→ │   │   │   │   │   │   │
│├───┬───┬───┬───┬───┬───┤
││ cow│dog│cat│vache│chien│chat│
│├───┬───┬───┬───┬───┬───┤
││   │   │   │   │   │   │
│├───┬───┬───┬───┬───┬───┤
│ε├───┬───┬───┬───┬───┬───┤
│ε├───┬───┬───┬───┬───┬───┤
└───┴───┴───┴───┴───┴───┘
      (keys vals)+pop French
      keys
      cow dog cat
      vals
      vache chien chat

```

From this we can easily display a dictionary:

```

↑,[1.5]/pop french
cow vache
sheep mouton
cat chat

```

The push function is overloaded: it can be used dyadically as above or monadically on an argument with 2 elements. This makes monadic push and pop – when applied to a dictionary – inverses of each other, so that $\text{French} \equiv \text{push pop French}$.

The dictionary can thus be flipped, making its values keys and its keys values:

```

French at 'dog'
chien
(pushpop French) at 'chien'
dog

```

We can also make a French-German dictionary from the polyglot word table:

```

(push tbl at 'French' 'German') at 'chien'
Hund

```

push and pop also give us convenient ways to turn a table into a vector of dictionaries:

```

(hdr cols) ← pop [SPLIT tbl
dics ← (hdr) push" cols
(dics) at 'English' 'French'
cow vache

```

The map function

We often want to look up values in one column of a table and read the corresponding values from another column. To find the German forms of some French words:

```
(push tbl at 'French' 'German') at 'chat' 'vache'
Katze Kuh
```

The function `map` provides a little syntactic sugar for this:

```
(tbl at 'French' 'German') map 'chat' 'vache'
Katze Kuh
```

The `spin` function

Converting rows to or from columns is helped by the `spin` function, which is its own inverse:

```
]display tbl at 'French' 'German'
|-----|-----|-----|-----|
|.-----|.-----|.-----|.-----|.
| |vache| |chien| |chat| | |Kuh| |Hund| |Katze| |
|-----|-----|-----|-----|
| ε-----|-----|-----|-----|
| ε-----|-----|-----|-----|

]display spin tbl at 'French' 'German'
|-----|-----|-----|-----|
|.-----|.-----|.-----|.-----|.
| |vache| |Kuh| | |chien| |Hund| | |chat| |Katze| |
|-----|-----|-----|-----|
| ε-----|-----|-----|-----|
| ε-----|-----|-----|-----|

Δ ≡ spin spin Δ ← tbl at 'French' 'German'
```

Thus to loop through the rows of a table:

```
:for en fr de :in spin tbl at 'English' 'French' 'German'
  []←'English: ',en,'; French: ',fr,'; German: ',de
:endfor
```

The above[3] becomes awkward where many columns require many local 'loop variables'. Instead one could loop through table rows as dictionaries:

```
:for word :in Δ[1] push" 1+Δ+[]SPLIT tbl
  []←'English: ',word at 'English'
:endfor
```

Selecting from tables

To return a table containing only selected columns, the function `select`.

To return a table with selected rows, the `for` function. Its right argument is either a boolean vector whose length is the table depth; or a vector of which the first element is a column name and subsequent elements are values to be matched.

```

tbl for 'English' 'cow' 'cat'
English French German
cow vache Kuh
cat chat Katze
tbl for 'c' = ↑tbl at 'French'
English French German
dog chien Hund
cat chat Katze
(tbl for 'English' 'cow' 'cat') select 'French' 'German'
French German
vache Kuh
chat Katze

```

The amend function

To add a new element, or replace an element of a table or dictionary with a new value, the amend function.

```

(French amend 'cow' 'la vache') at 'cow'
la vache
tbl amend 'Danish' ('kuh' 'hund' 'katte')
English French German Danish
cow vache Kuh kuh
dog chien Hund hund
cat chat Katze katte
tbl amend 'French' ('la vache' 'le chien' 'le chat')
English German French
cow Kuh la vache
dog Hund le chien
cat Katze le chat

```

Conclusion

These ‘syntactic sugar’ functions have been invaluable in simplifying the representation of logic in our application, and keeping our code compliant with Cannon’s Canon.

Notes

1. *Cannon’s Canon* is so dubbed by me because it was introduced to me by Ray Cannon.
2. The monadic function represented by \uparrow varies between APL dialects and between ‘migration levels’ in at least one interpreter. \uparrow here represents the first function, that returns the first element of its argument. The monadic function mix is here represented by \triangleright . The cut function that splits an array along its first axis (eg a table into rows), in some interpreters represented by the \downarrow glyph, is here represented as \square SPLIT.
3. For interpreters in which the `:for` loop as not been so extended:

```
:for Δ :in spin tbl at 'English' 'French' 'German' ◊ (en fr de)+Δ
```

Appendix – Function definitions

```

▽ Z←L amend R;newhds;newcols;seln;hds
[1] A sets one or more elements of semantic array L
[2] A defined for table and dictionary L
[3]
[4] A parse R
[5] :if 2=↑pR
[6] :andif (≡↑R)∈0 1
[7] (newhds newcols)+,"c"R A tbl amend 'col4' foo
[8] :elseif ^/(≡↑R)∈0 1
[9] :andif (↑"p"R)^.=2
[10] (newhds newcols)+spin R A tbl amend('col4' foo)('col5' bar)
[11] :endif
[12]
[13] :select ↑ppL
[14] :case 1 ◊ hds+↑L A dict
[15] :case 2 ◊ hds+L[[]IO;] A matrix
[16] :endselect
[17]
[18] seln←hdsenewhds
[19]
[20] :select ↑ppL
[21] :case 1 ◊ Z+push(seln/"pop L), "newhds newcols A dict
[22] :case 2 ◊ Z+(seln/L), newhds; ◊newcols A matrix
[23] :endselect
[24]
▽

```

```

▽ Z←L at cols;hdr;vals;DEFAULT
[1] A select cols from table, represented either as
[2] A - matrix with header row
[3] A - dictionary: keys val val val...
[4] DEFAULT+" " A for undefined values (where allowed)
[5] :if 2=ppL A table: error if cols not found
[6] (hdr vals)+(L[[]IO;])([]SPLIT[[]IO]1 0\L)
[7] :else A dictionary
[8] (hdr vals)+pop L
[9] vals+vals,cDEFAULT A default value if cols not found
[10] :endif
[11] :if 1=≡cols
[12] Z+(hdr<cols)>vals
[13] :else
[14] Z+vals[hdr<cols]
[15] :endif
▽

```

```

▽ Z←tbl for cv;col;vals;msk;hdr;bdy
[1] A table/dictionary syntax: tbl for 'col1' val1 val2 ...
[2] A returns a table or dictionary according to tbl
[3] A or cv may be a boolean mask
[4] :if 1=≡cv ◊ :andif 1=↑ppcv ◊ :andif ^/cv∈0 1
[5] msk←cv
[6] :else
[7] (col vals)+pop cv
[8] msk←(tbl at col)evals
[9] :endif
[10]
[11] :select ↑pptbl
[12] :case 1 A table
[13] (hdr bdy)+pop tbl
[14] Z+hdr push msk/"bdy
[15] :case 2 A dictionary
[16] Z+(1,msk)≠tbl

```

```
[17] :endselect
      ▽
```

```
      ▽ Z←L map R;to;from
[1]   (from to)+L
[2]   Z←to[from;R]
      ▽
```

```
      ▽ Z←pop R
[1]   Z←(↑R)(1↓R)
      ▽
```

```
      ▽ Z←L push R;A;B
[1]   A syntax sugar:
[2]   A 'abc' 'def' 'ghi'
[3]   A ↔ 'abc' push 'def' 'ghi'
[4]   A ↔ push ('abc') ('def' 'ghi')
[5]   :if 2=NC 'L'
[6]     Z←(←L),R
[7]   :else
[8]     (A B)←R
[9]     Z←(←A),B
[10]  :endif
      ▽
```

```
      ▽ tbl←tbl select cols
[1]   A select cols from tbl
[2]   tbl←(tbl[IO;]ecols)/tbl
      ▽
```

```
      ▽ Z←spin R
[1]   Z←SPLIT$R
      ▽
```

Compiling APL to JavaScript

Nick Nickolov (nick.nickolov@gmail.com)

This article is about github.com/ngn/apl[1], an APL to JavaScript compiler written in CoffeeScript. It gives an overview of the project and its dialect of APL, then proceeds to describe three major implementational obstacles and the design decisions taken to overcome them.

Introduction

The list of languages that compile to JavaScript is growing steadily for a good reason: almost every modern consumer device can run a browser with a JavaScript interpreter in it. And for the server there's NodeJS. As a language, JavaScript has its subset of good parts and if you restrict your coding to them, it's actually a very decent language. The execution model is reminiscent of Lisp with its lambdas and lexical closures, despite the superficially C-like syntax. As a matter of fact, getting rid of those curly braces, semicolons, and other noise is easy thanks to CoffeeScript, which gives a comfortable layer of syntactic sugar over raw JavaScript.

I started an open source project for an APL compiler targeting JavaScript. Though incomplete and of poor performance, it's good enough for experimenting with short programs, such as the Game of Life or the N Queens problem.

I made a conscious decision to deviate from the APL tradition in several ways.

Function definition

Lambdas are supported (function literals enclosed in curly braces, a.k.a. dfns) but the del (∇) syntax for function definition is not. So, instead of

```

 $\nabla$ R←A f B
    R←...
 $\nabla$ 

```

one is forced to write

```
f←{...}
```

Variable scoping

Scoping is always lexical. This means that the two occurrences of `a` below

```
f←{a←123}
g←{a←456}
h←{...}
```

are different variables and neither of them is accessible from within `h`, regardless of the order and nesting of `f`, `g`, `h` invocations.

A variable is considered to belong to the outermost ancestor scope where it is mentioned. So, the first two occurrences of `a` below are the same variable and the third `a` is distinct from them:

```
f←{
  a←123
  g←{a←456}
}
h←{a←789}
```

Note that assignment doesn't necessarily create a local variable. If the variable is already present in any enclosing scope, assignment will set *that* variable.

A variable must not be used before it is assigned a value. The compiler can detect most violations of that:

```
□←ι a      A compiler error
a←5
```

Phrasal forms

Expressions consisting of a sequence of two or three functions are said to form a hook or a fork respectively and the result is a new function defined as follows:

```
(fg)ω ↔ ωfgω
α(fg)ω ↔ αfgω
(fgh)ω ↔ (fω)g(hω)
α(fgh)ω ↔ (αfω)g(αhω)
```

Examples usages of phrasal forms are the arithmetic mean written as $+ / \div \rho$ and continued fraction evaluation as $(+\div)\backslash$.

Non-privileged primitives

Primitives are just like any other variable, only as if implicitly defined in the root scope. As a side effect, they can be overridden:

```

3@4      A returns 1.2618595071429148
@←{α+2×ω}
3@4      A returns 11

```

One use of this fact might be to introduce comparison tolerance, which isn't supported by default:

```

□CT+1e-13
≈←{( |α-ω) ≤ □CT × ( |α) ⌈ |ω}

```

It's okay to use single non-alphabetic characters as variable names, so the language can be augmented as one sees fit:

```

π←0.1
√←{ω*.5}
Σ←+/
½←{ω÷2}
±←+, -
€←{exchangeRate×ω}
≈←{( |α-ω) ≤ □CT × ( |α) ⌈ |ω}

```

Index origin fixed to 0

I took a side in this Holy War.

```

i3      A returns 0 1 2
□IO=0   A ok, no effect
□IO=1   A error

```

Line terminator ambiguity

A line terminator is treated as a statement separator if it occurs within {} or at the topmost level, but is considered whitespace if it's inside () or [].

Example:

```

glider←{
  a←3 3ρ(
    0 1 0
    0 0 1
    1 1 1
  )
  ω ωta
}

```

Absence of control structures

Control structures such as :If ... :Else ... :EndIf are not supported but can be emulated using lambdas, guards, and APL primitives. For instance


```
:If x
  y
:Else
  z
:EndIf
```

can be replaced with

```
{x:y◊z}0
```

and

```
:While x
  y
:EndWhile
```

with

```
{~x:1◊y◊0}*- 0
```

The rest of this article describes three challenging problems which I came across during the implementation of APL. Two of them I solved and the third one is yet to be taken care of.

Parsing APL expressions

Consider the APL expression

```
a b c
```

Depending on context, it could mean different things:

| | | | | | | | |
|------|---|------|---|------|---|-------|-----------------------------------|
| a←0 | ◊ | b←{} | ◊ | c←0 | ◊ | a b c | A a dyadic application of b |
| a←{} | ◊ | b←{} | ◊ | c←0 | ◊ | a b c | A two nested monadic applications |
| a←0 | ◊ | b←0 | ◊ | c←{} | ◊ | a b c | A compiler error |
| a←0 | ◊ | b←0 | ◊ | c←0 | ◊ | a b c | A construction of a new vector |
| a←{} | ◊ | b←{} | ◊ | c←{} | ◊ | a b c | A a fork |

So, it seems the semantics depend on what a, b, and c represent at runtime, verbs or nouns, and building a complete AST is impossible before we have that knowledge.

Fortunately, there exists a way around this at the cost of imposing a small restriction on variable usage. The restriction is as follows:

- Once we assign a noun to a variable, we are only allowed to assign nouns to it later.
- Once we assign a verb to a variable, we are only allowed to assign verbs to it later.

In other words, variables must preserve their lexical category.

Example:

```
x ← 0
x ← 1      A OK
x ← {}     A Compiler error

f ← {}
f ← +/     A OK
f ← 2 3    A Compiler error
```

This restriction separates the worlds of nouns and verbs at compile time. Every expression can then be inferred one of the two lexical categories so the compiler know what JavaScript code to render for it.

Representation of n-dimensional arrays

During the evolution of the project I went through a couple of different implementations of the basic APL data structure.

Nested representation

My first take on this was to use n nested levels of JavaScript arrays to represent an n-dimensional APL array. For instance:

```
APL: 2 3p16
JS: [[0, 1, 2], [3, 4, 5]]
```

Thus, subscripting maps nicely between the two languages:

```
APL: a[i;j;k]
JS: a[i][j][k]
```

but many of the primitives' implementations had to involve recursion or multiple loops, which apart from being inefficient was complicating matters unnecessarily, so I transitioned to a flat representation.

Flat representation

A flat representation is a single JavaScript array which contains all items of the APL array in lexicographic order of their indices (a.k.a. ravel order). Additional meta-information about the shape must be stored, too.

```
APL: 2 3p16
JS:
  var a = [0, 1, 2, 3, 4, 5];
  a.shape = [2, 3];
```

This way, indexing is not as plain as before:

```

APL: a[i;j;k]
JS: a[
    i * a.shape[0] * a.shape[1] +
    j * a.shape[0] +
    k
]

```

but it's still worth it, as the implementation of most primitives gets simpler and more efficient.

There turns out to be an even better data structure which I was lucky to come across.

Strided representation

In a strided representation, an APL array is a record of:

`data`: the elements, not necessarily in ravel order

`shape`: as usual, an integer array

`stride`: an integer for each axis, indicating how many items of `data` we have to skip over in order to move to an adjacent cell along that axis. Strides can be positive, negative, or zero.

`offset`: the position in `data` where element `[0;0;...;0]` is located

Example:

```

APL: 2 3⍶6
JS:
{
  data: [0, 1, 2, 3, 4, 5],
  shape: [2, 3],
  stride: [3, 1],
  offset: 0
}

```

and this is another representation of the same APL array

```

JS:
{
  data: [2, 1, 0, 5, 4, 3],
  shape: [2, 3],
  stride: [3, -1],
  offset: 2
}

```

Indexing is relatively straightforward:

```

APL: a[i;j;k]
JS:
data[
  offset
  + i * stride[0]
  + j * stride[1]
]

```

```
    + k * stride[2]
  ]
```

The advantage of this strided representation becomes apparent when working with large volumes of data. Functions like `transpose` ($\Phi\omega$), `reverse` ($\phi\omega$), or `drop` ($\alpha\downarrow\omega$) can reuse the `data` array and only care to give a new `shape`, `stride`, and `offset` to their result. A reshaped scalar, e.g. `1000000ρ0`, can only occupy a constant amount of memory, exploiting the fact that strides can be 0.

Continuation-passing style

The third problem, one which I haven't solved in code yet, arose from the way NodeJS does I/O. While in some cases NodeJS allows you to do

```
var content = fs.readFileSync('a.txt');
```

it strongly encourages the so-called continuation-passing style (CPS), demanding a callback:

```
fs.readFile('a.txt', function (content) {
  // ... Invoked later, after content becomes available
});
```

It's not only that. In a browser environment, if we want to be properly mimicking I/O, we again need CPS. Of course, we can use the blocking functions `prompt()` and `alert()`, but most users find those annoying because they prevent them from using other parts of the UI while entering text.

This is what happens when reading from `□`, for instance:

1. APL code tries to get the value of variable `□`.
2. The user is prompted to type something in an `<input/>`.
3. APL execution is suspended while the user is typing, looking up help information, using the on-screen keyboard, etc.
4. The user presses enter.
5. APL execution is resumed and the value obtained from `□` is processed.

The trouble is in step 3. How do we suspend execution to resume it later? We might as well try to make the compiler output CPS JavaScript code, but that appears get complicated rather quickly.

If only JavaScript had a `call-with-current-continuation` primitive like Scheme, we would be able to do just this:

```
var content = callcc(fs.readFile.bind(fs, 'a.txt'));
```

but alas, it doesn't. We can amend that through implementing a virtual machine (VM) in JavaScript with a `callcc` instruction that suspends execution, captures the current state with all its stack and variable bindings and presents it as a callable *resume* function. The compiler can then produce some intermediate bytecode for the VM instead of pure JavaScript.

What if we also expose `callcc` as an APL primitive? This would be so powerful that it could be used to implement an exception mechanism, break/continue, coroutines, generators, Prolog-style backtracking, etc in APL itself. Therefore, adding a VM looks like a lucrative prospect for `ngn/apl`.

1. `ngnAPL` <https://github.com/ngn/apl>

Boolean Reductions

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

A look at the boolean vector reductions - choosing the right one for the job in hand; and some potential speed-ups.

Some boolean scans

Most experienced APLers could list a number of boolean scans, some more than others, that they can include in algorithms, confident that they will fulfill a particular task. Some of these scans would not necessarily seem intuitive to a newcomer, even one conversant with the primitive itself.

Less-than and *not-equals* are possibly the most ubiquitous of these.

The less-than scan of a boolean vector turns off all ones in the vector except the first.

The not-equals scan of a boolean vector, reading from the left, flips the corresponding and subsequent items on or off wherever the argument vector is on.

Once we have learned what these do we tend not to question how they work. Thinking of them in similar terms to the apparent left-to-right processing of *and* and *or* scans, the not-equal scan is fairly clear. Not so less-than. This is because not-equals is, and less-than is not, associative over the boolean domain. We can only understand non-associative scans in terms of the individual reductions that constitute their results. The less-than reduction of a boolean vector is true only if the final element is the one-and-only one. Paradoxical except for the fact that the definition of scan demands it, the less-than scan gives the *first* one precisely because the less-than reduction requires only the *last*.

Reductions

I don't know how many scans are either particularly useful or easily explicable. But I'd guess that not many of my readers can give natural language descriptions of which boolean vectors will return true for more than a very few of the boolean *reductions*.

It may seem that there should be no good answer to this question. For each function infinitely many boolean vectors resolve to one and infinitely many resolve to zero. Is

it reasonable to assume a recognisable pattern to distinguish the two sets? Think of and-reduction as *all* and or-reduction as *any* and you can deduce two trivial examples.

I asked myself this question nearly two decades ago and after much experiment found answers to fourteen of the sixteen candidates.

If the mention of sixteen in the previous statement doesn't surprise you then please skip to the next section - Preliminary results.

Sixteen boolean functions?

APL traditionally implements either four, six or ten primitive *boolean* or *logical* functions depending on your point-of-view. The four undisputed are *and*, *or*, *nand* and *nor*. Even two of these now have additional, non-boolean definitions. We can add *equals* and *not-equals* by restricting their domains and *less-than*, *less-than-or-equals*, *greater-than-or-equals* and *greater-than* if we are willing to accept that $p > q$ can be read as *p and-not q* instead of, and having a different meaning from, *p greater-than q*. But this still only gives us ten.

A boolean or logical dyad takes two booleans and returns one. A single function must return a single one or zero for each boolean pair, of which there are four: (0 0)(0 1)(1 0)(1 1). Representing each of these these resultant four item booleans as a four-digit binary and there being sixteen of those: 0000 0001 0010 ... 1111; there must be sixteen functions to produce them.

Here they are in the order dictated by their results as above:

| b | n | f |
|------|----|---|
| 0000 | 0 | |
| 0001 | 1 | ^ |
| 0010 | 2 | > |
| 0011 | 3 | |
| 0100 | 4 | < |
| 0101 | 5 | |
| 0110 | 6 | ≠ |
| 0111 | 7 | ∨ |
| 1000 | 8 | ∩ |
| 1001 | 9 | = |
| 1010 | 10 | |
| 1011 | 11 | ≥ |
| 1100 | 12 | |
| 1101 | 13 | ≤ |
| 1110 | 14 | ⋈ |
| 1111 | 15 | |

The six missing, those not implemented as primitives, could be represented in the notation implemented as direct functions (dfns) in Dyalog APL as: $\{0\}$ $\{\alpha\}$ $\{\omega\}$ $\{\sim\omega\}$ $\{\sim\alpha\}$ $\{1\}$. But these only work for simple scalar arguments as they take no cognizance of the left, the right or, in two cases, either argument. We need to reference both of them to produce an array of the required structure valued as the identity or negation of one boolean function that we apply to the other argument.

| b | n | f | |
|------|----|----------------------------------|-------------------------------------|
| 0000 | 0 | $\{\alpha\wedge 0\wedge\omega\}$ | A false |
| 0011 | 3 | $\{\alpha\wedge 1\vee\omega\}$ | A left (left notwithstanding right) |
| 0101 | 5 | $\{\omega\wedge 1\vee\alpha\}$ | A right (left nevertheless right) |
| 1010 | 10 | $\{\omega=0\wedge\alpha\}$ | A negate right |
| 1100 | 12 | $\{\alpha=0\wedge\omega\}$ | A negate left |
| 1111 | 15 | $\{\alpha\vee 1\vee\omega\}$ | A true |

Preliminary results

In the following list the comment describes the essential character of logical vector ω when $1 < \rho\omega$ and the result of f/ω is true.

| b | n | f | |
|------|----|----------------------------------|--------------------------------|
| 0000 | 0 | $\{\alpha\wedge 0\wedge\omega\}$ | A never |
| 0001 | 1 | \wedge | A all ones |
| 0010 | 2 | $>$ | A odd leading ones |
| 0011 | 3 | $\{\alpha\wedge 1\vee\omega\}$ | A first is one |
| 0100 | 4 | $<$ | A last is the only one |
| 0101 | 5 | $\{\omega\wedge 1\vee\alpha\}$ | A last is one |
| 0110 | 6 | \neq | A odd ones |
| 0111 | 7 | \vee | A at least one one |
| 1000 | 8 | $\tilde{\vee}$ | |
| 1001 | 9 | $=$ | A even zeros |
| 1010 | 10 | $\{\omega=0\wedge\alpha\}$ | A last is parity of the length |
| 1011 | 11 | \geq | A even leading ones |
| 1100 | 12 | $\{\alpha=0\wedge\omega\}$ | A first is zero |
| 1101 | 13 | \leq | A last is not the only zero |
| 1110 | 14 | $\tilde{\leq}$ | |
| 1111 | 15 | $\{\alpha\vee 1\vee\omega\}$ | A always |

Those missing descriptions, *nand* and *nor*, caused this paper to be delayed by a period approaching two decades.

At "Iverson College 2012" at Cambridge we were discussing Roger Hui's presentation "How to write an elegant computer program" and moved on to the subject of the semantics of the boolean functions and how many of them can be used in unconventional ways to increase speed and elegance. The reference to $p \rightarrow q$ being equivalent to $p \wedge \sim q$ above being one such. And of course Simon Garland's memorable: "To be \geq to be, that is the question." being an obvious improvement on the original.

In conversation afterwards about the topic of this paper Roger mentioned that he had recently come up with efficient definitions for the missing pair.

These were in the form of an operator that implemented the four reductions: $\leq/ \geq/ \tilde{/} \tilde{\wedge}/$. But he had also made an effort to verbalise them. I include only those for my missing pair.

After many false starts my most concise wording of them is:

$\tilde{/}$ implies odd leading zeros else the last is the only one.

$\tilde{\wedge}/$ implies even leading ones else the last is the only zero.

They can be summed up in these two rank- and origin-independent direct functions:

```

 $\tilde{/} \leftrightarrow \{(+/\wedge\sim\omega)\{(2|\alpha)=\alpha<\omega-1\}\}\wedge/\rho\omega\}$ 
 $\tilde{\wedge}/ \leftrightarrow \{(+/\wedge\wedge\omega)\{(2|\alpha)\neq\alpha<\omega-1\}\}\wedge/\rho\omega\}$ 
    
```

or first-dimension oriented as:

```

 $\tilde{/}\omega \leftrightarrow \{(+/\wedge\sim\omega)\{(2|\alpha)=\alpha<\omega-1\}\}\wedge/\rho\omega\}$ 
 $\tilde{\wedge}/\omega \leftrightarrow \{(+/\wedge\wedge\omega)\{(2|\alpha)\neq\alpha<\omega-1\}\}\wedge/\rho\omega\}$ 
    
```

[$\wedge/$ and \wedge (and \wedge and \wedge) are idioms for selecting the first and last sub-arrays from an array]

see Appendix: Traditional function equivalents if these are incomprehensible to you.

So here they are in context:

| b | n | f | |
|------|----|----------------------------------|--|
| 0000 | 0 | $\{\alpha\wedge 0\wedge\omega\}$ | A never |
| 0001 | 1 | \wedge | A all ones |
| 0010 | 2 | $>$ | A odd leading ones |
| 0011 | 3 | $\{\alpha\wedge 1\vee\omega\}$ | A first is one |
| 0100 | 4 | $<$ | A last is the only one |
| 0101 | 5 | $\{\omega\wedge 1\vee\alpha\}$ | A last is one |
| 0110 | 6 | \neq | A odd ones |
| 0111 | 7 | \vee | A at least one one |
| 1000 | 8 | $\tilde{\vee}$ | A odd leading zeros else the last is the only one |
| 1001 | 9 | $=$ | A even zeros |
| 1010 | 10 | $\{\omega=0\wedge\alpha\}$ | A last is parity of the length |
| 1011 | 11 | \geq | A even leading ones |
| 1100 | 12 | $\{\alpha=0\wedge\omega\}$ | A first is zero |
| 1101 | 13 | \leq | A last is not the only zero |
| 1110 | 14 | $\tilde{\wedge}$ | A even leading ones else the last is the only zero |
| 1111 | 15 | $\{\alpha\vee 1\vee\omega\}$ | A always |

All this might seem very esoteric but these can prove useful in very different contexts.

APL coders can produce simple expressions to validate the resulting pattern of a multiple boolean check.

Interpreter writers can convert the description into fast code written in their favourite compiled language. Possibly your own interpreter already benefits from this.

APL coders can anticipate their vendors efforts and produce fast work-arounds for slow implementations.

Appendix: Traditional function equivalents

for the \tilde{v} / variants:

```

▽ r←norRed w;n
[1] A {(+/^-\ω){(2|α)=α<ω-1}+ρω}
[2] n←+/\-\ω
[3] r←(2|n)=n<~1+0⊥ρw
▽

```

for the missing boolean primitives:

```

▽ r←a left w
[1] A {α^ω=ω}
[2] r←a^w=w
▽

```

The others can easily be inferred as they differ to the same extent as their counterparts.

APLUnit - An APL unit test library

Gianfranco Alongi (gianfranco.alongi@gmail.com)

What is test first programming?

Test first programming, or TDD (Test Driven Development) [1] as it is also called, is a popular approach for development where you design your system in an incremental fashion by writing a test, and then adding the code for this test to pass.

One way to look at it, is that we figure out how we would like our code to work, set up a test so that we know when we are done, and then making it pass our test, thus making our code work as we intended it to.

Yet another way to look at it: we are constantly capturing our assumptions about our code, and documenting it with a language that allows the documentation to check it's own correctness.

And finally, I like to think of it as rock climbing. I start out at the base of the mountain, with a pretty good idea of where I want to go, and which path I'll take. So, I set out to reach the first point I want (I write my first test), and I go there (I make the test pass). I now have a nail in the rock and my safety line through the loop. I set out to the second goal, (I write my second test) and I make it there (I make it pass). Just like in rock climbing, it's dangerous to make it too far between the safe points.

The idea is that you should spend at most about 15 minutes per step. If the test you wrote was too hard then remove it; this would be the equivalent of admitting that maybe you should take another path up to the next stop. Go back, ponder your strategy and set out again (remove the test, write another one - a smaller step - and make it pass).

The way to do TDD is to always write a test first and then write code to make that test pass. As long as that test fails; you have a problem to think about. It follows the principle of separation of concerns; only concern yourself with one problem at a time. Once your test passes, you know it works, and you can prove it! You run the test. You run the test often to feel secure and comfortable; and sometimes just because you like the ego-boost of seeing all the tests pass.

But only write the minimal amount of code needed to make the test pass!

A testing library for APL

As I like the incremental development model which is a result of TDD. I always approach languages with TDD. To do this, I keep things simple and look for libraries with a minimal footprint, rich feature set and helpful error reports that facilitate the error localization. Unfortunately I could not find such a library for Dyalog APL, so, I set out and rolled my own TDD library.

This became what is now *APLUnit* a Test framework with a small syntactical footprint and rich feature set. This has evolved through technical, borderline theological and philosophical discussions with Morten Kromberg (CTO @ Dyalog), who represented the APL community while I worked on the initial development and style of the library.

Installation

APLUnit is written as a Dyalog SALT script - installation is done by downloading the UT.dyalog file onto your system and loading it into the environment using

```
⊞SE.SALT.Load './UT.dyalog -target=#'
```

A Test and it's expectation

A test is any traditional (tradfn) or direct (dfn) function which ends with '_TEST' as part of the name. All tests shall set the expectation of the test into the UT namespace variable `expect` (match the following) or `nexpect` (do not match), as in the trivial examples below, written as a tradfn and a dfn.

```
▽ Z ← user_defined_function_TEST
  #.UT.expect ← 1 2 3
  Z ← ι 3
▽

dfn_TEST ← {
  #.UT.nexpect ← 1 2 3
  1 + ι3
}
```

Running tests

Once loaded - the `#.UT.run` command is available and capable of running any test which is defined. You may run a single test by naming it directly

```
#.UT.run 'user_defined_function_TEST'
```

You can run an array of tests

```
tests ← 'user_defined_function_TEST' 'dfn_TEST'
#.UT.run tests
```

You can run all tests in a particular SALT file (dyalog script)

```
#.UT.run './test/known_bugs.dyalog'
```

Finally you can also run all tests from test files in a directory. Test files are SALT dyalog scripts with a name ending in '_tests.dyalog'- this enforces a standardized format making it easy to locate test files.

```
#.UT.run './bugs_reported_by_customers/'
```

Result reporting

If the expected value matches the produced result of the test the test is said to have passed, and APLUnit prints this for you on the screen:

```
#.UT.run '#.bowling_tests.parse_strike_frame_TEST'
Passed 0 m 0 s 0 ms
```

If the test fails, (expect and actual result does not match, or `nexpect` matches the output of the test), a graphical display is printed to help you understand the cause of the failure.

```
#.UT.run '#.bowling_tests.parse_strike_frame_TEST'
FAILED: #.bowling_tests.parse_strike_frame_TEST
Expected
┌───────────┐
│ 10 ──> strike │
└───────────┘
Got
10
```

If you ran a collection of tests through the array, file or directory mechanic, you get a test report nicely written on the screen at the end of the test, and the same Pass/FAILED output for every test executed.

```
#.UT.run './test/'
Passed 0 m 0 s 0 ms
Passed 0 m 0 s 0 ms
.....
.....
Passed 0 m 0 s 0 ms
Passed 0 m 0 s 0 ms
Passed 0 m 0 s 0 ms
-----
./test/fixed_bugs_tests.dyalog tests
▲ Passed: 141
```

```

⊙ Crashed: 0
▼ Failed: 0
○ Runtime: 0 m 0 s 10 ms

```

Crashes

When APLUnit runs a test, the default behaviour is to trap all errors and continue execution until reporting is done. If you wish to change this behaviour - you can control the trapping by setting the (s)top (a)t (c)rash configuration variable in the UT namespace to either 1

```
#.UT.sac ← 1 A stop at crash
```

or

```
#.UT.sac ← 0 A trap and run on
```

0 is the default value. This is to ensure that bulk-running of several tests allows you to get a full report without manual intervention.

If you do set `#.UT.sac←1` you must remember to `→⌈LC` once you fix your test.

Example

For this example I demonstrate how to use the library on a dynamic toy problem with a made up customer.

Setup

First I need to load the test framework into the APL environment.

```
⌈⌈SE.SALT.Load 'UT.dyalog'
#.UT
```

Now I need two SALT files (`sensor.dyalog` and `sensor_tests.dyalog`) with empty namespaces

```
#.sensor
```

which will contain the functionality, and

```
#.sensor_tests
```

which contains the tests for my implementation.

Coding

The customer EvilCorp wants the system to censor all words which has a high enough match with blacklisted words, nasty words, such as 'pony', 'flower' and 'sun'.

Thus, we create an initial test for one of them - the 'pony' word. In order to achieve this I add the first test to the `#.censor_tests` namespace:

```
:NameSpace censor_tests

censor_pony_TEST←{
  #.UT.expect← 'little xxxx friends'
  'pony' #.censor.run 'little pony friends'
}

:EndNameSpace
```

Now we run the tests in the current directory. It will find the `censor_tests.dyalog` file and execute the tests. We should expect a failure:

```
#.UT.run './'

CRASHED: censor_pony_TEST
Expected
  little xxxx friends
Got
  VALUE ERROR
  censor_pony_TEST[2] 'pony'#.censor.run'little pony friends'
                        ^
-----
./censor_tests.dyalog tests
  ▲ Passed: 0
  ● Crashed: 1
  ▼ Failed: 0
  ○ Runtime: 0 m 0 s 20 ms
```

We can now write the implementation function.

```
run←{
  w←a
  t←w
  i←(wεt)/ιpt
  t[-1+(ιpw)+i]+(pw)ρ'x'
  t
}
```

And we can re-run the tests:

```
#.UT.run './'
Passed 0 m 0 s 0 ms
-----
./censor_tests.dyalog tests
  ▲ Passed: 1
  ● Crashed: 0
  ▼ Failed: 0
  ○ Runtime: 0 m 0 s 6 ms
```

Now - we can write the test for a list of several censored words.

```
censor_many_TEST←{
#.UT.expect←'xxxx in the xxxshine'
'pony' 'sun' #.censor.process 'pony in the sunshine'
}
```

Re-running the tests will cause our newest test to fail:

```
#.UT.run './'
CRASHED: censor_many_TEST
Expected
  → xxxx in the xxxshine
Got
  → VALUE ERROR
  | censor_many_TEST[2] 'pony' 'sun'#.censor.process'pony in the sunshine'
  | ^
Passed 0 m 0 s 1 ms
-----
./censor_tests.dyalog tests
  ▲ Passed: 1
  ● Crashed: 1
  ▼ Failed: 0
  ○ Runtime: 0 m 1 s ^979 ms
```

We can now write the implementation for this process function:

```
process←{>run/α, cω}
```

Finally we re-run the tests:

```
#.UT.run './'
Passed 0 m 0 s 0 ms
Passed 0 m 0 s 0 ms
-----
./censor_tests.dyalog tests
  ▲ Passed: 2
  ● Crashed: 0
  ▼ Failed: 0
  ○ Runtime: 0 m 0 s 8 ms
```

They all passed. Now we know that we are done with the implementation.

Since we have tests for correctness, we are free to work the code over (refactoring), as long as we run all the tests often.

This will be done now, I will simplify the run function a little. I will inline the index (i) and remove the temporary variable (w).

```
run←{
  t←w
  t[-1+(iρα)+(αεt)/iρt]+(ρα)ρ'x'
  t
}
```

After this change, I rerun the tests.

```
#.UT.run './'
Passed 0 m 0 s 0 ms
Passed 0 m 0 s 0 ms
-----
./censor_tests.dyalog tests
▲ Passed: 2
⊙ Crashed: 0
▼ Failed: 0
○ Runtime: 0 m 0 s 8 ms
```

All is good!

This approach of starting from the bottom up, is called the 'Chicago School of TDD'.

Now, I can focus on the next requirement.

Final code

The final version of the censoring program from censor.dyalog

```
:NameSpace censor
  process←{ >run/α,cω }
  run←{
    t←w
    t[-1+(iρα)+(αεt)/iρt]+(ρα)ρ'x'
    t
  }
:EndNameSpace
```

The tests from censor_tests.dyalog:

```
:NameSpace censor_tests
censor_pony_TEST←{
  #.UT.expect←'little xxxx friends'
  'pony'#.censor.run'little pony friends'
}
}
```

```
    censor_many_TEST+{  
        #.UT.expect+'little xxxx in the xxxshine'  
        'pony' 'sun'#.censor.process'little pony in the sunshine'  
    }  
  
    :EndNameSpace
```

References

1. http://en.wikipedia.org/wiki/Test-first_programming/

NFL Passer Rating

Brian Becker

I like football, both the American version and the version the rest of the world calls football but the Americans call “soccer”. One of the statistics in American football that’s frequently mentioned is the passer (or quarterback) rating. I’ve heard sportscasters say the maximum passer rating is 158.3 – a rather odd number if you ask me. This piqued my curiosity and I turned to the font of all knowledge – the internet, where I found an interesting entry in en.wikipedia.org[1] which gave the following formula:

$$a = \left(\frac{\text{COMP}}{\text{ATT}} - 0.3\right) \times 5$$

$$b = \left(\frac{\text{YARDS}}{\text{ATT}} - 3\right) \times .25$$

$$c = \left(\frac{\text{INT}}{\text{ATT}}\right) \times 20$$

$$d = 2.375 - \left(\frac{\text{INT}}{\text{ATT}} \times 25\right)$$

Where:

ATT = Number of passing attempts

COMP = Number of completions

YARDS = Passing yards

YARDS = Touchdown passes

INT = Interceptions

Then, the above calculations are used to complete the passer rating:

$$\text{PasserRating}_{\text{NFL}} = \left(\frac{\text{mm}(a) + \text{mm}(b) + \text{mm}(c) + \text{mm}(d)}{6}\right) \times 100$$

Where: $\text{mm}(x) = \max(0, \text{mm}, (x, 2.375))$

I thought it might be interesting to model the formula in APL and look at the impact of the various inputs, for instance what happens if a passer completes only 50% of his passes, but every pass he completes is for a touchdown?

It’s fairly straightforward to translate the formula directly into APL:

```

▽ rating←att PasserRating(comp yards td int);a;b;c;d:mm
[1] mm←{0|2.375|ω}      A define the max/min function
[2] a←((comp÷att)-0.3)×5
[3] b←((yards÷att)-3)×0.25
[4] c←(td÷att)×20
[5] d←2.375-(int÷att)×25
[6] rating←((mm a)+(mm b)+(mm c)+(mm d)÷6)×100
▽

```

For instance, a quarterback who completes 17 of 20 attempts for 300 yards, 4 touchdowns, and no interceptions would have a “perfect” rating of 158.3.

```

      20 PasserRating 17 300 4 0
158.3333333

```

The above function works just fine, but it’s not particularly “APL-like”. By this I mean that it’s longer than it needs to be and doesn’t really make use of APL’s array handling capabilities.

One of the first things I noticed was that all of the terms (completions, yards, TDs and interceptions) are treated similarly...

- they’re each divided by the number of attempts
- the result of that operation is adjusted by some number (in some cases the number is 0)
- those results are then multiplied by some number
- then those results are added (or subtracted) from a number (again that number may be 0)
- the mm function is applied to each result

Then to complete the calculation the results are summed, that sum is divided by 6 and then multiplied by 100.

So, if instead of having 4 terms (comp, yards, tds, and int) we have one term “stats” which is a 4 element vector comprised of comp, yards, tds, and int, we can simplify the first operation to:

```

stats÷att

```

The next step is to adjust each result by some number, -0.3 for completions, -3 for yards, and 0 for each of TDs and interceptions. There are a few ways to do this in APL...

```
(stats ÷ att) - .3 3 0 0 A is the most straightforward
-.3 3 0 0 + stats ÷ att A removes the parentheses
.3 3 0 0 -~ stats ÷ att A uses subtraction and the commute operator ~
A which commutes (switches) the arguments
```

All of these statements produce equivalent results. So, you can use whichever seems most straightforward to you.

Each of these results is multiplied by a number: 5 for completions, .25 for yards, 20 for TDs, and -25 for interceptions. Why use -25? Because the interception result is subtracted from 2.375 in the subsequent operation.

```
0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att
```

Next the mm (max/min) function gets applied; it's so trivial we can skip writing a separate function...

```
0 [ 2.375 [ 0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att
```

Then those results are summed...

```
+ / 0 [ 2.375 [ 0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att
```

The sum is then divided by 6 and multiplied by 100, again there are several ways to accomplish this...

```
100 × (+ / 0 [ 2.375 [ 0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att) ÷ 6
(100 ÷ 6) × + / 0 [ 2.375 [ 0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att
100 × 6 ÷~ + / 0 [ 2.375 [ 0 0 0 2.375 + 5 .25 20 -25 × .3 3 0 0 -~ stats ÷ att
```

Putting it all together, we can rewrite our function as:

```
▽ rating←att PasserRating stats
[1] rating←100×6÷~+/0[2.375[0 0 0 2.375+5 .25 20 -25×.3 3 0 0-~stats÷att
▽
```

This is known as a trad-fn (traditional function) in Dyalog APL. Dyalog also has d-fns (dynamic functions). PasserRating written as a d-fn would look like:

```
PasserRating←{100×6÷~+/0[2.375[0 0 0 2.375+5 0.25 20 -25×.3 3 0 0-~ω÷α}
```

Once you have the PasserRating function, you can see the effect of the different inputs. For instance, what if the passer had the same results (17 completions, 300 yards, 4 TDs, and 0 interceptions) but the number of attempts varied?

```
{'Attempts' 'Rating';ω,[1.1](PasserRating=17 300 4 0)''ω}16+116
Attempts      Rating
17 158.3333333
18 158.3333333
19 158.3333333
20 158.3333333
21 158.3333333
22 158.1439394
23 155.3442029
24 152.7777778
25 148.3333333
26 144.2307692
27 140.4320988
28 136.9047619
29 133.6206897
30 130.5555556
31 127.6881172
32 125
```

Does the passer rating formula make sense? For instance, consider two quarterbacks who each have 20 attempts and pass for 400 yards. The first quarterback has only 10 completions, but each of those is for a touchdown. The second quarterback has 20 completions, but only 1 touchdown. Who's the higher rated quarterback? According to the formula, they both have the same rating.

```
20 PasserRating ``(10 400 10 0)(20 400 1 0)
135.4166667 135.4166667
```

Perhaps “passer” rating is less interesting than a “quarterback” rating which, in addition, could take into account things like:

- sacks – a quarterback who less aware of his surroundings and is sacked more frequently isn't as effective
- fumbles – similarly a quarterback who fumbles more often isn't as effective
- yards gained – most of today's quarterbacks have to be somewhat mobile

Can passer rating be improved upon? There are lots of possibilities and APL makes it easy, even fun, to explore them.

References

1. *Passer rating* http://en.wikipedia.org/wiki/Passer_rating

Dyalog's parser - a new parser in town

Dan Baronet (danb@dyalog.com)

In the following text I use terms specific to our trade. You won't find them in the dictionary but I assume the reader is familiar with words such as 'monad', 'global' (as a noun) and 'default' (verb). I also use quotes and angle brackets to help determine the type of the object I am referring to. 'Quotes' denote a variable or workspace and <angle brackets> refer to a function/operator or file. Often, the context is sufficient to remove ambiguities. *Emphasized* words have a special meaning. Definitions are especially marked up, too.

Introduction

A line (string) parser is a handy tool to carry around.

Such a tool should be able to accept a string as argument (its input) and be able to attribute meaning to its constituent parts by following a number of simple rules.

For example, in C, a function's list of arguments is given by "a left parenthesis, 0 or more non-blank strings separated by commas and a right parenthesis". The statement `ABC(2+3,x,y/z);` is a perfectly valid C statement, calling function ABC with 3 arguments.

Another example, in DOS (or Windows' command mode), a command (a keyword) takes 0 or more words as arguments, followed by 0 or more switches, a word being a series of non-blank characters and a switch being a special character (here /) followed by a letter, possibly followed by a colon (:), and a string.

The TREE command in DOS, for example, accepts 0 or 1 argument, possibly followed by the switches /F and/or /A.

Unix is similar, using dash (-) instead of / as switch delimiter.

There are many times when a similar situation arises and we must supply a known number of arguments and switches to a program.

Closer to home, there are cases where such a parser could come in handy. Imagine a program, REPORT, which accepts DATA as right argument and some Options as left

argument. The options could be e.g. "Title", "page width", "use page numbers". The program's header and first line would typically look like this:

```
▽ Options REPORT Data;Title;PW;P0;...
[1] (Title PW P0)+Options,(p,Options)↓' 70 0
[2] ...
```

The onus is on the user to remember the order of `Options` but with only 3 options it isn't so bad. Imagine now that there are a large number of options. It might be simpler to specify something like:

```
'Sales Report +usepagenos +pagewidth=80 +margin=10 10' REPORT data
```

The syntax is cleaner and the user doesn't have to remember all the options and their order. If a change occurs and new options are added they can be inserted easily. The problem of course is to make sense of that left argument.

This kind of problem arose in the 90s when STSC introduced user commands in the APL/PC product.

User commands are commands that the developer writes in APL and are called by the system using a right bracket (`[]`) syntax, similar to the `]SYSTEM` commands. A programmer writes command `XYZ` which is called in the session by writing `]XYZ`. If the command takes arguments and/or switches they are added after the command name. The programmer is responsible to parse the line and figure out the meaning. Simple commands with few arguments and switches take only a few lines to parse but many acceptable switches become quickly overwhelming and the lack of standard makes it confusing for the user. And the code to parse the line quickly shadows the important code.

When user commands came out with APL/PC there was no parser and I wrote one for them and used it for many years. I even wrote an article in *Vector*[1] about it. The problem was that there were no enclosed arrays at the time and the parser had to do all kinds of tricks to do its job like setting globals, using delimited strings and so on.

Today with arrays of enclosures and, in Dyalog APL's case, namespaces, it is easier to pack all the information into a tight object.

That's the purpose of this text. I will use user commands as example. This is in fact the parser used at Dyalog for their user command processor.

A few definitions

Generalities

A sentence is made of characters and divided into 0 or more sections.

Sections are separated from each other by one or more of a special character, the separator.

Each section contains a single field and a value. A value is 1 or a string.

A field has an ID. Some fields may have a name in which case their ID is their name. An unnamed field's ID is a unique number in the range 1 to N where N is the number of allowed unnamed fields.

Named fields are always optional. Unnamed fields can be compulsory. In a sentence, sections may contain named fields which may be repeated.

A named field is introduced with a special symbol followed by the name. If a section with a named field in it will have a value it will appear after the '=' sign after the name.

Example: here is a sentence with named and unnamed *sections*, the comma is the *section* separator:

```
una,unb,, $city=mtl,, $cnt=can,$nice
```

This *sentence* has 5 *sections*, 2 with unnamed *fields* and 3 with named *fields*. The *ID* of each one is 1, 2, *city*, *cnt* and *nice*. All *sections* but *nice* have a *value* specified. The first two *sections* have unnamed *fields* with *values* *una* and *unb*. The last three *sections* have named *fields* and the last one has no *value* specified, its *value* is 1.

Implementation

A parser should be able to determine if a *sentence* follows specific rules.

The class `Parser` produces a parser capable of recognizing if a sentence follows specific rules. Those rules are supplied at instantiation time.

The rules specify

- the number of unnamed sections.
- how many are compulsory.
- the list of named sections.

- whether they accept a value.

Example: the expression

```
CP←NEW Parser ('$city= $cnt= $nice' 'nargs=2')
```

produces a parser (CP) only capable of determining if a sentence, like the one above, follows the rules by applying its Parse method to a sentence, e.g.:

```
Data←CP.Parse 'una unb $city=mtl $cnt=can $nice'
```

If the sentence does **not** follow the rules the parser will signal an error. For example if a non-existent named field is specified or if a named field accepting a value is not given one (or vice versa) then it will signal an appropriate error. The validation is very strict.

If the *sentence* is valid, *Data* will be a regular namespace containing all the possible *sections* with their name and value. If a *section* is absent its value will be 0. If it is present without a value (e.g. *\$nice*) it will be 1 (not '1'). If a section is repeated only the last value is retained. To see all of them you can do *Data.SwD*. In the example above you would get

```
city  mtl
cnt   can
nice  1
_1    una
_2    unb
```

You can access the value of a *section* directly, e.g.

```
Data.city
mtl
```

The unnamed *sections* are given the *ID* *_1* and *_2*. This way you can access their value directly:

```
Data._1
una
```

The parser can be applied again to another *sentence*:

```
Datb←CP.Parse 'I love $city=Paris '
Datb.city
Paris
  Datb.SwD
city Paris
cnt    0
nice   0
```

```

_1      I
_2      love

```

Since `cnt` was not specified in the *sentence* its value is 0. Same for `nice`.

Unnamed sections

Optional unnamed sections can be done using 'S' with the number of arguments. 'S' stands for 'Short' to allow a shorter number of arguments. This makes them all optional as 0 is an acceptable number of arguments too. If CP above is defined as (note the 'S' after the 2)

```
CP+[]NEW Parser ('$city= $cnt= $nice' 'nargs=2S')
```

Then

```
Datc+CP.Parse 'great $nice $cnt=Canada '
```

would produce (note the 2nd argument is 0 because it is not in the *sentence*)

```

Datc.SwD
city      0
cnt      Canada
nice      1
_1      great
_2      0

```

Sections with spaces in them

If a *section* contains a *section delimiter* (a space here) in it there must be a way to tell the parser. The preferred way is to use yet another character to escape the space or to surround the section with a pair of enclosing special characters. An obvious character to use in this case is ". For example:

```
Datd+CP.Parse ' "Добрый день" $cnt=Russia '
```

produces

```

Datd.SwD
city      0
cnt      Russia
nice      0
_1      Добрый    день
_2      0

```

Parse will accept both ' and " as string delimiter as long as they are paired properly, i.e. 'a ... z' and "a ... z" are ok but 'a ... z" is not.

If a quote is part of the string the other quote can be used or you can double the quote

inside the quotes string, e.g. "I'm OK" or 'I'm OK'.

Quotes should also be used if the text includes a character used to introduce a named field (e.g. \$ above). Example: 'amount is \$20'.

The Dyalog parser

The Dyalog parser is located in `SE`.

In this parser the space is used as section delimiter. It cannot be changed.

Terms

Because of the context in which the parser is used an unnamed field is called an argument and a named field is called a switch or modifier.

In theory the *arguments* could appear anywhere in the sentence but Dyalog's parser does not allow it; all *arguments* must appear at the beginning of the sentence. This means that since only sections containing *modifiers* can appear at the end there is no need to quote the values if they contain spaces, i.e. in

```
Datf+CP.Parse 'huge $cnt=US of A '
```

US of A does not need to be quoted to include the spaces. Note that the trailing spaces are ignored.

On the other hand, since arguments have an ID you can specify them elsewhere in the sentence by simply using their ID followed by = and the value. The example above then becomes

```
Datf+CP.Parse '$cnt=US of A $_1=huge '
```

Features

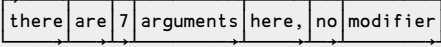
This parser has many features.

No need to specify the number of arguments if it is possibly unlimited. The class' argument is then a single string:

```
CP+NEW SE.Parser '$city= $cnt= $nice'
```

If you do not specify the number of arguments no `_n` variable will be stored in the resulting namespace. However, the list of arguments is **always** stored in variable `Arguments`. Example:

```
Datf+CP.Parse 'there are 7 arguments here, no modifier' Ano nargs=string
Datf.SwD
city 0
cnt 0
dsa 0
]disp Datf.Arguments
```



The character introducing the names must be specified.

It is the 1st char in the list (here #) and separates the names:

```
CP+NEW SE.Parser '#city= #cnt= #nice'
```

Minimum character needed to specify names

There is no need to enter the entire name, only the minimum suffices:

```
CP+NEW SE.Parser '+color = +country='
Datf+CP.Parse '+col=blue +cou=UK'
```

Here +col is sufficient to determine that it is color. Same with +cou for country.

If only +c or +co is used the parser won't be able to tell which one is meant and an error will be signalled.

On the other hand you may want to force the entry of a name to a minimum. You use parentheses for that:

```
CP+NEW SE.Parser '-color= -country(ofresidence)='
```

Here -color can be abbreviated to -col but -countryofresidence can only be entered with a minimum of -country. This is useful when forcing the user to enter the whole name because of a security problem, e.g.

```
CP+NEW SE.Parser '/file= /delete()'
```

Here /file can be entered as a single /f but we don't want the user to enter /d alone by mistake and a full /delete is required.

Only the '(' is important and the last ')' is ignored but it is tolerated.

Case insensitive

Normally modifiers' names are used "as is" but you may want to enter them in lower or uppercase. If you do

```
CP+NEW SE.Parser ('$City= $Cnt= $nice' 'nargs=2 upper')
Datg+CP.Parse 'I love $cITY=Paris '
Datg.SwD
CITY Paris
CNT 0
NICE 0
_1 I
_2 love
Datg.CITY
Paris
```

all names are uppercased. There is no way to get them in lowercase form.

Minimum-maximum number of arguments

It is possible to add an 'S' to the number of arguments to specify that they are all optional, i.e. that 0 to n can be entered (here 5):

```
CP+NEW SE.Parser ('/file= /delete' 'nargs=5S')
```

It is also possible to use n1-n2 to specify a minimum (here 2 to 5):

```
CP+NEW SE.Parser ('/file= /delete' 'nargs=2-5')
```

If the number of arguments is not from 2 to 5 the parser will issue an error, either 'too few arguments' or 'too many arguments'. Using 'S' is the same as 0-n.

It is possible to merge extra arguments together.

For example if the last section contains spaces it must be used like this:

```
CP+NEW SE.Parser (' 'nargs=3') A note no modifiers accepted
Dath+CP.Parse ' Joe Blough "42 Penny Lane E." '
```

If there is nothing following the 3rd section we can tell the parser that it is "Long" and quotes are not needed (but still accepted). Note the L after the 3:

```
CP+NEW SE.Parser (' 'nargs=3L')
Dath+CP.Parse ' Joe Blough 42 Penny Lane E. ' A no quotes needed at the end
ldisp Dath.SwD A note the spaces are preserved
```

| | |
|----|------------------|
| _1 | Joe |
| _2 | Blough |
| _3 | 42 Penny Lane E. |

This feature is useful when expecting a single long argument:

```
Log+NEW SE.Parser ('-file=' 'nargs=1L')0
Dath+Log.Parse ' Joe Blough 42 Penny Lane E. -file=\tmp\log.txt'
]disp Dath.SwD
```

| | |
|------|-----------------------------|
| file | \tmp\log.txt |
| _1 | Joe Blough 42 Penny Lane E. |

The number of arguments can be both, "Long" and "Short". There is no restriction in that respect. The rules may specify less than, say, 3 (Short), but merge any argument above 3 with the 3rd one (Long). This would be specified as

```
CP1+NEW SE.Parser (' ' 'nargs=3SL')
```

There is no limit on the number of arguments

As noted before it is possible to specify that there is no limit on the number of arguments simply by not specifying the 'nargs=' field in the 2nd string (or eliding the 2nd string completely).

```
CP2+NEW SE.Parser '/file=/del'
```

It is also possible to enter 'nargs=99999' to signify 'a large number of arguments'.

The difference is in the resulting namespace which will only contain the _1, _2, ... variables if nargs=n has been specified.

Although there is no limit, in order to limit the number of variables defined in the resulting namespace (like Dath, above), the number of variables produced is limited to 15, i.e. _1, _2, ..., _15 will be there but _16 and up won't be. The list of all arguments is always kept in Arguments inside the namespace so they are always available. For example:

```
CP+NEW SE.Parser '+s1' A no nargs=
Dati+CP.Parse 'Joe Blo 42 Penny Lane E. tel 0 44 12345 890, and more '
pDati.Arguments
16
]disp Dati.Arguments
```

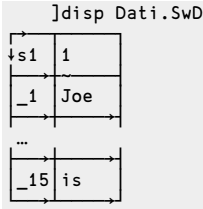
| | | | | | | | | | | | | |
|-----|-----|----|-------|------|----|-----|---|----|-------|-----|-----|------|
| Joe | Blo | 42 | Penny | Lane | E. | tel | 0 | 44 | 12345 | 890 | and | more |
|-----|-----|----|-------|------|----|-----|---|----|-------|-----|-----|------|

```
]disp Dati.SwD
```

| | |
|----|---|
| s1 | 0 |
|----|---|

```
CP+NEW SE.Parser ('+s1' 'nargs=999S')
Dati+CP.Parse 'Joe Blo 42 Penny Lane E. tel 0 44 12345 890, and more +s'
]disp Dati.Arguments
```

| | | | | | | | | | | | | |
|-----|-----|----|-------|------|----|-----|---|----|-------|-----|-----|------|
| Joe | Blo | 42 | Penny | Lane | E. | tel | 0 | 44 | 12345 | 890 | and | more |
|-----|-----|----|-------|------|----|-----|---|----|-------|-----|-----|------|



Ambivalent modifiers

Sometimes modifiers accept a value, sometimes they don't. If their nature is ambivalent you can specify it at parser creation time, using square brackets around = to mean "maybe", like this:

```
CP+[]NEW []SE.Parser '+s1[=]'
```

Here, `s1` is a modifier that may be specified with or without a value:

```

Datj+CP.Parse '+s'
Datj.SwD   A s1 is on the line without a value
s1 1
Datj+CP.Parse '+s=abc'
Datj.SwD
s1 abc

```

Validation

List member

The parser is able to perform minimalistic validation on the values entered with modifiers. For example, if modifier `s1` above accepts any of the values in 'ab' 'cde' 'fgjk' then we can create a parser to validate it like this:

```
CP+[]NEW []SE.Parser '+s1=ab cde fgjk'
```

and using it is as before:

```

Datj+CP.Parse '+s=ab'
Datj.SwD
s1 ab

```

except that if we enter a value not in the list we get:

```

Datj+CP.Parse '+s=abc'
invalid value for switch <s1> (must be ONE of "ab cde fgjk")
Datj+CP.Parse'+s=abc'
^

```


Set member

The values can also be checked against a list of characters and ensure they all belong to the list. We use \in instead of $=$ for this. For example, if modifier `vowel` below accepts any character in the set `'aeiou'` then we can create a parser to validate it like this:

```
CP+NEW SE.Parser '+vowel ∈aeiou'
```

And using it is as before:

```
Datk+CP.Parse '+v=aoaaee'
Datk.SwD
vowel aoaaee
```

except that if we enter a character not in the list we get:

```
Datk+CP.Parse '+s=aeY'
invalid value for switch <s1> (must be ALL in "aeiou")
Datk+CP.Parse '+s=aeY'
^
```

Default values

By default all fields have the value 0 to mean “not specified on the line”. When a modifier (or even an argument) is not specified we may wish to give it a value by default. For example, you may wish to use the value `'abc'` for modifier `s1` if it not on the line. In APL the code to do this would look like

```
:if 0≡v+Datj.s1 ⋄ v+'abc' ⋄ :endif
```

There are 2 ways to get a default value with the parser. The first one involves telling the parser at creation time:

```
CP+NEW SE.Parser '+city:London'
Datl+CP.Parse '+c=Kbh'
Datl.city
Kbh
Datl+CP.Parse 'blah' A no +city specified
Datl.city
London
```

The second method involves using a function (called `Switch`) in the resulting namespace.

That function takes the name of a modifier and returns its value when called monadically.

When called dyadically it returns its left argument if the modifier's value is 0 (e.g. not in the statement).

```
CP+NEW SE.Parser '+city='
Datl+CP.Parse '+c=Toronto'
Datl.Switch 'city' A city has the value "Toronto" as specified
Toronto
'NY' Datl.Switch 'city' A city was specified, it is returned
Toronto
Datl+CP.Parse 'blah' A no +city specified
Datl.Switch 'city' A no city means 0
0
'NY' Datl.Switch 'city' A no city can mean NY when not specified
NY
```

Switch has the advantage over the :default syntax in that it can turn strings representing numbers into numbers.

```
CP+NEW SE.Parser '+age:18'
Datl+CP.Parse '+a=70'
70=Datl.age A 'age' is '70'
70
0 0
Datl+CP.Parse 'blah' A no +age specified, its value is '18'
18=Datl.age A this is a string, not a number
18
0 0
```

The parser cannot tell whether '18' is meant to be a string or a number. Switch, on the other hand, is smart about it:

```
CP+NEW SE.Parser '+age=' A we don't specify a default value here
Datl+CP.Parse '+a=70'
70=Datl.Switch 'age' A this is character
70
0 0
70=+18 Datl.Switch 'age' A this is numeric, thanks to Switch
70
1
Datl+CP.Parse 'blah' A no +age specified
18=+18 Datl.Switch 'age' A this is numeric, thanks to Switch
18
1
```

Note that the result is a numeric vector, not a scalar.

If you try to turn a non-numeric modifier into a number Switch will also complain:

```
Datl+CP.Parse '+a=seventy'
666 Datl.Switch 'age'
value must be numeric for age
666 Datl.Switch 'age'
^
```

Other features

There are a few more features left:

Prefixing names

Modifier names cannot start with a number but if you use a prefix for them it can be made to work:

```
CP+[]NEW []SE.Parser ( '+007[=]')
switches must be valid identifiers
CP+[]NEW []SE.Parser('+007[=]')
^
CP+[]NEW []SE.Parser ( '+007[=]' 'prefix=Δ')
Datm+CP.Parse 'whatever +007=JB'
Datm.SwD
007 JB
Datm.[]nl-2
Arguments SwD Δ007
Datm.Δ007
JB
Datm.Switch '007'
JB
```

Not requiring space before modifiers

Since names start with a special character there is no real need to force a space to delimit them. An example is DOS commands which may be abutted as in DIR /T/A; here /A follows /T without any space in between.

If this can be allowed it can be specified as in

```
CP+[]NEW []SE.Parser ( '/sw1 /sw2' 'allownospace')
```

Changing the error number when things go wrong

When the parser refuses to accept a set of rules it signals an error in the 700-710 range. If this can interfere with the calling program it can be changed using error= to specify the lower range value:

```
CP+[]NEW []SE.Parser ( '/sw1' 'error=800')
```

Propagating the modifiers

Sometimes it is necessary to pass the modifiers received to another program which uses similar modifiers.

For example, in SALT, the program Snap uses many modifiers, some of which are

passed along to the program `Save`. Both use some same modifiers. The modifier `-noprompt` is one of them. When `Snap` calls `Save` it has to pass along that modifier in the command string. Assuming all the modifiers and arguments are in namespace `A`, one thing it could do is

```
Save cmdstring, A.noprompt / ' -noprompt'
```

Because there are many modifiers to pass along this statement would be in fact much more complicated, especially when modifiers have values.

The arguments namespace contains a function, `Propagate`, which will generate a string defining the switches as they were submitted.

For example, if `-noprompt` was specified on the `Snap` command line, doing `A.Propagate 'noprompt'` would return `'-noprompt'`. If `-noprompt` was not specified then it would return `' '`. If a modifier to be propagated has a value the function will reproduce it verbatim, e.g. if `-nop -file=\ab\c` is used then doing `A.Propagate 'noprompt file'` would return `'-noprompt -file=\a\b\c'`.

Example: going back to the `REPORT` example we can see that writing:

```
▽ Options REPORT Data;all;Parse;...
[1] Parse←{ (□new □se.Parser α).Parse ω}
[2] all← '+margin= +usepagenos +pagewidth=' Parse Options
[3] :if all.usepagenos ...
```

is easier to read and modify. We can now call this program like this:

```
'Sales Report +usepagenos +pagewidth=80 +margin=10 10' REPORT data
```

Another example, coding the `DIR` command in DOS (we use a prefix because of /4):

```
pDIR←□new □se.Parser ('/a=/b/c/d/n/o=/p/q/r/s/t=/w/x/4' 'allow prefix=S')
```

Epilogue

This tool is a bit elaborate but covers many aspects of line parsing. Many years of programming convinced me of its usefulness. I have programmed variants of this code in several languages but none as advanced as in Dyalog APL. If you write your own user commands this will prove to be very helpful.

If your version of Dyalog APL does not have all these features try to use the user command `juupdate` to update your version of `SALT` and `User Commands`. This should work with all versions of Dyalog APL starting at V13.1.

References

1. See "Tools, Part 1. Basics" in *Vector* 19.4 (April 2003) for details

ELI: a simple system for array programming

Hanfeng Chen (wukefe@gmail.com)

Wai-Mee Ching (waimee_ching@yahoo.com)

Abstract: ELI is an interactive array-oriented programming language system based on APL. In addition to flat array operations of ISO APL, ELI features basic list for non-homogeneous data, temporal data, symbol type and control structures. By replacing each APL character with one or two ASCII characters, ELI retains APL's succinct and expressive way of doing array programming in a dataflow style. A scripting file facility helps a user to easily organize programs in a fashion similar to C with convenient input/output.

1. Introduction

Ken Iverson first developed the APL notation for teaching applied mathematics at Harvard; later APL became machine executable through the work of L. Breed and colleagues at IBM T.J. Watson Research Center. Iverson intends APL to be a tool of thought for communicating algorithmic ideas precisely (see his Turing Award Lecture [7]). This leads to two unique features of APL: i) arrays are first class citizens, ii) a comprehensive set of array operations as language primitives and a special character set enforcing an one character one symbol rule with uniform syntax. That results in succinct expressions and a dataflow style of programming, i.e. organizing tasks as chains of monadic or dyadic functions and operators acting on arrays. As a consequence, APL is remarkably productive and versatile; it has been used profitably in areas ranging from finance, actuarial, computer-aided design, logistics, manufacturing to that of research in physics, econometrics and biometrics ([8] has a recent example). In contrast, the array language MATLAB follows the FORTRAN style and its main application areas center on scientific computation and engineering.

We see three factors contributing to APL's decline despite its power to rapidly implement and deliver commercial applications: i) introduction of two nested array systems post-ISO APL fragmented APL community without an increase of markets; ii) its special character set and iii) no free version of APL is (J is both free and uses ASCII characters but it is more terse and difficult to learn than APL). The aim of ELI is to re-introduce APL-style programming to a wider audience by a free and simple array

programming system which is easy to learn and convenient to use. The project[3] started a decade ago, and reactivated recently. We hope ELI will attract new comers to APL family of languages who may later move on to commercial APL products.

ELI has all flat arrays operations specified in the ISO APL standard [1]; it does not have nested array feature of IBM APL2 and other APL vendors. However, ELI has lists and list operations to deal with irregular/non-homogeneous data. In addition, ELI has temporal data, symbol type and C-like control structures. In ELI, each APL character is replaced by one or two ASCII characters in a way that is intuitive and retains the succinctness of original symbolic representation of APL code. Finally, ELI has scripting file facility to conveniently transfer data and code as well as to organize programs in a way similar to the use of #include in C, which is further helped by a short function definition form.

While MATLAB is popular in scientific/engineering applications, a simple array language of APL lineage offers unique advantage in programming complex systems such as database systems. ELI is easy to learn, has useful features beyond original APL and retains its original programming style. Being free, ELI can let more people appreciate the fact that simplicity of rules and notation in a programming language leads to greater programming productivity.

In section 2, we describe the basic features of ELI system and the symbol representation, in section 3, we explain the scripting file facility; details can be found in the Primer [4] on our web-site. We discuss future development plan for ELI before conclusion. ELI is written in C++ using Microsoft MFC, hence is first available on Windows, but it soon will be ported to Mac OS X and Linux. ELI is interpreted, but we already have a compiler written in ELI with some restrictions (exclusion of the execute function and the general use of lists). The compiler, translates ELI code into ANSI C code (with this in mind, ELI prohibits variable names to be reused for function names, nor a function name to be renamed as a variable). Once the compiler has compiled itself, it will be offered for general use. ELI can also be used as a tool for generating parallelized code for multi-core desktop as well as for other parallel machines similar to what have been done using APL [6]; the route to parallelism is not a parallel implementation of the interpreter but to automatically parallelize the translated C code by the compiler. We also intend to implement a simple column-based database system in ELI.

2. The Array Language ELI

ELI is based on APL1 as described in the ISO APL standard [1]; it replaces each APL character symbol by one or two ASCII characters. A sample is listed in the table below

(for a complete list of ELI symbols see [4], which comes with the system) where the original APL symbol is in parentheses.

| | | | | | |
|----|-----|----------------------------|-----|------|----------------------------|
| # | (ρ) | shape/reshape | #. | (⊖) | matrix inverse/divide |
| ? | (ε) | where/member | ?. | (?) | roll/deal |
| ! | (ι) | interval/index of | !. | (!)" | execute/drop |
| ^ | (^) | count/and | ^. | (&) | first/take |
| & | (∨) | or | &. | (*) | transpose |
| ⌊ | (⌊) | floor/min | <. | (.) | enclose/encode |
| ⌈ | (⌈) | ceiling/max | >. | (3) | grouping/decode |
| * | (×) | signum/multiply | *. | (*) | exponential/power |
| % | (÷) | inverse/divide | %. | (9) | natural log/log |
| | () | abs value/modulo | . | (!) | factorial/binomial |
| \. | (⍋) | scan,expand along 1st axis | + | (>) | format |
| \$ | (φ) | reverse/rotate | \$. | (*) | reverse/rotate on 1st axis |
| > | (⤴) | grade down/greater | @ | (C) | circle function |
| < | (⤵) | grade up/less | -> | (F) | branch |
| <- | (←) | assign | <= | (H) | less or equal |

An APL symbol usually represents either a monadic or a dyadic primitive function depending on whether it has one or two operands, and it is the same in ELI, the name(s) next to each symbol in the table above is the name of the monadic (and the dyadic) primitive function it represents. While ELI symbols lost the exquisite beauty of APL font, it retains the essential spirit of one character one symbol of APL notation. We see that the ELI symbols, all consist of special characters, are simple and suggestive, hence easy to remember and do not require a blank between symbols and names/data as in original APL. This is in contrast to the more elaborate multi-character symbol scheme of J and the introduction of word symbols in Q [2]. The current ELI symbol representation differs partly from that in [3], though it is conceived with the same pragmatic considerations: i) give frequently used primitives single character representation while less frequent ones with a second character '.', ii) for two-character symbols not ending in '.' the character pair should be self-suggestive.

The arithmetic, logical and relational primitives in APL/ELI are called scalar functions since they act on arrays by an extension of their action on each (pair of) scalar element(s) in that array(s). To take cube roots of a group of numbers, or take different roots of a single number (%3 is 1%3), we enter

```

1 2 8 1000 81 125 *.%3
1 1.259921 2 10 4.3267487 5
1024*.%1 2 3 5 10
1024 32 10.079368 4 2
  _1*.0.5
0j1
  *.0j1*@1
  _1

```

For a dyadic scalar function f , the outer product of f is written as $.:f$. For example, to see how 1000 dollars will grow under 3%, 5% and 8% of annual interests rates in

10 years, we do

```

1000 *1.03 1.05 1.08 .:*. !10
1030 1060.9 1092.727 1125.5088 1159.2741 1194.0523 1229.8739 1266.7701
1304.7732 1343.9164
1050 1102.5 1157.625 1215.5063 1276.2816 1340.0956 1407.1004 1477.4554
1551.3282 1628.8946
1080 1166.4 1259.712 1360.489 1469.3281 1586.8743 1713.8243 1850.9302
1999.0046 2158.925

```

Please note that a line of code in ELI, as in APL, executes from right to left, one primitive or derived function at a time with equal precedence but respects parentheses. For two dyadic scalar function f and g , $f:g$ is the inner product of f and g . $+:*$ ($+.×$ in APL) is the well known matrix multiplication, but there are other equally useful inner products such as $^:=$. For a dyadic scalar function f the reduce operator produces a derived function $f/$ and the scan operator produces $f\.$ To illustrate, $+/V$ is the sum of V while $+\V$ is the partial sums of V , and the or scan $\&\B$ is a vector derived from boolean vector B by turning all 1s once it encounters a 1. Now, suppose `emp_Div` is a 3000 by 2 character matrix indicating which division an employee belongs to where each row is one of the rows from the variable representing divisions

```

Div<-4 2#'HQMKSLLIT'
Div
HQ
MK
SL
IT

```

To count employees in each division, we simply do

```

+/.emp_Div ^:=&.Div

```

where $\&.Div$ is the transpose of `Div` and the result of inner product is a 3000 by 4 boolean matrix indicating the division an employee belongs; finally, $+/.$ sums along the first axis gives a 4 elements vector showing the number of employees in each division.

A data of symbol type is entered with a back-tick ` followed by a character string (possibly empty); a character string which can form a symbol obeys the same rule as that governing a name for variables.

```

#s3<-`abc `ddl `comp
3
2 2#s3
`abc `ddl
`comp `abc

```

There are 6 temporal data types. For date and second, we have the following examples:

```
2012.10.15+!7
2012.10.16 2012.10.17 2012.10.18 2012.10.19 2012.10.20 2012.10.21 2012.10.22
23:10:50+30
23:11:20
```

Eli provides lists to organize non-homogeneous data: a *list* is a group of *items*, each of which can be a scalar or an array, separated by ‘;’ and a list L can be assigned to a group of variables where the number of variables is equal to #L:

```
#L<-(`abc `ddl `comp;1 2 3)
2
  L
<`abc `ddl `comp
<1 2 3
  L[1]
`abc `ddl `comp
  L[2]
1 2 3
  (s;n)<-L
  s
`abc `ddl `comp
  n
1 2 3
```

To enter a list of one item, we employ the monadic primitive function enclose <.:

```
#L<-<.:2 4#!12
1
  L
<1 2 3 4
5 6 7 8
```

The ELI programming environment, follows that of APL, is called a workspace. After you installed ELI and click on the ELI icon, a window pops up with a line

```
CLEAR WS
```

indicating a clear workspace, i.e. there is no user variable or defined function yet (on right most of the top bar there is a Help button, click on it to access the Primer [4] for a basic description of ELI); but it has default system variables such as []IO, the index origin, which is set to 1 and can be changed to 0:

```
!10
1 2 3 4 5 6 7 8 9 10
[ ]IO<-0
!10
0 1 2 3 4 5 6 7 8 9
```

One is either in execution mode, i.e. executing an ELI expression, including assignments to variables as we have seen in previous examples; or in definition mode

to define a user function. To switch to the definition mode, you type @. followed by the intended function head. A user defined function can have one or two arguments, or no argument, it can return a result or no result and all these are specified by the function head (sect. 3.1 in [4]). One can use a list to effectively input more than two arguments:

```
`sales bk_load (sa;cu;it;am;py;dt;sp)
```

Once you finish writing your function, type a matching @. to get out of the editor and back to the execution mode of the interpreter. The function you just edited is available for testing, but it is not yet saved. To save, the workspace should already have a name, if not, then do

```
)wsid ABC
)save
```

Of course, variables will also be saved. A saved workspace can be loaded later by

```
)load ABC
```

ELI does not support the)copy command to copy in a workspace on top of an existing workspace. Instead, if a workspace has no suspension, it can be)out ..(see section below), and then bring in back by the command)in .. which differs from)load in that existing variables and functions previously existed would remain, but in case of conflict, those in conflict will be replaced by newer ones.

ELI has control structures quite similar to those in C (control structures are not prescribed in [1], and not implemented in current IBM APL2 but present in the products of other APL vendors). There are five reserved words in ELI for control structures if, else, case, for and while and simple statements can be grouped together by a pair of curly brackets. We illustrate their use with a recursive function rprime for finding primes up to n:

```
p<-rprime n;i
p<-2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
if (n<=100) p<-(n>p)/p
else {p<-#p<-rprime _.n*.0.5
      b<-n#0
      for (i:1;pl) b<-b&n#(-p[i])^1
      p<-p,1!.(~b)/!n
}
```

This shows how control structures improve program readability when there are alternate choices and irregular iteration while ELI boolean vector operations still entice a succinct dataflow style coding.

J[10] pushes APL to be more abstract in the direction of functional programming, but aims of ELI are fairly pragmatic: to make classical APL more accessible to general public and easier to mix with other application environments, and easy communication such as pasting code in e-mails. ELI also avoids the complexity of nested arrays in APL2 by providing lists for non-homogeneous data.

3. Scripting Files and Testing

While workspace is great for program development, as you can save not only clean code but partially debugged code as well, it takes quite lot of space (in fact, the old APL idea of workspace is what we call IDE for other conventional languages today). When your workspace contains no unresolved error, i.e. the system indicator `)SI` is empty, ELI (follows IBM APL2) let you output the content of your clean workspace to a transfer file by the system command `)out fnam`; and later the content of that file can be brought back by `)in fnam`. However, a transfer file must first come from some workspace. It is rather inconvenient to input a large amount of data or functions into an APL workspace. Hence, we introduced the scripting file facility in ELI to directly bring data and functions, written in ordinary text files, into ELI system. We note that scripting file facility exists in another APL dialect, A+, developed at Morgan Stanley[9], but there is no workspace facility in A+.

A scripting file can contain not only function definitions, as you would write them in the ELI editor in the definition mode of the ELI interpreter, but also values for variables. To do that you put

```
&ABC I 2 50 80
...
&
```

The first line is the variable name followed by its type (B:boolean; I:integer; E:floating point; J: complex number; C:character; S:symbol; D:datetime) and the rank and shape of that variable, then followed by the value of the variable in ravel order. To load in a scripting file S, type

```
)fload S
```

There is a companion command `)fcopy S` for copying in a script file which behaves similar to `)copy` of a workspace. You can even put in executable statements such as `'RPRIME 120'`(where `RPRIME` must be defined earlier) in a script file; that statement would then be executed at the time of loading/copying. Hence, we can say that scripting files provide a batch mode execution for ELI. A scripting file can also include other system commands such as `)fload ...`, `)fcopy ...` to bring in other scripting files. For scripting files, ELI also has a short-form function definition facility for simple

functions which do not access global variables as follows

```
{fnam: ...}
```

where `fnam` is the name of a function; `z` is the result, or the last expression is the result, of the function, `x` is the right argument, and `y` is the left argument if present, and all other variables are local; comment lines must be outside of the function body `{..}`. There is a standard library `standard.esf` which includes the following functions:

```
{avg: (+/x)%_1^.#x} //row-wise average of a numeric array
{gmean: (*/x)*.##x} //geometric mean of a numeric vector
{intersect: (y?x)/y} //y intersects x, those in y which are from x
{less: (~y?x)/y} //elements in vector y which are not in x
{xor: 2|y+x} //exclusive or of boolean vectors y and x
{last: x[#x]} //last element of a vector
{triml: (&\x~=' ')/x} //trim leading blanks off a character vector
{trimr: $(&\r~=' ')/r<-$x} //trim trailing blanks off a character vector
{stddev: ((+/(x-avg x)*.2)%#,x)*.05} //standard deviation of vector x
{median: ((0.5*w[m]+w[m+1]),w[m<-_1[[]IO+~.0.5*#x]][[]IO+2|#w<-x[<x]])} //median
```

Following examples in this script file, one can easily extend the ELI language to have many built-in functions which may or may not present in other array languages such as MATLAB or R. Since a scripting file can contain `)fcopy` command lines in the beginning, the short function form can be utilized by users in an specific application area to group commonly used functions in a scripting file similar to the use of `#include` to bring in domain specific libraries in C. In addition to using scripting file for input/output, we also have provided a link between ELI and Excel.

For APL users who have legacy APL programs which do not use nested arrays, we offer an APL program to translate APL source codes into ELI scripting files. This, indeed, is our main tool for testing the ELI interpreter. The backbone of our testing procedure follows the two-pronged strategy of [5], i.e. unit test and a large suite of application programs translated from APL. In both cases, we utilized scripting files to automate the process of generating input suites as well as comparing output files for correctness. Yet testing an interpreter is more complicated than testing a compiler as it has to account for erratic ways a user types in codes, and that can only be tested by the usage of many users.

4. Conclusion

We have described an array-oriented programming system called ELI, which is derived from APL but uses ASCII characters. It remains to be simple, succinct with expressive power and encourages a dataflow style of programming as in APL. The

addition of control structures aids to better present complex code; and the new scripting file facility provides convenient means of input/output and let a user organize programs similar to the use of `#include` in C.

We hope the easy availability of such an array programming system will let more people appreciate the fact that simplicity of rules and notation in a programming language leads to greater programming productivity. ELI can be used both as a tool for speedy implementation of highly complex applications as well as for rapid experimentation in building prototypes in search for an ideal design.

References

1. International Organization for Standardization, ISO Draft Standard APL, ACM SIGAPL Quote Quad, vol.4, no.2, December, 1983.
2. Jeffery Borrer, q for Mortals, a tutorial in Q programming, Continuux LLC, New York, 2008.
3. W.-M. Ching, The Design and Implementation of an APL dialect, ELI, APL Berlin 2000 Proc., Berlin, 2000, p69-76. (<http://fastarray.appspot.com/> for documents and executable)
4. W.-M. Ching, A Primer for ELI, a system for programming with arrays, preliminary version, 2011.
5. W.-M. Ching and Alex Katz, The Testing of an APL Compiler, ACM SIGAPL Quote Quad, vol.20, no.1, 1993, p55-62.
6. W.-M. Ching and Da Zheng, Automatic Parallelization of Array-oriented Programs for a Multi-core Machine, Int'l Jour. of Parallel Programming, vol. 40, no.5, 514-531, 2012.
7. Ken. Iverson, Notation as a Tool of Thought, Comm. ACM, vol.23, no.8, 444-465, 1980.
8. Lars Wentzel, CPAM, Array Structured Product Data at Volvo Cars, Conf. Proc., International Conf. on APL, Berlin, Germany, 2010, p199-208.
9. A+ developed at Morgan Stanley <http://www.aplusdev.org>
10. J <http://www.jsoftware.com>

J

J-ottings 56

Trig Time

by Norman Thomson

A general objective of J-ottings over the years has been to draw attention to the considerable number of mathematical or mathematical type routines which are built into J primitives thereby leading to significant reductions of programming effort. One such feature is the versatility of j which although primarily a complex number constructor is adaptable to other circumstances in which objects are defined by pairs of numbers, for example betting odds (see J-ottings 54).

A further example concerns transformations of a 2-dimensional plane by means of matrices of the form

$$M = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}$$

where a and b are real numbers. A transformation such as

$$M = \begin{pmatrix} x \\ y \end{pmatrix}$$

is called a **similitude**, that is a transformation which results in the combination of an **anti-clockwise rotation about the origin** and an **enlargement** of the objects described by the coordinates. (For a clockwise rotation exchange b and -b). Given $\det = .-/ .*$ standing for determinant, $\det M$ is $a^2 + b^2$ and its square root is the enlargement E. The rotation component is represented by M divided by E, resulting in a matrix of the form

$$M = \begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix}$$

where t is the anti-clockwise angle of rotation.

Now although a similitude could be applied to a triangle whose points are, say, (0,0), (2,1) and (0,1) by

```
]M=.2 2$2 _3 3 2 NB. similitude matrix
2 _3
3 2
```



```

]tri=.2 3$0 2 0 0 1 1
0 2 0
0 1 1
M +/ .*tri
0 1 _3
0 8 2
    
```

to give the transformed triangle (0,0), (1,8), (-3,2), clearly M is defined by the number pair (a,b) and so can be represented compactly as a j pair, in which case enlargement E and rotation t are given by :

```

10 o. 2j3
3.60555          NB. enlargement=sqrt of 2^2 + 3^2
(%10&o.)2j3
0.5547j0.83205  NB. (cos x)j(sin x) where tan x=3%2
    
```

Instead of using a matrix inner product to transform points, simple multiplication is all that is required, so that the previous triangle transformation is given by

```

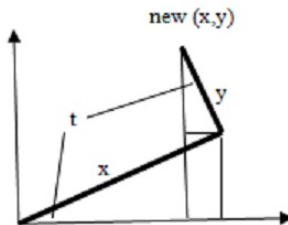
each=.&>
2j3*each 0j0 2j1 0j1  NB. triangle transformation
0 1j8 _3j2
    
```

Also since multiplication is commutative, a product such as 2j3*2j1 has two geometric interpretations, viz. the similtude 2j3 transforms the point (2,1) to the point (1,8) and the similtude 2j1 transforms the point (2,3) to (1,8).

For rotation without enlargement the transformed coordinates of the point (x, y) in the new frame of reference are

$$\begin{pmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cos t - y \sin t \\ x \sin t + y \cos t \end{pmatrix}$$

which can be confirmed by elementary trigonometry:



The components of displacement from {x, y} are thus

$$\begin{pmatrix} x \cos t - y \sin t \\ x \sin t + y \cos t \end{pmatrix} \text{ or } \begin{pmatrix} x(\cos t - l) - y \sin t \\ x \sin t + y(\cos t - l) \end{pmatrix}$$

which can be written

$$-(1 - \cos t)\left(\frac{x}{y}\right) - \sin t\left(\frac{x}{y}\right)$$

The object of this rearrangement will become apparent shortly.

Rotations in three dimensions

Here the geometry is more complicated and j no longer helps. Take those rotations in which any line through the origin may be chosen as axis. Define such a rotation by any point on it other than the origin, and normalise this so that the defining point lies on the unit sphere. The results of this normalisation are the direction cosines of the axis of rotation, that is the cosines of the angles which this axis makes separately with each of the coordinate axes :

```
dircos=.% %:(+/@:*)      NB. direction cosines
dircos 3 4 5
0.424264 0.565685 0.707107
```

A necessary preliminary is to obtain the cross-product of the rotation vector and a point to be rotated. To my knowledge there is no primitive which delivers cross-products directly, however it is a reasonably straightforward to write a verb xp. First stitch 3 4 5 (defining the axis) to 1 2 3, a point to rotated, and use the 'all but one' technique described in J-ottings 52 to obtain the submatrices obtained by progressively eliminating one row at a time. The determinant of each of these 2x2 matrices is required with a suitable adjustment for alternating signs leading to :

```
xp=.4 : '1 _1 1*det each<"(2) 1+\.(dircos x),.y'
```

One other requirement is an identity matrix of appropriate length :

```
id=.=@i.@#      NB. identity matrix
```

Now suppose the anti-clockwise angle of rotation looking outwards from the origin is t. By a pleasing analogy with the two dimensional case the displacement components are

```
- (1 - cos t) times <a vector> - sin t times <a cross-product>
```

Where 'a vector' is the result of the matrix multiplication

$$\begin{pmatrix} 1-\lambda_1^2 & -\lambda_1\lambda_2 & -\lambda_1\lambda_3 \\ -\lambda_1\lambda_2 & 1-\lambda_2^2 & -\lambda_2\lambda_3 \\ -\lambda_1\lambda_3 & -\lambda_2\lambda_3 & 1-\lambda_3^2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

in which the λ s are the direction cosines of the axis of rotation. The parameters defining a rotation are thus an axis (three coordinates) joined to the angle t , and it seems natural to take this 4-element vector as the left argument of a rotation verb. Also many people are more comfortable with degrees rather than radians, so define :

```
dtor=.180%~o.          NB. degrees to radians
```

rm defines the rotation matrix above and rmdata multiplies it with the coordinates of the data point being rotated :

```
rm =.(id - */~)@dircos      NB. rotation matrix
rm 3 4 5
 0.82 _0.24 _0.3
_0.24 0.68 _0.4
_0.3 _0.4 0.5
rmdata=.rm@dircos@({:@[) +/ .* ]
(3 4 5,dtor 60) rmdata 1 2 3
_0.56 _0.08 0.4
```

(As an aside, taking the z axis as axis of rotation (0 0 1) so that $\lambda_3 = 1$ and $\lambda_1 = \lambda_2 = 0$ gives

```
rm 0 0 1
1 0 0
0 1 0
0 0 0
```

which multiplies $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ to give $\begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$, while the cross-product of $\begin{pmatrix} 0 & x \\ 0 & y \\ 1 & z \end{pmatrix}$ is $\begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}$ so that this reduces to the formula given earlier for the two-dimensional case.)

Next define m1 and m2, bearing in mind that t has to be extracted as the 4th element of the rotation vector:

```
m1=-.o.@(2&o.@({:@[)) NB. (1-cos t)
m2=.1&o.@({:@[)      NB. sin t
```

Finally reflect the formula for the displacement components in

```
rotate=.] - (m1 * rmdata) + m2 * }:@[ xp ]
(1 0 0,dtor 90)rotate 1 2 3
1 3 _2
```

A couple of further checks helps confirm understanding :

(a) Rotate the point (1,2,3) through a *clockwise* angle of 90° about the x-axis:

```
(1 0 0,dtor _90)rotate 1 2 3
1 _3 2
```

(b) Undo a clockwise rotation of (1,2,3) with an anti-clockwise one :

```
axis=.73$10 NB. choose an axis at random
(axis,dtor 60)rotate (axis,dtor _60)rotate 1 2 3
1 2 3
```

Multiple data points are dealt with by, for example

```
(<3 4 5,dtor 60)rotate each 1 2 3;2 3 1;3 1 2
1.03505 2.5299 2.55505
3.03722 1.56268 1.52753
1.82258 0.31773 3.25227
```

For practical uses consider that every minute of every day we all perform rotations on a merry-go-round called Earth which itself rotates continuously within an even larger solar system which itself gyrates around another even larger system and so on. Alternatively, from an earth-bound view, units of this larger system are in a perpetual state of 3-D rotation about what appears to us as a fixed axis. Perhaps there's an idea here for next time ...

L-systems in J

R.E. Boss (*r.e.boss@planet.nl*)

Introduction

There are not many examples of repeated squaring [1], but I stumbled upon a new one in 2006, solving a problem posed by Bron [2]. To explain it I use some examples and some theory, but then a new method is introduced in J. The term L-system in the title is short for Lindenmayer system [3] [4].

Rabbit sequence

This is an infinite sequence [5] of booleans which has the property that if one changes each 0 to a 1 and every 1 to the pair 1 0, the sequence does not change.

```
1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1
1 0 1 0 1 1 0 1 1 0 1 ...
```

Notice that this is the Algae example [6] in [3].

Lindenmayer systems

An L-system is a parallel rewriting system, i.e. consisting of an *alphabet* \mathbf{V} of symbols which can be replaced, a set of (production) *rules* \mathbf{P} such that each rule replaces a particular symbol of \mathbf{V} in a string of symbols (from \mathbf{V}), and an *initial symbol* \mathbf{S} to which the rules subsequently are applied.

All rules are applied at the same time, therefore the system is *parallel*.

In fact \mathbf{P} is a mapping from $\mathbf{V} \rightarrow \mathbf{V}^*$, the set of all strings with symbols from \mathbf{V} , including the empty string. If for some $a \in \mathbf{V}$ one has $\mathbf{P}(a)=s$, with s some string, this is also denoted by $a \rightarrow s$.

If for a symbol c in \mathbf{V} one has $\mathbf{P}(c) = c$, then c is called a *constant*.

If a symbol in \mathbf{V} is not a constant, then it is called a *variable*.

In the example of the rabbit sequence, \mathbf{V} is the set of booleans $\{0,1\}$.

The rules are $\mathbf{P}(0) = 1$ and $\mathbf{P}(1)= 1 0$ and the initial symbol \mathbf{S} is 0.

L-systems in J

To apply J [7] in producing an L-system, I prefer the alphabet **V** to be a set of integers, say {0,1,...,n-1}, such that (mostly) **S** is represented by 0.

Then the mapping **P** can be represented by an array of boxes, all of which contain an array of integers from **V**. The production rules are then $P(k) = ; k \{P$.

So for the rabbit sequence we get $V =: 0\ 1$, $S =: 0$ and $P =: 1; \ 1\ 0$.

Now the verb to produce the next generations of **S** is easy to construct:

```

P {~ S
+--+
|1|
+--+

P {~^:2 S
+----+
|1 0|
+----+
    
```

But after that we get a problem:

```

P {~^:3 S
|length error
| P {~^:3 S
    
```

This can easily be repaired by the following verb:

```

P ([:; {~)^:3 S
1 0 1
    
```

And now we can get a rabbit sequence of any length:

```

P ([:; {~)^:20 S
1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1
1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1
0 1 1 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1
0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1
    
```

Which can be given a bit more sophisticated by $P ([:; {~)^:(]`('S"_))\ 20$

Notice that if the production rules all have the same length, no boxing is needed, in which case the ; in the expression is replaced by a , .

More examples

Some examples from [3] are Cantor dust [7] and Koch curve [8].

The first is given in J by $V =: 0\ 1, S =: 0$ and $P =: 0\ 1\ 0; 1\ 1\ 1$ so

```
P ([:; {~}^:(])` (S"_) 3
0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0
```

Where 0 represents black and 1 represents white.

The Koch curve [8] is given by

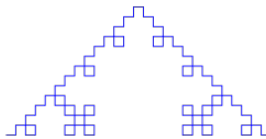
```
V=:0 1 2, S =: 0
```

and

```
P=: 0 1 0 2 0 2 0 1 0; 1; 2
```

```
P ([:; {~}^:(])` (S"_) 3
0 1 0 2 0 2 0 1 0 1 0 1 0 2 0 2 0 1 0 2 0 1 0 2 0 2 0 1 0 2 0 1 0 2 0 2 0
1 0 1 0 1 0 2 0 2 0 1 0 1 0 1 0 2 0 2 0 1 0 1 0 1 0 2 0 2 0 1 0 2 0 1 0 2
0 2 0 1 0 2 0 1 0 2 0 2 0 1 0 1 0 1 0 2 0 2 0 1 0 2 0 1 0 2 0 2 0 1 0 1 0
1 0 2 0 2 0 1 0 2 0 1 0 2 0 2 0 1 ...
```

With the graph below.



Given these examples, anybody can make her own.

Repeated squaring

Now the question (with which I perhaps whetted your appetite) in the introduction is still open, what about repeated squaring?

If you look at the verb $P ([:; {~}^:(])` (S"_)$ it is obvious the behaviour is rather linear. In each iteration a string is used to apply **P** on the individual symbols of the string and the results are concatenated.

Now consider P^k which is the mapping applied k times (on S which is 0). It is easy to construct in J by $(;@:{&. > <}^:(k-1)~ P$.

If we apply this to the rabbit sequence (V =: 0 1, S =: 0,P =: 1; 1 0) we get

```
(;@:{&.> <)^:(<4)~ P
+-----+-----+
|1          |1 0      |
+-----+-----+
|1 0        |1 0 1     |
+-----+-----+
|1 0 1      |1 0 1 1 0  |
+-----+-----+
|1 0 1 1 0|1 0 1 1 0 1 0 1|
+-----+-----+
```

As soon as (some of) these P^k are known, repeated squaring can start:

binind=: I.@|. @#: NB. The binary indices are determined

appl=: (;@:{&.> <) NB. Applying the main verb

rs=: 4 : '> { . appl/ appl^:(binind y) x' NB. x equals P and y is the generation giving the complete solution

The figures show that the performance is what you would expect, much better, while the output is the same:

```
ts'P ([:; {-}^:(J`S"_)35'
0.50670696 1.6777933e8

ts'P rs 35'
0.022058907 79695232

(rs=:P ([:; {-}^:(J`S"_)]) 35
1
```

In this example we can do even better, since for the rabbit sequence we have $P^{k+1} = P^k, P^{k-1}$, indeed Fibonacci!

```
ts'3 : '' ;|. appl/ appl^:(J`P"_) binind y-2''35'
0.012114756 46154880
```

So this gives a solution which is about 40 times as fast and 3.5 times as lean as the original one.

Final example, Gray codes

The least I owe the reader is the solution I found for the problem of Bron [2]. He asked for the most efficient verb to generate the Gray code, see [9] and [10]. Fortunately he also allowed to generate the decimal values of that code, which for the first 64 numbers look like:


```
G=: 0 1 3 2 6 7 5 4 12 13 15 14 10 11 9 8 24 25 27 26 30 31 29 28 20 21 23
22 18 19 17 16 48 49 51 50 54 55 53 52 60 61 63 62 58 59 57 56 40 41 43 42
46 47 45 44 36 37 39 38 34 35 33 32 .
```

The fractal structure of the code becomes apparent if you do

```
2--/\ G
1 2 _1 4 1 _2 _1 8 1 2 _1 _4 1 _2 _1 16 1 2 _1 4 1 _2 _1 8 1 2 _1 _4 1 _2
_1 32 1 2 _1 4 1 _2 _1 8 1 2 _1 _4 1 _2 _1 16 1 2 _1 4 1 _2 _1 8 1 2 _1
_4 1 _2 _1 .
```

To not let grow the numbers too quickly, I transformed it in

```
G1=: (** 2^ . 2* |) 2--/\ 64{.G
1 2 _1 3 1 _2 _1 4 1 2 _1 _3 1 _2 _1 5 1 2 _1 3 1 _2 _1 _4 1 2 _1 _3 1 _2
_1 6 1 2 _1 3 1 _2 _1 4 1 2 _1 _3 1 _2 _1 5 1 2 _1 3 1 _2 _1 _4 1 2 _1
_3 1 _2 _1 .
```

By the way, this representation of the (binary reflected) Gray code has a very useful interpretation: each number gives by its absolute value the coordinate where two consecutive code words differ, and by its sign how they differ: + if 0→1 and - if 1→0.

So I defined **V** to be the set of integers and $S=:1$. But most important was **P**. To produce G1, I defined

```
P=: 1; 1 2 _1; 3; 4; 5; 6;(…); _6; _5; _4; _3;1 _2 _1
```

Notice that in fact **P** is infinite, which is indicated by the (...).

Notice also that $(P\{~ -n) -: |.- n\{ P$ for $n > 0$, so in general we have $P=:3 : '1; (, [:|. -@|.&.>) 1 2 _1; ;/3+ i.y' n$ for some n , like

```
[P=:3 : '1; (, [:|. -@|.&.>) 1 2 _1; ;/3+ i.y' 6
+-----+-----+-----+-----+-----+-----+-----+-----+
|1|1 2 _1|3|4|5|6|7|8|_8|_7|_6|_5|_4|_3|1 _2 _1|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

The only thing one has to do is to transform this resulting fractal in the Gray code again, ie to invert $(** 2^ . 2* |)$ and precede it by 0 :

```
+/\0,(** 2%~ 2^|) P rs 6
0 1 3 2 6 7 5 4 12 13 15 14 10 11 9 8 24 25 27 26 30 31 29 28 20 21 23 22
18 19 17 16 48 49 51 50 54 55 53 52 60 61 63 62 58 59 57 56 40 41 43 42
46 47 45 44 36 37 39 38 34 35 33 32
```

which is equal to the original G. This, in principle, gives the impressive performance results in [2].

Conclusions

Using L-systems in J is a powerful technique for generating all kinds of fractal like structures. In a next paper I will describe how to transform these arrays in nice graphs using the plot facility of J.

Bibliography

1. RepeatedSquaring.
[Online]. <http://www.jsoftware.com/jwiki/Essays/Repeated%20Squaring>.
2. Bron. [Online]. <http://www.jsoftware.com/jwiki/Puzzles/Gray%20Code>.
3. Lindenmayer1. [Online]. <http://en.wikipedia.org/wiki/L-system>.
4. Lindenmayer2. [Online].
<http://mathworld.wolfram.com/LindenmayerSystem.html>.
5. RabbitSequence.
[Online]. <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibrab.html>.
6. Algae. [Online]. http://en.wikipedia.org/wiki/L-system#Example_1:_Algae.
7. CantorDust. [Online]. http://en.wikipedia.org/wiki/L-system#Example_3:_Cantor_dust.
8. KochCurve. [Online]. http://en.wikipedia.org/wiki/L-system#Example_4:_Koch_curve.
9. GrayCode1. [Online]. http://en.wikipedia.org/wiki/Gray_code.
10. GrayCode2. [Online]. <http://www.jsoftware.com/jwiki/Essays/Gray%20Code>.

Backgammon tools in J

4: Ace-point Bearoffs

Howard A. Peelle (hapeelle@educ.umass.edu)

J programs are presented as analytical tools for expert backgammon to compute the probability of winning a two-sided bearoff with stacks of pieces on both ace points, to generate binary probability trees, and to solve a related problem.

Utility programs:

```
ELSE =: `
WHEN =: @.
X =: [
O =: ]
```

The probability of the first player (X) winning is 1 minus the probability for the second player (O) -- which is the sum of probabilities for rolling double dice and single dice -- or else the default 1 when there are no pieces left:

```
Ace =: (1: - Double + Single) ELSE 1: WHEN (X<1:)
```

The probability of double dice is 1%6 times the result of `Ace` for the second player (now first, due to switching turns) with the same number of pieces to bear off and four less for the first player (now second). The probability of single dice is similar, but two less:

```
Double =: (1:%6:) * O Ace X-4:
Single =: (5:%6:) * O Ace X-2:
```

An alternative definition:

```
Ace =: (1: - Double + Single)~ ELSE 1: WHEN (<1:)
```

```
Double =: (1:%6:) * (Ace (-4:))
Single =: (5:%6:) * (Ace (-2:))
```

For example, the winning probability for the first player with 8 pieces vs. 6 pieces is:

```
8 Ace 6
0.301055
```

Here is a table of such probabilities for all odd-numbered stacks:

```

5.2 ": 1 3 5 7 9 11 13 15 Ace"0/ 1 3 5 7 9 11 13 15
1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
0.17 0.86 1.00 1.00 1.00 1.00 1.00 1.00 1.00
0.00 0.25 0.79 0.98 1.00 1.00 1.00 1.00 1.00
0.00 0.02 0.30 0.75 0.96 1.00 1.00 1.00 1.00
0.00 0.00 0.05 0.33 0.72 0.93 0.99 1.00 1.00
0.00 0.00 0.00 0.08 0.35 0.70 0.91 0.99 1.00
0.00 0.00 0.00 0.01 0.10 0.36 0.68 0.90 1.00
0.00 0.00 0.00 0.00 0.02 0.12 0.37 0.67 1.00
    
```

Results for even numbers one larger (Ace"0/~ +:>:i .8) are the same. These results can be confirmed by program Pwin in [3] using probability distributions. For example, 8 0 0 0 0 0 Pwin 6 0 0 0 0 0 is 0.301055.

Now generalize these programs to generate a binary tree for any input probability P (and its complement) to a given level L:

```

P =: [
L =: ]

BTree =: (Left,Right) ELSE 1: WHEN (L<1:)

Left =: P * P BTree L-1:
Right =: -.@P * P BTree L-1:
    
```

This can be used to generate probabilities for n-roll bearoffs. For example, to bear off 3 pairs of pieces:

```

1r6 BTree 3
1r216 5r216 5r216 25r216 5r216 25r216 25r216 125r216
    
```

The sum of any p BTree n is 1, and the previous level p BTree n-1 can be reconstructed by _2 +/\ p BTree n. The whole tree is:

```

1r6 BTree"0 i.4
1 0 0 0 0 0 0 0
1r6 5r6 0 0 0 0 0 0
1r36 5r36 5r36 25r36 0 0 0 0
1r216 5r216 5r216 25r216 5r216 25r216 25r216 125r216
    
```

Problem

Here is a related conundrum to solve: Given a stack of pieces on the ace-point, what is the probability of bearing off the final pieces with a doubles on the last roll? [This problem was posed by Walter Trice, well-known backgammon author and world-class player.] See end of article for answer.

It is easy to modify BTree to produce an asymmetric binary tree by truncating a branch:

```
BTreeA =: (LeftA , RightA) ELSE 1: WHEN (L<1:)
  LeftA =: P * P BTreeA L-2:
  RightA =: -.@P * (P BTreeA L-1:) ELSE 0: WHEN (L<2:)
```

Use it to explore the problem:

```
+ / 1r6 BTreeA 8      or      + / (1%6) BTreeA 8
479891r1679616      0.285715
```

Or, use a variant of Ace (above):

```
DoubleOffAce =: (D + S) ELSE 1: WHEN (<1:)
  D =: (1:%6:) * DoubleOffAce@(-4:)
  S =: (5:%6:) * DoubleOffAce@(-2:) ELSE 0: WHEN (<3:)
```

For instance, DoubleOffAce 15 is 0.285715.

Answer

The probability of bearing off a stack on the ace-point with doubles is 2%7.

+/"1 (1r6 BTreeA)"0 i.10 shows oscillating convergence, or DoubleOffAce"0 i.20 (in pairs).

References

1. Peelle, Howard A. "Backgammon Tools in J (Part 1) Bearoff Expected Rolls", Vector, Vol. 24, No. 2&3
2. Peelle, Howard A. "Backgammon Tools in J (Part 2) Wastage", Vector, Vol. 24, No. 4
3. Peelle, Howard A. "Backgammon Tools in J (Part 3) Two-sided Bearoff Probabilities", Vector, Vol. 25, No. 4

Fibonacci and golden spirals

Cliff Reiter (reiterc@lafayette.edu)

A spiral of squares with Fibonacci edge lengths is created in J. Spiral Fibonacci and Golden curves are also explored.

1. Introduction

During renovations of an abandoned farmhouse much demolition debris was created. Almost all of it went to the dump, but occasionally a piece of old wood was saved. Such a pine board seemed well suited to make a two faced clock for a partition exposing post and beam construction. I considered several natural and mathy designs for the clock. I decided a golden ratio rectangle would have a nice overall form and soon focused on a Fibonacci spiral of squares that would approximate the golden ratio and illustrate the proof of a Fibonacci identity.

The Fibonacci sequence may be defined by $F_1=1$, $F_2=1$, and $F_j=F_{j-1}+F_{j-2}$ for integer indices j greater than 2. The Fibonacci sequence has many remarkable properties, surprising applications and lovely connections to nature. [1],[2],[4],[6],[7]

We can implement a verb to generate the positive Fibonacci sequence of specified length (greater than or equal to 2) as follows. The verb iterates the process of appending the sum of the last two items an appropriate number of times.

```
pos_fib_seq=:3 : '(,[:+/ _2&{.)^(y-2) 1 1'
pos_fib_seq 8
1 1 2 3 5 8 13 21
```

If we place two 1 by 1 squares adjacent along an edge then the assembly forms a 1 by 2 rectangle. If we put a 2 by 2 square adjacent to the length 2 edge of that, the new assembly forms a 2 by 3 rectangle. If we put a 3 by 3 square adjacent to the length 3 edge, then the new assembly forms a 3 by 5 rectangle. We can continue this process resulting in a F_j by F_{j+1} assembly at the j^{th} stage.

We organize the placement of the edges so that we adjoin edges sequentially on the south edge, east edge, north edge and then west edge and then continue repeatedly. A spiral pattern of squares with edge lengths given by the Fibonacci numbers develops.

See Figure 1 for a hand sketch of the idea we have in mind for the spiral of squares and a Fibonacci spiral. Notice the spiral curve is tangent to the intersection points as shown. We will describe one way to define such a spiral curve in Section 3.

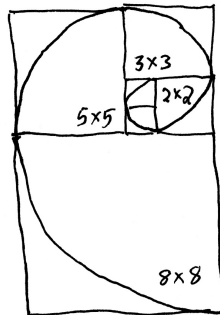


Figure 1. Rough sketch of a spiral of Fibonacci squares and a spiral curve.

In the next section we will discuss J verbs to create these spiral assemblies of squares. Then we will look at spiral curves that "fit" the pattern of squares.

2. Assembling the Fibonacci spiral of squares

We begin by defining matrix `fs_sq1` (Fibonacci spiral square of size 1) with rows giving the vertices of the unit square with diagonal corners at (0,0) and (1,1).

```
]fs_sq1=#:0 2 3 1
0 0
1 0
1 1
0 1
```

The main function `draw_fib_sqs`, defined below, organizes the overall assembly. Its argument is the number of squares to create. It initializes three global variables and then updates them at each stage. At each stage, the next square will be a translated version of the unit square `fs_sq1` multiplied by the current Fibonacci number. Since the details depend upon the side we adjoin the next square, we organize by cases, according to the stage modulo 4. The three global variables are updated using the conjunction `nx_fs_sq`.

The three global variables are the bounds of the assembly, `fs_wsen`, (Fibonacci spiral: west-south-east-north), and the list of squares, `fs_sqs`, that form the spiral

pattern and the list of arcs, `fs_arcs`, giving a spiral curve. We will not fully discuss the arcs until the next section. Looking at the core conjunction, `nx_fs_sq` (next Fibonacci spiral square), it updates the bounds of the assembly at each stage using the global variable `fs_wsen` by adding or subtracting a Fibonacci number corresponding to the just joined square. The conjunction also adds a new square, `nsq` to the global list `fs_sqs` that gives a list of the coordinates of each of the squares. And the list of arcs, `fs_arcs` is updated. Notice that the new square `nsq` is translated so that the lower left has the appropriate coordinates. As we noted, the details of how to do that depend upon the side we adjoin the next square.

For example, when we adjoin a new square to the west, the first two entries of `fs_wsen` give the position of the lower left corner. However, when adjoining to the east, the upper right coordinates are available from `fs_wsen` but we need to subtract the current Fibonacci number from those coordinates to obtain the lower left. The conjunction `nx_fs_sq` accomplishes these tasks. It also presumes a global variable `F_j` giving the current Fibonacci number exists. Note that we use `fibs` as a local list of positive Fibonacci numbers and index to it to obtain `F_j`. The for-loop runs through all those Fibonacci numbers; however, its indices are one off from the standard indices of the Fibonacci numbers.

The arguments to `nx_fs_sq` are as follows: `x` which gives a unit vector specifying the direction in which `wsen` should be modified, `m` which gives the indices of `wsen` that give a corner of the square, `n` which is used to translate the first entry in the new square to be the lower left corner of `nsq`, and `y` which gives the indices of the points of `nsq` on the arc that should be drawn to get the Fibonacci spiral. The main function `draw_fs_sqs` also plots the spiral of squares.

The functions require that we load two scripts from the add-ons to give some utilities for interactively plotting polygons and defining a nice colour sequence. If you want to duplicate these experiments you should have downloaded the add-ons required for the scripts below (currently these are available for J6.02, 32 bit [3]). A script containing the J expressions in this note may be found at [5].

```
load '~addons/graphics/fvj3/dwin2.ijs'
load '~addons/media/image3/image3.ijs'

draw_fs_sqs=:3 : 0
fibs=.pos_fib_seq y
fs_wsen=: 0 0 0 1
fs_sqs=: i. 0 4 2
fs_arcs=: i.0 8
for_J. i. y do.
  F_j=:J{fibs
  select. 4|J
```



```

case. 0 do. NB. add onto west
_1 0 0 0 (0 1 nx_fs_sq 0 1) 2 0
case. 1 do. NB. add onto south
0 _1 0 0 (0 1 nx_fs_sq 0 0) 3 1
case. 2 do. NB. add onto east
0 0 1 0 (2 3 nx_fs_sq 1 0) 0 2
case. 3 do. NB. add onto north
0 0 0 1 (2 3 nx_fs_sq 1 1) 1 3
end.
end.
range=. (2 3{fs_wsen)-0 1{fs_wsen
WIN_WH=:range*<.<./0.8*(_2{"wd'qm')%range
fs_wsen dwin 'Fibonacci Spiral'
(Hue *(i.%])#fs_sqs) dpoly fs_sqs
)

nx_fs_sq=:2 : 0
:
fs_wsen=:fs_wsen+x*F_j
nsq=.((m{fs_wsen)-F_j*m-:2 3)+ "1 fs_sq1*F_j
fs_sqs=:fs_sqs,nsq
fs_arcs=: fs_arcs,((0{nsq)-n*F_j),(2#2*F_j),,y{nsq
)

```

Now we can run an 8 square spiral. The result is shown in Figure 2 and the list of squares and bounds from the global variables are given below.

```

draw_fs_sqs 8

fs_wsen
_6 _9 15 25

<"2 fs_sqs
+-----+-----+-----+-----+
|_1 0|_1 _1|0 _1|_1 1|_6 _1|_6 _9| 2 _9|_6 4|
| 0 0| 0 _1|2 _1| 2 1|_1 _1| 2 _9|15 _9|15 4|
| 0 1| 0 0|2 1| 2 4|_1 4| 2 _1|15 4|15 25|
|_1 1|_1 0|0 1|_1 4|_6 4|_6 _1| 2 4|_6 25|
+-----+-----+-----+-----+

```

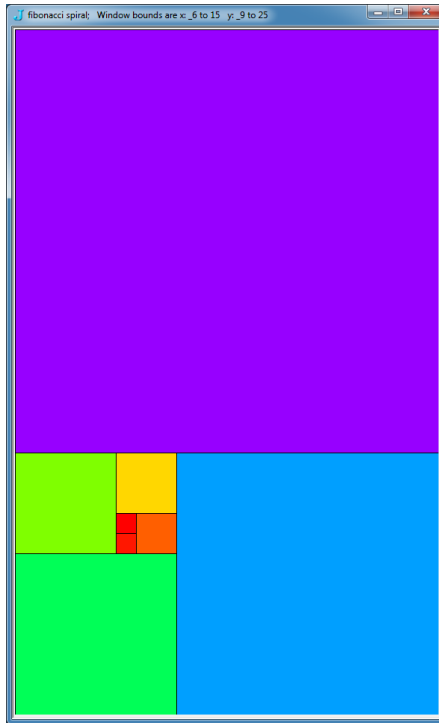


Figure 2. A spiral of eight Fibonacci square.

Now suppose we consider the area of the assembled rectangle in two ways. Suppose the last square added had an edge length of F_n , then the other edge of the assembled rectangle has length $F_n + F_{n-1} = F_{n+1}$; thus, the total area of the assembled rectangle is $F_n F_{n+1}$. On the other hand, the area of the assembled rectangle is the sum of the area of all the squares which is $F_1^2 + F_2^2 + F_3^2 + \dots + F_n^2$. Thus, we have given a geometric proof that $F_n F_{n+1} = F_1^2 + F_2^2 + F_3^2 + \dots + F_n^2$. This identity is well known and commonly appears in lists of Fibonacci identities [1, 2, 4, 6, 7]. Figure 3 shows one face of the pine board clock illustrating that fact.



Figure 3. A clock showing the Fibonacci spiral of squares.

3. The Fibonacci spiral

A simple Fibonacci spiral can be drawn that places a quarter circle in each square [1, 2, 4, 5, 6, 7]. The add-on *graphics/fvj3/dwin2.ijs* does not have a utility for drawing arcs, so we will define one similar to the style of the add-on. For the right argument we use the same parameters as the `glellipse` function. Namely, we give a corner of the drawing window, the extent of a box that bounds the ellipse, and start and end points for the arc (given as points on a ray from the center, not necessarily points on the ellipse). The left argument is a boxed array giving the RGB colour and the pen style for the arc. Notice that `darc` converts its data to window coordinates using the global function `SC` so it requires some adjustments since the extent needs to be scaled, but it is not a coordinate.

```

    darc=:3 : 0"1
(0 0 0;1 0) darc y
:
wd 'psel ',WIN_nam
glrgb >{.x
glpen >{:x
'a b c d'=.4 2$y
'A B C D'=.SC a,(a+b),c,:d
glarc x:^:_1 A,(B-A),C,D
glpaint ''
)

```

The result of the following expressions is shown in Figure 4.

```
draw_fs_sqs 8
(0 0 0;3 0) darc fs_arcs
```

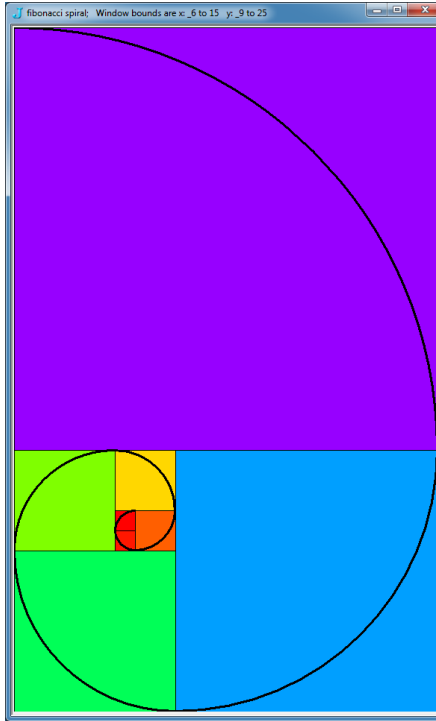


Figure 4. A Fibonacci spiral of square and a spiral curve consisting of quarter circles.

Since we are drawing arcs of circles the curvature is constant on the interior of each square. However, the curvature discontinuously changes at each intersection point. One might hope for a smooth spiral with continuous derivatives of all orders. Such a curve seems plausible although we have not seen one in the literature. However, in the next section we will describe a smooth golden spiral that approximates the Fibonacci spiral.

A variant of the above Fibonacci spiral is the Great Fibonacci spiral. It is created by

drawing the complements of the arcs (in their circles) that we used in the Fibonacci spiral. This can easily be done by interchanging the start and end points of the arcs as below.

```
draw_fs_sqs 8
(0 0 0;3 0) darc 0 1 2 3 6 7 4 5{"1 fs_arcs
```

The result is not shown but is worth exploring.

4. The golden spiral

The golden spiral is a spiral given in polar coordinates by $r = a e^{b\theta}$ where $b = 2 \ln(\tau)/\pi$ and $\tau = (1 + \sqrt{5})/2$ is the golden ratio.[4], [6], [8] We take $a = 1/(2 \tau)$ to fit our orientation of the spiral of squares. The argument to `draw_golden_spiral` gives the ending angle (in radians).

```
draw_golden_spiral=:3 : 0
(0 0 0;1 0)draw_golden_spiral y
:
wd 'psel ',WIN_nam
glrgb >{.x
glpen >{:x
gr=.->:%:5
]b=(^ . gr)%1r2p1
]r=. ^@:(b&*)
X=.r * cos
Y=.r * sin
]a=. %2*gr
z=. a *(X,.Y) y *(i.%)1000
glines ,x:^_1 SC 2{"1 z
glpaint''
)
```

In Figure 5 we see the result of the following expressions.

```
draw_fs_sqs 8
(0 0 0;3 0) darc fs_arcs
(255 255 255;2 0) draw_golden_spiral 5p1
```

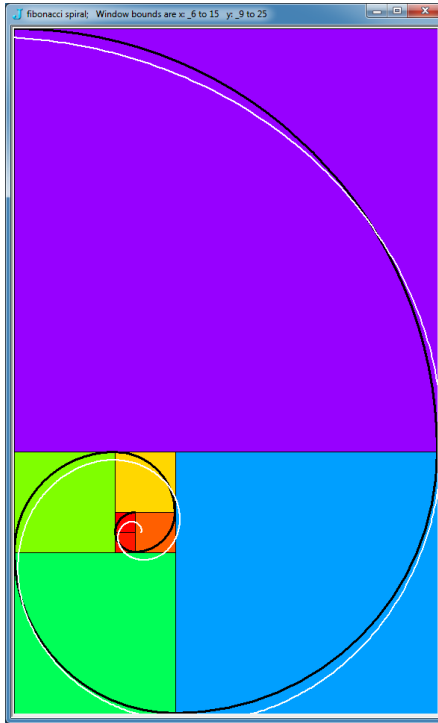


Figure 5. Fibonacci and golden spirals on eight squares.

Notice in Figure 5 that the golden spiral does not seem to be a close approximation to the Fibonacci spiral, especially near the center.

In Figure 6 we see the result of the following expressions where we plot four additional squares.

```
draw_fs_sqs 12
(0 0 0;3 0) darc fs_arcs
(255 255 255;2 0) draw_golden_spiral 7p1
```

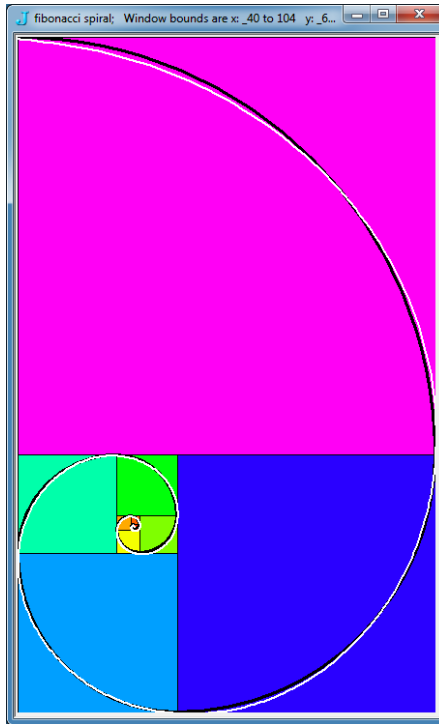


Figure 6. Fibonacci and golden spirals on 12 squares.

In Figure 6 the golden spiral is a good approximation for the large squares. Notice that as we move outward along the golden spiral the curve bends more slowly than the circle at the beginning of the square and it bends faster than the circle toward the end of the square. Many other geometric designs based upon the Fibonacci numbers and the golden ratio may be found.[4], [6]

References

1. Richard A. Dunlap, The Golden Ratio and Fibonacci Numbers, World Scientific, 1997
2. Vener E. Hoggatt, Jr., Fibonacci Numbers, The Fibonacci Association, 1969.
3. Jsoftware, J6.01c, with Image3 and FVJ3 addons, <http://www.jsoftware.com>. 2007
4. Alfred S. Posamentier and Ingmar Lehmann, The Glorious Golden Ratio, Prometheus Books, 2012.

5. Cliff Reiter, Fibonacci and Golden Spirals Script, http://webbox.lafayette.edu/~reiterc/j/vector/fib_spiral.html, 2013.
6. Hans Walser, The Golden Section, translated by Peter Hilton et al, The Mathematical Association of America, 2nd ed., 1996, translation, 2001.
7. Wikipedia: Fibonacci Numbers, http://en.wikipedia.org/wiki/Fibonacci_number
8. Wikipedia: Golden Ratio, http://en.wikipedia.org/wiki/Golden_ratio