Contents

Editorial

Mankind only sets itself such problems as it can solve; since, looking at the matter more closely, it will always be found that the task itself arises only when the material conditions for its solution already exist or are at least in the process of formation.
Karl Marx, A Contribution to the Critique of Political Economy(1859)

OK so I stole the above quote form the preface to David Lewis-Williams - The Mind in the Cave(2002). In this book the author seeks a methodology for analysing palaeolithic art.

In putting together this issue the quotation seemed to be appropriate where the material conditions exist for Robert Pullman to employ *“Notation as a tool for proof”*; Dan Baronet to use APL as *“A tool for thought”*; and Brian Becker to give us *“One reason APL is cool”* in their respective articles.

Then for material conditions in the process of formation there is Stephen Taylor’s article *“Impending kOS”* in which he recounts Arthur Whitney and his team‘s progress towards kOS.

Later, at the August meeting of BAA London, Stephen Taylor remarked that his current apprentice Dhrusham Patel, had in his debut article *“Table Diff”* found himself modifying his examples to clarify his thoughts for the reader.

This year is the thirtieth since the first publication of *Vector* and I should like to mark this in some way in the next issue. To this end I would welcome any suggestions, anecdotes, photographs or whatever.

Earlier this year at the AGM in May there was a suggestion that *Vector* should publish a section of useful links to resources and material of general interest to the APL community. Here again I would like to invite suggestions.

John Jacob

News

BAA: Chairman’s Report 2014

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

Once again our production team has been working to produce this edition which for the second time comes with some 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, please let us know?

BAA Chairman - Paul Gosvenor

This year our AGM was held on Friday 23rd May, for the first we also ran it as a webinar to allow even more of you to attend, especially those outside the UK. Those of you who elected to come to our meeting in person attended the group meeting held at the same time in the Albion in London as we have done in the past.

I hope that you will have received our new BAA Newsletter. We aim to send out a regular bulletin to keep all of our members up to date with what is going on and we can include any news or links that are appropriate from you. Please let us know if there is anything that you would like including and we will do our best to distribute. If you have not seen the Newsletter please check your spam filter in case it has gone in there. We hope you enjoy!

I hope you are pleased with our journal and look forward to seeing some of you at the forthcoming conferences and just to finish off, a comment from Roger;

> “I started in 1966 on an APL machine that weighed 15,000 tons, when I travelled from Hong Kong to San Francisco on the S.S. President Wilson of the American President Lines.”
>
> — Roger Hui

Thank goodness for the microchip …..

BAA AGM Minutes 2014

John Jacob (editor@vector.org.uk)

Minutes of the British APL Association AGM 2014 held on-line by webinar and at The Albion, 3 New Bridge Street, London EC4 on Friday 23 May 2014

1. The **Minutes of the 2013 AGM** (as published in Vector 26:1) were accepted by general consensus of those present.

2. **Report from the Chairman**

   With apologies from Paul Grosvenor, Peter Merrit (Acting Chair) presented the Chairman's report on his behalf.

   We continue to see some colour added into *Vector*, I hope everyone finds that a good thing to see. The first BAPLA newsletter went out in December 2013 and are planning to send a further two throughout the year so if anyone has anything to announce then we are willing to include that as a service free to members.

   BAA London continues to meet regularly and my thanks to their team for allowing us to piggy back this AGM on their meeting.

   We have started to broadcast our meetings as webinars where appropriate and for those of you listening in today a special welcome. Its early days but the production team are trying and make this as easy as possible for everyone. We are hoping to publish some of the talks that are captured. Jake is in the process of editing those at the moment.

   This year sees the thirtieth aniversary of the first publication of *Vector* which is quite an achievement. Later in the year we would like to publish a bumper edition and we would like as many BAPLA members as can to contribute. These contributions can take any form you like, one-liners; doodles; embarassing photographs; or old articles. It really does not matter this is your chance to get involved in making this quite an issue.

   Asked what were the deadlines for submission to *Vector*. John Jacob(Jake) confirmed that there wasn't a deadline as such rather that an issue of *Vector* depended on accumulating sufficient copy to send for print.

Asked if *Vector* were available in PDF form. John Jacob confirmed that PDF version was made available to sustaining members at print time, with plans to make it publicly available at a later stage.

3. **Report from the Treasurer & Membership Secretary** (Nicholas Small)

Very little change in our financial situation with total receipts just under £3,000 and payments just under £3,000. Cost of posting *Vector* overseas being the most significant part. Chris Hogan (Auditor) confirmed the accounts.

Number of *Vectors* subscribed for is down by thirty-five. Five of these accounted for by Soliton ceasing subscription. The overall circulation of *Vector* was around 325 with about ten copies to libraries.

Nicholas Small confirmed that dues for BAPLA membership fell due when at the end of a volume of *Vector*.

Peter Merit confirmed that a large package had been included in the Dyalog conference pack.

British APL Association - summary of annual accounts

Summary of income and expenditure/receipts and payments:

Income/Receipts	**2013/14 (R&P)**	**2012/13 (R&P)**	**2011/12 (R&P)**
Subscriptions	2,964.42	7,142.75	3,793.00
Other	0.00	217.00	0.00
Total receipts	2,964.42	7,359.75	3,793.00
Expenditure/Payments			
Meetings	0.00	0.00	0.00
Administration	17.25	0.00	36.00
Vector production and despatch	2,639.01	2,265.85	3,882.00
Conferences and seminars	113.92	0.00	0.00
Other	157.13	104.40	69.00
Total payments	2,927.31	2,370.25	3,987.00
Assets summary:			
Bank and other balances	12,246.10	12,195.61	7,049.00
Debtors	1,197.50	3,527.50	3,183.00
Creditors	-6,425.00	-7,955.00	-3,165.00
Net assets	7,018.60	7,768.11	7,066.00

BAPLA membership at May 2014 (after Vector 26:1)
(volume 25 figures in parentheses)

	UK		**FOREIGN**		**TOTAL**	
	Number	**Vectors**	**Number**	**Vectors**	**Number**	**Vectors**
Sustaining*	5(5)	23(23)	5(5)	42(42)	10(10)	65(65)
Corporate*	0(1)	0 (5)	2(2)	15(15)	2(3)	15(20)
Corp. Ind*	5(5)	5 (5)	2(2)	2(2)	7(7)	7(7)
Individual	48(56)	47(55)	129(147)	129(147)	177(203)	176(202)
Non-voting	14(16)	14(16)	0(0)	0(0)	14(16)	14(16)
Life	0(0)	0 (0)	0(0)	0(0)	0(0)	0(0)
Library	1(1)	1 (1)	2(4)	2(4)	3(5)	3(5)
Russians			11(11)	11(11)	11(11)	11(11)
APL Groups			12(12)	34(34)	12(12)	34(34)
						325(360)

*Add the Vector numbers in these rows to get the total subscribed for by corporate and sustaining members

4. **Committee for 2014/2015**: It was suggested that as no applicants for posts had been received that the existing committee be returned, proposed Paul Grosvenor seconded by Ronny Simon.

5. **Appointment of Auditor**: The current auditor (Chris Hogan) was proposed by John Jacob and seconded Ronny Simon. Accepted by those present.

6. **General Questions**

There was a discussion as to whether *Vector* should be encouraging vendors to share more of their material as a way to encourage a much more a holistic community for APL rather than the different vendors encouraging their own communities. There was general agreement that a section in Vector with useful links to sites or other material and some initial candidates were identified:

- LinkedIn group: APL- A Programming Language https://www.linkedin.com/groups/APL-Programming-Language-1805002
- [kx] Technology Network https://www.linkedin.com/groups?home=&gid=773547
- Iversonians https://www.linkedin.com/groups/Iversonians-44369?home=&gid=44369
- APLWiKi - http://aplwiki.com
- comp.lang.apl - http://groups.google.com/forum/#!forum/comp.lang.apl
- BAA London - http://groups.google.com/forum/#!forum/baa-london

Dyalog Ltd

Morten Kromberg

2014 has been another year of increasing activity for everyone at Dyalog Ltd! One of the encouraging trends that we have noticed is that array language user meetings seem to be on the rise. In addition to the monthly BAA London meetings, which we try to attend regularly, members of the team have participated in three APL User Meetings in Europe this spring: SwedAPL in Stockholm and the FinnAPL Forest Seminar in April, and APL Germany in Stuttgart in May (all of these groups meeting twice a year). July was incredibly hectic – in addition to hosting our own seminar for Dyalog users in New York, we dispatched delegations to Iverson College in Cambridge, UK, and the J Conference in Toronto, Canada.

Many of our presentations at these meetings have focused on one of the most exciting new language features of version 14.0, futures and isolates. These aim to put the power of parallel hardware at the fingertips of both expert and novice users by making it easy to make asynchronous function calls.

Version 14.0

Version 14.0 is the most significant new release since v11.0 added support for object-oriented programming. It has been available on all supported platforms (Microsoft Windows, IBM AIX, Intel and ARM Linux) since June 30th. Highlights of version 14.0 include:

- Performance enhancements in the language engine; early adopters have reported speed-ups of 10-30% without application code changes.
- File system speed-ups and functional enhancements, including the ability to automatically compress file components and read several components in a single operation.
- Several new primitive language features, including the operators rank and key, the function tally, and function trains (similar to those in the J language). Many of these enhancements have the potential to further enhance application throughput (and simplify your code)!
- New APL language constructs designed to make it straightforward to use distribute computation across multiple processors without relying on locks or semaphores for synchronisation (futures and isolates).

- An experimental compiler that can reduce interpreter overhead of small utility functions.
- Syncfusion libraries for Windows Presentation Foundation and Javascript are bundled with Dyalog version 14.0, making it significantly easier to build state-of-the-art applications for desktop and web deployment.
- Data Binding with Microsoft .NET components allows APL applications to share data in real time with WPF GUI components and other tools that support data binding.
- An interface to the R framework for statistical computing.

New Platforms, and the Remote IDE

It is our intention to add support for several new platforms. If all goes according to plan, then version 14.1 will add official support for MAC OSX in Q1 of 2015 (contact us if you would like to help us test the new platform in Q4). Android is probably next, and we are also looking at iOS and "Windows Modern". A key component of the plan is to provide a new graphical development environment, the Remote Integrated Development Environment (RIDE), that will provide consistent functionality on all platforms. One of the main features of the RIDE is that you can run the RIDE and your APL engine on different machines. For example, you will be able to use a Windows-based RIDE to develop and maintain applications running under AIX and Linux – or on remote or inaccessible Windows Servers.

New Web Site and Social Media Channels

On the same day that v14.0 was released, we launched a completely reworked website. If you want to read more about version 14.0 and download the 500+ pages of related documentation and tutorials, then please visit http://www.dyalog.com. The site includes a blog, which will have frequent contributions from members of our development team, many of them involving Raspberry Pi-driven robots. We also launched active presences on Twitter, Facebook and LinkedIn, in addition to our own forums at http://forums.dyalog.com. We hope that these new initiatives will make it much easier to stay informed about our activities and both new and old functionality of our products. Please follow Dyalog on one or more of these channels to receive regular updates from us – none of the resources require you to have a licence for any of our products.

Another successful Annual Programming Contest

This year, http://studentcompetitions.com hosted and ran the contest for us, and this definitely extended our reach. About 40 students submitted Phase I solutions, and nearly 20 made it all the way through both phases of the contest. This year's winner of the $2,500 grand prize is Emil Bremer Orloff from the University of Aarhus, Denmark, who also wins a trip to the Dyalog User Meeting in Eastbourne. The winner of the new category for non-students, where the prize is free conference attendance, is Iryna Pashenkovska from SimCorp Ukraine. We hope to see them both next month!

Come to the Dyalog User Meeting!

For the first time in the last decade, the Dyalog User Meeting 2014 returns to the UK; it will be held on the south coast of England in Eastbourne on 21st-25th September . With a month to go we already have over 100 registered participants, so we are on track to set a new all-time record! By the time this goes to press, hotel rooms at the conference hotel will almost certainly be sold out; fortunately there are many alternatives nearby.

Last year we recorded about 25 of the main conference sessions – for a total of nearly 16 hours of viewing. If you were not fortunate enough to attend the meeting at Deerfield Beach in Florida, make sure to visit http://video.dyalog.com and watch the recordings from this and several earlier conferences. Highlights of the 2013 meeting include:

- The Stormwind Simulator – by Tomas Gustafsson, winner of the main category in Apps4Finland competition (more at https://www.facebook.com/Apps4Finland)
- Computer Science Outreach and Education with APL – by Aaron Hsu of the University of Indiana
- Social Skills for Programmers – by our own John Scholes

We'll be recording as many sessions as we can in Eastbourne. However, if you want to network with other array language users in addition to watching presentations on both new and mature applications of Dyalog APL, or attend tutorials and workshops on version 14.0 features and associated tools, Eastbourne will be the place to be in the penultimate week of September!

Optima Systems Ltd – Industry News August 2014

Paul Grosvenor – Managing Director

First off let me welcome Mike Mingard to the team. Mike has just joined us as our Graphics Designer and UI expert. Since joining us he has already had a huge impact on many of our developments. They are now looking good and behaving better!

Talking of behaving better our three trainees (aka "The Three Blind Mice") are coming on very well and now getting involved with many of our clients. We have even had a few come back to say how impressed they were. Now I don't know how much they paid them but that does not happen very often !

We also welcome our new Apprentice, Callum, who joins us after completing his "A" level qualifications. We hope that Callum will be a great addition to the team overall. He has completed his APL course at Dyalog; next stop will be the user conference in Eastbourne. Callum has a blog for anyone to see how he gets on.

On the subject of user conferences we sent a group of five staff members to the Dyalog conference in Miami last year which was very enjoyable and uplifting. Our three trainees showed off their robots which rather surprisingly made it across the Atlantic without too much damage. Follow their progress on their blog.

Our COSMOS™ data visualisation product continues to move from strength to strength with a number of contracts now starting. Most of the product activity has been in America through our partner company Galileo Analytics but we hope to start a sales pipeline in Europe and UK shortly.

Our Swedish subsidiary Data Analytics Sweden AB from which much of our R&D work will be performed is now up and running. It is from here that the COSMOS™ product and other technologies will get built, tested and distributed.

We can 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. With the addition of Mike we now also have a fully-fledged graphic design facility to complete our design package.

We expect the next twelve months to be even more exciting than the last !

4xtra Alliance - News

by Chris Hogan (chris.hogan@4xtra.com)

For those of you who have started reading this issue of Vector by turning to the back cover (I will resist the temptation to make some joke about "isn't that what all APLers do?" - ah I've gone and done it), you might notice the apparent disappearance of HMW Computing as a sustaining member and a "new" one appearing in its place - the 4xtra Alliance. So time for a little explanation.

Firstly, HMW hasn't vanished, but the new arrangement reflects what has actually been happening for several years.

HMW Computing started over 30 years ago, with the somewhat more long winded name of "HMW Programming Consultants". We changed our name to HMW Computing back in the late 80s to show we were (and really always had been) doing more than "mere" programming and that by that time we were primarily supplying 4xtra - a foreign exchange trading system 4X- Tra - rather than being a team of freelance consultants.

Indeed 4xtra continued to be our primary focus for over seventeen years. During that time HMW's personnel line-up changed significantly and the use of 4xtra seemed to go into decline. I'm afraid that although we were pioneers in the field of electronic trading systems, we simply weren't a large enough organization to compete (we thought) when the bigger players moved into the field.

So we were a little surprised when a client came back to us because our APL system could still out perform a new platform written in C++ using a Sybase database. The problem was then to support this client at short notice. So we turned to two former employees of HMW, John Jacob and Phil Last, who by then were running their own companies. The solution was a joint venture between HMW, John Butler Associates and Phil Last Limited to support the client.

This arrangement has adapted to changes so well it has survived the client being taken over, Phil converting to a sole trader and Jake becoming an employee of Optima systems.

Now of course Jake is the editor of Vector, Phil the events officer and I (Chris Hogan) am, for my sins the auditor of the British APL Association, although I do try to help out elsewhere too, if not on the committee.

We have decided that it is finally about time the sustaining membership of the BAA reflects the reality of the way we've been working for the past 14 years. So from now on the membership will be in the name of the 4xtra Alliance, rather than selfishly showing only HMW.

You might also have noticed that 4xtra has a different address than HMW - we thought we had better update our details with another change which happened almost three years ago. Hamilton House, our offices since 1987 was an excellent location when most of our clients were in the City of London and we had anything up to 14 people in the office at once, but proved less so when our clients have become more scattered geographically and most of our work is now done remotely. So in 2011 faced with increasing costs and changes to the terms and conditions of our lease, we sadly vacated the offices which had been our base for 25 years.

Beyond these formalities nothing has changed. Jake, Phil and I still operate as three separate entities, but we assist each other with our APL activities and band together as 4xtra whenever we need more resources and it suits the needs of our clients. So you should be able to see all three of us if you come along to the next BAA London meeting.

APL2000 User Conference 2014

APL2000 Welcomed Attendees in Fort Lauderdale, Florida

On March 23-25, 2014, APL enthusiasts gathered at the Gallery One Fort Lauderdale– A Doubletree Suites by Hilton, for the APL2000 User Conference 2014. The hotel was beautifully situated along the scenic Intracostal Waterway, 8 miles from the Fort Lauderdale/Hollywood International Airport (FLL) and 3 blocks from the beach.

Conference attendees reflected the diversity among users of APL2000 software. They are diverse both in the broad span of industries in which they work as well as the size of their businesses. APL2000 customers are industry leaders in the fields of finance, insurance, healthcare, aerospace engineering, employee benefits, airline and travel and many others both in the US and abroad.

The conference had a full agenda focusing on new developments in APL2000 products and various topics of interest to APL programmers including multi-threading options to increase processing performance, using the C# Script Engine to access the .NET framework directly from APL+Win and techniques to incorporate APL+Win applications in cross-platform solutions.

A comprehensive, 3-day “Introduction to APL” class, taught by Kevin Weaver, was held simultaneously with the APL2000 User Conference.

At the APL2000 Conference two years ago Professor Spyros Magliveras, a noted cryptology expert from the Center for Cryptology and Information Security at Florida Atlantic University gave a very interesting presentation on his use of APL in cryptology. For the past several years, under APL2000’s Education Program, APL2000 has provided Florida Atlantic University with APL+Win licenses at no-cost to Professor Magliveras’ students. APL2000 was pleased to welcome two of his students, Olga Shukina and Jessie Adamski, who gave presentations about how they used APL+Win to complete their Master’s theses.

Conference Session Descriptions

Catching Up on APL+Win (John Walker)

This presentation highlighted the new enhancements in APL+Win version 12, 13 and 14 since the APL2000 User Conference 2012.

Performance improvement for repetitive catenation; Improved support for Windows visual styles in APL+Win; APL+Win ActiveX engine Unicode execution methods; Improved APL Session Logging; `⎕CSE` – Interface to the APLNext C# Script Engine; `:FOREACH` control structure.

Multi-threading in APL+Win (Jairo Lopez, Joe Blaze, Pik Ng)

An overview of multi-threading topics (including operation grouping and independence, data marshalling, asynchronous execution and performance monitoring) were presented.

- APLNext Application Server for multi-machine processing
- APLNext Supervisor for multi-cpu processing
- APLNext C# Script Engine for multi-core processing

Windows Event Log and APL+Win (Brian Chizever)

What is the Windows Event Log? Why would you want to use it? Techniques and sample APL+Win code to use the Windows Event Log were provided.

Using APL to Manage Google Earth (John Magill)

Google Earth is a readily available tool with many useful features and potential. However, the syntax is rather cumbersome and not particularly dynamic. APL+Win provides an easy way to produce Google Earth maps and use them dynamically for strategic decision making. John Magill demonstrated the PATMIR III program he developed in APL+Win with funding from the World Bank and the Government of Mexico.

APL+Win `⎕CSE` System Function Interface to the APLNext C# Script Engine - Part 1 (Jairo Lopez, Frank Yang, Joe Blaze)

The ⎕CSE system function empowers the APL+Win developer with direct access to 100% of the .Net Framework 4.5 without the need for Visual Studio. The CSE implementation rationale, features, object model and documentation were presented.

APL+Win ⎕CSE System Function Interface to the APLNext C# Script Engine - Part 2 (Jairo Lopez, Frank Yang, Joe Blaze)

Advanced CSE features (e.g. defining .Net classes, GUI tools in .Net, consuming .Net events) were presented including detailed CSE examples for symmetric encryption, variable precision arithmetic, Linq queries, XML serialization, Windows event log and Windows Active Directory.

APL+Win as a Web Server (Jairo Lopez, Joe Blaze, Pik Ng)

APL+Win is a terrific tool to implement complex algorithms. Deploying an algorithm to browser- or mobile-based users is easy when APL+Win is exposed as a web service. Depending on the expected deployment scope, APL+Win functions in workspaces on a server can be exposed as a web service using Windows Communication Foundation (WCF) or APLNext Application Server technologies. The APLNext Application Server is now available in the traditional APL+Win web-server-based version and the new APL+Win module integrated with Microsoft IIS.

Thor - An APL Expert System to Assess Corporate Health (Eric Baelen)

Originally written in the 1980's for Touche Ross Audits to help assess non-financial risk, Eric was recently asked to update it. Eric answered questions like “What's it like to take an APL system written for the Intel 8086 processor and move it to APL+Win, the Internet and a javascript GUI?”. While making this presentation, Eric took attendees down memory lane as he shared with us his 40 year relationship with APL.

Workspace Recovery (Brain Chizever)

Once you release your application to a user, what do you do when they say "it won't even start"! Learn how to use the Crash Recovery Mechanism to handle these problems.

Using .Net with ⎕CSE Made Easy - Part 1 (Eric Lescasse)

How about if you could use the new ⎕CSE feature (almost) without having to learn .Net, Visual Studio and C#?

The presentation showed you how to create Objects in APL+Win which support multi-level inheritance, visual inheritance and multi-cast events, etc. It showed how you can easily document these objects and use them with ⎕wi. It showed how you can programmatically convert .Net Framework C# objects with all their properties, methods, events and documentation into such APL+Win Objects and start using them with the good old ⎕wi that we all know how to use! This new

APL+Win Object technology is called APL+Win zObjects.

Using .Net with ⎕CSE Made Easy - Part 2 (Eric Lescasse)

After the theory, the practice: This presentation showed practical applications using APL+Win zObjects. Various APL+Win examples were shown as well as applications that would not be possible to write with just APL+Win. Among other things, attendees saw a number of very impressive .Net controls embedded in simple APL+Win forms and how easy it is to use them. The benefits and limitations of this new APL+Win zObjects approach were discussed. Eric Lescasse provided a copy of this workspace to conference attendees.

APL+Win Interfaces: R statistical package (Ajay Askoolum, Joe Blaze)

Using work originally developed by Ajay Askoolum, Joe Blaze has extended the interface between APL+Win and the R statistical and graphics package to use the R.Net SDK and the new APL+Win C# Script Engine. Adding R functionality to APL+Win, such as R-based calculations and charts, were illustrated and a sample workspace was provided.

APL+Win Development Roadmap (APL2000 Team)

This session included a discussion of APL2000 priorities and development possibilities. An overview of current trends in the IT world was presented. The session provided an open forum for an audience Q&A session with APL2000 developers.

Accessing a Remote APL+Win COM Server from Excel (Joe Blaze, Pik Ng, Tesa Carlson)

Using ‘service moniker’ support in Excel 2003+, an Excel workbook can transmit requests to and receive responses from a remote APL+Win COM server via a simple WCF web service which exposes a ‘metadata exchange’ endpoint.

APL+Win Implementation and Comparison of Error Correcting Algorithm Performance (Olga Shukina)

This APL+Win-based project performed data transmission across noisy channels with recovery of the message first by using the Golay code, and then by using the first-order Reed-Muller code. The main objective of this thesis is to determine which code among the above two is more efficient for text message transmission by applying the two codes to exactly the same data with the same channel error bit probabilities. Comparison of the error-correcting capability and the practical speed of the Golay code and the first-order Reed-Muller code was documented.

Tags: APL and .NET Access to Your Personal Metadata Cloud (Jeremy Main)

Use the APL+Win ⎕CSE system function and other utilities to access ALL the metadata in ALL your files including documents, pictures, music and video. Using Microsoft Powershell via the APL C# Script Engine and other .NET assemblies were discussed as they pertain to metadata.

APL2000 – A Full-Service Software Development Company (Sonia Beekman)

Although you are most familiar with APL+Win, APL2000's flagship product, APL2000 is a full-service software company providing comprehensive consulting and training. An overview of APL2000 Products and Services was presented.

Driving MS Office (Eric Baelen)

APL+Win does a great job driving Microsoft Office (Word, Powerpoint and Excel). This presentation was an overview of several MS Office toolkit workspaces distributed to APL2000 customers.

APL+Win Interfaces (Joe Blaze, Frank Yang, Melissa Farmer)

APL+Win ⎕NFE System Function: Accessing Encoded Text Files A character encoding is a '1 to 1' mapping of abstract glyphs (characters) to values that represent those glyphs. The values resulting from the encoding of glyphs can be persisted and transmitted without ambiguity. The new ⎕NFE system function supports reading and writing of native files encoded as ASCII, UTF-8, UTF-16 and UTF-32. APL+Win server used by RDBMS Stored Procedures Relational databases can support pre-compiled methods called stored procedures. The technology for calling APL+Win functions from such a stored procedure using the Microsoft SQL server was presented and a sample project and workspace was provided to attendees. This technology can be used to embed APL+Win functions in database structures such as an XMLA server, column-oriented configurations or distributed big data deployments (e.g. Hadoop). APL+Win ⎕EDITEX System Function: Editor for Heterogeneous Data A prototype of a new APL+Win editor for heterogeneous and nested data was illustrated which uses the latest WPF GUI technology and directly interfaces with APL+Win to perform all array operations. Attendees received a working copy of the new editor.

Computing Automorphism Groups of Projective Planes (Jessie Adamski)

APL+Win was utilized to generate the full automorphism group of finite Desarguesian projective planes. This was done using homologies and the Frobenius automorphism, which was found by using the planar ternary ring derived from a coordinatization of the plane.

Sunday Seminar (Jairo Lopez, Frank Yang, Tesa Carlson, Joe Blaze)

The Sunday Seminar portion of the conference has traditionally explored a few topics in greater detail. Frank and Jairo discussed the power and simplicity of the new APL+Win C# Script Engine providing several sample workspaces. Tesa and Joe presented a prototype application system which uses HTML5 and javascript for the GUI, Microsoft ASP.Net for the middleware and APL+Win as a web service to support the server-side algorithms and data persistence.

Group Social Events at the Conference

In addition to all the interesting sessions, the APL2000 User Conference provided an opportunity to enjoy the camaraderie of other APLers. Attendees were treated to a special evening at the Ft. Lauderdale Antique Car Museum. The museum owns 39 Packard motor cars from the 1900's to the 1940's. A delicious dinner was served in the middle of the 18,000 square foot building surrounded by the beautiful cars and the thousands of pieces of automotive memorabilia.

This unique venue was the perfect place for a scavenger hunt. Everyone had fun searching for the answers.

Doug Masto, APL2000's Business Manager and car buff gave an interesting PowerPoint presentation with historical photos showing the early attempts to traverse the United States by Packard automobiles.

The Scavenger Hunt Winning Team

This quote from a conference attendee summed it up best:

> "I thoroughly enjoyed the conference. Everything about it was excellent. Kevin was a wonderful instructor, the sessions I attended were very informative, the materials and flash drive are great resources, the venue and location provided an exciting but relaxing atmosphere, and everyone in the APL community was very pleasant and a joy to be around."

General

SwedAPL April 2014

Gilgamesh Athoraya (gil@optima-systems.co.uk)

The Swedish APLers are perhaps not very numerous, but after probing and prodding a little bit, we managed to find each other. A first meeting was planned and executed on the first week of April and was attended by representatives from seven companies. This is our story.

The meeting

The meeting was held in CGM's office in Stockholm and by 10am the meeting room was full of anticipation and every seat was occupied. All eyes turned on me as I was stalling while waiting for the last couple of attendees to find their way in. A few minutes later we got started and after welcomes and greetings, Joakim Hårsman (CGM) launched his presentation.

Not one iota

Joakim started off by talking about how a seemingly simple and innocent fix to a bug in the APL interpreter can have unexpected, widespread knock-on effects in applications. The case in point is a fix to the result of `⍳⍬` in Dyalog APL. Where previously the following statement was true `1≡⍳⍬` it was corrected and is now `(⊂⍬)≡⍳⍬`.

This can and has caused issues in many applications that rely on the incorrect behaviour and Joakim told us of a tool they have developed to help in identifying candidates in the code that could suffer from this.

RIDE vs Dyalog+

Joakim continued to show us an alternative to the standard Dyalog IDE. He has developed an Emacs mode that he calls Dyalog+. By using sockets to communicate between Emacs and Dyalog, he demonstrated how to use either IDE to edit and fix functions. This is particularly interesting for those who are familiar with Emacs and/or often program in many different languages as Emacs can be used as a single IDE for all/most development work.

Dyalog+ doesn't currently offer the same features as the official RIDE (Remote IDE) from Dyalog, but Morten Kromberg (Dyalog) said the protocol for RIDE may be available once it is released and would enable users to create their own IDE to hook into sessions remotely.

Aplensia

After a brief break, Lars Wentzel took over and presented Aplensia. The consulting firm is formed of seven APLers, most of whom have been working with APL since the early 90s. They are now managing four major systems, one for Swedbank and three for Volvo, all of which have been migrated from APL2 mainframe to Dyalog APL on Windows servers.

Aplensia was the launch customer of the Dyalog File Server (DFS) which they have used successfully to replace a DB2 file server. They were also the ones to request Integrated Windows Authentication (IWA) via Conga (Dyalog's communication tool), which is now available to all (introduced in Conga v2.3).

Peter Simonsson talked about migrating from APL2 to Dyalog APL. He mentioned dialectal variations that required some attention, such as different interpretation of indexing:

```
A B[X] ? (A B)[X] or A (B[X])
```

A tool was developed to semi-automate the translation of code (about 890k LOC) to the Dyalog APL dialect.

They used WPF to emulate the original screens (about 400 screens) to make the transition as unobtrusive as possible for the users.

Ways of working

After lunch break Gianfranco Alongi (Ericsson) gave a talk about how he confronted managers' traditional views on efficiency of developers. His story of how they grudgingly gave in to pair-programming only to face the concept of mob-programming was both inspiring and hilarious at the same time. You can read more about this in his article in this same issue of *Vector*.

News from Dyalog

Next up was Morten Kromberg with a summary of new features and tools. He talked about the upcoming RIDE which will make it much easier to debug remote sessions as well as dynamically start/stop and monitor remote sessions. They have added support for .NET data-binding which will make it easier to share data between APL and .NET components. He mentioned the Syncfusion GUI package which is going to be bundled with Dyalog APL v14.0, offering WPF and JS components. There are improvements and speed-ups to Dyalog Component Files (DCF) as well as a release of DFS v2.0.

He concluded his presentation with a demonstration of Futures and Isolates, the new features that will make it easier to harness the power of your hardware by parallelising the execution of operations in separate, external processes.

Cosmos and big data

Finally, Paul Grosvenor (Optima Systems) talked about Cosmos: a graphical, analytical tool that doesn't give you the answers, but helps you find the questions. The system is using MiServer in the back end and a flash UI on the client side. It is a system designed to be easy to use and without requiring deep technical knowledge, but powerful enough to let the user explore data in an intuitive way. I demonstrated the system briefly and talked about the difficulties of taming data quantities that grow bigger faster than you can say analytics.

Group photo

Back row: Stefan Lindén, Peter Simonsson, Ylva Ljungdell, Lars Wentzel, Mikael Blomgren, Joakim Hårsman, Ronnie Sommer
Middle row: Tina Leijding, Alvin Mattson, Gitte Christensen, Gianfranco Alongi, Paul Grosvenor
Front row: Sargon Athoraya, Gilgamesh Athoraya, Morten Kromberg, Gunnar Jörtsö

Others present

In addition to the presentations we also had representatives from Sandvik. Tina Leijding presented the company briefly. Their system was built back in 1984, runs on APL2 and is hosted by IBM. They have a small team of APLers (five plus one consultant) and are looking to expand over the year.

A big group of APLers from CGM took the opportunity to pop in and out during the day, most of whom were wearing Rosetta stone t-shirts with javascript on one side and APL on the other.

My brother, Sargon Athoraya, attended the meeting to learn more about what I do for a living and claims to have been able to follow most of the presentations without nodding off.

Next meeting

The meeting was greatly appreciated and after a conversation about content and frequency of meetings it was agreed that we will aim at half-yearly meetings with the next one planned for beginning of October (preliminary date is 9 Oct 2014). It will be hosted by Aplensia in Gothenburg and open to the general public. More information will follow closer to the date.

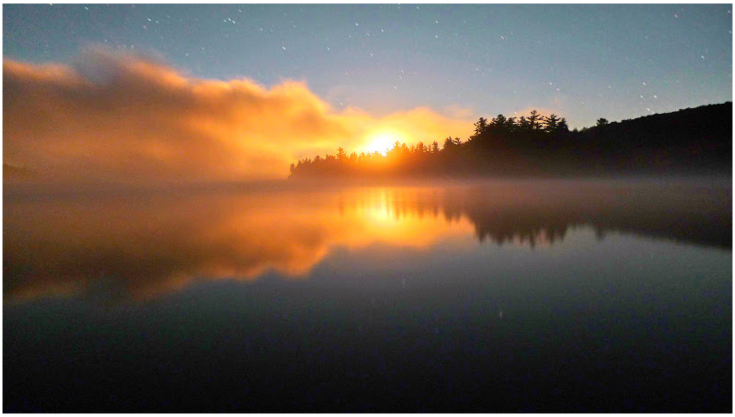
LinkedIn

SwedAPL is a group on LinkedIn. Feel free to join the group to stay in touch with Swedish APLers.

Minnowbrook conference review: September 14–18, 2013

Steve Mansour

I was honored to be invited to the 2013 APL Implementers Conference at Minnowbrook this year. I had been busy teaching statistics at the University of Scranton in addition to working on my doctorate in industrial engineering at Lehigh University and was unable to attend APL conferences for several years. I was privileged to have this opportunity to share my presentation, “Taming Statistics with Defined Operators” with such an esteemed group.


Minnowbrook Moonset

It was good to see everyone again. It was surprising how many APLers in the group began in 1969, including Roy Sykes, Ron Murray and Bob Smith. Jim Brown was not included in this group because he started using APL in 1967! There was a little sad news: we were told that Phil Benkard had died on July 24, 2010. I shared an office with Phil briefly during my tenure at IBM and I learned a lot from him.

The meals at the Minnowbrook Lodge were phenomenal and the bar was well-stocked for the ‘evening seminars’, which lasted well into the night. Although it rained on one day, the weather was perfect on the last two days of the conference. We were graced by the presence of Tess and Grace, two black-and-white Border Collies belonging to Roy Sykes. We had a free afternoon on Tuesday, September 17th. Many of the attendees took a two-hour boat ride on the lake while several of us hiked up to Castle Rock.

We owe Garth Foster a debt of gratitude for his connection with Syracuse University and the Minnowbrook Conference Center. He said the FORTRAN people could get rid of APL by blowing up Minnowbrook since there was almost as much programming expertise there as when Ken Iverson was in a room by himself. Why do we have meetings in such a remote place? Garth said the remoteness allows us to brainstorm and reflect without distraction. Special thanks to Roy Sykes for organizing the conference.

Jim Brown spoke about the early years of APL2 as trying to improve perfection. He used the analogy of extending a circle to a sphere. He spoke of the debates with Iverson about nested arrays and how two versions of APL emerged with boxed and enclosed arrays. It appears that APL took a 'fork' in the road although many feared a 'train' wreck. Dijkstra referred to APL as "a mistake carried through to perfection."

David Liebtag demonstrated new enhancements to his nested array editor, which is now a part of Dyalog APL. Bob Smith spoke about "Progress on NARS2000" where he discussed factoring and number-theoretic primitive functions, the where and array lookup primitive functions and the variant primitive operator. Later he gave a separate talk entitled, "2-by-2 Syntax Analyzer."

Bob Bernecky discussed performance problems with various sizes of arrays. Treating scalars, small arrays and large arrays differently versus a one-size-fits-all approach leads to significant performance improvements. The current optimization status shows that scalars can be improved by as much as 1300x, small arrays by 20x and large arrays by 10x.

Jacob Brickman gave us a lesson in constructing the real numbers starting from set theory, and he discussed extending the number system beyond the complex field, to quaternions and octonions. There was a heated discussion of using the notation `0J1` to represent instead of `0I1` and extending this notation to quaternions and octonions. I guess some APLers don't like this form of 'J' notation. IBM saw valid reasons for using either 'I' or 'J' to denote irrational numbers; the reason that IBM chose 'J' was that there was less ambiguity than 'I' when it was written on a board in a classroom.

Bob Smith and I moderated an open discussion of expanding the domain of *iota*. The index generator function could be expanded to 'sequence' with an optional step. Although John Scholes was not there, some mention was made of his `a..b` notation to accomplish this. Dyadic *iota* could be extended to allow a matrix left argument; the result would be the index pairs which correspond to each element of the right argument. Another possibility would be to indicate the row which corresponds to the vector on the right. The variant operator would allow a comparison method, e.g., trailing blanks or handling special cases.

Morten Kromberg from Dyalog discussed the Version 14 language features. New operators include *rank* and *key* as well as *trains*, *forks* and *atop*. In Dyalog APL, function trains of length two are equivalent to the *atop* operator: `⍺(fg)⍵` is equivalent to `f⍺g⍵`. One significant benefit of this is that an expression like `?1E6⍴6` can be written `1e6(?⍴)6`, making it easy for the implementer to special-case the operation and avoid creating the million-element intermediate result. An extended version of dyadic *iota* is in the works as well as a new function, tally (monadic ≢), which counts the number of items or rows in a matrix and produces a scalar result. While he was explaining the tally function, he stopped and looked at the screen for a moment, and commented that he didn't know who would actually write a tally like that, with three lines instead of four. Someone (Bob Smith, I think) immediately replied that "that's the way that cartoon characters write a tally." Well done.

The new constructs *isolates* and *futures*, which let programs run on different processors, were also discussed. Morten also demonstrated the new parallel operator by executing code containing `⎕dl 3` three times in four seconds by assign each to a different isolate.

Dog & Pi

Ray Polivka and Mike Van Der Meulen held an open discussion about APL on alternative platforms, Raspberry Pi, Robots, natural language interfaces and mobile platforms. The Raspberry Pi is an inexpensive processor that can be programmed in APL as well as Python. Morten Kromberg demonstrated Raspberry Pi by giving commands to a robot using APL. This got the attention of the dogs, Tess and Grace, who were both amused and a little scared. Later we saw a demonstration on the web of helicopters flying in formation with no central authority; their motions were affected by the position and motion of the neighboring helicopters. (Kind of like the free market!)

Ray Polivka and Bob Bernecky discussed education and getting more people into array-oriented languages. This was followed by the usual lament that the average age of APLers was going up by one year every year. Ray Polivka has been teaching high school and college students for three years using a classroom in a rent-free model house. He found that the language bar was a tremendous help and that because students are generally open-minded, they had no problem with the symbols or scanning rules. Students are less concerned with how APL is used than who uses it.

This was followed by a discussion of how APL was used in industry. This included finding deadbeats in the utility industry, product reliability, mortgage finance, insurance, actuarial, medical imaging data management, petrochemical analysis, travel reservation systems, process control and ticket sales. Even Bill Gates' investment management was done in APL.

Shannon Bailey of Native Cloud Systems brought back memories of the early 80's with visuals of the old IBM 3279 terminal and the original IBM PC when she presented "APL— A Love Story." She was introduced to APL at Marist College where everyone had to use APL for Computer Science 101. In that class she was able to rewrite a multi-player game from scratch in STSC APL*PLUS PC in eight weeks. Her current work includes a cloud-based system that supports an APL-like programming language and transaction-based Native Cloud Objects for distributed arrays.

Mike Van Der Meulen wowed us with a demo of his experimental APL application using voice recognition software. He asked the computer about the weather, had the computer translate "When is the next train" into Chinese, had the computer play "Hey Jude" and asked it to show a picture of Ken Iverson. There was one glitch. When he asked it the question "Who invented FORTRAN?", the computer showed a picture of the Roman god Bacchus!

Ron Murray revealed that although dead men tell no tales, dead processes talk! In distributed systems the failure of a computer you didn't know existed may cause your system to fail. To get to the heart of the matter one must be able to change the system without stopping it.

Adrian Smith from across the pond showed how to embed a matrix into a database by inserting vectors as items into columns. His son Richard showed how just-in-time compiling in the parse table step improved performance in "Adventures in JIT computing APL."

Paul Grosvenor demonstrated COSMOS, a data visualization tool largely written in APL. COSMOS is a top-down, drill-down system used to analyse medical data. The system talks to statistical language R and has often been referred to as a 'thesis generator'.

Bob Armstrong got CoSy with FORTH and showed us how to go FORTH and multiply. He also challenged the global warming community with a hot topic: "How to Calculate the Temperature of the Earth for a Libertarian Society."

On the last night of the conference, the highlight was jazz guitarist Stanley Jordan, who presented us with “Music and APL.” He used APL and the circle of fifths to measure distance between various musical scales. He demonstrated sonification by showing us how to generate music from stock charts and listen to patterns. Finally, he did a MIDI edit using APL to generate music and rhythm on several tracks. Afterwards he entertained us on the guitar with several original compositions. His final comment was: “You don’t speed-read poetry. APL is like poetry; everything has meaning. You may have to read it several times to understand it.”

Minnowbrook 2013

Impending kOS

by Stephen Taylor (sjt@5jt.com)

It began badly. We were walking along the South Downs Way in early summer, the sun glittering on the English Channel on our right, the Weald of Sussex stretching away to our left. "How big," asked Arthur, "should a text editor be?"

I've known Whitney most of my life. I know what he does. I know his stupid questions. And still I can't resist trying to give helpful answers. "I don't know. One could find out, surely? What do Emacs and Vim weigh – tens of megabytes?"

"I've got a text editor in four lines of K. Just need to add Copy and Paste."

Ah, we're back to that. Of course. K is the language part of kdb+, Arthur's frighteningly fast column-store database, used by trading rooms to handle huge real-time data flows from financial exchanges. It started off at Morgan Stanley in the 1980s as an APL stripped for speed and became A+[1], and for two decades the bank's development environment for trading applications. For the last twenty years it has evolved as kdb+, trading as Kx Systems, Inc.[2] It's a two-orders-of-magnitude sort of thing: two orders of magnitude faster than industry-standard database, two orders of magnitude smaller code volume. Four lines of K equate to about four hundred lines of C.

The kdb+ interpreter is tiny: about 100Kb. (And yes, kdb+ programs are interpreted, not compiled.) As the code base improved, kdb+ releases became faster – and smaller. Kdb+ has sharp elbows. Impatient with the speed of Windows, kdb+ wins a ×3 performance improvement by managing memory itself.

Whitney is no respecter of rules. One of the scariest things I ever did as a young man was following him through central Toronto on a bicycle.

An apocryphal story. At the first of the three universities he claims to have been thrown out of, Whitney's class was given an assignment: write a program that will print the most successive prime numbers possible with limited CPU time and limited green-striped paper. (Yes, that long ago.) His solution won by a handsome margin and was disqualified on two counts. In the first place he had ignored everything the class had been taught about modularisation and code re-use. He just wrote code optimised to solve one problem spectacularly fast. He had also noticed the problem did not specify printing spaces between the primes. The printouts were a sea of ink. And his code looked like woodgrain.

> As a rule, it was the fittest who perished; the misfits,
> Forced by failure to emigrate into unsettled niches,
> Who altered their structure and prospered.
> — WH Auden

Kdb+ is a testament to the rewards available from finding the right abstractions. K programs routinely outperform hand-coded C. This is of course, impossible, as *The Hitchhiker's Guide to the Galaxy* likes to say. K programs are interpreted into C. For every K program there is a C program with exactly the same performance. So how do K programs beat hand-coded C? As Whitney explained at the Royal Society in 2004, "It is a lot easier to find your errors in four lines of code than in four hundred."

What would computing be like if it were all done this way? The decades-long sleigh-ride of Moore's Law[3] has ended. What if we could get another two orders of magnitude of performance out of the hardware?

This question has been asked before, notably by Alan Kay at the Viewpoints Research Institute.[4]

Whitney means to find out. The first phase of the project is to escape the bloated embrace of the operating systems and run kdb+ on the bare metal.

> "If you keep on chipping at that rust, eventually you'll reach flat, bright metal." – Herman Wouk, *The Caine Mutiny*

Whitney started replacing calls to Linux, working, alone as always, in his garage office. Characteristically, it's a simple workplace: a pool table, a desk, a chair and a PC with a single monitor. When I saw it in 2007 it was running Windows XP and had five windows open: two MS-DOS and three Notepad. Brutally simple IDE. Doubtless things have improved since.

Oleg and Pierre had heard of Whitney and kdb+. They study computer science in St Petersburg. (Russia, not Florida.) With a great deal of trepidation and some support from a teacher they wrote asking Whitney what he was doing. He replied with some C code he was working on.

Everyone knows how C programs look: tall and skinny. Whitney's don't. I first encountered them in the 1980s. I was working for I.P. Sharp Associates in Sydney. My boss wanted to port the SHARP APL interpreter onto the fast new Hewlett-Packard HP1000 minicomputer. I recommended Whitney, then the youngest member of the IPSA systems-programming team, for the job. (Probably the best thing I've done in my professional life.)

There was, of course, a catch. The interpreter was a 500Kb program developed over 15 years and supported by an 11-man team. The original language had been considerably extended – most recently with 'general' or 'nested' arrays – and all the extensions had to be ported too. Although the target machine was attractively fast, most of the speed disappeared for programs larger than 80Kb. The interpreter had not just to be ported, but also made six times smaller. Game over?

Whitney's strategy was to implement a core of the language – including the bits everyone thought most difficult, the operators and nested arrays – and use that to implement the rest of the language. The core was to be written in self-expanding C. As far as I know, the kdb+ interpreter is built the same way.

Unlike the tall skinny C programs in the textbooks, the code for this interpreter spills sideways across the page. It certainly doesn't look like C.

In Sydney we assigned Whitney two coding assistants. Not that he needed or wanted help, but when he eventually left we'd need *some* idea how it all worked. His assistants had a very hard time. They would struggle through the week, get their assignments half finished, then on Monday discover Whitney had dropped in over the weekend, rewritten most of the interpreter, and included their assignments. (The interpreter got finished. A decade later I saw one of the HP machines still running on Westpac's trading floor. Not long after that, Whitney started work at Morgan Stanley on what became A+.)

Whitney sent Oleg and Pierre some of the C code he was working on, and notes on a problem he didn't know how to solve. They emailed back a solution, coded in his style. A partnership was born: a garage in California, a school in Russia.

Whitney demonstrated his "research K interpreter" at the Iverson College meeting[5] in Cambridge in 2011. We had visitors from Microsoft Research. The performance was impressive as always. The tiny language, mostly familiar-looking to the APL, J and q programmers participating, must have impressed the visitors. Perhaps conscious that with the occasional wrong result from an expression, the interpreter could be mistaken for a post-doctoral project, Whitney commented brightly, "Well, we sold ten million dollars of K3 and a hundred million of K4, so I guess we'll sell a billion dollars worth of this."

Someone asked about the code base. "Currently it's 247 lines of C." Some expressions of incredulity. Whitney displayed the source, divided between five text files so each would fit entirely on his monitor. "Hate scrolling," he mumbled.

At Iverson College in 2013 he demonstrated the new graphics layer, z – 9Kb of code to replace the X Windows system. For the first time we saw the kOS desktop, solid black with a Tolkienesque legend top left: *one system/all devices*. Arrayed on the right edge, the icons of five kOS apps. He launched the text editor app and then wrote a new one, working out the key callbacks in front of us and explaining them as he worked. As he defined each callback the new app acquired it: no compile, load, install cycle. In eight lines of K he had replicated the core function of Notepad. At this point, with the new z layer in place, kOS weighed 62Kb.

Last autumn the kOS team recruited a fourth member, Geo, and in November announced it had removed the last connection to Linux. kOS was running on bare metal. Whitney announced the project would now go dark and return, perhaps in the summer of 2014, with a platform on which apps can be built.

kOS is coming.[6] Nothing will be the same afterwards.

References

1. http://www.aplusdev.org
2. http://www.kx.com
3. http://www.en.wikipedia.org/wiki/Moore's_law
4. http://www.vpri.org/pdf/tr2007008_steps.pdf
5. http://www.sites.google.com/site/iversoncollege
6. http://www.kparc.com

Searching for the state in which Wonderful Things are inevitable

by Gianfranco Alongi (gianfranco.alongi@gmail.com)

"Fear leads to anger, anger leads to hate, hate leads to suffering."

- Master Yoda

I love this quote, fear is the mind killer; our minds are like parachutes, they don't work unless they are open. If we feel threatened, we get defensive, we stop listening, we stop thinking clearly, and will make emotionally tainted decisions which sub optimize the value for the company/customer. In short - we will take decisions which protect our ego, instead of solving the problem we get paid to work on.

As programmers working with APL 'The Tool Of Thought' - it should be top priority to keep our thoughts unclouded by fear, so we can focus on writing suspiciously powerful and yet alarmingly beautiful APL code.

My team at Ericsson has been through a lot (God bless them) since I joined in the late summer of 2011. By now, I guess there is nothing they can fear any more. More or less forcefully subjected to all kinds of things like TDD, Pair Programming, Crush Sessions, Hero Avoidance, endless Code Dojos and now the latest addition which is Mob Programming - the learning is endless.

I am on the never ending quest of continuously improving everyone around me, so that they will get better than me. Why? Because the best way to get better is by working with those who are better. You can observe, ask, and mimic. Instead of trying to figure everything out yourself, you can leverage the fact that someone can give you distilled knowledge mixed with wisdom, this is accelerated learning.

If we all continuously strive to improve those who we work with, someone is always trying to improve you in return as you are trying to improve them. It is but a matter of time before this little select group is the best of the best.

What I describe is undeniably sensible, although it does require a lot of courage and trust. What I will describe now, is the different practices I have used with my team and other teams in order to nudge the team dynamics in the right direction.

Pair Programming (PP) was the first thing I introduced; two developers working on the same problem, on the same machine. There is much written about PP already, but let me mention the interesting discussions and what I observed. A common misconception that needs to be buried when it comes to PP, is that supposedly productivity would be halved.

This is only true if the productivity bottle neck is the typing speed.

If two developers have a 1:1 relation between their productivity and typing speed, then yes: removing a keyboard would put your productivity at 50%. However, this is definitely not the case.

Studies by Microsoft and IBM have shown multiple times that the so called Read/Write Ratio (R/W Ratio) is our main concern when it comes to working in large software systems. In short, the R/W Ratio in large software systems is ~10-15. This means that on average, a developer needs to spend 10-15 times more time reading than writing code.

Clearly, the main problem is the time needed in understanding.

Two minds will reduce the comprehension time dramatically, there is so much synergy when working in a pair. Just to mention a few things that happen

i. We stop following the wrong chain of thought very early.
ii. We do not get stuck. There is always an alternative idea to try.
iii. We teach each other tips/tricks related to how we work.
iv. We have fun.
v. We share system knowledge and knowledge about the ongoing work.
vi. We expose ourselves.

The sixth point (vi) We expose ourselves) is the most valuable and also the toughest one. Exposing ourselves to our colleagues can be scary. A lot of negative thoughts based on fear can pop up. The ego can get hurt, suddenly all our weaknesses which we have learnt to live with become a very apparent and very real issue. Typing speed, system specific knowledge, tool proficiency, coding skills, thinking speed, work habits, every aspect of work will be exposed to your PP partner.

If we do not expose ourselves to others, how can they help us improve?

In order to do this, we must be capable of facing our own fears, we must vanquish them, for they prevent us from growing.

> Master Yoda - "That place… is strong with the dark side of the Force. A domain of evil it is. In you must go."

> Luke - “What’s in there?”
>
> Master Yoda - “Only what you take with you.”

Much like Master Yoda told Luke to face his innermost demons, we must do so. Whenever we feel unease, a fear of exposing something related to our way of working or technical expertise - we must take this demon, and put it into our ‘Book Of Demons’. By systematically putting all your technical weaknesses into this list, you not only admit that you have a problem, but you also build a very practical checklist of things to improve.

So, when do we have time to improve? There is no time like the present!

One thing I have found profoundly successful is to practice weaknesses in a Code Dojo setting. Our team has 2 hours weekly, dedicated to deliberate practice. That is; my whole team gets paid two hours per week to practice, so they can perform better. Combine the Demon books from every team member into one huge list, next, take the first item - and whoever is the most proficient in that item prepares a lecture with some exercises for the team to improve on the next Code Dojo.

This works well when the team is motivated or guided by a team member taking point, but what if the team does not have this motivation?

As part of coaching another team in another part of Ericsson, I wanted to nudge the team into doing Pair Programming. At first the team was reluctant, but once I had tapped into the primitive parts of our psyches - they were doing PP.

We are animals, and we have all been living in tribes and clans for quite some time; physical tokens, recognition and rituals mean a lot to us. If we wish to change a behaviour then we can leverage these facts and utilize them. I gave the team I was coaching a challenge:

> ‘For every time someone does PP in the team, the team gets one Golden Star from me. If you manage to get 10 golden stars within a 2 week period, you get a certificate from me.’

The next time we met, they had done some PP and I held my word, a golden star was put on their whiteboard. The first collective recognition of the team, one of many to come. This does not only bond the team together, making the members appreciate that they are working towards the same goal, it also turns the greatest fear of exposure into a game.

Over time, the team managed to collect 10 stars for PP, and so I ceremoniously produced a diploma, which was given to the team during very formal circumstances with music and a short speech.

Ceremonies matter, strong emotions and psychological mechanisms are at play here, and it was very obvious that the team were happy to have exposed themselves to each other so much.

Another team I coached could not even get started with PP, there was an obvious fear of exposure to each other. The team consisted mainly of older developers who had been in their comfort zone for quite a while now. No one immediately expressed concerns for being exposed, but all the reasons for not doing PP were just blatant excuses. Master Yoda comes to mind again;

> “You must unlearn what you have learned”
>
> - Master Yoda

This quote is also a golden nugget I carry with me. Traditionally, people were taught to be subject experts, with a lot of gravitas and a fat nice paycheck to go with it. We learn to crave to be the best, we like being the best, our egos require it. We have learned to be heroes - and will protect this position and feeling that goes with it. Unfortunately, this is actually a counter productive thing.

What this team needed, was to unlearn being heroes. Because the hero does not wish to be perceived as weak. The team got a challenge:

> ‘For every technical area of improvement you practice together as a team, you get a Brain (brain printed on a magnet) from me. If you manage to get seven brains within the time I am coaching you, I will give you a certificate for your outstanding performance.’

Suddenly, the personal weaknesses become a positive driving force for the team. The team practices the ‘area of improvement’ (a more eloquent way of saying weakness), and gets a Brain token. Soon enough, the team members were ‘sharing ideas’ (admitting problems) they had for improvements, so they could get those Brains. Exposing their weaknesses was now a good thing, a game. The team fulfilled the challenge, and as promised, a ‘Tiger Challenge’ certificate was printed and delivered with music and a speech.

Today Pair Programming is a given in the team, no one sees this as anything but a positive force. In November 2013, I had Woody Zuill[1] stay at my place for a week. Woody told me about Mob Programming and how his team practiced it, I immediately wanted to try this out. Mob programming is PP taken to the next level.

The whole team, working on the same thing, on the same computer.

Surely this is crazy? It is crazy alright, crazy good.

In crisis mode, when you need to get something done quickly, you always gather the best people into the same physical location and give them a lot of space and freedom (with added pressure of course). Why don’t we do like this all the time instead?

My team has practiced this for a substantial period, and it works extremely well for Trouble Reports (error corrections). For normal development we still do a lot of PP, but we pull together into the Mob when the pressure and complexity starts mounting.

When working as a mob, no task will ever halt until it’s done, no team member burns out due to stress as everyone shares the load. We all know what we are doing, and we all know what has been done. No meetings are necessary as we are all there, we all take decisions together, this also means that we can quickly undo a decision. The most important thing is not to keep everyone busy by being stuck and working overtime. The most important thing is to get the most valuable solution out the door as quickly as possible. Every good thing from Pair Programming is magnified tenfold in the Mob.

But, what comes next? Evolution never stops, progress is inevitable. From what I've heard, Dyalog Ltd and Optima Systems Ltd practice something they call ’Swing programming‘ where one developer from each company is traded for a while! I just hope they will write an article on this and share their findings, it sounds really interesting!

Pair-/Mob-/and Swing-programming aside, the search for better should never stop, so let me leave you with this last quote:

> “All the right people and expertise,
> in the right place,
> at the right time.”

- Woody Zuill on Mob Programming.

References

1. Zuill, Woody http://zuill.us/WoodyZuill/

APL

One reason that APL is so cool

Brian Becker

The code shown in this article was not intended to be the most elegant or efficient means to solve the problem presented, but rather to demonstrate that APL's suitability for quick, ad hoc, data analysis and problem solving.

I had the good fortune to learn APL when I was but a freshman in high school. I found APL to be a great tool to solve problems. The phrase “APL as a Tool of Thought” has been around for quite some time and it still holds true. I’ve never viewed myself as a programmer, but as a problem solver. APL enables me to take a solution I conceive in my mind and translate it into a form executable by a computer with the least effort. Over the years, I’ve been lucky enough to work on some rather interesting problems, but in my spare time, I’ve also found APL to be fun for recreational computing. I’ll leave it to the gentle reader to assess just how much of a geek this makes me.

One such opportunity presented itself recently. Our local newspaper, the Rochester Democrat and Chronicle, along with other entities in the Rochester area, including the Rochester Institute of Technology (RIT), had been conducting a contest for several weeks called “Picture The Impossible”. It consisted of 7 weeks of challenges and puzzles all relating to aspects of the Rochester area and its history. Monday through Friday there featured puzzles on the web. There were weekly excursions or challenges that one could participate in around the local area and on Sunday there was a crossword puzzle and another challenge or puzzle. The puzzle of October 25, 2009 is the subject of this article.

This puzzle consisted of sets of scales and weights to be assigned to various points on the scales. For instance, given the scale:

```
┌───┴───┐
A   ┌───┴───┬───┐
    B       C   D
And weights of: 2 3 7 and 12
```

Each point will be assigned a weight and the force that point applies is its weight times the distance from the fulcrum. The example above gives the following:

```
A = B + C + D
B = C + (2 x D)
```

So, it’s pretty easy to work out that A=12, B=7, C=3, and D=4.

The challenge in the newspaper consisted of 6 such puzzles with up to 9 points. Because of some scheduling constraints, I had less than an hour to solve all the problems. While I’m a pretty good puzzle solver, I decided that the quickest way was to use APL. I figured that I could easily represent the algebra as a set of assertions and then run every combination of weights through those assertions until I found a set that worked. The first part was to build something that would generate all the combinations of weights. I remember that the number of combinations is the factorial of the number of elements, so nine elements would result in 362,880 possible combinations.

Now, had I paid more attention in school those many years ago, I’d probably have the “create all combinations” algorithm committed to memory. But, the way I thought of the problem is that combinations of set of nine weights is each of those weights concatenated with all the combinations of the other eight weights, then the combinations of a set of eight weights is each of those weights concatenated with all the combinations of the other seven weights, then the combinations of a set of seven weights… wait a minute… this is recursive! So, the terminal case is when you get down to a single item, and the only combination is the item itself. That’s easy enough to code.

```
 ∇ r←allcombinations v
[1]    →(1=⍴v)↓l1 ⋄ r←v ⋄ →0
[2]   l1:r←v cat¨allcombinations¨(⊂v)~¨v
 ∇
```

```
 ∇ r←a cat b
[1]    →(1=≡b)↓l1 ⋄ r←a,b ⋄ →0
[2]   l1:r←a cat¨b
 ∇
```

Why did I write “cat”? Well, I started out with using APL concatenation, the “,” function. But that resulted in a nested result that wasn’t quite what I was looking for, as shown below. DISPLAY is a wonderful utility that displays an array with its structure. The result below is using the APL concatenate primitive function instead of “cat”.

```
  DISPLAY allcombinations 1 3 5 7 ⍝ using ,
┌→────────────────────────────────────────────────────────────────────────────
│ ┌→────────────────────────────────────────────────────────┐ ┌→──────────────
│ │   ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ │ │   ┌→──────────
│ │ 1 │   ┌→──┐ ┌→──┐ │ │   ┌→──┐ ┌→──┐ │ │   ┌→──┐ ┌→──┐ │ │ │ 3 │   ┌→──┐ ┌→
│ │   │ 3 │5 7│ │7 5│ │ │ 5 │3 7│ │7 3│ │ │ 7 │3 5│ │5 3│ │ │ │   │ 1 │5 7│ │7
│ │   │   └~──┘ └~──┘ │ │   └~──┘ └~──┘ │ │   └~──┘ └~──┘ │ │ │   │   └~──┘ └~
│ │   └∊──────────────┘ └∊──────────────┘ └∊──────────────┘ │ │   └∊──────────
│ └∊────────────────────────────────────────────────────────┘ └∊──────────────
└∊────────────────────────────────────────────────────────────────────────────
```

Using cat, I got…

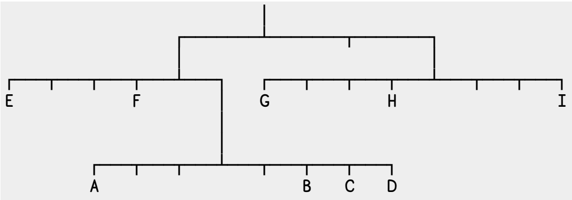
```
  DISPLAY  allcombinations 1 3 5 7 ⍝ using cat
┌→──────────────────────────────────────────────────────────────────────────
│ ┌→────────────────────────────────────────────────────────────────────┐
│ │ ┌→────────────────────┐ ┌→────────────────────┐ ┌→────────────────────┐ │
│ │ │ ┌→──────┐ ┌→──────┐ │ │ ┌→──────┐ ┌→──────┐ │ │ ┌→──────┐ ┌→──────┐ │ │
│ │ │ │1 3 5 7│ │1 3 7 5│ │ │ │1 5 3 7│ │1 5 7 3│ │ │ │1 7 3 5│ │1 7 5 3│ │ │
│ │ │ └~──────┘ └~──────┘ │ │ └~──────┘ └~──────┘ │ │ └~──────┘ └~──────┘ │ │
│ │ └∊────────────────────┘ └∊────────────────────┘ └∊────────────────────┘ │
│ └∊────────────────────────────────────────────────────────────────────┘
└∊──────────────────────────────────────────────────────────────────────────
```

Which is a lot closer to what I wanted…

However, I was still stuck with a nested array that had as many levels as the number of elements.

```
  ∇ r←n chunk a
[1]    r←∊a
[2]    r←((⍴r)⍴n↑1)⊂r
 ∇
 DISPLAY 4 chunk allcombinations 1 3 5 7
┌→──────────────────────────────────────────────────────────────────────────
│ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→─────
│ │1 3 5 7│ │1 3 7 5│ │1 5 3 7│ │1 5 7 3│ │1 7 3 5│ │1 7 5 3│ │3 1 5 7│ │3 1 7
│ └~──────┘ └~──────┘ └~──────┘ └~──────┘ └~──────┘ └~──────┘ └~──────┘ └~─────
└∊──────────────────────────────────────────────────────────────────────────
```

Perfect! So, I've got the "all combinations" part of the problem solved. Now, let's build the rest… The problem for this example is:



Weights are: 2 4 5 8 10 13 17 18 23

This gives the following relations:

```
3A = 2B+3C+4D
4E+F = A+B+C+D
3I = 4G+H
A+B+C+D+E+F = 2(G+H+I)
```

This turns into the following APL function. I won't go into all the aspects of APL's right to left execution, etc. If you know APL, it would be redundant, if you don't know APL, there are plenty of resources to learn it. But basically, it tests each assertion. If an assertion fails, the function exits returning a result of 0. If all assertions pass, the set of weights is displayed and a 1 is returned.

```
 ∇ r←p5 arg;a;b;c;d;e;f;g;h;i
[1]    (a b c d e f g h i)←arg
[2]    →(r←(3×a)=2 3 4+.×b c d)↓0
[3]    →(r←(f+4×e)=a+b+c+d)↓0
[4]    →(r←(3×i)=h+4×g)↓0
[5]    →(r←(2×g+h+i)=a+b+c+d+e+f)↓0
[6]    ⎕←arg
 ∇
```

So, now we have something to generate all combinations, and something to solve for a single combination. It would be easy to use the APL each operator (¨) to run the solution against all combinations, but I wanted to have it stop as soon as it found a solution and not evaluate all the combinations. So, I wrote a simple “solve” program…

```
 ∇ solve z;cnt
[1]    cnt←0
[2]   lp:→((⍴z)<cnt←cnt+1)⍴0
[3]    →(p5 cnt⊃z)↓lp
 ∇
```

This will check the assertions against all the combinations until either all combinations have been checked, or a solution is found. The solution, if found, is displayed and the program exits.

Putting it all together:

```
 solve 9 chunk allcombinations 2 4 5 8 10 13 17 18 22
22 17 4 5 10 8 13 2 18
```

So, A=22, B=17, C=4, D=5, E=10, F=8, G=13, H=2, I=18. I’ll leave it to the reader to verify the result.

I was able to solve all 6 problems in the paper in less than 30 minutes, about 28 of which were spent thinking about the problem and coding the solution. Well, in truth, I didn’t solve them, APL solved them, but I got the results I needed in probably less time than it would have taken me to do manually. Besides, it was fun!

The goal of this paper is to demonstrate that APL is a terrific tool for solving problems quickly. There are no doubt better ways to code this in APL, but elegance wasn’t my goal. I had a problem to solve and limited time to do it within. This has always been one of APL’s strengths and whether for recreational or commercial purposes, APL remains a tool of thought.

Notation as a tool of proof

Robert Pullman (rpullman@gmail.com)

APL is used to analyze the symmetries of magic squares. `⎕IO` of 1 used throughout.

1. A Primer On Magic Squares

The classic magic square of order N is an arrangement of `⍳N*2` into an N by N matrix A such that the sum of each of the N rows, N columns, and both diagonals equals the same value - `(N×1+N*2)÷2`. There is no solution for `N=2` so presume `N>2`.

2000 years ago the Chinese knew of magic squares of order 3. There are 8 magic squares out of 9! (362880) possible 3 by 3 squares.

In the 17th century Bernard de Bessy determined that there are 7040 magic squares of order 4.

There are many techniques for constructing magic square but no simple way of counting the number of magic squares of order N. In 1973 an MIT grad student, Richard Schoeppel, solved for N=5, 2202441792 solutions. Schoeppel wrote an Assembler program which took a week to determine the answer. In a private communication he wrote:

> “… The approach was straightforward, filling in the cells in a particular order and backtracking. There was a little bit of hardware advantage: The DEC10 had multi-level indexed indirect addressing, making it easy to add a few numbers in a single instruction. Another hack was the instruction for reversing the bits in a register. This made it fast to determine quickly the possible solutions to X+Y=K, using bit masks of the remaining available numbers.
>
> Two other people independently confirmed the count, using different strategies. One Japanese gentleman sent me his thesis, with a long table of pieces of squares that he then assembled. My counting program would probably run in a few minutes today if converted to C.”

For N>5 the number of solutions is unknown. There are estimates of the values, for example `10*19` for N=6. If this is accurate I wonder how long it would take that C program to solve for N=6.

2. Symmetries of magic squares

There is a common definition of ‘ distinct" magic squares, e.g. from “Solving Magic Squares”[1]

> “…there are exactly 880 distinct 4x4 magic squares, not counting rotations and reflections…”

There are 7040 magic squares of order 4. The above claims that for a magic square A there are 8 rotations and reflections, which reduces 7040 to 880 solutions. This factor of 8 is attributed to de Bessy, but de Bessy's paper was published posthumously in 1693.

If A is a magic square then so are the reflections `⌽A`,`⊖A`,and `⊖⌽A`. Since `⌽` and `⊖` are commutative there are just the 4 distinct magic squares. `⍉` applied to each of the 4 doubles the result to 8. 90 degree rotation of A is `⍉⊖A`, 180 is `⊖⌽A`, and 270 is `⍉⌽A`.

Actually there are 32 magic squares which can be derived from a magic square of order 4. There are 220 basic solutions from which all 7040 can be derived.

More generally, for a magic square A of order N (>2) one can derive `(!⌊N÷2)×2*2+⌊N÷2` magic squares. This formula has been known for some time by mathematicians, Schoeppel for one, also Benson and Jacoby. In the following sections a reasonable, if not rigorous, proof of this is offered. APL provides a convenient notation for this proof.

3. Definitions & Lemmas

Isomorphic vectors. V and W are isomorphic if and only if `V[⍋V]≡W[⍋W]`.

Lemma 3.1: If V and W are isomorphic integer vectors, `(+/V)=+/W` by the associative property of addition.

Isomorphic squares. A and B are isomorphic if and only if each row of A isomorphic to a row of B, each column of A to some column of B, `1 1⍉A` (main diagonal) with `1 1⍉B`, and `1 1⍉⌽A` (opposite diagonal) with `1 1⍉⌽B`.

Lemma 3.2: If A is a magic square and A and B are isomorphic, then B is a magic square. Follows from 3.1 applied to each row, column, and diagonal of B.

Lemma 3.3: If A and B are isomorphic and `A[I;K]∊B[J;]` then row I of A is isomorphic with row J of B. Proof: `A[I;K]` can only be in one row of B, so since A and B are isomorphic that row must be isomorphic with `A[I;]`.

Corollary: `A[K;I]∊B[;J]` then column I of A if isomorphic with column J of B.

4. Symmetric transforms

4.1. T1 of a magic square A

For any pair I,J such that I<J≤⌊N÷2, apply these row switches:

```
A[I,J,(N+1-I),(N+1-J);]←A[J,I,(N+1-J),(N+1-I);]
```

Rows and columns of the result are isomorphic with rows and columns of A but the diagonals are not.

Then switch columns in the same way:

```
A[;I,J,(N+1-I),(N+1-J)]←A[;J,I,(N+1-J),(N+1-I)]
```

Rows and columns of the result are again isomorphic.

The diagonals of the result are also isomorphic with the same diagonals of A since we have switched 4 pairs of diagonal items on the upper left (A[I;I] & A[J;J]), upper right (A[I;N+1-I] & A[J;N+1-J]), lower left (A[N+1-J;I] & A[N+1-I;J]) and lower right (A[N+1-I;I] & A[N+1-J;J]). So the result is also a magic square.

Through a series of switches any permutation of the first ⌊N÷2 items on the upper left diagonal can be accomplished.

So !⌊N÷2 distinct magic squares can be derived from A via T1.

4.2 T2 of a magic square A

For any I≤⌊N÷2, switch row I with row N+1-I:

```
A[I,(N+1-I);]←A[(N+1-I),I;]
```

Rows and columns of the result are isomorphic with A, diagonals are not, since two items of each diagonal are no longer on the same diagonal.

Then switch column I with column N+1-I.

```
A[;I,(N+1-I)]←A[;(N+1-I),I]
```

Rows and columns are again isomorphic.

Diagonals are isomorphic to the same diagonals since the only change is A[I;I] has switched with A[N+1-I;N+1-I] and A[N+1-I;I] has switched with A[I;N+1-I]. So the result is a magic square.

Each of first ⌊N÷2 rows can be switched, so there are 2*⌊N÷2 distinct magic squares which can be derived from A via T2.

4.3 T1 and T2 are disjoint

Since `(N+1-I)>⌊N÷2` no T2 can satisfy `I<J<⌊N÷2`. So there are `(!⌊N÷2)×2*⌊N÷2` distinct magic squares which can be derived from A via T1 and T2.

4.4 Closure Under T1 and T2

If A and B are isomorphic magic squares, B can be derived from A.

If `A[I;J]=B[I;J]` then (by lemma 3.3) row I of A is isomorphic with row I of B, and column J of A with column J of B.

Since the main diagonals are isomorphic we can apply T1 and T2 to obtain C such that `C[I;I]=B[I;I]` for all `I≤⌊N÷2`.

So row I of C are isomorphic with row I of B and column I of C with column I of B.

Since the diagonals are isomorphic it follows that `C[I;N+1-I]=B[I;N+1-I]` and `C[N+1-I;I]=B[N+1-I;I]`.

So row `N+1-I` of C is isomorphic with row `N+1-I` of B and column `N+1-I` of C with column `N+1-I` of B.

If N is even this shows that `C[I;J]=B[I;J]` for all I,J so `C≡B`.

If N is odd there is exception of the middle row and middle column. For `I≠(N+1)÷2`, `N-1` items in row I and column I match, which forces `C[I;(N+1)÷2]=B[I;(N+1)÷2]` and `C[(N+1)÷2;I]=B[(N+1)÷2;I]`. That leaves just the central item `[(N+1)÷2,(N+1)÷2]` which is also forced to match and so `C≡B`.

4.5 T3: Reflections

If A is a magic square and `B←⌽A` or `B←⊖A` then B is a magic square.

Under `⊖`, `B[;I]` is isomorphic with `A[;I]` and `B[N+1-I;]≡A[I;]`.

Under `⌽`, `B[I;]` is isomorphic with `A[I;]` and `B[;N+1-I]≡A[;I]`.

In each `1 1⍉B` is isomorphic with `1 1⍉⌽A` and `1 1⍉⌽B` with `1 1⍉A`, so B is a magic square.

However B is not isomorhpic to A since `1 1⍉B` is not isomorphic with `1 1⍉A`.

On the other hand if both are applied, say `B←⌽⊖A` then B is isomorphic with A.

It follows that ⊖A or ⌽A cannot be arrived at by T1 and T2 so reflection doubles the number of solutions obtained by T1 and T2.

So the number of solutions via T1, T2, and T3 is (!⌊N÷2)×2*1+⌊N÷2

4.6 T4: Transpose

For any magic square A, B←⍉A is also a magic square with (1 1⍉A)≡1 1⍉B.

The rows of A are isomorphic with the columns of B, and the columns of B with the rows of A. So B is certainly not isomorphic with A and, by transitivity, not with any isomorphism of A.

Suppose C is isomorphic with A. Since (1 1⍉A)≡1 1⍉B by transitivity 1 1⍉B is isomorphic with 1 1⍉C, so 1 1⍉B cannot be isomorphic with 1 1⍉⌽C or 1 1⍉⊖C. So B≢⌽C and B≢⊖C.

This completes the proof that (!⌊N÷2)×2*2+⌊N÷2 distinct magic squares which can be derived from any one solution.

5. Footnote: Associative Magic Squares

An associative magic square has the property that the sum of any item and its diametric opposite is 1+N*2. This property is preserved under any of the four transforms.

There are no associative magic squares of order N if 2=4|N. This was shown by A.H. Frost in 1878. The proof is too detailed to present here. See "Associative magic square"[2]

References

1. "Solving Magic Squares" http://mathpages.com/home/kmath295.htm
2. "Associative magic square" http://en.wikipedia.org/wiki/Associative_magic_square

A tool of thought

Dan Baronet (danb@dyalog.com)

I am often asked “what is APL good for”? I reply that APL is good for almost anything but that it is also very good at prototyping. With it you can experiment and use it as a tool for thinking about the problem at hand. It is easy in APL to manipulate data and build tools to get a better view of the problem and come up with solutions. In the following text we will use APL to think of a solution to a problem involving calculations to solve a mathematical problem. The problem originates from Kakuro[1], a popular puzzle found in newspapers, where you need to know the sets of numbers making up a solution.

The problem

In this problem we need to come up with all the sets of N unique positive single digit numbers (1..9) making up a particular sum S. N and S are the key numbers here, they will be the input to our problem. The output is all the possible sets. The format is unimportant; it could be e.g. a list of sets or a square matrix, N wide.

Most of the code should work in any modern APL. However, the examples were created with Dyalog Version 14. When features are used which are available in Dyalog APL only this is mentioned. `⎕IO←1` is assumed.

For example, there are only 2 sets of 4 unique digits 1 to 9 adding up to 12: (1 2 3 6) and (1 2 4 5).

Attempt #1

The first thought is the easiest: brute force. Can we generate all the possibilities and screen out unwanted ones?

We need to form sequences of N numbers, each from 1 to 9. For example, pairs are (1,1) (1,2) (1,3)…(2,1) (2,2)… (9,9), 81 combinations in all. We can use *catenate* (`,`) to put numbers together:

```
      i3 ← ⍳3  ⍝ define a vector of the numbers 1, 2 and 3
      i3 , i3
1 2 3 1 2 3
```

Not quite what we want, we want to catenate each number to each other:

```
      i3 ,¨ i3
```

```
 1 1  2 2  3 3
```

In Dyalog APL V14 there is a new user command that allows us to box enclosed arrays automatically to better see their nature:

```
      ]box on
Was OFF
      ⍳3 ,¨ ⍳3
┌───┬───┬───┐
│1 1│2 2│3 3│
└───┴───┴───┘
```

Again this is not what we want, what we want is to do the catenation for each element in ⍳3 to each other element in ⍳3, like this:

```
      1 ,¨ ⍳3
┌───┬───┬───┐
│1 1│1 2│1 3│
└───┴───┴───┘
      2 ,¨ ⍳3
┌───┬───┬───┐
│2 1│2 2│2 3│
└───┴───┴───┘
      3 ,¨ ⍳3
┌───┬───┬───┐
│3 1│3 2│3 3│
└───┴───┴───┘
```

APL allows us to do this nicely, distributing the function , without looping, using *jot-dot* (∘.) :

```
     ⍳3 ∘., ⍳3
┌───┬───┬───┐
│1 1│1 2│1 3│
├───┼───┼───┤
│2 1│2 2│2 3│
├───┼───┼───┤
│3 1│3 2│3 3│
└───┴───┴───┘
```

That's better. This will work is all modern APLs. In Dyalog we can use *commute* (⍨) to avoid repeating the argument. Commute normally swaps (commutes) the arguments of a function so a∊⍨b becomes b∊a, but when used monadically it repeats the argument so +⍨a becomes a+a:

```
      ∘., ⍨ ⍳3
┌───┬───┬───┐
│1 1│1 2│1 3│
├───┼───┼───┤
│2 1│2 2│2 3│
├───┼───┼───┤
│3 1│3 2│3 3│
└───┴───┴───┘
```

Let's do it for the numbers 1 to 9:

```
      ∘., ⍨ ⍳9
┌───┬───┬───┬───┬───┬───┬───┬───┬───┐
│1 1│1 2│1 3│1 4│1 5│1 6│1 7│1 8│1 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│2 1│2 2│2 3│2 4│2 5│2 6│2 7│2 8│2 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│3 1│3 2│3 3│3 4│3 5│3 6│3 7│3 8│3 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│4 1│4 2│4 3│4 4│4 5│4 6│4 7│4 8│4 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│5 1│5 2│5 3│5 4│5 5│5 6│5 7│5 8│5 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│6 1│6 2│6 3│6 4│6 5│6 6│6 7│6 8│6 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│7 1│7 2│7 3│7 4│7 5│7 6│7 7│7 8│7 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│8 1│8 2│8 3│8 4│8 5│8 6│8 7│8 8│8 9│
├───┼───┼───┼───┼───┼───┼───┼───┼───┤
│9 1│9 2│9 3│9 4│9 5│9 6│9 7│9 8│9 9│
└───┴───┴───┴───┴───┴───┴───┴───┴───┘
```

Let's keep this list in a variable and find the sum of each and let's find those that add up to, say, 6:

```
      v← , ∘.,⍨ ⍳9  ⍝ turn the table into a list with ravel (,)
      sum← +/¨ v    ⍝ sum each set
      six← 6=sum    ⍝ find the 6s
      six/v         ⍝ extract them
┌───┬───┬───┬───┬───┐
│1 5│2 4│3 3│4 2│5 1│
└───┴───┴───┴───┴───┘
```

Let's write a function to find pairs adding up to a specific number. Here we'll use Dyalog's dynamic functions:

```
      pairs←{ok←⍵=+/¨all←, ∘., ⍨ ⍳9 ⋄ ok/all}
      pairs 6
┌───┬───┬───┬───┬───┐
│1 5│2 4│3 3│4 2│5 1│
└───┴───┴───┴───┴───┘
```

There are 2 problems with this code:

#1, the rule says digits must be unique, so let's add code to only keep the numbers that are different:

```
      pairs←{ok←⍵=+/¨all←, ∘.,⍨ ⍳9 ⋄ ok←ok ∧ ≠/¨all ⋄ ok/all}
      pairs 6
┌───┬───┬───┬───┐
│1 5│2 4│4 2│5 1│
└───┴───┴───┴───┘
```

That's better, but we still need to solve problem #2: some are duplicates, e.g. (1 5) and (5 1) are the same. We should remove them. Let's create a sorting function to reorder the sets:

```
     Sort←{ω[⍋ω]}
```

and use it in our function to reorder each set and use *unique* (∪) to extract the unique ones:

```
      pairs←{ok←ω=+/¨all←,∘.,⍨⍳9 ⋄ ok←ok∧≠/¨all ⋄ ∪ Sort¨ ok/all}

      pairs 6     ⍝ pairs that add up to 6
┌───┬───┐
│1 5│2 4│
└───┴───┘
      pairs 9     ⍝ pairs that add up to 9
┌───┬───┬───┬───┐
│1 8│2 7│3 6│4 5│
└───┴───┴───┴───┘
```

Looks good. What about triples? We can use `∘.,` twice:

```
      i3 ∘., i3 ∘., i3    ⍝ generate a 3 x 3 x 3 of 1, 2 and 3s
┌─────┬─────┬─────┐
│1 1 1│1 1 2│1 1 3│
├─────┼─────┼─────┤
│1 2 1│1 2 2│1 2 3│
├─────┼─────┼─────┤
│1 3 1│1 3 2│1 3 3│
└─────┴─────┴─────┘
┌─────┬─────┬─────┐
│2 1 1│2 1 2│2 1 3│
├─────┼─────┼─────┤
│2 2 1│2 2 2│2 2 3│
├─────┼─────┼─────┤
│2 3 1│2 3 2│2 3 3│
└─────┴─────┴─────┘
┌─────┬─────┬─────┐
│3 1 1│3 1 2│3 1 3│
├─────┼─────┼─────┤
│3 2 1│3 2 2│3 2 3│
├─────┼─────┼─────┤
│3 3 1│3 3 2│3 3 3│
└─────┴─────┴─────┘
     triples←{ (ω=+/¨all)/ all←, i ∘., i ∘., i←⍳9 }
     triples 6
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│1 1 4│1 2 3│1 3 2│1 4 1│2 1 3│2 2 2│2 3 1│3 1 2│3 2 1│4 1 1│
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
```

Some doubles are still there - we can't use ≠ this time. `≠/` on more than 2 numbers is meaningless:

```
      ≠/4 1 1        ⍝ same as 4≠ (1≠1) or  4≠FALSE!!!
1
```

We'll use the nub (unique) of each set to see if it is valid with function `{ω≡∪ω}`: if the digits are unique we'll keep the set. We'll then sort each set and keep the unique ones:

```
      clean←{ ∪Sort¨ ({ω≡∪ω}¨ω)/ω }
      triples←{all←(ω=+/¨all)/all←,i ∘., i ∘., i←ι9 ⋄ clean all}
      triples 6
┌─────┐
│1 2 3│
└─────┘
```

That's better; we now need to write a function that will do it for any number of digits. The left argument will be the number of digits required. That means looping over `∘.,` until we have the proper number of iterations. In Dyalog the *power* operator (⍣) will help with this, it will do the looping for us:

```
      (i3  ∘.,  i3  ∘.,  i3) ≡ (i3 (∘., ⍣ 2) i3)
1
```

So we can write (for N digits we need to run `∘.,` N-1 times)

```
    NCat←{ω (∘., ⍣(α-1)) ω}
```

Or, since the argument is the same on both sides, we can use ⍨:

```
   NCat←{ ∘., ⍣(α-1)⍨ ω}
    ntuple←{ok←ω=+/¨all←, α NCat ι9 ⋄ all←ok/all ⋄ clean all }
```

Let’s find how many ways we can make twelve with four different numbers:

```
 4 ntuple 12
┌───────┬───────┐
│1 2 3 6│1 2 4 5│
└───────┴───────┘
      ρ4 ntuple 12
2
```

Just two. There should also be only one way for 9 digits to add up to 45 (all the numbers 1 to 9):

```
      ρ9 ntuple 45
WS FULL
NCat[0] NCat←{∘.,⍣(α-1)⍨ω}
                ^
```

Oops! Looks like we have a problem Houston. We're trying to generate

```
      9 × 9*9
3486784401
```

more than 3 billion numbers! This is too big on my machine, even with `⎕wa=64039748`.

```
      6 ntuple 35
┌───────────┬───────────┬───────────┬───────────┐
│1 4 6 7 8 9│2 3 6 7 8 9│2 4 5 7 8 9│3 4 5 6 8 9│
└───────────┴───────────┴───────────┴───────────┘
      7 ntuple 39
WS FULL
NCat[0] NCat←{∘.,⍣(⍺-1)⍨⍵}
             ^
```

Looking at

```
      i3 ∘., i3 ∘., i3
┌─────┬─────┬─────┐
│1 1 1│1 1 2│1 1 3│
├─────┼─────┼─────┤
│1 2 1│1 2 2│1 2 3│
├─────┼─────┼─────┤
│1 3 1│1 3 2│1 3 3│
└─────┴─────┴─────┘
…
```

we can see that the numbers in the boxes are the indices of each box. APL has a primitive to produce the indices of any structure: *iota* (⍳) :

```
      (i3 ∘., i3 ∘., i3 ) ≡ ⍳ 3 3 3
1
```

This primitive should take less space to generate and is a lot faster than looping over :

```
      ntupleB←{ok←⍵=+/¨all←,⍳ ⍺⍴9 ⋄ all←ok/all ⋄ clean all}
```

Let's see how much faster it is. There is a user command in Dyalog that allows us compare timings:

```
      ]runtime  "6 ntuple 35"  "6 ntupleB 35"  -compare

  6 ntuple 35  → 2.4E¯1 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  6 ntupleB 35 → 1.4E¯1 | -43% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

But it still suffers from space problems:

```
      7 ntupleB 39
WS FULL
ntuple2[0] ntupleB←{ok←⍵=+/¨all←,⍳⍺⍴9 ⋄ all←ok/all ⋄ clean all}
                                  ^
```

But wait, maybe we can do it another way. How about using encode? Here are the 81 pairs again:

```
      1+ 9 9 ⊤ ¯1+ ⍳9*2
1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2

      3 4 5 6 7 8 9 1 2 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 9
      5 5 5 5 5 5 5 6 6 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1

      9 9 9 9 9 9 9 9
      2 3 4 5 6 7 8 9
```

Again, that would also require too many numbers when N is greater than 6. This brute force method fails for large N, let's see if we can modify it.

Attempt #2

Maybe we can refine the process. Let’s have a look at pairs again:

```
      pairs
{ok←⍵=+/¨all←,∘.,⍨⍳9 ⋄ ok←ok∧ ≠/¨all ⋄ ∪Sort¨ok/all}
```

We don’t really need `Sort` nor the *unique* (`∪`) after. We can eliminate a lot of cases by reordering the checks and taking into account that the numbers in the sets must be in increasing order:

```
      pairs2← {all←(</¨all)/all←, i9 ∘., i9←⍳9 ⋄ (⍵=+/¨all)/all}
      pairs2   9
┌───┬───┬───┬───┐
│1 8│2 7│3 6│4 5│
└───┴───┴───┴───┘
```

Triples are similar. We cannot use `</` on 3 numbers but since the last ones are already ordered we can use `</` on the first 2 . We could write

```
       triples2←{
       all←(</¨2↑¨all) /all←, i9 ∘., (</¨all)/all←,i9 ∘., i9←⍳9
       (⍵=+/¨all)/all
    }
      triples2  19
┌─────┬─────┬─────┬─────┬─────┐
│2 8 9│3 7 9│4 6 9│4 7 8│5 6 8│
└─────┴─────┴─────┴─────┴─────┘
```

Seems to work. Quadruples would work similarly. We should use a function, like Ncat, to generate the combinations, something like

```
      Gen←{ ((</¨2↑¨all)/ all←,(⍳9) ∘., ⍵}
```

We can use the user command `]ROWS` (newly introduced in Version 14.0 of Dyalog) to cut the output to the width of the window (paper here):

```
      ]rows -style=cut
Was -style=long
      Gen ⍳9
```

```
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬──·
│1 2│1 3│1 4│1 5│1 6│1 7│1 8│1 9│2 3│2 4│2 5│2 6│2 7│2 8│2 9│3 4│3 ·
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴──·
      Gen Gen ι9
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬·
│1 2 3│1 2 4│1 2 5│1 2 6│1 2 7│1 2 8│1 2 9│1 3 4│1 3 5│1 3 6│1 3 7│·
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴·
      Gen Gen Gen ι9
┌───────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┬──·
│1 2 3 4│1 2 3 5│1 2 3 6│1 2 3 7│1 2 3 8│1 2 3 9│1 2 4 5│1 2 4 6│1 ·
└───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┴──·
```

Let's try it:

```
      ntuple2←{ ok←ω=+/¨all← Gen⍣(α-1) ι9 ⋄ ok/all}
      3  ntuple2  19
┌─────┬─────┬─────┬─────┬─────┐
│2 8 9│3 7 9│4 6 9│4 7 8│5 6 8│
└─────┴─────┴─────┴─────┴─────┘
```

The ultimate test: can it find the only solution to a 9 digits sequence adding up to 45 `+/ι9`?

```
      9  ntuple2  45
┌─────────────────┐
│1 2 3 4 5 6 7 8 9│
└─────────────────┘
```

It works! We now have a solution. Mission accomplished. Let's see how much space is needed to run it; the user command `]spaceneeded` will provide that information:

```
      ]space  "9 ntuple2  45"
85824
```

Not bad. How much CPU does it take?

```
      ]runtime "9 ntuple2 45" -repeat=1s

* Benchmarking "9 ntuple2 45", repeat=1s
                     Exp
 CPU (avg):  2.835227273
 Elapsed:    2.866477273
```

3 ms. That's good enough.

Out of curiosity, could we have done better?

Attempt #3

Looking carefully at pairs we see that the first number can be 1 to 9 and that the second number can be whatever remains (as long as it is in the range 1 to 9) but not the same, i.e. for pairs adding up to a specific Sum e.g. 10, we have 1 and (10-1), 2 and (10-2), 3 and (10-3), etc. In APL ⎡:

```
      (⍳9) ,¨ 10-⍳9
┌───┬───┬───┬───┬───┬───┬───┬───┬───┐
│1 9│2 8│3 7│4 6│5 5│6 4│7 3│8 2│9 1│
└───┴───┴───┴───┴───┴───┴───┴───┴───┘
```

The sets should be ordered in increasing order so we can limit the numbers 1 to 4 as first number because the largest combination we will have will be n followed by n+1, i.e. 10=n+n+1 or n=4.5, or 4 since we only deal with integers.

```
      pairs3← {i,¨ ⍵-i← ⍳⌊ (⍵-1)÷2 }
      pairs3  9
┌───┬───┬───┬───┐
│1 8│2 7│3 6│4 5│
└───┴───┴───┴───┘
      pairs3  10
┌───┬───┬───┬───┐
│1 9│2 8│3 7│4 6│
└───┴───┴───┴───┘
```

Fine. How about triples? Using the same idea we find that the largest set will be n,(n+1),(n+2) so we can use the numbers from 1 to `⌊(Sum-3)÷3` followed by all the pairs of `Sum` minus that number. For example

```
      Sum←10
      (Sum-3) ÷ 3   ⍝ we start with the numbers 1 and 2
2.333333333
      1,¨ pairs3  Sum-1
┌─────┬─────┬─────┬─────┐
│1 1 8│1 2 7│1 3 6│1 4 5│
└─────┴─────┴─────┴─────┘
      2,¨ pairs3  Sum-2
┌─────┬─────┬─────┐
│2 1 7│2 2 6│2 3 5│
└─────┴─────┴─────┘
```

Not quite. We should ignore any pair starting with a number smaller or equal to our first number.

Let’s modify `pairs3` to accept a (optional) left argument specifying the starting numbers to skip:

```
      pairs3← {⍺←0 ⋄ i,¨ ⍵-i← ⍺↓ ⍳⌊ (⍵-1)÷2 }
      pairs3  9
┌───┬───┬───┬───┐
│1 8│2 7│3 6│4 5│
└───┴───┴───┴───┘
      1 pairs3  9
┌───┬───┬───┐
│2 7│3 6│4 5│
└───┴───┴───┘
```

```
      1,¨ 1 pairs3  Sum-2
┌─────┬─────┐
│1 2 6│1 3 5│
└─────┴─────┘
```

```
      2,¨ 2 pairs3  Sum-2
┌─────┐
│2 3 5│
└─────┘
```

Triples?

```
      triples3←{i← ⍳⌊ (⍵-3)÷3 ⋄ i,¨ i pairs3¨ ⍵-i }
      triples3  9
┌───────────┬───────┐
│┌─┬───┬───┐│┌─┬───┐│
││1│2 6│3 5│││2│3 4││
│└─┴───┴───┘│└─┴───┘│
└───────────┴───────┘
```

Not quite, we need to use *each* (¨) twice:

```
      triples3←{i← ⍳⌊ (⍵-3)÷3 ⋄ i,¨¨ i pairs3¨ ⍵-i }
      triples3  9
┌─────────────┬───────┐
│┌─────┬─────┐│┌─────┐│
││1 2 6│1 3 5│││2 3 4││
│└─────┴─────┘│└─────┘│
└─────────────┴───────┘
```

That’s better but this enclosing business is getting out of hand. Let’s work with matrices:

```
      pairs3B←{⍺←0 ⋄ i,⍪⍵-i←⍺↓⍳⌊(⍵-1)÷2}
      pairs3B  9
1 8
2 7
3 6
4 5
      2 pairs3B  9
3 6
4 5
      triples3B← {i← ⍳⌊ (⍵-3)÷3 ⋄ ↑⍪/i,¨ i pairs3B¨ ⍵-i }
      triples3B  9
1 2 6
1 3 5
2 3 4
```

Seems to work. But there is a pattern here. It looks like a recursive definition.

- If we want a single digit set then the set is the sum if it is below 10.
- If we want a N digit set then it is all the digits from 1 to `⌊(Sum-+/⍳N-1)÷N` followed by the N-1 digit set of `Sum` minus that number.

We’ll need to adjust the starting number and remove any number smaller or equal to the first digit. We need to supply that number as argument, something like:

```
    ntuple3← {
  ⍝ Generate all monotonic combinations of ⍺ numbers adding to ⍵
    (nn sn)←2↑⍺      ⍝ # of numbers needed, numbers to skip
    nn=1 : (⍵≤9)⌿ ⍪⍵ ⍝ solution for 1 number is ⍵ if ≤9
  ⍝ More than 1 #, drop any number ≤ sn
    n1←sn↓⍳8⌊0⌈⌊(⍵- +/ ⍳ nn-1)÷nn ⍝ all possible starting #
    0∊⍴n1 : 0 nn⍴0  ⍝ no solution?
  ⍝ All are starting # followed by the new combination
    ↑⍪/ n1 ,¨ ((nn-1),¨n1) ∇¨ ⍵-n1
 }
```

This solution returns a matrix instead of a list of vectors. Let’s try it:

```
      4 ntuple3  12
1 2 3 6
1 2 4 5
      9 ntuple3  45
1 2 3 4 5 6 7 8 9
```

How does it compare with the previous solution?

```
      ]runtime "9 ntuple2  45 " "9 ntuple3  45" -compare

  9 ntuple2 45  → 2.9E¯3 |   0% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
* 9 ntuple3 45  → 5.3E¯5 | -99% □
      ]space  "9 ntuple3  45"
3620
```

Quite a difference. With just a little more effort we improved our original solution a lot. And there are even faster solutions.

Conclusion

APL provides us with an environment where we can experiment “hands on” to study situations, verify results and make better decisions. With it we can even come up with prototypes that we can use to make further forays into our problem. There is a lot of thinking that could have been done mentally but being able to use the computer really is a bonus.

If you want, there is a video accompanying this article you can have a look at. Go to YouTube and look for “Brute force method to finding all the sets in a row of Kakuro”. You can also try this link: http://youtu.be/bJssWsdXjmY

Have fun!

References

1. Kakuro

Table Diff

Dhrusham Patel (dhrusham.patel@equiniti.com)

The objective of `tablediff` is to compare two tables: an old table and a new table, where the new table is assumed to have been derived from the old table by way of row-edits, insertions and deletions. The result should therefore identify and distinguish between these types of modification. To achieve this, the function compares two tables and returns a pair of aligned tables. In addition to aligning the matched rows, the aligned tables contain empty rows corresponding to insertions and deletions. The process of matching rows is based on solving the Longest Common Subsequence (LCS) problem[1] for rows of the tables.

Here is an example of the function in operation:

```
old new2  (old tablediff new)
  A  A  A    v  v  v     A  A  A
  B  B  B    w  w  w                v  v  v
  C  C  C    -  B  B                w  w  w
  D  D  D    C  -  C     B  B  B    -  B  B
  E  E  E    -  D  -     C  C  C    C  -  C
  F  F  F    x  x  x     D  D  D    -  D  -
  G  G  G    y  y  y     E  E  E
  H  H  H    H  H  H     F  F  F
  I  I  I    D  I  D     G  G  G
  J  J  J    M  M  M                x  x  x
  K  K  K    z  z  z                y  y  y
  L  L  L                H  H  H    H  H  H
  M  M  M                I  I  I    D  I  D
  N  N  N                J  J  J
  O  O  O                K  K  K
                         L  L  L
                         M  M  M    M  M  M
                         N  N  N
                         O  O  O
                                    z  z  z
```

As can be seen above: rows align according to where there is a match, exact or partial. Empty rows in the `old` and `new` tables represent row insertions and row deletions respectively. Note that partial matches have also been aligned; representing where rows have been edited. In particular, notice the instances where partial matches have been aligned despite better matches being available, as doing so would yields a better alignment for the tables as a whole.

The cells contain strings, i.e. character vectors. In this example all the strings have length 1, merely for convenience in presentation.

The key variation in this function from the standard LCS problem is that it tolerates partial matches, thereby allowing the function to trace minor row edits. By calculating degree of match in the range 0 – 1, we find the highest-scoring common subsequence.

The strategy is to match rows in `new` to their originals in `old`; then expand the two tables to align them.

The function has four steps:

1. Find row matches: both partial and exact.
2. Generate and select candidate solutions to evaluate.
3. Find the best match: i.e. the highest-scoring common subsequence.
4. Align matching rows by creating a pair of boolean expansion vectors.

Step 1: Tabulate all row matches

```
      matches←(↓old)∘.≡↓new
```

Split the tables and use an outer-product match to find which rows in `new` match which rows in `old`. But we want to honour partial matches, where a row has been edited.

```
      matches←(↓old)∘.(≡¨)↓new
```

Now each cell of the result is a 3-element boolean. Sum each cell and divide by the number of columns to get a score for each match in the range 0 – 1.

```
      rnew cnew←⍴new
      ms←(+/¨(↓old)∘.(≡¨)↓new)÷cnew ⍝ match scores
```

But this can be written more simply[2]:

```
      ms←(old+.≡⍉new)÷cnew ⍝ match scores
```

The resulting table of row match scores `ms` represents the likeness of each row from the `old` table compared with each row of the `new` table.

Step 2: Generate and select candidate solutions to evaluate

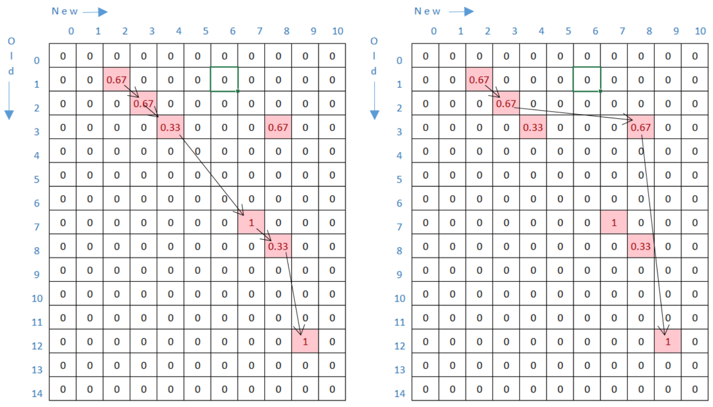


Figure 1: Two solution paths through the match scores table.

The above two figures show two possible match paths. Origin 0, the first matches rows 2 3 4 7 8 9 of `new` to rows 1 2 3 7 8 12 of `old`. The second matches rows 2 3 7 8 of `new` to 1 2 3 12 of `old`.

Because rows are not moved (only inserted, deleted or edited) a match path must specify progressively rising indexes of old and new.

The challenge is to identify all possible match paths and find the highest-scoring.

We start by tabulating all possible selections of rows of `new`. In the figures above, the selections from `new` correspond to columns with matches: in both cases the selection is 0 0 1 1 1 0 0 1 1 1 0.

The expression `(⍴new/2)⊤⍳2*⍴new` gives all possible selections from new:

```
      disp←{'.⎕'[⍵]}
      disp 60↑[1] (⍴new/2)⊤⍳2*⍴new ⍝ first 60 of 2048 cols
............................................................
............................................................
............................................................
............................................................
............................................................
...............................⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
...............⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕................⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
.......⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕
...⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕
.⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕.
⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.
```

We can eliminate selections that select rows of `new` that have no matches at all.

```
      disp {ω/⍨∧⌿~(~∨⌿×ms)⌿ω} (⍴new/2)⊤⍳2*⍴new
................................................................
................................................................
................................⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
................⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕................⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
........⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕⎕⎕⎕........⎕⎕⎕⎕⎕⎕⎕⎕
................................................................
................................................................
....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕....⎕⎕⎕⎕
..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕..⎕⎕
.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕.⎕
................................................................
```

That reduces the number of selections to test from 2048 to 64.

The more matches the better. So we'll try the most promising selections first.

```
      disp sstt←{ω[;⍒+⌿ω]} {ω/⍨∧⌿~(~∨⌿×ms)⌿ω} (⍴new/2)⊤⍳2*⍴new
................................................................
................................................................
⎕.⎕⎕⎕⎕⎕.....⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕..........⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕..........⎕⎕⎕⎕⎕.....⎕.
⎕⎕.⎕⎕⎕⎕.⎕⎕⎕⎕....⎕⎕⎕⎕⎕⎕....⎕⎕⎕⎕⎕⎕......⎕⎕⎕⎕......⎕⎕⎕⎕....⎕....⎕..
⎕⎕⎕.⎕⎕⎕⎕.⎕⎕⎕.⎕⎕⎕...⎕⎕⎕.⎕⎕⎕...⎕⎕⎕...⎕⎕⎕...⎕...⎕⎕⎕...⎕...⎕....⎕...
................................................................
................................................................
⎕⎕⎕⎕.⎕⎕⎕⎕.⎕⎕⎕.⎕⎕.⎕⎕..⎕⎕.⎕⎕.⎕⎕..⎕.⎕⎕..⎕..⎕..⎕⎕..⎕..⎕...⎕....⎕....
⎕⎕⎕⎕⎕.⎕⎕⎕⎕.⎕⎕⎕.⎕⎕.⎕.⎕.⎕⎕.⎕⎕.⎕.⎕.⎕.⎕.⎕..⎕..⎕.⎕.⎕..⎕...⎕....⎕.....
⎕⎕⎕⎕⎕⎕.⎕⎕⎕⎕.⎕⎕⎕.⎕⎕.⎕..⎕⎕⎕.⎕⎕.⎕..⎕⎕.⎕..⎕...⎕⎕.⎕..⎕...⎕....⎕......
................................................................
```

From 2048 possible selections of rows of `new`, we've found 64 to evaluate, and sorted the most promising to the left.

Step 3: Find best solution

A candidate solution is a selection of rows from `new`, all of which match rows from `old`. Because table rows have not been reordered, the solution must pick out successive rows of old.

Each selection of `new` will produce zero or more match paths. The criterion that row matches are monotonically rising (i.e. terms of solution must be consecutive, but not necessarily contiguous, terms of the original sequences) means that it is possible for there to be zero match paths. The longest match path corresponds to the longest common subsequence.

Scoring the matches introduces an additional variable for consideration: we now want the highest-scoring common subsequence. Note that if there are partial row matches the longest common subsequence is not necessarily the highest-scoring common subsequence.

Each selection of rows of new gives a corresponding selection of the match-scores table, in which to look for match paths. The `soln` function returns the highest-scoring match path, and its score.

```
soln←{
    ⍝ best solution (origin-0) and score for match scores ⍵ (matrix)
     ⎕IO←0
     where←{⍵/⍳⍴⍵}
     cmbn←{↑,⊃∘.,/⍵,⊂⊂⍬}                  ⍝ combine lists
     rr←{∧/↑>/1 ¯1↓[1]¨⊂⍵}                ⍝ rising rows
     mrr←{⍵⌿⍨(rr ⍵)∧∧/⍵=⌈\⍵}              ⍝ monotonically rising rows
     rows←mrr cmbn where¨↓[0]×⍵           ⍝ solns are sequences of rows
     0∊⍴rows:⍬ 0                          ⍝ no solution, zero score
     nc←⊃⌽⍴⍵                              ⍝ count cols in ⍵
     scores←+/(,⍵)[(rows×nc)+[1]⍳nc]      ⍝ score by soln
     (scores⍳⌈/scores)∘⊃¨(↓rows)(scores)
 }
```

To find the highest-scoring match path we apply `soln` to the selections in turn, starting with the most promising. But we don't need to evaluate every selection. We can stop when the next selection could not produce a higher-scoring match path. We don't need to apply soln to see that. If the next selection has only four flags and the highest score so far is 4.66, then we need look no further, because the maximum score for a selection with four flags is 4.

```
i←0 ⋄ end←↑⌽⍴sstt ⋄ (sc mo mn)←0 ⍬ ⍬      ⍝ score maskold masknew
:while sc<+/sstt[;i]                      ⍝ scope to improve?
    ∆←(soln ∆/ms),⊂∆←sstt[;i]             ⍝ evaluate next
    (sc mo mn)←(sc<↑∆) ⌽ (sc mo mn) ∆
:until end=i←i+1
```

After the loop, `mo` and `mn` contain a pair of boolean vectors which represent the positions of matching row numbers for each of the old and new tables.

```
      mo mn    ⍝  match booleans
┌→─────────────────────────────────────────────────────────┐
│ ┌→────────────────────────────┐ ┌→────────────────────┐ │
│ │0 1 1 1 0 0 0 1 1 0 0 0 1 0 0│ │0 0 1 1 1 0 0 1 1 1 0│ │
│ └~────────────────────────────┘ └~────────────────────┘ │
└∊─────────────────────────────────────────────────────────┘
```

Step 4: Align matching rows

Once the best solution is found, our final step is to create two boolean expansion vectors that will align the two tables.

Three ways to do this:

Looping function

This function recognises patterns in the two booleans `mo` and `mn` and assembles (column-wise) a 2-row table representing a pair of expansion vectors.

```
Z←2 0ρθ
:Repeat
  :Select ⊃¨mo mn
     :Case 0 0 ⋄ x←1 0   ⍝ Row deleted
     :Case 0 1 ⋄ x←1 0   ⍝ Row deleted
     :Case 1 0 ⋄ x←0 1   ⍝ Row inserted
     :Case 1 1 ⋄ x←1 1   ⍝ match
  :EndSelect
 Z,←x
 (mo mn)↓⍨¨←x
:Until ∨/⊃¨0=ρ¨ mo mn
Z,←↑~ mo mn
expansion←↓Z
```

The select structure above represents a mapping that can be expressed as a short D function:

```
x←{0 0≡⍵:1 0 ⋄ ⌽⍵}⊃¨ mo mn
```

Recursive function

The loop can be rewritten concisely as a D function with tail recursion.

```
stephen←{
         0∊≢¨⍵:↑~⍵
         x←(0 1)(1 1)(1 0)⊃⍨(1 0)(1 1)⍳⊂⊃¨⍵  ⍝ inserted, preserved, deleted
         (⍪x),∇ x↓¨⍵
     }
expansion←↓stephen mo mn
```

Morten's non-looping function

```
morten←{
     rn← ≢ ¨⍵
     ord←⍋(2×∊+\¨⍵)-(∊⍵)
     0 m←(⊂⊂ord)⌷¨(rn/0 1)(∊⍵)
     {((~m∧0≠⍵)/0=⍵)}¨0 1
 }
expansion←morten mo mn
```

This function takes the booleans as its right argument. It "orders the enlisted booleans so matching rows are adjacent." Morten has discussed his function in more detail over on a blog post [3].

Finally, the resulting expansion vectors are used to align the tables.

```
      ,¨/expansion (expansion⍀¨old new)
  1   A   A   A    0
  0                1   v   v   v
  0                1   w   w   w
  1   B   B   B    1   -   B   B
  1   C   C   C    1   C   -   C
  1   D   D   D    1   -   D   -
  1   E   E   E    0
  1   F   F   F    0
  1   G   G   G    0
  0                1   x   x   x
  0                1   y   y   y
  1   H   H   H    1   H   H   H
  1   I   I   I    1   D   I   D
  1   J   J   J    0
  1   K   K   K    0
  1   L   L   L    0
  1   M   M   M    1   M   M   M
  1   N   N   N    0
  1   O   O   O    0
  0                1   z   z   z
```

Listing

Putting the four steps together:

```
∇ Z←old tablediff new;⎕IO;rnew;cnew;ms;aps;wps;sstt;i;end;sc;mo;mn;∆
 ⎕IO←0
 rnew cnew←⍴new

 ⍝ 1. Tabulate match scores
 ms←cnew÷⍨old+.≡⍉new

 ⍝ 2. Generate subsequences to test
 aps←{(ω/2)⊤⍳2*ω}                               ⍝ all possible selections
 wps←{ω/⍨∧⌿~(~∨⌿×ms)⌿ω}                         ⍝ with possible solutions
 sstt←{ω[;⍒+⌿ω]} wps aps rnew                   ⍝ subsequences to test

 ⍝ 3. Select subsequence with highest score
 i←0 ⋄ end←↑⌽⍴sstt ⋄ (sc mo mn)←0 ⍬ ⍬           ⍝ score, maskold, masknew
 :while sc<+/sstt[;i]                           ⍝ scope to improve?
    ∆←(soln ∆/ms),⊂∆←sstt[;i]                   ⍝ evaluate next
    (sc mo mn)←(sc<↑∆) ⌽ (sc mo mn) ∆
 :until end=i←i+1

 ⍝ 4. Align the tables vertically
 Z←old new⍀⍨¨morten mo mn
∇
```

Scope for improvement

1. The problem is symmetrical. That is:

   ```
   old tablediff new ↔ ⌽ new tablediff old
   ```

   The longest common subsequence cannot be longer than `⌊/⊃¨⍴¨old new`. So only the shorter table should be searched for subsequences. In this example `new` is the shorter table. `tablediff` can be improved by switching `old` and `new` if the latter is longer then correspondingly reversing the 2-element result.

2. `cmbn←{↑,⊃∘.,/⍵,⊂⊂⍬}` is space hungry and inefficient for large tables. Instead, Morten recommends a depth-first search.[4].
3. The match scores might be calculated faster if the strings were first hashed to integers. Or, if there are many repeated elements, to indexes to a list of the unique elements.

Acknowledgements

I am indebted to Morten Kromberg of Dyalog, my colleague Stephen Taylor, Mike Thomas of Bedarra and Arthur Whitney of Kx for help with this work. Any errors are of course mine.

References

1. Longest Common Subsequence
   en.wikipedia.org/wiki/Longest_common_subsequence_problem
2. I have Morten Kromberg to thank for spotting this equivalence. Watch out: this inner product returns wrong results in both APL+Win 12.0 and APLX 4.1.6. In these interpreters use the longer outer-product expression, which returns a correct result.
3. Morten's blog post:
   dyalog.com/blog/2014/07/aligning-diff-output-2/
4. John Scholes on Depth-first searching in D
   youtube.com/watch?v=DsZdfnlh_d0

A letter from Dijkstra on APL

Roger K.W. Hui

Acknowledgments. I would like to thank Bob Bernecky, Nicolas Delcros, Jay Foad, and Eric Iverson for comments on the manuscript.

Nick Nickolov brought to my attention comments by Dijkstra on APL [1] that I had not seen before. I contacted the author of the website and obtained a copy of Dijkstra’s letter, transcribed below:

Burroughs
PROF DR EDSGER W DIJKSTRA   PLATAANSTRAAT 5
RESEARCH FELLOW   5671 AL NUENEN THE NETHERLANDS

Dr A.Caplin
[street address]
CROYDON, Surrey
United Kingdom

Tuesday 12 January 1982

Dear Dr Caplin,

thank [*sic*] you for your letter dated 31 May (?) 1981. You were right in your reference to an APL “cult”: some adore it and others abhor it with very few in between. Allow me to offer you another explanation for that phenomenon.

I think that most people (be it subconsciously) realize that “ease of use” is not the most significant aspect. Experience has show that, provided people are sufficiently thrilled by a gadget, they are willing to put up with the most terrible interfaces. Much more important is that the tool shapes the one who trains himself in its usage, just as the words we use shape our thoughts and the instrument forms the violinist. I think that a major reason for shunning APL is that many people are repelled by the influence APL has on its devotees. They implement the prayer “Dear Lord, don’t let me become like them” by ignoring it.

A typical characteristic of the APL devotee is, for instance, his closeness to an implementation of it. I know of a visiting professor at an American University [*sic*] who, trying to teach APL, bitterly complained about the absence of APL terminals. He was clearly unable to teach it without them. And you, too, write to me that you would like to meet me in your part of the world, so that you can “demonstrate APL” to me. This is in sharp contrast to people who prefer programming languages that can be adadequately [*sic*] “demonstrated”—i.e. shown, taught and discussed—with pencil and paper.

The fact that the printed or written word is apparently not the proper medium for the propagation of APL may offer a further explanation for its relative isolation; at the same time that fact may be viewed as one of its major shortcomings.

Your writings made me wonder in which discipline you got your doctor’s degree.

```
   With my greetings and best wishes,
                            yours ever,
                              (signed) Edsger W. Dijkstra

PS. I apologize for the quality of my signature; having broken my right arm
I have to sign with my left hand.
                                   EWD
```

I find Dijkstra's comments deeply ironic, because Ken Iverson invented his notation as a means of communications among people [2], and it was only years later that the notation was implemented on a computer at which time it became APL. Moreover, Dijkstra encountered "Iverson notation" no later than August 1963 before there was an implementation [3]. Even with APL, perhaps especially with APL, one can reasonably do non-trivial things without ever executing it on a computer.

I have read at least one of Dijkstra's EWDs in which he wrote programs using formal methods, at the end of which is derived a provably correct program. As I read it/them, I thought to myself, "APL should have been natural for Dijkstra". One can argue what "provably correct program" means. To me, it means what a typical mathematician means when he/she says a theorem has been proven. I know it is far from saying that the program will produce a correct result in all circumstances (compiler/interpreter has a bug, somebody pulled the plug, cosmic ray strikes a transistor, etc.), but I believe I am using "prove" in the same sense that Dijkstra did.

Like Dijkstra's "visiting professor at an American university", I would be distressed if I had to teach a course on APL without an APL machine. Were it a course on formal methods, one can get by without a machine; but even in a course on formal methods executability would be an asset, because executability keeps you honest, a faithful servant that can be used to check the steps of a proof. Were it a general programming course, it seems extreme to eschew the use of a machine in showing, teaching, and discussing. It would be like trying to learn a natural language without ever conversing with a speaker of that language.

Herewith, two examples of using APL in formal manipulations. Further such examples can be found in Iverson's Turing Award Lecture [4]. A proof is here presented as in [4], a sequence of expressions each identical to its predecessor, annotated with the reasoning.

A Summary of Notation is provided at the end.

Example 1: Ackermann's Function

The derivation first appeared in 1992 [5] in J and is transcribed here in Dyalog APL.

Ackermann’s function is defined on non-negative integers as follows:

```
   ack←{
     0=α: 1+ω
     0=ω: (α-1) ∇ 1
     (α-1) ∇ α ∇ ω-1
   }

   2 ack 3
9
   3 ack 2
29
```

Lemma: If `α ack ω ↔ f⍢(3∘+) ω`, then `(α+1)ack ω ↔ f⍣(1+ω)⍢(3∘+) 1` .

Proof: By induction on `ω` .

`(α+1) ack 0`	basis
`α ack 1`	definition of ack
`f⍢(3∘+) 1`	antecedent of lemma
`f⍣(1+0)⍢(3∘+) 1`	⍣
`(α+1) ack ω`	induction
`α ack (α+1) ack ω-1`	definition of ack
`f⍢(3∘+) (α+1) ack ω-1`	antecedent of lemma
`f⍢(3∘+) f⍣(1+ω-1)⍢(3∘+) 1`	inductive hypothesis
`¯3∘+ f 3∘+ ¯3∘+ f⍣(1+ω-1) 3∘+ 1`	⍢
`¯3∘+ f f⍣(1+ω-1) 3∘+ 1`	+
`¯3∘+ f⍣(1+ω) 3∘+ 1`	⍣
`f⍣(1+ω)⍢(3∘+) 1`	⍢
	QED

Using the lemma (or otherwise), it can be shown that:

```
0∘ack = 1∘+⍢(3∘+)
1∘ack = 2∘+⍢(3∘+)
2∘ack = 2∘×⍢(3∘+)
3∘ack = 2∘*⍢(3∘+)
4∘ack = */∘(⍴∘2)⍢(3∘+)
5∘ack = {*/∘(⍴∘2)⍣(1+ω)⍢(3∘+) 1}
```

Example 2: Inverted Table Index-Of

Presented at the 2013 Dyalog Conference [6].

A *table* is a set of values organized into rows and columns. The rows are records. Values in a column have the same type and shape. A table has a specified number of columns but can have any number of rows. The extended *index-of* on tables finds record indices.

```
   tx                          ty                          tx ⍳ ty
┌──────┬─┬───┬──┐          ┌──────┬─┬───┬──┐          3 1 5 2 5 5
│John  │M│USA│26│          │Min   │F│CN │17│
├──────┼─┼───┼──┤          ├──────┼─┼───┼──┤             tx ⍳ tx
│Mary  │F│UK │24│          │Mary  │F│UK │24│          0 1 2 3 4
├──────┼─┼───┼──┤          ├──────┼─┼───┼──┤
│Monika│F│DE │31│          │John  │M│UK │26│             ty ⍳ ty
├──────┼─┼───┼──┤          ├──────┼─┼───┼──┤          0 1 2 3 4 4
│Min   │F│CN │17│          │Monika│F│DE │31│
├──────┼─┼───┼──┤          ├──────┼─┼───┼──┤
│Max   │M│IT │29│          │Mesut │M│DE │24│
└──────┴─┴───┴──┘          ├──────┼─┼───┼──┤
                           │Mesut │M│DE │24│
                           └──────┴─┴───┴──┘
```

An inverted table is a table with the values of a column collected together. *Comma-bar each* (⍪¨) applied to an inverted table makes it look more like a table. And of course the columns have the same *tally* (≢). A table can be readily inverted and *vice versa*.

```
   x
┌──────┬─────┬───┬──────────────┐
│John  │MFFFM│USA│26 24 31 17 29│
│Mary  │     │UK │              │
│Monika│     │DE │              │
│Min   │     │CN │              │
│Max   │     │IT │              │
└──────┴─────┴───┴──────────────┘
   ⍪¨x                     ≢¨x
                       5 5 5 5
┌──────┬─┬───┬──┐
│John  │M│USA│26│
│Mary  │F│UK │24│
│Monika│F│DE │31│
│Min   │F│CN │17│
│Max   │M│IT │29│
└──────┴─┴───┴──┘

   invert ← {↑¨↓⍉⍵}
   vert   ← {⍉↑⊂⍤¯1¨⍵}

   x  ≡ invert tx
1
   tx ≡ vert x
1
```

A table has array overhead per element. An inverted table has array overhead per column. The difference that this makes becomes apparent when you have a sufficiently large number of rows. The other advantage of an inverted table is that column access is much faster.

An important computation is `x index-of y` where `x` and `y` are compatible inverted tables. Obviously, it can not be just `x⍳y`. The computation obtains by first *verting* the arguments (un-inverting the tables) and then applying `⍳`, but often there is not enough space for that.

```
   ⍪¨x                 ⍪¨y
┌──────┬─┬───┬──┐    ┌──────┬─┬──┬──┐
│John  │M│USA│26│    │Min   │F│CN│17│
│Mary  │F│UK │24│    │Mary  │F│UK│24│
│Monika│F│DE │31│    │John  │M│UK│26│
│Min   │F│CN │17│    │Monika│F│DE│31│
│Max   │M│IT │29│    │Mesut │M│DE│24│
└──────┴─┴───┴──┘    │Mesut │M│DE│24│
                     └──────┴─┴──┴──┘

   x ⍳ y
4 4 4 4

   (vert x) ⍳ (vert y)
3 1 5 2 5 5
```

We derive a more efficient computation of *index-of* on inverted tables:

```
(vert x) ⍳ (vert y)                (a)
({⍉↑⊂⍤¯1¨⍵}x) ⍳ ({⍉↑⊂⍤¯1¨⍵}y)    (b)
(⍉↑⊂⍤¯1¨x) ⍳ (⍉↑⊂⍤¯1¨y)          (c)
(⍉↑x⍳¨x) ⍳ (⍉↑x⍳¨y)              (d)
```

(a) The indices obtain by first uninverting the tables, that is, by first applying `vert` .

(b) Replace `vert` by its definition.

(c) Replace the D-fn by its definition. We see that `⊂⍤¯1` plays a major role. `⊂⍤¯1` encloses, or alternatively computes a scalar representation.

(d) For purposes of *index-of* `x⍳¨x` and `x⍳¨y` have the same information as `⊂⍤¯1¨x` and `⊂⍤¯1¨y` , but are much more efficient representations (small integers *v* the data itself).

Point (d) illustrated on column 0:

```
   ⊂⍤¯1⊢x0←0⊃x
┌──────┬──────┬──────┬──────┬──────┐
│John  │Mary  │Monika│Min   │Max   │
└──────┴──────┴──────┴──────┴──────┘
   x0 ⍳ x0
0 1 2 3 4

   ⊂⍤¯1⊢y0←0⊃y
┌──────┬──────┬──────┬──────┬──────┬──────┐
│Min   │Mary  │John  │Monika│Mesut │Mesut │
└──────┴──────┴──────┴──────┴──────┴──────┘
   x0 ⍳ y0
3 1 0 2 5 5
```

That is, the function `{(⍉↑⍺⍳¨⍺)⍳(⍉↑⍺⍳¨⍵)}` computes `index-of` on inverted tables.

I believe that in another language a derivation such as the one above would be very long (in part because the program would be very long), possibly impractically long.

Summary of Notation

The following table lists the APL notation used in the paper. A complete language reference can be found in [7]. D-fns are described in [7, pp. 112-127] and [8].

↔	equivalent (extralingual)
←	assignment
⍺	left argument
⍵	right argument
×	times
*	exponentiation
⍴	reshape; `n⍴s` makes `n` copies of `s`
⍉	transpose
⍳	index-of
⊂	enclose
⊃	pick
↑	mix (disclose)
↓	split (enclose rows)
⍪	table, ravel the major cells
≡	match
≢	tally, the length of the leading dimension
⊢	right (identity function)
`f∘g`	function composition
`a∘f`	currying (fix left argument)
`f∘a`	currying (fix right argument)
`f⍤r`	rank operator; `f` on rank `r` subarrays
`f⍣n`	power operator; `n` applications of `f` ; the n-th iterate of `f` (`f⍣¯1` is the inverse of `f`)
`f⍢g`	dual operator; `g⍣¯1∘f∘g` (not yet implemented in Dyalog APL)
`f/`	reduce (fold)
`f¨`	each (map)
`{⍺ … ⍵}`	D-function
∇	D-function: recursion
:	D-function: guard

References

1. Daylight, Edgar Graham, A Letter about APL, 2012-04-05. http://www.dijkstrascry.com/node/90
2. Iverson, Kenneth E., A Personal View of APL, IBM Systems Journal, Volume 30, Number 4, 1991-12. http://www.jsoftware.com/papers/APLPersonalView.htm
3. Iverson, Kenneth E., Formalism in Programming Languages, Communications of

the ACM, Volume 7, Number 2, 1964-02. See the last question in the discussion. http://www.jsoftware.com/papers/FPL.htm

4. Iverson, Kenneth E., Notation as a Tool of Thought, Communications of the ACM, Volume 23, Number 8, 1980-08. http://www.jsoftware.com/papers/tot.htm
5. Hui, Roger K.W., Three Combinatoric Puzzles, Vector, Volume 9, Number 2, 1992-10; also in Ackermann's Function, J Wiki Essay, 2005-10-14. http://www.jsoftware.com/jwiki/Essays/Ackermann%27s%20Function
6. Hui, Roger K.W., Rank & Friends, 2013 Dyalog Conference, 2013-10-22. http://www.dyalog.com/dyalog_13/presentations/D08_Rank_and_Friends/friendsscript.htm
7. Dyalog Limited, Dyalog APL Programmer's Guide & Language Reference, Version 13.1, 2012. http://docs.dyalog.com/13.1/Dyalog%20APL%20Programmer%27s%20Guide%20&%20Language%20Reference.pdf
8. Scholes, John, D: A Functional Subset of Dyalog APL, Vector, Volume 17, Number 4, 2001-04. http://archive.vector.org.uk/art10007770

Written in honor of Ken Iverson's 93rd birthday.

Legacy code, survival strategies and Fire

Kai Jaeger (kai@aplteam.com)

This article explains why Fire for Dyalog APL came into existence at all, and why I spent about 5 – unpaid – month of work on this project over the last three years. The name "Fire" points to the two main features: FInd and REplace. Fire is specifically designed to support programmers who have to deal with legacy code: code that is typically quite old, with little or outdated documentation if any, without test cases, no clear structure and/or design and more often than not without any kind of reasonable modularisation, often with none of the original authors being around anymore.

Maintaining and enhancing a legacy system is a big challenge in any case, in particular because rewriting it seems to be the only reasonable thing to do but that's not an option because either the client (or management) is not prepared to accept this or it's too complex or has too many interfaces to other systems to allow this.

Rewriting an application can also pose a threat to APL because it might be used as an excuse to get rid of APL altogether, although clearly the problems causing the headache are not an intrinsic feature of APL at all, they are caused by bad decisions made by ordinary humans. In most if not all cases the client is part of the problem, not the solution. When asked to pay for improving the situation you are very likely to get an answer along the lines of "How many new features are you are going to add? None?! Forget it!"

The only way to survive in a situation like that in the long run is to add test cases and to improve design, modularisation and documentation step by step whenever you touch the code. In the beginning this will look like a simple waste of resources without gaining anything, but it will improve the situation, leading to better and more stable code.

In particular adding test cases will make it easier to carry out changes because you will become more and more confident that those changes will not break the application in many different ways, a typical problem with legacy code.

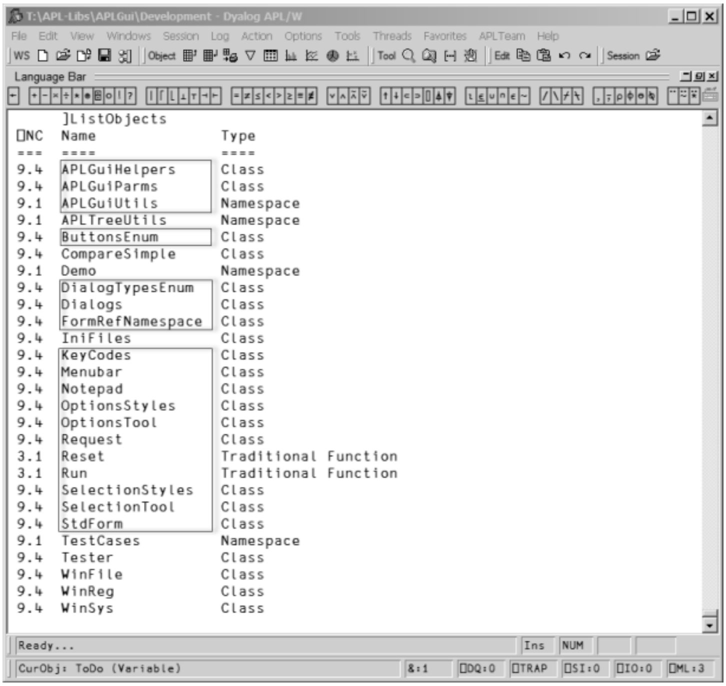
For that reason spending time on improving the code base is also in the interest of the client/management, although they would probably be incapable of realising this if told, so you better keep your mouth shut and hide behind feature enhancements and bug fixes.

Improving a code base often requires changes on a large scale like renaming plenty of objects etc., something that one would prefer to carry out automatically, at least to some extent. Fire is designed to support a programmer in this respect. Let’s discuss some typical problems and how they can be solved with Fire.

Case study I.

Imagine a workspace where you started developing quite a number of classes dedicated to solve a certain problem (GUI utilities in this case), with all these classes situated in the root together with a couple of general classes addressing common problems and used by the GUI-related classes, and a namespace called “Demo” which holds quite a number of functions designed to demonstrate certain aspects of the GUI-related classes.

This is how the namespaces in the root might look:


T:\APL-Libs\APLGui\Development - Dyalog APL/W
File Edit View Windows Session Log Action Options Tools Threads Favorites APLTeam Help
Language Bar

```
      ]ListObjects
⎕NC  Name               Type
===  ====               ====
9.4  APLGuiHelpers      Class
9.4  APLGuiParms        Class
9.1  APLGuiUtils        Namespace
9.1  APLTreeUtils       Namespace
9.4  ButtonsEnum        Class
9.4  CompareSimple      Class
9.1  Demo               Namespace
9.4  DialogTypesEnum    Class
9.4  Dialogs            Class
9.4  FormRefNamespace   Class
9.4  IniFiles           Class
9.4  KeyCodes           Class
9.4  Menubar            Class
9.4  Notepad            Class
9.4  OptionsStyles      Class
9.4  OptionsTool        Class
9.4  Request            Class
3.1  Reset              Traditional Function
3.1  Run                Traditional Function
9.4  SelectionStyles    Class
9.4  SelectionTool      Class
9.4  StdForm            Class
9.1  TestCases          Namespace
9.4  Tester             Class
9.4  WinFile            Class
9.4  WinReg             Class
9.4  WinSys             Class
```

```
Ready...                                   Ins  NUM
CurObj: ToDo (Variable)          &:1  ⎕DQ:0  ⎕TRAP  ⎕SI:0  ⎕IO:0  ⎕ML:3
```

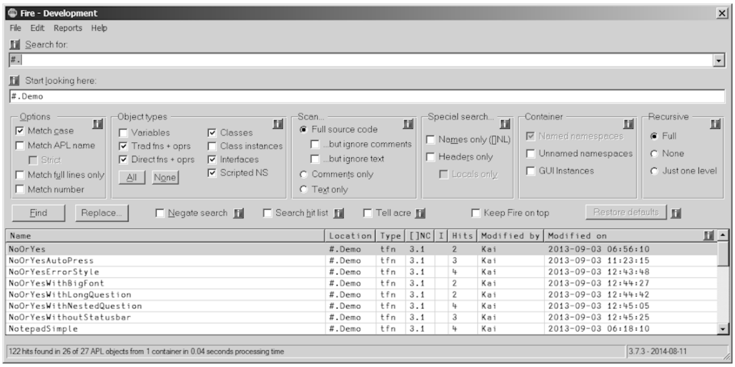
In the future I want to use Phil Last’s excellent code management system acre[1]. In order to do so I decided to restructure the code so that all the GUI-related classes and namespace scripts go into an ordinary namespace `#.GUI` . That is all those shown in boxes.

That's relatively easy to achieve because I wrote my own tool that allows me to move scripts around; unfortunately the Workspace Explorer is still not capable of doing this. After that the root looks like this:

```
      ]ListObjects -n=9
⎕NC   Name           Type
===   ====           ====
9.1   APLTreeUtils   Namespace
9.4   CompareSimple  Class
9.1   GUI            Namespace
9.1   Demo           Namespace
9.4   IniFiles       Class
9.1   TestCases      Namespace
9.4   Tester         Class
9.4   WinFile        Class
9.4   WinReg         Class
9.4   WinSys         Class
```

But that is not enough: the functions in the `#.Demo` namespace as well as all the test cases still try to address the GUI utilities in the root. Those references need to be changed, and that is a perfect task for Fire.

The first goal is to find out how many functions and how many lines of code need to be changed. For this we enter `#.` in the "Search for" box and `#.Demo` into the "Start looking here" box as shown here:



Search for: #.

Start looking here: #.Demo

Options: [✓] Match case; [ ] Match APL name; [ ] Strict; [ ] Match full lines only; [ ] Match number

Object types: [ ] Variables; [✓] Trad fns + oprs; [✓] Direct fns + oprs; [✓] Classes; [ ] Class instances; [✓] Interfaces; [✓] Scripted NS; All; None

Scan...: ● Full source code; [ ] ...but ignore comments; [ ] ...but ignore text; ○ Comments only; ○ Text only

Special search...: [ ] Names only (⎕NL); [ ] Headers only; [ ] Locals only

Container: [✓] Named namespaces; [ ] Unnamed namespaces; [ ] GUI Instances

Recursive: ● Full; ○ None; ○ Just one level

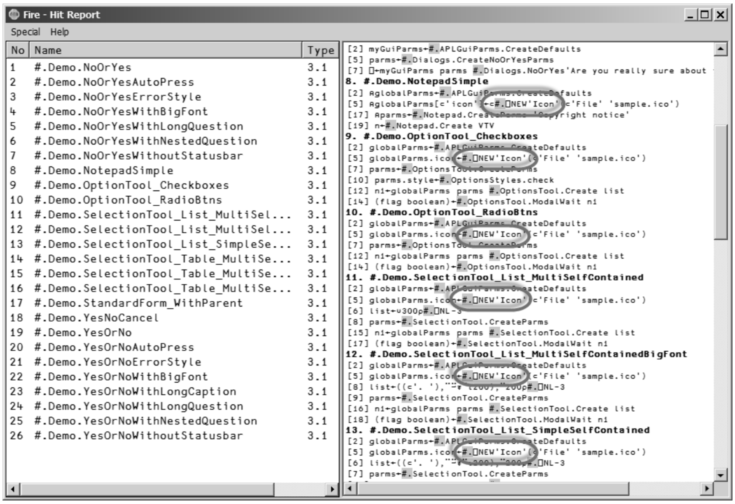
[ ] Negate search; [ ] Search hit list; [ ] Tell acre; [ ] Keep Fire on top

Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
NoOrYes	#.Demo	tfn	3.1		2	Kai	2013-09-03 06:56:10
NoOrYesAutoPress	#.Demo	tfn	3.1		3	Kai	2013-09-03 11:23:15
NoOrYesErrorStyle	#.Demo	tfn	3.1		4	Kai	2013-09-03 12:43:48
NoOrYesWithBigFont	#.Demo	tfn	3.1		2	Kai	2013-09-03 12:44:27
NoOrYesWithLongQuestion	#.Demo	tfn	3.1		2	Kai	2013-09-03 12:44:42
NoOrYesWithNestedQuestion	#.Demo	tfn	3.1		4	Kai	2013-09-03 12:45:05
NoOrYesWithoutStatusbar	#.Demo	tfn	3.1		3	Kai	2013-09-03 12:45:25
NotepadSimple	#.Demo	tfn	3.1		4	Kai	2013-09-03 06:18:10

122 hits found in 26 of 27 APL objects from 1 container in 0.04 seconds processing time | 3.7.3 - 2014-08-11

There are 26 functions with 122 hits (see the status bar of Fire) which potentially need to be changed.

With the next step we check whether the hits we got are really what we are after. The report can be created via the "Report hits" command from the "Reports" menu. It gives an excellent overview:



Fire - Hit Report

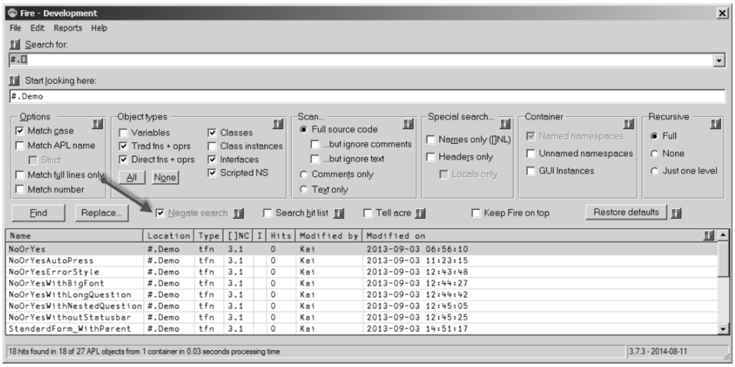
Special Help

No	Name	Type
1	#.Demo.NoOrYes	3.1
2	#.Demo.NoOrYesAutoPress	3.1
3	#.Demo.NoOrYesErrorStyle	3.1
4	#.Demo.NoOrYesWithBigFont	3.1
5	#.Demo.NoOrYesWithLongQuestion	3.1
6	#.Demo.NoOrYesWithNestedQuestion	3.1
7	#.Demo.NoOrYesWithoutStatusbar	3.1
8	#.Demo.NotepadSimple	3.1
9	#.Demo.OptionTool_Checkboxes	3.1
10	#.Demo.OptionTool_RadioBtns	3.1
11	#.Demo.SelectionTool_List_MultiSel...	3.1
12	#.Demo.SelectionTool_List_MultiSel...	3.1
13	#.Demo.SelectionTool_List_SimpleSe...	3.1
14	#.Demo.SelectionTool_Table_MultiSe...	3.1
15	#.Demo.SelectionTool_Table_MultiSe...	3.1
16	#.Demo.SelectionTool_Table_MultiSe...	3.1
17	#.Demo.StandardForm_WithParent	3.1
18	#.Demo.YesNoCancel	3.1
19	#.Demo.YesOrNo	3.1
20	#.Demo.YesOrNoAutoPress	3.1
21	#.Demo.YesOrNoErrorStyle	3.1
22	#.Demo.YesOrNoWithBigFont	3.1
23	#.Demo.YesOrNoWithLongCaption	3.1
24	#.Demo.YesOrNoWithLongQuestion	3.1
25	#.Demo.YesOrNoWithNestedQuestion	3.1
26	#.Demo.YesOrNoWithoutStatusbar	3.1

```
[2] myGuiParms←#.APLGuiParms.CreateDefaults
[5] parms←#.Dialogs.CreateNoOrYesParms
[7] ⎕←myGuiParms parms #.Dialogs.NoOrYes'Are you really sure about
8. #.Demo.NotepadSimple
[2] ⍝globalParms←#.APLGuiParms.CreateDefaults
[5] ⍝globalParms[⊂'icon']←⊂#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[17] ⍝parms←#.Notepad.CreateParms 'Copyright notice'
[19] n←#.Notepad.Create VTV
9. #.Demo.OptionTool_Checkboxes
[2] globalParms←#.APLGuiParms.CreateDefaults
[5] globalParms.icon←#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[7] parms←#.OptionsTool.CreateParms
[10] parms.style←#.OptionsStyles.check
[12] n1←globalParms parms #.OptionsTool.Create list
[14] (flag boolean)←#.OptionsTool.ModalWait n1
10. #.Demo.OptionTool_RadioBtns
[2] globalParms←#.APLGuiParms.CreateDefaults
[5] globalParms.icon←#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[7] parms←#.OptionsTool.CreateParms
[12] n1←globalParms parms #.OptionsTool.Create list
[14] (flag boolean)←#.OptionsTool.ModalWait n1
11. #.Demo.SelectionTool_List_MultiSelfContained
[2] globalParms←#.APLGuiParms.CreateDefaults
[5] globalParms.icon←#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[6] list←∪300⍴#.⎕NL-3
[8] parms←#.SelectionTool.CreateParms
[15] n1←globalParms parms #.SelectionTool.Create list
[17] (flag boolean)←#.SelectionTool.ModalWait n1
12. #.Demo.SelectionTool_List_MultiSelfContainedBigFont
[2] globalParms←#.APLGuiParms.CreateDefaults
[5] globalParms.icon←#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[8] list←((⊂'. '),¨⍕¨⍳200),¨200⍴#.⎕NL-3
[9] parms←#.SelectionTool.CreateParms
[16] n1←globalParms parms #.SelectionTool.Create list
[18] (flag boolean)←#.SelectionTool.ModalWait n1
13. #.Demo.SelectionTool_List_SimpleSelfContained
[2] globalParms←#.APLGuiParms.CreateDefaults
[5] globalParms.icon←#.⎕NEW'Icon'(⊂'File' 'sample.ico')
[6] list←((⊂'. '),¨⍕¨⍳200),¨200⍴#.⎕NL-3
[7] parms←#.SelectionTool.CreateParms
```

As you can see there are quite a number of functions that carry #.⎕NEW in them – these statements must remain unchanged. However, those functions also carry hits we are interested in. What's the best way to deal with this situation?

First we create a hit list with objects that do not contain the string #.⎕" . We can achieve this with these settings:



Fire - Development

File Edit Reports Help

Search for: #.⎕

Start looking here: #.Demo

Options: [✓ Match case] Match APL name, Strict, Match full lines only, Match number

Object types: Variables, [✓ Trad fns + oprs], [✓ Direct fns + oprs], [✓ Classes], Class instances, [✓ Interfaces], [✓ Scripted NS]

Scan...: [✓ Full source code] ...but ignore comments, ...but ignore text, Comments only, Text only

Special search...: Names only (⎕NL), Headers only, Locals only

Container: [✓ Named namespaces] Unnamed namespaces, GUI Instances

Recursive: [✓ Full] None, Just one level

[✓ Negate search] Search hit list, Tell acre, Keep Fire on top

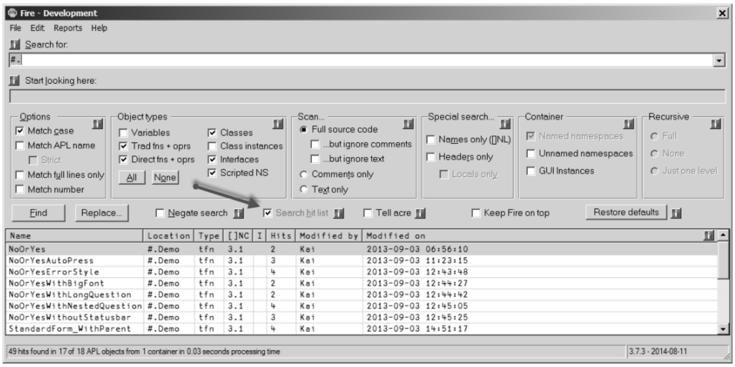
Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
NoOrYes	#.Demo	tfn	3.1		0	Kai	2013-09-03 06:56:10
NoOrYesAutoPress	#.Demo	tfn	3.1		0	Kai	2013-09-03 11:23:15
NoOrYesErrorStyle	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:43:48
NoOrYesWithBigFont	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:44:27
NoOrYesWithLongQuestion	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:44:42
NoOrYesWithNestedQuestion	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:45:05
NoOrYesWithoutStatusbar	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:45:25
StandardForm_WithParent	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:51:17

18 hits found in 18 of 27 APL objects from 1 container in 0.03 seconds processing time

3.7.3 - 2014-08-11

Note that the "Negate search" option was ticked, therefore Fire just lists functions that do not contain the string #.⎕.

Now we untick "Negate search" and tick "Search hit list". Then we search for #. as shown here:



Fire - Development

File Edit Reports Help

Search for: `#.`

Start looking here:

Options: [✓ Match case] ☐ Match APL name ☐ Strict ☐ Match full lines only ☐ Match number

Object types: ☐ Variables [✓ Trad fns + oprs] [✓ Direct fns + oprs] [✓ Classes] ☐ Class instances [✓ Interfaces] [✓ Scripted NS]

Scan...: [✓ Full source code] ☐ ...but ignore comments ☐ ...but ignore text ○ Comments only ○ Text only

Special search...: ☐ Names only (⎕NL) ☐ Headers only ☐ Locals only

Container: [✓ Named namespaces] ☐ Unnamed namespaces ☐ GUI Instances

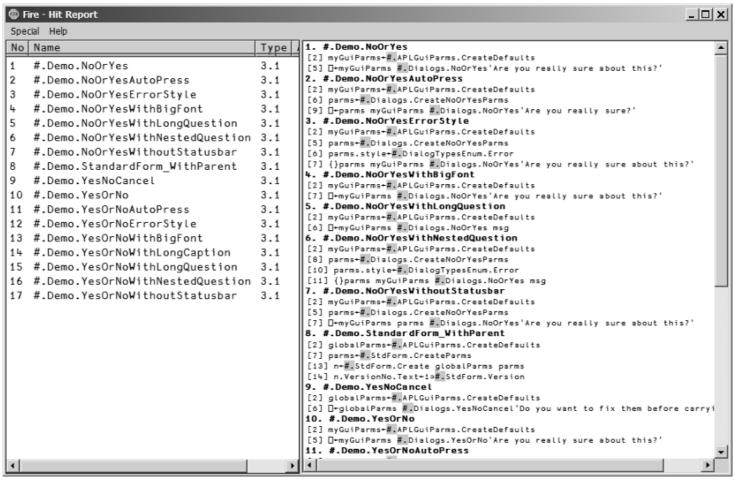
Recursive: ○ Full ○ None ○ Just one level

☐ Negate search [✓ Search hit list] ☐ Tell acre ☐ Keep Fire on top

Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
NoOrYes	#.Demo	tfn	3.1		2	Kai	2013-09-03 06:56:10
NoOrYesAutoPress	#.Demo	tfn	3.1		3	Kai	2013-09-03 11:23:15
NoOrYesErrorStyle	#.Demo	tfn	3.1		4	Kai	2013-09-03 12:43:48
NoOrYesWithBigFont	#.Demo	tfn	3.1		2	Kai	2013-09-03 12:44:27
NoOrYesWithLongQuestion	#.Demo	tfn	3.1		2	Kai	2013-09-03 12:44:42
NoOrYesWithNestedQuestion	#.Demo	tfn	3.1		4	Kai	2013-09-03 12:45:05
NoOrYesWithoutStatusbar	#.Demo	tfn	3.1		3	Kai	2013-09-03 12:45:25
StandardForm_WithParent	#.Demo	tfn	3.1		4	Kai	2013-09-03 14:51:17

49 hits found in 17 of 18 APL objects from 1 container in 0.03 seconds processing time | 3.7.3 - 2014-08-11

That results in 17 functions. This is again the “Report hits” report:



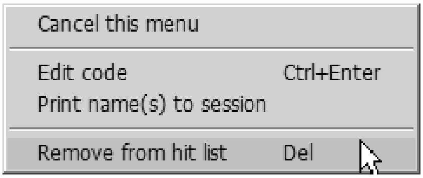
Fire - Hit Report

Special Help

No	Name	Type
1	#.Demo.NoOrYes	3.1
2	#.Demo.NoOrYesAutoPress	3.1
3	#.Demo.NoOrYesErrorStyle	3.1
4	#.Demo.NoOrYesWithBigFont	3.1
5	#.Demo.NoOrYesWithLongQuestion	3.1
6	#.Demo.NoOrYesWithNestedQuestion	3.1
7	#.Demo.NoOrYesWithoutStatusbar	3.1
8	#.Demo.StandardForm_WithParent	3.1
9	#.Demo.YesNoCancel	3.1
10	#.Demo.YesOrNo	3.1
11	#.Demo.YesOrNoAutoPress	3.1
12	#.Demo.YesOrNoErrorStyle	3.1
13	#.Demo.YesOrNoWithBigFont	3.1
14	#.Demo.YesOrNoWithLongCaption	3.1
15	#.Demo.YesOrNoWithLongQuestion	3.1
16	#.Demo.YesOrNoWithNestedQuestion	3.1
17	#.Demo.YesOrNoWithoutStatusbar	3.1

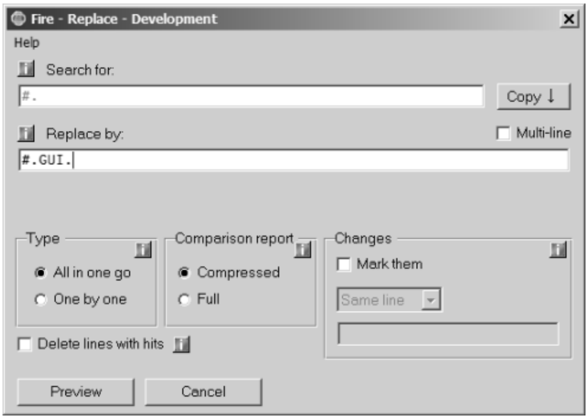
```
1. #.Demo.NoOrYes
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [5] ⎕←myGuiParms #.Dialogs.NoOrYes'Are you really sure about this?'
2. #.Demo.NoOrYesAutoPress
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [6] parms←#.Dialogs.CreateNoOrYesParms
 [9] ⎕←parms myGuiParms #.Dialogs.NoOrYes'Are you really sure?'
3. #.Demo.NoOrYesErrorStyle
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [5] parms←#.Dialogs.CreateNoOrYesParms
 [6] parms.style←#.DialogTypesEnum.Error
 [7] {}parms myGuiParms #.Dialogs.NoOrYes'Are you really sure about this?'
4. #.Demo.NoOrYesWithBigFont
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [7] ⎕←myGuiParms #.Dialogs.NoOrYes'Are you really sure about this?'
5. #.Demo.NoOrYesWithLongQuestion
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [6] ⎕←myGuiParms #.Dialogs.NoOrYes msg
6. #.Demo.NoOrYesWithNestedQuestion
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [8] parms←#.Dialogs.CreateNoOrYesParms
 [10] parms.style←#.DialogTypesEnum.Error
 [11] {}parms myGuiParms #.Dialogs.NoOrYes msg
7. #.Demo.NoOrYesWithoutStatusbar
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [5] parms←#.Dialogs.CreateNoOrYesParms
 [7] ⎕←myGuiParms parms #.Dialogs.NoOrYes'Are you really sure about this?'
8. #.Demo.StandardForm_WithParent
 [2] globalParms←#.APLGuiParms.CreateDefaults
 [7] parms←#.StdForm.CreateParms
 [13] n←#.StdForm.Create globalParms parms
 [14] n.VersionNo.Text←1⊃#.StdForm.Version
9. #.Demo.YesNoCancel
 [2] globalParms←#.APLGuiParms.CreateDefaults
 [6] ⎕←globalParms #.Dialogs.YesNoCancel'Do you want to fix them before carryi
10. #.Demo.YesOrNo
 [2] myGuiParms←#.APLGuiParms.CreateDefaults
 [5] ⎕←myGuiParms #.Dialogs.YesOrNo'Are you really sure about this?'
11. #.Demo.YesOrNoAutoPress
```

This is indeed a big step forward: apparently only stuff that needs to be changed is reported.



However, if there were still some items in the hit list we don’t want to change we could easily remove them from the hit list via the Hit Report’s context menu:

After pressing the “Replace” button we get the “Replace” dialog box were we can enter this:



Search for: `#.`

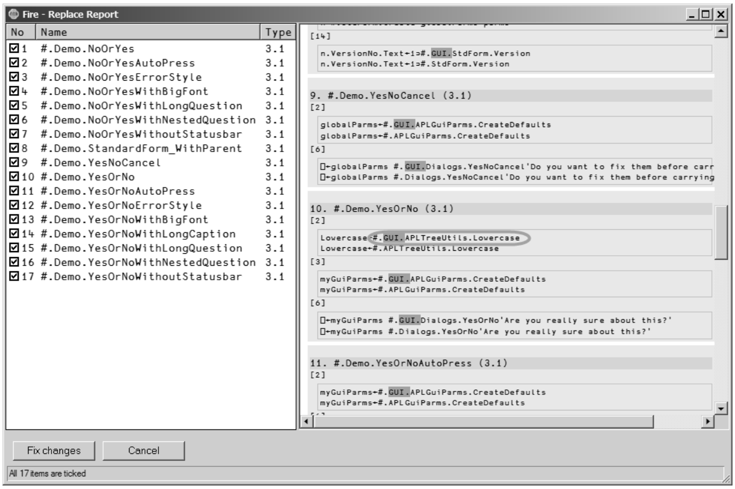
Replace by: `#.GUI.`

Type: [✓ All in one go] One by one

Comparison report: [✓ Compressed] Full

Delete lines with hits: ☐

After clicking on “Preview” we get this:



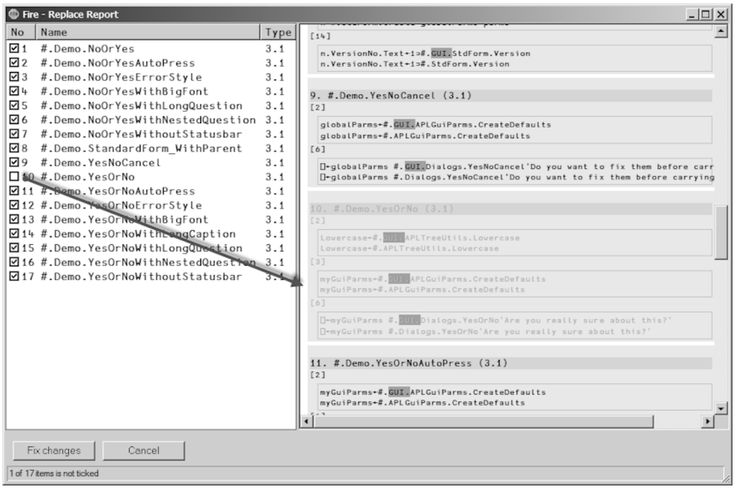
No	Name	Type
☑1	#.Demo.NoOrYes	3.1
☑2	#.Demo.NoOrYesAutoPress	3.1
☑3	#.Demo.NoOrYesErrorStyle	3.1
☑4	#.Demo.NoOrYesWithBigFont	3.1
☑5	#.Demo.NoOrYesWithLongQuestion	3.1
☑6	#.Demo.NoOrYesWithNestedQuestion	3.1
☑7	#.Demo.NoOrYesWithoutStatusbar	3.1
☑8	#.Demo.StandardForm_WithParent	3.1
☑9	#.Demo.YesNoCancel	3.1
☑10	#.Demo.YesOrNo	3.1
☑11	#.Demo.YesOrNoAutoPress	3.1
☑12	#.Demo.YesOrNoErrorStyle	3.1
☑13	#.Demo.YesOrNoWithBigFont	3.1
☑14	#.Demo.YesOrNoWithLongCaption	3.1
☑15	#.Demo.YesOrNoWithLongQuestion	3.1
☑16	#.Demo.YesOrNoWithNestedQuestion	3.1
☑17	#.Demo.YesOrNoWithoutStatusbar	3.1

```
[14]
 n.VersionNo.Text←1⊃#.GUI.StdForm.Version
 n.VersionNo.Text←1⊃#.StdForm.Version

9. #.Demo.YesNoCancel (3.1)
[2]
 globalParms←#.GUI.APLGuiParms.CreateDefaults
 globalParms←#.APLGuiParms.CreateDefaults
[6]
 ⎕←globalParms #.GUI.Dialogs.YesNoCancel'Do you want to fix them before carr
 ⎕←globalParms #.Dialogs.YesNoCancel'Do you want to fix them before carrying

10. #.Demo.YesOrNo (3.1)
[2]
 Lowercase←#.GUI.APLTreeUtils.Lowercase
 Lowercase←#.APLTreeUtils.Lowercase
[3]
 myGuiParms←#.GUI.APLGuiParms.CreateDefaults
 myGuiParms←#.APLGuiParms.CreateDefaults
[6]
 ⎕←myGuiParms #.GUI.Dialogs.YesOrNo'Are you really sure about this?'
 ⎕←myGuiParms #.Dialogs.YesOrNo'Are you really sure about this?'

11. #.Demo.YesOrNoAutoPress (3.1)
[2]
 myGuiParms←#.GUI.APLGuiParms.CreateDefaults
 myGuiParms←#.APLGuiParms.CreateDefaults
```

All 17 items are ticked

Everything is fine except number 19 which is the function `#.Demo.YesOrNo` : apart from two lines that we want to change indeed (3 and 6) there is also a line we don’t want to change: line number 2.

One way to deal with this is to change the function anyway and then to fix line 2 in the editor afterwards. Here however we use a different approach which is much more appropriate in case we want to exclude not just one but quite a number of changes: we simply untick the checkbox number 10 in the tree view. That leads to this:



No	Name	Type
☑ 1	#.Demo.NoOrYes	3.1
☑ 2	#.Demo.NoOrYesAutoPress	3.1
☑ 3	#.Demo.NoOrYesErrorStyle	3.1
☑ 4	#.Demo.NoOrYesWithBigFont	3.1
☑ 5	#.Demo.NoOrYesWithLongQuestion	3.1
☑ 6	#.Demo.NoOrYesWithNestedQuestion	3.1
☑ 7	#.Demo.NoOrYesWithoutStatusbar	3.1
☑ 8	#.Demo.StandardForm_WithParent	3.1
☑ 9	#.Demo.YesNoCancel	3.1
☐ 10	#.Demo.YesOrNo	3.1
☑ 11	#.Demo.YesOrNoAutoPress	3.1
☑ 12	#.Demo.YesOrNoErrorStyle	3.1
☑ 13	#.Demo.YesOrNoWithBigFont	3.1
☑ 14	#.Demo.YesOrNoWithLongCaption	3.1
☑ 15	#.Demo.YesOrNoWithLongQuestion	3.1
☑ 16	#.Demo.YesOrNoWithNestedQuestion	3.1
☑ 17	#.Demo.YesOrNoWithoutStatusbar	3.1

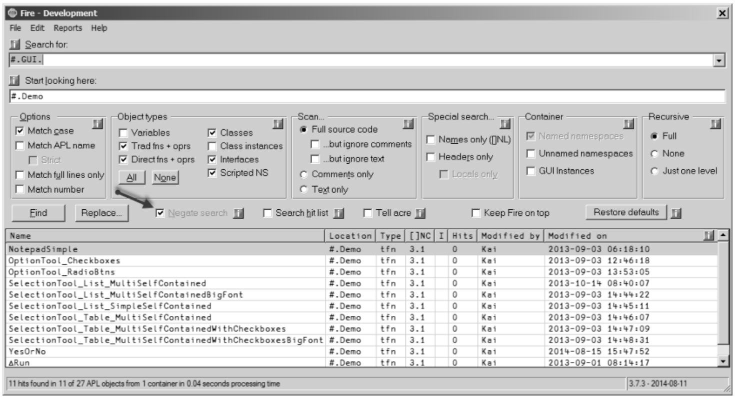
```
[14]
  n.VersionNo.Text←1⊃#.GUI.StdForm.Version
  n.VersionNo.Text←1⊃#.StdForm.Version

9. #.Demo.YesNoCancel (3.1)
[2]
  globalParms←#.GUI.APLGuiParms.CreateDefaults
  globalParms←#.APLGuiParms.CreateDefaults
[6]
  ⎕←globalParms #.GUI.Dialogs.YesNoCancel'Do you want to fix them before carr
  ⎕←globalParms #.Dialogs.YesNoCancel'Do you want to fix them before carrying

10. #.Demo.YesOrNo (3.1)
[2]
  Lowercase←#.GUI.APLTreeUtils.Lowercase
  Lowercase←#.APLTreeUtils.Lowercase
[3]
  myGuiParms←#.GUI.APLGuiParms.CreateDefaults
  myGuiParms←#.APLGuiParms.CreateDefaults
[6]
  ⎕←myGuiParms #.GUI.Dialogs.YesOrNo'Are you really sure about this?'
  ⎕←myGuiParms #.Dialogs.YesOrNo'Are you really sure about this?'

11. #.Demo.YesOrNoAutoPress (3.1)
[2]
  myGuiParms←#.GUI.APLGuiParms.CreateDefaults
  myGuiParms←#.APLGuiParms.CreateDefaults
```

1 of 17 items is not ticked

`#.Demo.YesOrNo` is now greyed, indicating that the function will not be processed any more.

Finally we press the “Fix changes” button. First part of the task is done – the majority of the necessary changes have been carried out already.

In order to address the remaining problems we first want to get a list of objects that do *not* contain any reference to `#.GUI.` because those that do are the ones we have just changed, so we are not interested in them. In order to achieve that we tick the "Negate search" check box and repeat the search:



Search for: `#.GUI.`

Start looking here: `#.Demo`

Options: ☑ Match case ☐ Match APL name ☐ Strict ☐ Match full lines only ☐ Match number

Object types: ☐ Variables ☑ Trad fns + oprs ☑ Direct fns + oprs ☑ Classes ☐ Class instances ☑ Interfaces ☑ Scripted NS

Scan...: [✓ Full source code] ☐ ...but ignore comments ☐ ...but ignore text; Comments only; Text only

Special search...: ☐ Names only (⎕NL) ☐ Headers only ☐ Locals only

Container: ☑ Named namespaces ☐ Unnamed namespaces ☐ GUI Instances

Recursive: [✓ Full] None; Just one level

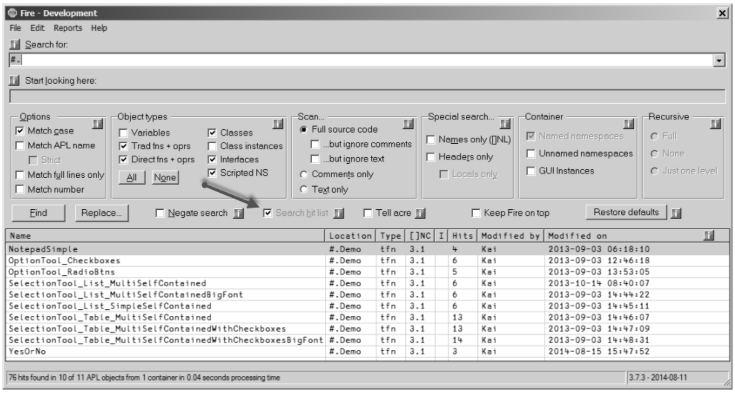
☑ Negate search ☐ Search hit list ☐ Tell acre ☐ Keep Fire on top

Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
NotepadSimple	#.Demo	tfn	3.1		0	Kai	2013-09-03 06:18:10
OptionTool_Checkboxes	#.Demo	tfn	3.1		0	Kai	2013-09-03 12:46:18
OptionTool_RadioBtns	#.Demo	tfn	3.1		0	Kai	2013-09-03 13:53:05
SelectionTool_List_MultiSelfContained	#.Demo	tfn	3.1		0	Kai	2013-10-14 08:40:07
SelectionTool_List_MultiSelfContainedBigFont	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:44:22
SelectionTool_List_SimpleSelfContained	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:45:11
SelectionTool_Table_MultiSelfContained	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:46:07
SelectionTool_Table_MultiSelfContainedWithCheckboxes	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:47:09
SelectionTool_Table_MultiSelfContainedWithCheckboxesBigFont	#.Demo	tfn	3.1		0	Kai	2013-09-03 14:48:31
YesOrNo	#.Demo	tfn	3.1		0	Kai	2014-08-15 15:47:52
∆Run	#.Demo	tfn	3.1		0	Kai	2013-09-01 08:14:17

11 hits found in 11 of 27 APL objects from 1 container in 0.04 seconds processing time   3.7.3 - 2014-08-11

Eleven objects do not contain any reference to `#.GUI` as of yet. These still need to change. Now let’s search this hit list for any references to `#.`

Note that the "Search hit list" check box is ticked now; that restricts the search the the objects in the hit list:



Search for: #.

Options: [✓ Match case], Match APL name, Strict, Match full lines only, Match number

Object types: Variables, [✓ Trad fns + oprs], [✓ Direct fns + oprs], [✓ Classes], Class instances, [✓ Interfaces], [✓ Scripted NS]

Scan...: [✓ Full source code], ...but ignore comments, ...but ignore text, Comments only, Text only

Special search...: Names only (⎕NL), Headers only, Locals only

Container: [✓ Named namespaces], Unnamed namespaces, GUI Instances

Recursive: Full, None, Just one level

Negate search, [✓ Search hit list], Tell acre, Keep Fire on top

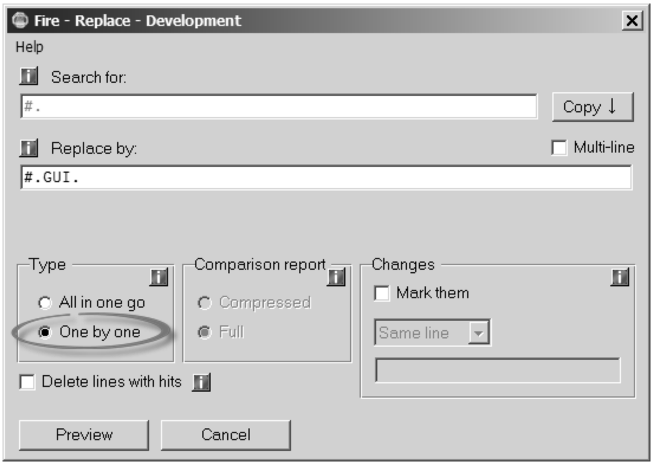
Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
NotepadSimple	#.Demo	tfn	3.1		4	Kai	2013-09-03 06:18:10
OptionTool_Checkboxes	#.Demo	tfn	3.1		6	Kai	2013-09-03 12:46:18
OptionTool_RadioBtns	#.Demo	tfn	3.1		5	Kai	2013-09-03 13:53:05
SelectionTool_List_MultiSelfContained	#.Demo	tfn	3.1		6	Kai	2013-10-14 08:40:07
SelectionTool_List_MultiSelfContainedBigFont	#.Demo	tfn	3.1		6	Kai	2013-09-03 14:44:22
SelectionTool_List_SimpleSelfContained	#.Demo	tfn	3.1		6	Kai	2013-09-03 14:45:11
SelectionTool_Table_MultiSelfContained	#.Demo	tfn	3.1		13	Kai	2013-09-03 14:46:07
SelectionTool_Table_MultiSelfContainedWithCheckboxes	#.Demo	tfn	3.1		13	Kai	2013-09-03 14:47:09
SelectionTool_Table_MultiSelfContainedWithCheckboxesBigFont	#.Demo	tfn	3.1		14	Kai	2013-09-03 14:48:31
YesOrNo	#.Demo	tfn	3.1		3	Kai	2014-08-15 15:47:52

76 hits found in 10 of 11 APL objects from 1 container in 0.04 seconds processing time | 3.7.3 - 2014-08-11

That has reduced the number of objects down to 10.

We already know that all remaining objects do need to be changed but we also know that some of them have references to `#.` which must remain unchanged. There is no escape route; this problem cannot be solved automatically. However, Fire can still be of great help in this situation.

After a click on the “Replace” button we modify the default setting of the “Replace” dialog box:



Search for: #.

Replace by: #.GUI.

Multi-line

Type: All in one go [✓ One by one]

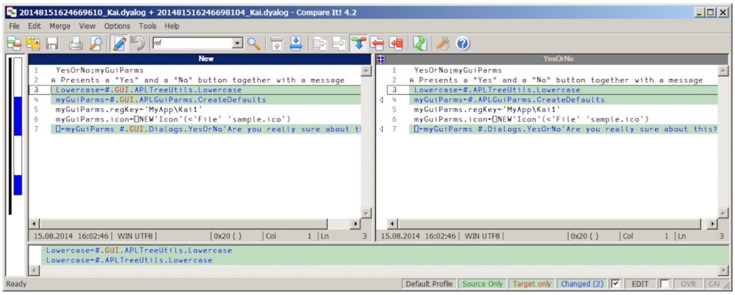
Comparison report: Compressed, Full

Changes: Mark them, Same line

Delete lines with hits

Note that here we assume that you have installed the excellent 3rd-party tool “CompareIt!” on your machine. If that is not the case than a very basic built-in comparison tool will be used.

This allows the user to accept – or deny – changes on a hit-by-hit bases for one object after the other:



20148151624669610_Kai.dyalog + 201481516246698104_Kai.dyalog - Compare It! 4.2

New

```
 YesOrNo;myGuiParms
⍝ Presents a "Yes" and a "No" button together with a message
 Lowercase←#.GUI.APLTreeUtils.Lowercase
 myGuiParms←#.GUI.APLGuiParms.CreateDefaults
 myGuiParms.regKey←'MyApp\Kai1'
 myGuiParms.icon←⎕NEW'Icon'(⊂'File' 'sample.ico')
 ⎕←myGuiParms #.GUI.Dialogs.YesOrNo'Are you really sure about t
```

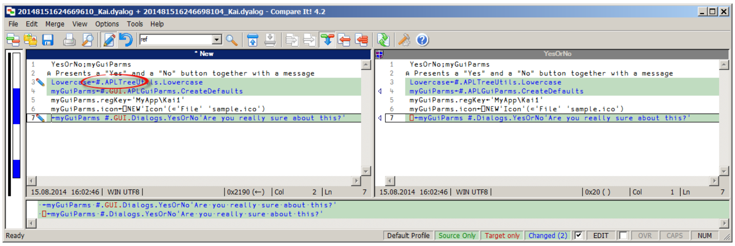
YesOrNo

```
 YesOrNo;myGuiParms
⍝ Presents a "Yes" and a "No" button together with a message
 Lowercase←#.APLTreeUtils.Lowercase
 myGuiParms←#.APLGuiParms.CreateDefaults
 myGuiParms.regKey←'MyApp\Kai1'
 myGuiParms.icon←⎕NEW'Icon'(⊂'File' 'sample.ico')
 ⎕←myGuiParms #.Dialogs.YesOrNo'Are you really sure about this?
```

```
15.08.2014  16:02:46 | WIN UTF8 |   |0x20 ( )  |Col   1 |Ln   3
15.08.2014  16:02:46 | WIN UTF8 |   |0x20 ( )  |Col   1 |Ln   3
·Lowercase←#.GUI.APLTreeUtils.Lowercase
·Lowercase←#.APLTreeUtils.Lowercase
Ready   Default Profile | Source Only | Target only | Changed (2) | EDIT | OVR | CAI
```

The highlighted areas can be moved from the right pane to the left by clicking at the arrow(s) but not the other way around: the icon shown in the caption of the right pane indicates that the right pane is read-only. You can also simply edit the code in the left pane.

Either way, we should end up with the following and the problem is solved.



20148151624669610_Kai.dyalog + 201481516246698104_Kai.dyalog - Compare It! 4.2

* New

```
 YesOrNo;myGuiParms
⍝ Presents a "Yes" and a "No" button together with a message
 Lowercase←#.APLTreeUtils.Lowercase
 myGuiParms←#.GUI.APLGuiParms.CreateDefaults
 myGuiParms.regKey←'MyApp\Kai1'
 myGuiParms.icon←⎕NEW'Icon'(⊂'File' 'sample.ico')
←myGuiParms #.GUI.Dialogs.YesOrNo'Are you really sure about this?'
```

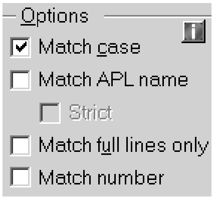
YesOrNo

```
 YesOrNo;myGuiParms
⍝ Presents a "Yes" and a "No" button together with a message
 Lowercase←#.APLTreeUtils.Lowercase
 myGuiParms←#.APLGuiParms.CreateDefaults
 myGuiParms.regKey←'MyApp\Kai1'
 myGuiParms.icon←⎕NEW'Icon'(⊂'File' 'sample.ico')
 ⎕←myGuiParms #.Dialogs.YesOrNo'Are you really sure about this?'
```

```
15.08.2014  16:02:46 | WIN UTF8 |   |0x2190 (←)  |Col   2 |Ln   7
15.08.2014  16:02:46 | WIN UTF8 |   |0x20 ( )    |Col   1 |Ln   7
·←myGuiParms·#.GUI.Dialogs.YesOrNo'Are·you·really·sure·about·this?'
·⎕←myGuiParms·#.Dialogs.YesOrNo'Are·you·really·sure·about·this?'
Ready   Default Profile | Source Only | Target only | Changed (2) | EDIT | OVR | CAPS | NUM
```

Case study II.

With version 3.3.0 Fire itself changed its user interface: a new check box “Strict” became available:



Options

- ☑ Match case
- ☐ Match APL name
  - ☐ Strict
- ☐ Match full lines only
- ☐ Match number

The option is active only when the “Match APL name” check box is ticked. With both check boxes ticked an entry like `Foo.` in “Search for” is rejected by Fire. With just “Match APL name” ticked but not “Strict” you can search for `Foo.`, `.Foo` or `.Foo.` and it will find such strings while a reference to `Foo` without a dot won’t be found. In April 2014 I realized that this is a very useful feature and added it immediately to Fire.

That posed a problem: Fire comes with a large set of test cases: at the time of writing 121. The vast majority fires up the GUI and then sets properties. This is how a typical test case looks:

```
R←Test_Search_001(stopFlag batchFlag);n;⎕TRAP
⍝ Search for "a" everywhere with "Names only"
 ⎕TRAP←(999 'C' '. ⍝ Deliberate error')(0 'N')
 R←1

⍝ Preconditions
 1 #.Fire.Run 0
 n←#.Fire.GUI.n

 n.SearchFor.Text←'a'
 n.LookIn.Text←'#'

 n.Case.State←0
 n.APL_Name.State←0
 n.FullLineOnly.State←0
 n.AsNumber.State←0

 n.Vars.State←1
 n.FnsOprsTrad.State←1
 n.FnsOprsDirect.State←1
 n.Classes.State←1
 n.Interfaces.State←1

 n.ScriptedNamespaces.State←1

 n.Code.State←1
 n.NoComments.State←1
 n.NoText.State←0
 n.CommentsOnly.State←0
 n.TextOnly.State←0

 n.NamesOnly.State←1
 n.HeaderOnly.State←0
 n.LocalsOnly.State←0

 n.NamedNamespaces.State←1
 n.UnnamedNamespaces.State←0
 n.GuiInstances.State←1

 n.Recursive.State←1
 n.RecursiveOneLevel.State←0
 n.RecursiveNone.State←0

 n.Negate.State←0
 n.ReuseSearch.State←0
 n.acre.State←0
 n.acre.State←0

 {}∆Select n.StartBtn
 ∆Process n.Form
 →PassesIf(0<0⊃⍴n.HitList.ReportInfo)

⍝ Tidy up
  CloseFire
 R←0                ⍝ Okay
```

These lines pose the problem:

```
n.Case.State←0
n.APL_Name.State←0
n.FullLineOnly.State←0
```

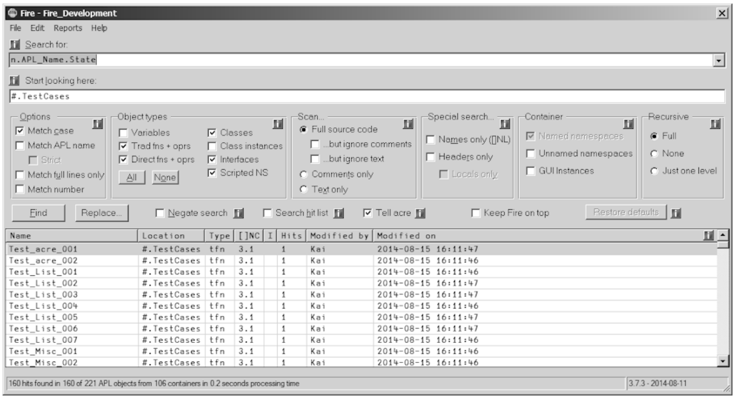
After

```
n.APL_Name.State←0
```

there should be a line:

```
n.StrictOnNames.(Active State)←0
```

That can be achieved with Fire quite easily. First we search for `n.APL_Name.State` just in `#.TestCases` :



Search for: n.APL_Name.State

Start looking here: #.TestCases

Options: [✓ Match case] Match APL name; Strict; Match full lines only; Match number

Object types: Variables; [✓ Trad fns + oprs]; [✓ Direct fns + oprs]; [✓ Classes]; Class instances; [✓ Interfaces]; [✓ Scripted NS]

Scan...: [✓ Full source code]; ...but ignore comments; ...but ignore text; Comments only; Text only

Special search...: Names only (⎕NL); Headers only; Locals only

Container: [✓ Named namespaces]; Unnamed namespaces; GUI Instances

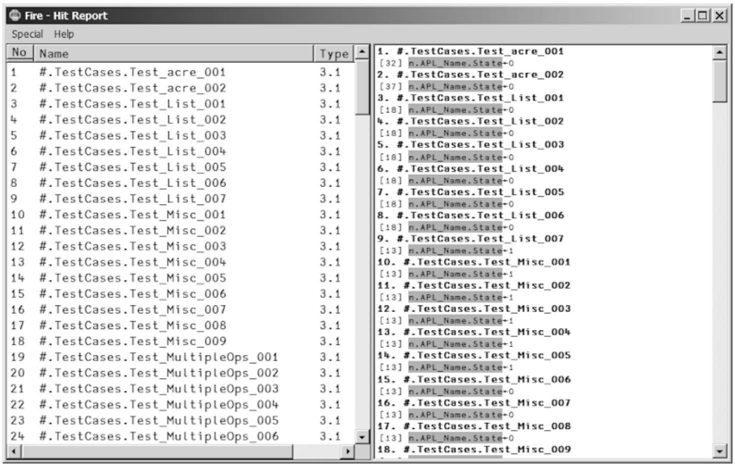
Recursive: [✓ Full]; None; Just one level

Negate search; Search hit list; [✓ Tell acre]; Keep Fire on top

Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
Test_acre_001	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_acre_002	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_List_001	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_List_002	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_List_003	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_List_004	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_List_005	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_List_006	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_List_007	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_001	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_002	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46

160 hits found in 160 of 221 APL objects from 106 containers in 0.2 seconds processing time | 3.7.3 - 2014-08-11

“Report hits” confirms that we are on the right path, although it also shows that sometimes the “APL Name” check box is ticked and sometimes it isn’t, so we need to carry out the work in two steps:



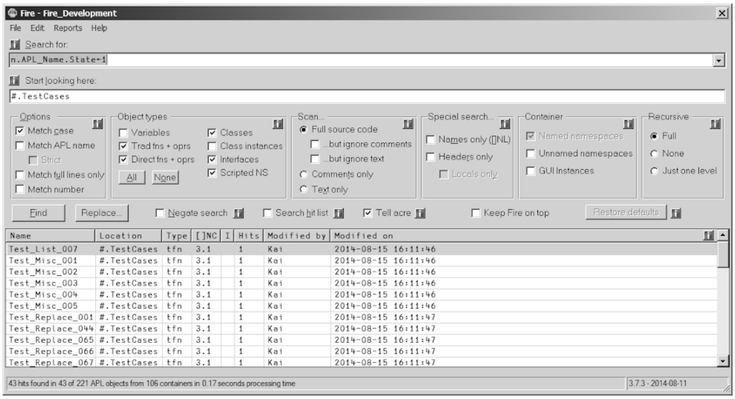
No	Name	Type
1	#.TestCases.Test_acre_001	3.1
2	#.TestCases.Test_acre_002	3.1
3	#.TestCases.Test_List_001	3.1
4	#.TestCases.Test_List_002	3.1
5	#.TestCases.Test_List_003	3.1
6	#.TestCases.Test_List_004	3.1
7	#.TestCases.Test_List_005	3.1
8	#.TestCases.Test_List_006	3.1
9	#.TestCases.Test_List_007	3.1
10	#.TestCases.Test_Misc_001	3.1
11	#.TestCases.Test_Misc_002	3.1
12	#.TestCases.Test_Misc_003	3.1
13	#.TestCases.Test_Misc_004	3.1
14	#.TestCases.Test_Misc_005	3.1
15	#.TestCases.Test_Misc_006	3.1
16	#.TestCases.Test_Misc_007	3.1
17	#.TestCases.Test_Misc_008	3.1
18	#.TestCases.Test_Misc_009	3.1
19	#.TestCases.Test_MultipleOps_001	3.1
20	#.TestCases.Test_MultipleOps_002	3.1
21	#.TestCases.Test_MultipleOps_003	3.1
22	#.TestCases.Test_MultipleOps_004	3.1
23	#.TestCases.Test_MultipleOps_005	3.1
24	#.TestCases.Test_MultipleOps_006	3.1

```
1. #.TestCases.Test_acre_001
[32] n.APL_Name.State←0
2. #.TestCases.Test_acre_002
[37] n.APL_Name.State←0
3. #.TestCases.Test_List_001
[18] n.APL_Name.State←0
4. #.TestCases.Test_List_002
[18] n.APL_Name.State←0
5. #.TestCases.Test_List_003
[18] n.APL_Name.State←0
6. #.TestCases.Test_List_004
[18] n.APL_Name.State←0
7. #.TestCases.Test_List_005
[18] n.APL_Name.State←0
8. #.TestCases.Test_List_006
[18] n.APL_Name.State←0
9. #.TestCases.Test_List_007
[13] n.APL_Name.State←1
10. #.TestCases.Test_Misc_001
[13] n.APL_Name.State←1
11. #.TestCases.Test_Misc_002
[13] n.APL_Name.State←1
12. #.TestCases.Test_Misc_003
[13] n.APL_Name.State←1
13. #.TestCases.Test_Misc_004
[13] n.APL_Name.State←1
14. #.TestCases.Test_Misc_005
[13] n.APL_Name.State←1
15. #.TestCases.Test_Misc_006
[13] n.APL_Name.State←0
16. #.TestCases.Test_Misc_007
[13] n.APL_Name.State←0
17. #.TestCases.Test_Misc_008
[13] n.APL_Name.State←0
18. #.TestCases.Test_Misc_009
```

Next we repeat the search for

```
n.APL_Name.State←1
```

simply because that will result in lesser hits. Indeed we get just 43 hits:



Search for: n.APL_Name.State←1

Start looking here: #.TestCases

Options: [✓ Match case] Match APL name; Strict; Match full lines only; Match number

Object types: Variables; [✓ Trad fns + oprs]; [✓ Direct fns + oprs]; [✓ Classes]; Class instances; [✓ Interfaces]; [✓ Scripted NS]

Scan...: [✓ Full source code]; ...but ignore comments; ...but ignore text; Comments only; Text only

Special search...: Names only (⎕NL); Headers only; Locals only

Container: [✓ Named namespaces]; Unnamed namespaces; GUI Instances

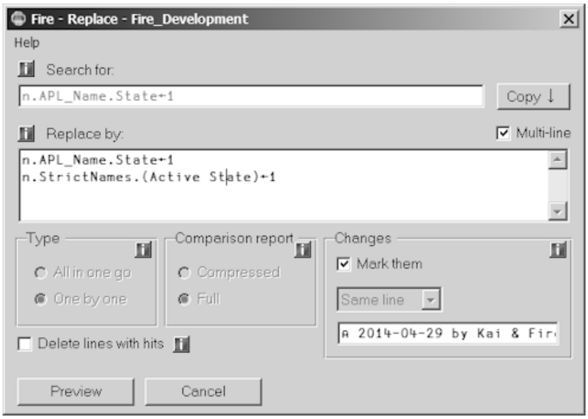
Recursive: [✓ Full]; None; Just one level

Negate search; Search hit list; [✓ Tell acre]; Keep Fire on top

Name	Location	Type	[]NC	I	Hits	Modified by	Modified on
Test_List_007	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_001	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_002	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_003	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_004	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Misc_005	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:46
Test_Replace_001	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_Replace_044	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_Replace_065	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_Replace_066	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47
Test_Replace_067	#.TestCases	tfn	3.1		1	Kai	2014-08-15 16:11:47

43 hits found in 43 of 221 APL objects from 106 containers in 0.17 seconds processing time

3.7.3 - 2014-08-11



Search for:

```
n.APL_Name.State←1
```

Replace by: [✓ Multi-line]

```
n.APL_Name.State←1
n.StrictNames.(Active State)←1
```

Type: All in one go [✓ One by one]

Comparison report: Compressed [✓ Full]

☐ Delete lines with hits

Changes: [✓ Mark them] — Same line

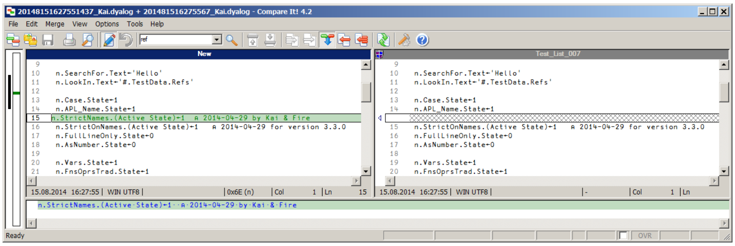
```
⍝ 2014-04-29 by Kai & Fir
```

In “Replace” we first tick the “Multi-line” check box and then repeat the search string followed by a second line with the statement (line) we want to add:

Note that the “Mark them” check box is ticked and the combo box underneath carries “Same line”. That means that the added line will be marked, by default by the string:

```
⍝ 2014-04-29 by {⎕AN} & Fire
```

After a click on “Preview” we can check the results for one function after the other:


20148151627551437_Kai.dyalog + 201481516275567_Kai.dyalog - Compare It! 4.2
File Edit Merge View Options Tools Help
ref
Ready
OVR

New

```

 n.SearchFor.Text←'Hello'
 n.LookIn.Text←'#.TestData.Refs'

 n.Case.State←1
 n.APL_Name.State←1
 n.StrictNames.(Active State)←1  ⍝ 2014-04-29 by Kai & Fire
 n.StrictOnNames.(Active State)←1   ⍝ 2014-04-29 for version 3.3.0
 n.FullLineOnly.State←0
 n.AsNumber.State←0

 n.Vars.State←1
 n.FnsOprsTrad.State←1
```

15.08.2014 16:27:55 | WIN UTF8 | 0x6E (n) | Col 1 | Ln 15

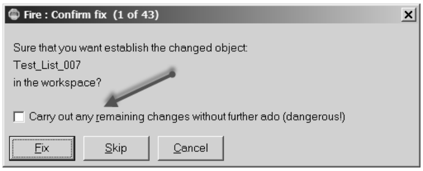
Test_List_007

```

 n.SearchFor.Text←'Hello'
 n.LookIn.Text←'#.TestData.Refs'

 n.Case.State←1
 n.APL_Name.State←1

 n.StrictOnNames.(Active State)←1   ⍝ 2014-04-29 for version 3.3.0
 n.FullLineOnly.State←0
 n.AsNumber.State←0

 n.Vars.State←1
 n.FnsOprsTrad.State←1
```

15.08.2014 16:27:55 | WIN UTF8 | - | Col 1 | Ln

```
n.StrictNames.(Active·State)←1··⍝·2014-04-29·by·Kai·&·Fire
```

For a large number of changes this can turn out to be annoying. However, if you feel confident that everything will just be fine you can tick the "Carry out any remaining changes without further ado" check box and you are done:



Sure that you want establish the changed object:
Test_List_007
in the workspace?

☐ Carry out any remaining changes without further ado (dangerous!)

Of course that is dangerous, so keep a backup! In a separate step we can repeat this for

```
    n.APL_Name.State←0
```

and add

```
n.StrictOnNames.(Active State)←0
```

We have changed more than 100 functions within a matter of minutes.

Note that the “Strict” option has a different meaning when you search for # or ##. Let’s look at an example: in a WS we have just 4 objects, two class scripts (APLTreeUtils and WinFile, both members of the APLTree project[2]) and two functions located in an ordinary namespace #.MyApp:

```
∇ Run;A
[1]    A←#.APLTreeUtils
[2]    ∆WSID←A.Uppercase ⎕WSID
[3]    #.WinFile.PolishCurrentDir
[4]    WorkHorse ⍬
     ∇
∇ {r}←WorkHorse.WorkHorse dummy
[1]    r←⍬
[2]    ⎕←##.WinFile.PWD
     ∇
```

The coding of the functions does not make too much sense but they are good enough to highlight the topic. Note that Run carries references to # and ## while Workhorse carries only a reference to ## .

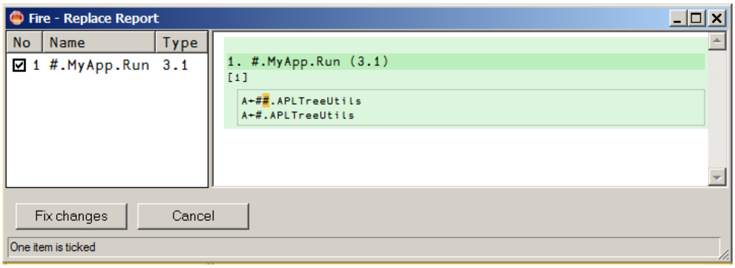
Now let’s assume that we want to use these two functions in a user command by copying the code over to a user command script. Although with Workhorse that would work without further ado, Run might or might not run because it relies in WinFile and APLTreeUtils to live in #. If they are not to be found in # the user command generates a VALUE ERROR.

Of course a user command should not make such assumptions: instead it should refer to the parent for utilities and stuff. In practice however this scenario might well occur because the application might have been written without considering it to run as a user command one day. It is also quite easy to try to address an object with ##. but occasionally to address it as #. anyway. Worse, running the application in normal mode would not reveal such problems because both ways would work just fine.

In short, to convert an application into a user command we need to find all references to #. and convert them to ##. while all references to ## can be left alone. The problem is that searching for # or #. would not help because it would also find ## or ##. .

“Name” and “Strict” to the rescue: with both options ticked Fire will perform some sort of special search that deals with the problem. In our case Fire would ignore

Workhorse because it does not contain any reference to # . The function Run would change; here the change preview:

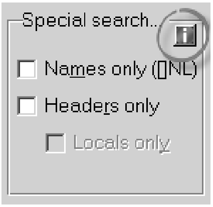


No	Name	Type
☑ 1	#.MyApp.Run	3.1

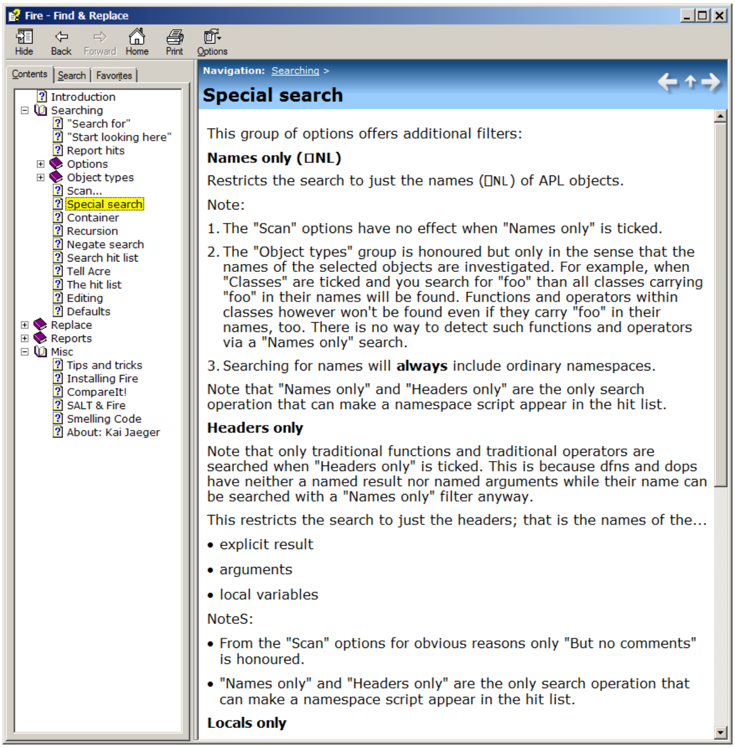
```
1. #.MyApp.Run (3.1)
[1]
  A←##.APLTreeUtils
  A←#.APLTreeUtils
```

One item is ticked

Finally I want to draw your attention to the boxes displaying an "i" for information: these are links to Fire's help file. For example, clicking at this box:



brings up the help page that is associated with that very topic:



Special search

This group of options offers additional filters:

Names only (⎕NL)

Restricts the search to just the names (⎕NL) of APL objects.

Note:

1. The "Scan" options have no effect when "Names only" is ticked.
2. The "Object types" group is honoured but only in the sense that the names of the selected objects are investigated. For example, when "Classes" are ticked and you search for "foo" than all classes carrying "foo" in their names will be found. Functions and operators within classes however won't be found even if they carry "foo" in their names, too. There is no way to detect such functions and operators via a "Names only" search.
3. Searching for names will **always** include ordinary namespaces.

Note that "Names only" and "Headers only" are the only search operation that can make a namespace script appear in the hit list.

Headers only

Note that only traditional functions and traditional operators are searched when "Headers only" is ticked. This is because dfns and dops have neither a named result nor named arguments while their name can be searched with a "Names only" filter anyway.

This restricts the search to just the headers; that is the names of the...

- explicit result
- arguments
- local variables

NoteS:

- From the "Scan" options for obvious reasons only "But no comments" is honoured.
- "Names only" and "Headers only" are the only search operation that can make a namespace script appear in the hit list.

Locals only

Over time you might find all these information boxes distracting. No problem, unticking the menu command Help > Show info buttons make them disappear:

Converted into a Word document Fire's help file comprises 33 pages. Scanning a workspace and trying to change selected objects can be a surprisingly complex business.

Although this article describes just a few of the features of Fire I hope you agree that these already proved how valuable Fire can be when dealing with legacy code. However, Fire offers many features which make it also a useful tool when dealing with non-legacy code as well.

Fire is part of the APLTree project[2,3] and as such sort of Open Source[4]: you can use it freely, you can contribute to the code basis or even take a copy and modify it for your own purposes and do whatever you like with that code.

Fire has its own page on the APL wiki[5] and can be downloaded from there[6].

References

1. acre's home page on the APL wiki: http://aplwiki.com/acre
2. "Sharing code: the APLTree project" by Kai Jaeger, Vector 25-3, http://archive.vector.org.uk/art10500730
3. The APLTree project on the wiki: http://aplwiki.com/CategoryAplTree
4. The APLTree project license: http://aplwiki.com/AplTreeLicensing
5. Fire's home page on the APL wiki: http://aplwiki.com/Fire
6. The APLTree download page: http://download.aplwiki.com/apltree/

Writing a simple Japanese dentist office system in APL2

Kyosuke Saigusa, APL Consultants of Japan Ltd.

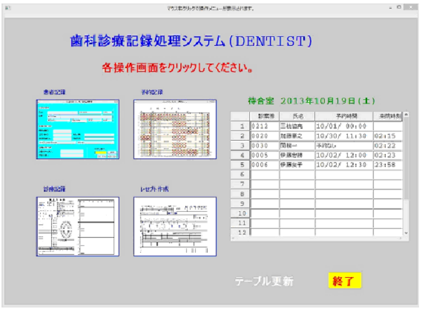
Introduction

We have written several office systems for small businesses mainly to explore the potentials of APL2. Some of them have been used daily for ten years or more. They all run under IBM workstation APL2 runtime. Users do not know what language is being used, nor need to. I would like to introduce my current work which aims to address a rather wider range of users for the first time. It is an office system for the dentists in general working under the Japanese health insurance setup. It is said that we have more dentist offices than the number of convenience stores in this country and many of them cannot afford to use expensive commercial systems.

System Outline

It consists of the following four modules as represented in the figure shown in Fig.1.

1. Client information(Upper left box on the left)
2. Reservation information(Upper right box on the left)
3. CARTE information(Lower left box on the left)
4. RECEPT* information(Lower right box on the left)
   *note: document to submit to the national insurance union for insurance claims.


マウス右クリックで操作メニューが表示されます。
歯科診療記録処理システム(DENTIST)
各操作画面をクリックしてください。
患者記録
予約記録
診療記録
レセプト作成
テーブル更新
終了

待合室 2013年10月19日(土)

	診察券	氏名	予約時間	来院時刻
1	0212	三枝協亮	10/01/ 00:00	
2	0020	加藤崇之	10/30/ 11:30	02:15
3	0030	関根一	予約なし	02:22
4	0005	伊藤忠博	10/02/ 12:00	02:23
5	0006	伊藤良子	10/02/ 12:30	23:58
6				
7				
8				
9				
10				
11				
12				

Fig. 1: Startup screen to enter the four modules

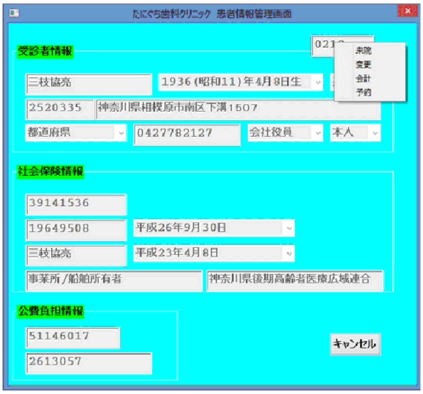
In the screen, the left side four boxes represent the referenced modules and the right side table shows a list of clients in the waiting room.

Upper two boxes are mainly used by clerks at reception desk, and the lower boxes are for dentists to use. These can be concealed as required.

The system operates on Microsoft Windows XP and later versions. It is installed at respective user sites but linked via VPN to our centre for maintenance.

Client information module

The following screen records client personal information regarding the means of contact, social insurance and government subsidies. This is used to record the time the client has arrived at the dentist office, to alter the personal information recorded, to issue bills and prescriptions after treatment and to link to the reservation module by way of a popup menu.



021
三枝協亮
1936 (昭和11) 年4月8日生
2520335
神奈川県相模原市南区下溝1507
都道府県
0427782127
会社役員
本人
39141536
19649508
平成26年9月30日
三枝協亮
平成23年4月8日
事業所/船舶所有者
神奈川県後期高齢者医療広域連合
51146017
2613057

Fig.2: Client information screen

For a new client, a unique client number is automatically assigned. Initial information and later alterations can be entered manually or for some items by way of menus. Automatic translation of national post code is also implemented.

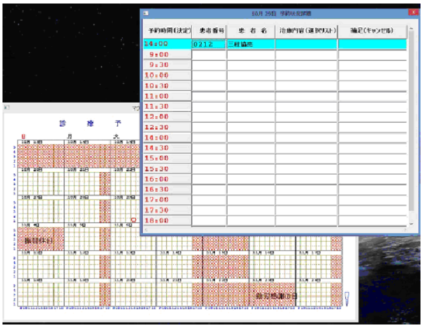
By way of mutually exclusive control of the AP 211 of the IBM APL2 interpreter, multiple screens can be operated concurrently by clerks for different clients. The recorded data is accessed for read and write by different modules simultaneously as well. The data takes the form of APL2 general arrays and hence it is flexible, powerful and easy to handle by APL2.

Reservation information module

This module can be started via the link from the client information screen to enter or to alter the reservation date and time. As any desired date box is clicked, the reservation status list is shown for the date. As you click the time zone on the left of the list, the set of client name and number is automatically entered and upon confirmation, it is recorded in the database. Aside from the client number and the name, you can also specify the kind of treatment and expected time required to finish the treatment as an option (probably by dentist).

When the reservation screen is entered directly from the main screen, it only works as a reference and the contents cannot be altered.

The calendar is shown by weeks and calculated by APL2 to begin with the current week. The national and public holidays are centrally entered, based on the public web information by the central maintenance via remote access. Scheduled operating dates and off-days/time information can be entered and maintained at individual dentist offices. The calendar can be scrolled back and forth between the current week and any pre-specified future week. Each hour on the calendar is made to accommodate up to five clients, divided optionally by fifteen or thirty minutes.



予約時間(決定)	患者番号	患者名	治療内容(選択リスト)	備忘(キャンセル)
14:00	0212	三枝龍亮		
9:00				
9:30				
10:00				
10:30				
11:00				
11:30				
12:00				
12:30				
14:00				
14:30				
15:00				
15:30				
16:00				
16:30				
17:00				
17:30				
18:00				

Fig 3: Reservation information screen

Dentists are entitled to charge additional fees from the clients and the insurance union, if the treatment is done at an irregular time.

CARTE information module

Each client (patient)'s ailment and treatment information is recorded in the same format as the officially designated form of the ministry of labour and welfare for compatibility with the manual systems. This is a single sheet form which records ailment information on the first (front) page and treatment information in the second (rear) page. These two pages are shown side by side on the screen, so that the related information can be viewed at a glance and new data entered with minimum errors. The records can be viewed historically by scrolling. The advantage of this approach is that it can eliminate the need to store an increasing volume of physical paper documents in the office and to search for the appropriate page of the documents in a short time. The target document page is displayed on the screen any time for reference and for hard copy printout if required. The documents are dually recorded in the center at specified intervals for back-ups and recovery of user data in case accidents occur. The dentist can enter the name of the ailment and code and the ailing teeth, if any, by mouse click on the illustration of the teeth in the first page. Treatment is entered from the multiple selection popup menu in tree structure, together with the date of treatment and the fees in points with appropriate calculations.

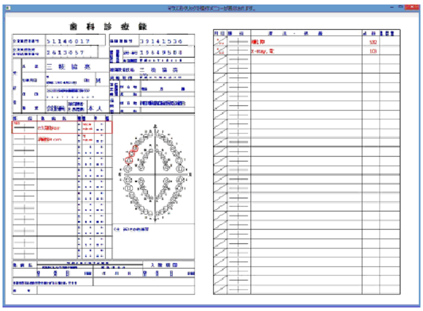


Fig.4-1: CARTE information screen

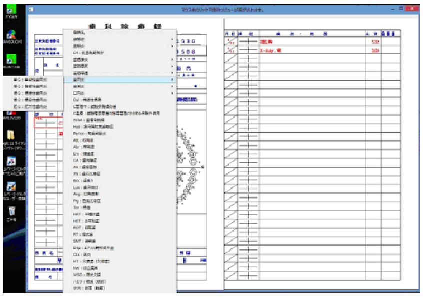


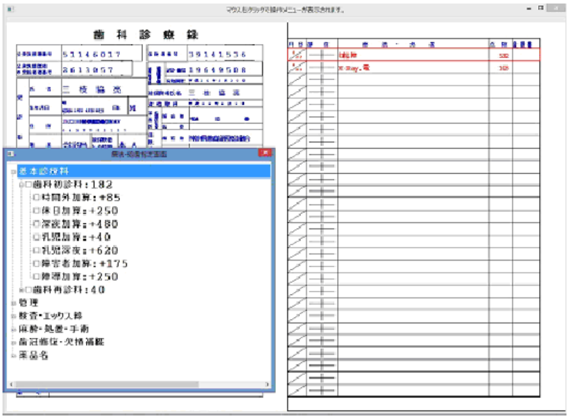
Fig.4-2: Popup menu to enter ailment

Patient's personal information is entered as you click the client number from the list of clients in the waiting room, which is shown as you click the blank space

The registered ailment names and codes are shown in the popup menu by category in tree structure to select from to avoid errors in entry.

The names of the ailment are recorded in historical order in the next line without limit. The lines scroll automatically as the space runs out for new lines or manually when the past records are referenced. The treatment information on the right page relating to the selected ailment line framed in red on the left page also changes automatically.

Both pages permit alterations and deletion by the discretion of the dentists. The treatment information is entered or altered by selecting the target line on the right page. Each line records five items: date of treatment, treated teeth, treatment, the fees in points and other information. Today's date is automatically recorded but if this column is clicked, a popup calendar allows the alteration of the date. When the treated teeth column is clicked, a popup menu appears to select the teeth to treat from the list of ailing teeth recorded on the left page for this ailment. When the treatment column is clicked, a popup menu appears, showing the possible kind of treatment, care or medicine in tree structure for multiple selections to replace the previously selected list of treatments for the same day if any. De-selection works this way. Points are automatically calculated according to the rules set by the national dentist union and insurance union.



歯　科　診　療　録

5 1 1 4 6 0 1 7

3 9 1 4 1 5 5 6

2 6 1 3 0 5 7

1 9 6 4 9 5 0 8

基本診療料
- 歯科初診料：182
  - 時間外加算：+85
  - 休日加算：+250
  - 深夜加算：+480
  - 乳児加算：+40
  - 乳児深夜：+620
  - 障害者加算：+175
  - 障哪加算：+250
- 歯科再診料：40

管理

検査・エックス線

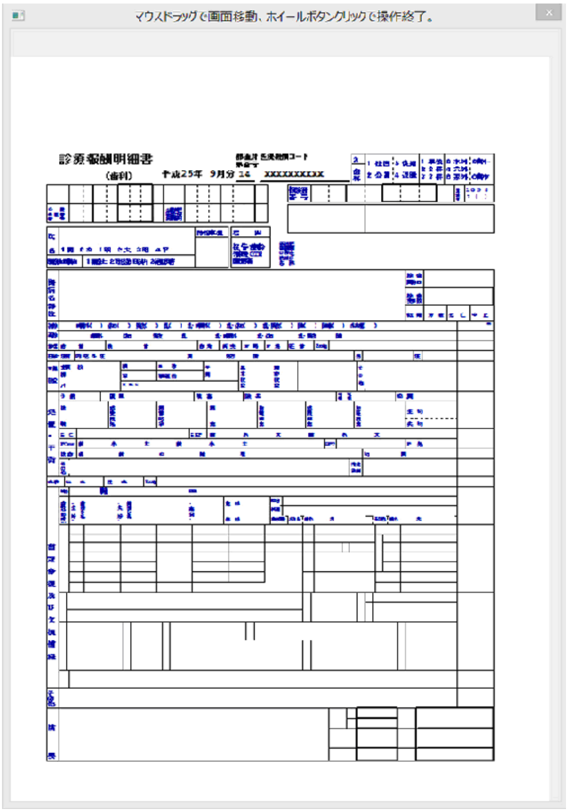
麻酔・処置・手術

歯冠修復・欠損補綴

薬品名

Fig.4-3: Popup menu to enter treatment and care

RECEPT information module (currently being built)



マウスドラッグで画面移動、ホイールボタンクリックで操作終了。

診療報酬明細書

（歯科）

平成25年　9月分

XXXXXXXXXX

Fig 5: RECEPT information screen

RECEPT is an official document dentists are required to submit to the national insurance union for insurance claims on monthly bases for each client treated during the past month. Theoretically, RECEPT is created automatically from CARTE, if the information there is all correct. In fact this is where dentist offices spend much time to eliminate entry errors to avoid rejection and repeated re-submission and penalties. Therefore, this document can be viewed for any errors with references to the details recorded in the CARTE before submission to the insurance union by hard copy or optionally via internet.

The form of the RECEPT is printed in very small characters , therefore when it is viewed on the screen, the system can augment the displayed size of the copy and permits scrolling of the drawing by mouse drags to view the entire copy in details and also for the last-minute corrections.

Why APL2 is suitable to write small systems

Through my observation of the development of APL2, I understood that IBM has used a tremendous amount of energy and brains to make it a practical tool to develop computer applications to the present day level. To us in the Far East, symbolic representations of the primitive functions should be the best choice to convey the precise meanings at a glance like Chinese characters, which we have been using over centuries. Aside from the philosophy of the language itself, I must say that existence of the auxiliary and associated processors have played indescribably important rolls to make interface programming very easy and simple. Hence it is easy to introduce internet access for the reservation by way of PC or smart phones and the like with the use of QR code or regular bar code. Here APL2's AP 119 socket interface and HTML helps.

Writing programs in APL is quite private in the sense that it depends much on personal interpretations of the effects of limitless combinations of simple functions, analogous to the game of GO. This aspect of the language makes it difficult to produce extremely large application systems by groups of programmers without failure.

On the other hand, small systems require only a small team of application experts and a single well-trained and qualified APL programmer.

Small systems can grow into more comprehensive systems in time as I aim to make this dental system eventually to cover the entire segment of professional medical doctor's office systems in town because they all run under the same public insurance system.

J

J-ottings 57 Heavens above!

by Norman Thomson (ndt4@btinternet.com)

J-ottings 56 decribed the development of a verb `rotate` which, given a left argument (axis, angle) delivers the result of performing a 3-D rotation of a point whose coordinates form the right argument. *axis* is defined by the three coordinates of any point on it other than the origin, so the left argument is a 4-item list. Here, repeated from J-ottings 56, is the verb `rotate` along with its subverbs :

```
   rotate=.] - (m1 * rmdata) + m2 * }:@[ xp ]
       rmdata=.rm@dircos@(}:@[) +/ .* ]       NB. rotn matrix*data
           rm =.(id - */~)@dircos             NB. rotation matrix
               id=.=@i.@#                     NB. identity matrix
           dircos=.% %:@(+/@:*:)              NB. direction cosines
       xp=.4 : ‘1 _1 1*det each<”(2) 1+\.(dircos x),.y’
                                          NB. normalised cross product
           det=.-/ .*                         NB. determinant
           each=.&>
       m1=.-.@(2&o.@({:@[))                   NB. (1-cos angle)
       m2=.1&o.@({:@[)                        NB. sin angle
```

David Edwards has pointed out that the verb xp the above delivers the normalised cross-product; if a conventional cross-product is required as part of another application, magnitude must be taken into account by e.g.

```
    mag=.%:@:(+/)@:*:                         NB. magnitude
    xprod =.4 : ‘(mag x)*x xp y’              NB. cross-product
    0 4 5 xprod 2 1 3
7 10 _8
```

In this article `rotate` is used to perform some basic calculations in astronomy. Trig ratios as well as conversions to and from radians and degrees are frequently required, and so it is convenient to define in advance a few utility verbs :

```
   dtor=.180%~o.                              NB. degrees to radians
   rtod=.dtor^:_1                             NB. radians to degrees

   sin=.1&o.@dtor                             NB. sine (angle in degs)
   cos=.2&o.@dtor                             NB. cosine (angle in degs)
   asin=.rtod@(_1&o.)                         NB. arcsine in degrees
   acos=.rtod@(_2&o.)                         NB. arccosine in degrees
```

Plotting star movements

This is a special case of 3-D rotation in which all data points in the heavens are identified by two rather than than three parameters. Astronomers measure star positions as observed from Earth in angular rather than Cartesian measure. Specifically the two angles used are **altitude A** which corresponds to celestial latitude, and **azimuth Z** which correponds to longitude in terrestrial measurement. The stars themselves lie on the surface of a sphere called the **celestial sphere** which is continuously rotating about the extended Earth axis and on which every star has a latitude and a longitude which are called respectively declination D and **right ascension ra**. Analogous to the Greenwich meridian on Earth the celestial sphere requires an arbitrary zero line or **celestial meridian** from which **ra** is measured. This is conventionally taken to be the first point in Aries, which is observable as the rightmost star in the constellation Cassiopea. Azimuth is often measured in sidereal hours, minutes and seconds rather than degrees; the significance of sidereal is that a sideral year is one day longer than a solar year, that is the fixed stars appear to rotate at a slightly slower speed than the sun, the difference being about four minutes per day. Stars rise in the east and set in the west, and so to an Earth-bound observer looking outwards to the Pole Star, the celestial sphere appears to rotate in an anticlockwise direction.

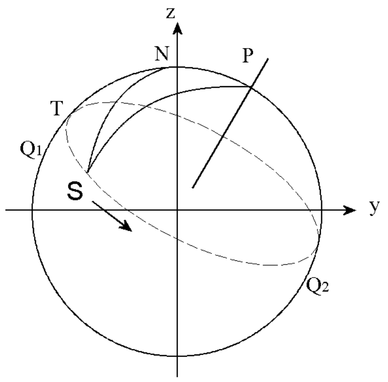
To convert star positions defined by A and Z into (x, y, z) coordinates assume the x-axis runs west to east, the y-axis south to north, and the z-axis upward. The plane x=0 is then a meridian on a fixed celestrial sphere from which Z is measured clockwise. (x, y, z) coordinates are then given by

$$x = \cos A \sin Z; \quad y = \cos A \cos Z; \quad z = \sin A$$

Inverting these formulae to convert from (x, y, z) coordinates to (A, Z) coordinates :

$$A = \sin^{-1} z \ ; \quad Z = \cos^{-1}\frac{y}{\sqrt{1-z^2}} \quad \text{or} \quad \sin^{-1}\frac{x}{\sqrt{1-z^2}}$$

Transits



A star is said to transit or culminate when it is at its highest point in the sky when seen by an observer on Earth, at which time x=0. The diagram below shows a circle of celestial longitude through the transit point T of a star S as it traverses its daily circuit shown as a dotted line :

N is the zenith, P is the Pole Star. As S moves on from transit, the curved triangle NPS comes out of the page towards the reader.

Side SP is `90°-d` where `d` is S's declination.
Side NP is `90°-l` where `l` is the observer's latitude.

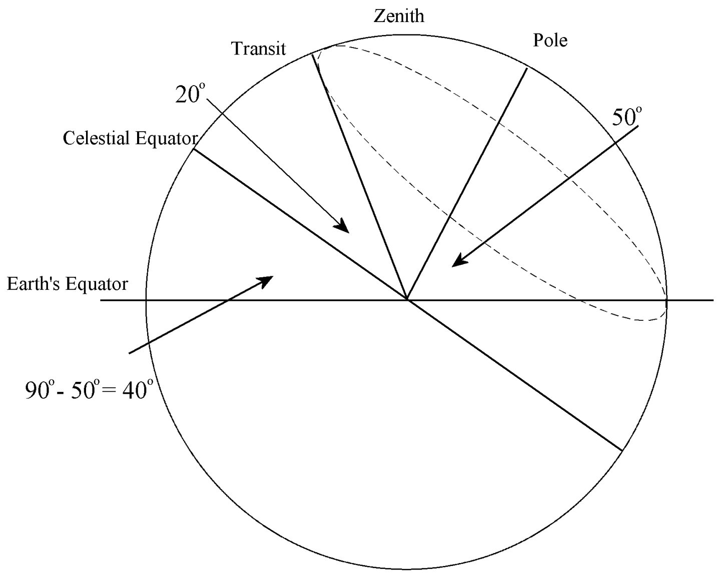
The lengths of both of these remain fixed as S progresses.

Q1 and Q2 are points on the celestial equator.

Quantities which change as the star S proceeds along its course are :

> side NS = 90° – A where A is the altitude;
>
> angle NPS = the angle of rotation about the polar axis, known as the *hour angle*;
>
> angle TNS = the terrestrial azimuth Z based on the transit plane as zero meridian.

The diagram below shows the same cross-section of the celestial sphere through the plane x=0 for a specific star with declination 20° observed from a latitude of 50° North. 40 + 20 = 60, and so the (x,y,z) transit coordinates are (0,cos (180-60)°, sin (180-60)°), that is (0,cos 120°, sin 120°).



This diagram can be generalised to show that the altitude at transit is (90°- l) + d provided that d < l as in the case of the star illustrated. This star transits south, that is to the left of the zenith and dips below the terrestrial horizon for at least part of its circuit. If d > l a star is circumpolar and transits north. Here the altitude at transit is (90° + l) – d, or combining the two cases, the altitude of every star at transit is

```
    90° - abs(l - d) .
```

Plotting star positions

Unlike locations on a geographical map, a star's position has time as a parameter, which can be either local time – where was a star six hours ago? – or time by year – where was it three months ago at the current time of day? The star sphere appears to Earth observers to revolve from east to west around the pole, completing a revolution in a sidereal day which is shorter by 1/365th of a day (that is approximately four minutes) than the solar day. Thus the position of a star six hours ago (¼ of a day) is the same as its position three months ago (¼ of a year).

For example, consider the star illustrated above with declination 20°, and ask what are the (x,y,z) coordinates of its position six hours earlier, that is when the hour angle is -90° . The list '0, cos x, sin x' is required sufficiently frequently in defining axes and points that it is convenient to have a verb

```
    cs=.0,cos,sin           NB. e.g. rotn. axis in y-z plane
    ((cs 50),dtor _90)rotate cs 120
0.939693 0.219846 0.262003
```

This result can be confirmed by spherical trigonometry applied to triangle NPS (see next section).

The next step is to make the hour angle a parameter (clockwise 90° = anti-clockwise -90°):

```
    v=.monad :'((cs 50),-dtor y)rotate cs 120';
```

and plot values as this moves towards transit at 10° intervals :

```
    v each 90 80 70 60 50 40 30 20 10 0
0.94    0.22    0.262
0.925   0.0948  0.367
0.883  _0.0264  0.469
0.814  _0.14    0.564
0.72   _0.243   0.65
0.604  _0.332   0.725
0.47   _0.404   0.785
0.321  _0.457   0.83
0.163  _0.489   0.857
0      _0.5     0.866
```

More generally, it is useful to convert time to angular measure with 24 hours being equivalent to a complete rotation, which suggests three more utility verbs :

```
   ttor=.o.&(%&43200@(60&#.))    NB. time (hms) to radians
   ttor 12 0 0                   NB. check 12 hrs = pi rads
3.14159
```

```
   dtot=.60 60 60&#:@(*&240)     NB. deg to time (hms)
   dtot 180                      NB. check 180 deg.= 12 hours
12 0 0
```

```
   atod=.%&3600@(60&#.@(3&{.))   NB. angle(deg,min,sec) to degrees
   atod 49 15                    NB. check 49o 15' 0" = 49.25
49.25
```

The cooordinates of the above star 15 and a half minutes after transit, are given by

```
    ((cs 50),ttor 0 15 30)rotate cs 120
_0.0635044 _0.498354 0. 864645
```

that is a little bit to the west, a shade less south and a bit lower, all as expected.

Spherical Trigonometry

The cos formula for a spherical triangle ABC states that if a, b and c are sides measured in angles, and A, B and C are the angles between the sides with A opposite a, etc. then

```
    cos a = cos b cos c - sin b sin c cos A
```

Although the sides are nominally measured as angles they are nevertheless *lengths* – length being defined by the angle subtended at the centre of the sphere.

There are two forms of 'Pythagoras' theorem' in spherical trigonometry, viz.

```
    if cosA = 90° (n.b. do not confuse this A with A = altitude)
    if cosc = 90°
```

Applying this to the first diagram above, triangle NPS has sides SP=(90° – d), NS=(90° – A) and NP=(90° – l), and angle NPS is 180° – Z, so in general .

In the worked example angle P was chosen to be -90° and so the first 'Pythagorean' form applies, that is, for the star with declination 20° observed at 50° latitude, the altitude six hours earlier is given by :

```
    sin A = sin 50° sin 20°
```

The values of sin A and cos A are thus given by,

```
    ]sinA=.*/sin 50 20          NB. sin Altitude = z coordinate
0.262003
    ]cosA=.cos asin sinA        NB. cos Altitude
0.965067
```

The sine formula in spherical trigonometry states that

$$\frac{\sin\ a}{\sin\ A} = \frac{\sin\ b}{\sin\ B} = \frac{\sin\ c}{\sin\ C}$$

Apply this to triangle TNS. The dotted line is a circle of latitude and so angle NTS = 90°. If the hour angle P is 90° the side it subtends at the celestial equator is sin 90° = 1, hence the dotted line TS at declination 20° has length cos 20°.

$$\text{Thus}\ \frac{\cos\ 20^0}{\sin\ Z} = \frac{\cos\ A}{1}$$

```
   ]Z=.asin(cos 20)%cosA        NB. azimuth
76.8322
```

which enables the x and y coordinates to be found using formulae given earlier :

```
   cosA*(sin,cos)Z              NB. x,y coords
0.939693 0.219846
```

The case of the sun

Unlike other stars whose declination is constant, the sun's declination varies in the course of a year from -23.5° to +23.5° and back again. At sunrise and sunset the sun's altitude is zero, so side NS of triangle NPS is 90° and now the second 'Pythagorean' form applies to give

$$\sin d = \cos l \cos Z$$

at these times.

From this

$$\cos Z = \frac{\sin d}{\cos l}$$

Then using the sin formula,

$$\sin P = \frac{\sin Z}{\cos d}$$

where P is the hour angle.

Consider London (latitude of 51° 30′) on the 21st December when the sun's declination is -23° 30′, and its altitude at noon, that is at transit, is (90° - 51° 30′) - 23° 30′ = 15°00.

```
    ]Z=.acos(sin 23.5)% cos atod 51 30    NB. azimuth
50.17
    ]P=.dtot asin(sin Z)%cos 23.5         NB. time to noon
3 47 27.2637
```

that is, in midwinter the sun is above the horizon for about 100÷360 of the day or 7½ hours.

Now use `rotate` to spin the sun from noon for 3 hrs 47 minutes and 27.26 seconds:

```
    lat=.atod 51 30
    dec=.atod -23 30
    tim=.3 47 27.26
    alt=.90-|lat-dec

   ((cs lat),ttor tim)rotate cs 180-alt
_0.7679 _0.6405 1.278e_7
```

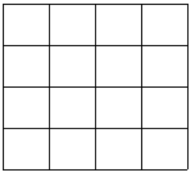
As expected the sun is south and west at altitude zero.

Both of the examples used above have involved special 'Pythagorean' cases which help to clarify principles from which more extended spherical trigonometry calculations can be made.

Squares, neighbours, probability, and J

John C. McInturff

The following problem is taken from the Mathcounts School Handbook for 2012-2013[1].



Two unit squares are chosen at random, without replacement, from the 4 x 4 grid shown. What is the probability that the squares do not share a side? Express your answer as a common fraction.

A student at the 6 through 8 grade level will, most likely, devise some method for counting the neighbors, and divide this number by the total number of possibilities.

The purpose here is to illustrate two methods that apply to a square of 4 sides that can be easily communicated and executed on a computer using J, then illustrate a generalisation to n sides.

Method 1

This method, suggested by Colin A. Hedges, a high school mathematics teacher, involves classifying the square into 3 mutually exclusive categories: Execution is shown for `n=. 4`.

Corners , C	The chance of selecting a corner is `C=. 4%*:n.`
Sides, S	These are bordering squares but not corners. The chance of selecting such a square is `S=. 4*(n-2)% *:n.`
Interior ,I	These squares are landlocked squares. The chance of selecting an interior square is `I=. (*:(n-2))% *:n`

Let `Z=. C,S,I.`

```
    +/Z
1
```

It is seen that they sum to unity, as they should.

After selecting a cell, there are `k=. <: (*: n)` cells remaining. A corner square has 2 neighbors, a side square has 3 neighbors, and an interior square has 4 neighbors. Therefore the conditional probability of selecting one of these squares is `X=. 'c s i'=. 2 3 4 % k` . The conditional probability that the square has no neighbor is `Y=. -.X` . It is seen that `X+Y` sum to unity in each of the three cases:

```
   X+Y
1 1 1
```

The inner product, (`ip=. +/ . *`), of `Z` and `X`. gives the unconditional probability that two squares share a side. The inner product of `Z` and `Y` is the probability they do not. These two probabilities, `W`, sum to unity as they should, and are shown below expressed both as a decimal and as a common fraction.

```
   ]W=. (Z ip X),(Z ip Y)
0.2 0.8

   +/W
1


   x: W
1r5 4r5
```

The monadic verb, `Fo` , simplifies the above formulation. It produces the probability that a square shares a side for any square having n sides. The expression `( -. Fo n)` produces the complementary probability,e.g.

```
   Fo=. 4 % (* >:)
   Fo 4
0.2

   (Fo 4);(-.Fo 4)
┌───┬───┐
│0.2│0.8│
└───┴───┘

   Fo 2 3 4 5
0.666667 0.333333 0.2 0.133333
```

Method 2

This method is simple and straightforward and may be the method a student in grades 6 through 8 may use. It compares every square with every other square and counts the neighbors it encounters in the process. The following pattern emerges and conclusions soon become evident:

Each row except the last has the same pattern wherein each square, except the last, has 2 neighbors, a row neighbor to its immediate right, and a column neighbor below. The last square in a row only has a column neighbor. For the 4 x4 matrix given this pattern, p1 would be p1=. 2 2 2 1 (for a total of 7 neighbors).

The last row, being at the bottom has no neighbors below it, and the last square in the last row has no neighbors, only a single, row, neighbor. Therefore, this last row has the pattern, p2 .

```
p2=. 1 1 1 0.
```

The 4x4 matrix would then have the following neighbor pattern, p :

```
   ]p=. > p1;p1;p1;p2
2 2 2 1
2 2 2 1
2 2 2 1
1 1 1 0
```

The total number of neighbors, N, is easily seen to be (N=. +/,p) or 24.

The total number of possible neighbors, T, is n squares taken 2 at a time. Expressed in J, this is (T=. 2!n or 120). A student, not knowing J but knowing the formula for T could reduce T to (15 *16)%2 or 120.

I f W1 is the probability of having a neighbor, then (W1=. N%T) or 0.2. The probability of not having a neighbor is (W2=. -.W1) , or 0.8. It is seen that this result is the same as that produced using Method 1.

Generalization and Simplification

The above conclusions, which were applied to a matrix having n=4 sides, also apply to a matrix having n sides. The above pattern, p , can be re-expressed as a tally of neighboring squares.

```
   p=. (neighbors having p1 type patterns) + (neighbors having p2 type patterns)
   p=          2(n-1)(n-1) +1(n-1)         +                   1(n-1)

   p= 2(n-1)(n-1)+2(n-1)
   p= 2(n-1)(n-1+1)
   p= 2(n)(n-1)
      p=. 2*n*(n-1)
      p=. h n
```

Although the last two expressions produce the number of neighbors, and both are executable, the last expression is more simply expressed using the monadic verb, h , where

```
   h=. [: +: (* <:)
```

The expression (F n) is the probability of p where (F=. h%g) and (g=. 2!*:) , the total number of possible neighbors, or equivalently, n squares taken 2 at a time.

```
   F 2 3 4 5
0.666667 0.333333 0.2 0.133333
```

The expression `(-. F n)` is the probability of no neighbors.

```
   W -: (F n);(-. F n)
1
```

It is seen that Method 2 produces the same result as Method 1.

References

1. Mathcounts Foundation. “2012-2013 Mathcounts School Handbook. Contains 300 Creative Math Problems That Meet NCTM* Standards For Grades 6-8.” Alexandria, VA, USA. Mathcounts. p. 27.
   *National Council of Teachers of Mathematics

All integer partitions: J programs compared

by Howard A. Peelle (hapeelle@educ.umass.edu)

J programs to generate all partitions of an integer are presented and compared.

Whoever wants to go about generating all partitions
not only immerses himself in immense labor,
but also must take pains to keep fully attentive,
so as not to be grossly deceived.

-- Leonhard Euler,
De Partitione Numerorum (1750)

Introduction

Historically, there has been great interest in partitions, especially computing the number of partitions of an integer. Relatively recently, algorithms and programs for generating all integer partitions have appeared but have not been expressly compared in a common language. Here, a dozen or so J programs are presented developmentally and then compared by speed, space, and spread.

Note: Programs are organized generally in chronological order by author of algorithm, with alternative coding to contrast efficiency, but with minimum explanation. If you are still learning J, please consult the JSoftware website [1] and read appropriate tutorials, notably [7] and [8]. If you are only interested in the best programs, skip to Comparisons and see Appendix for their final definitions.

```
     6
    1 5
    2 4
    3 3
   1 1 4
   1 2 3
   2 2 2
  1 1 1 3
  1 1 2 2
 1 1 1 1 2
1 1 1 1 1 1
```

A partition of an integer n is represented here as a list of parts: positive integers whose sum is n. All integer partitions of n are all the distinct partitions of n with parts in ascending (non-decreasing) or descending (non-increasing) order.

Partitions may be listed in any order but usually are in ascending or descending base value. For example, all partitions of 6 in ascending order, with ascending parts:

Note: The following utility names will be used often throughout, but their definitions will not be repeated.

```
    ELSE =: `
    WHEN =: @.
    EACH =: &.>
```

Skiena's Algorithm

The first program is a simplified coding of Steven Skiena's algorithm [2] using double recursion to produce partitions of input integer n beginning with input largest part p. It joins two arrays vertically: The top is the result of p joined horizontally onto partitions of n - p with the new largest part being the smaller of n - p and p; the bottom is the partitions of n with largest part p - 1. In tacit J:

```
Skiena =: Parts ELSE Ones WHEN Under2        NB. (n) Skiena (p)
    Ones =: ,:@#
    Under2 =: 2 > ]
    Parts =: Top , Bottom
        Top =: ] ,. - Skiena - <. ]
        Bottom =: Skiena <:
```

For example, partitions of 5 beginning with 3 in a table (padded with 0s):

```
   5 Skiena 3
3 2 0 0 0
3 1 1 0 0
2 2 1 0 0
2 1 1 1 0
1 1 1 1 1
```

All partitions of 6 in descending order:

```
   6 Skiena 6                  NB. Skiena~ 6
6 0 0 0 0 0
5 1 0 0 0 0
4 2 0 0 0 0
4 1 1 0 0 0
3 3 0 0 0 0
3 2 1 0 0 0
3 1 1 1 0 0
2 2 2 0 0 0
2 2 1 1 0 0
2 1 1 1 1 0
1 1 1 1 1 1
```

This is the shortest program here, but it is inefficient in speed and space.

Knuth's Algorithms

Donald Knuth proffered two algorithms for generating partitions in [3].

Knuth

Knuth's algorithm P computes partitions successively, starting with n and ending with a list of 1s: "If a partition isn't all 1s, it ends with (x+1) followed by zero or more 1s, for some x≤1; therefore, the next smallest partition in lexicographic order is obtained by replacing the suffix (x+1)1…1 by x…xr for some appropriate remainder r≤x. The process is quite efficient if we keep track of the largest subscript q [of partition a] such that aq not equal 1…" [3 page 37]. Coded explicitly in J, the program below performs classic scalar processing to produce a table of all partitions in descending order:

```
Knuth =: 3 : 0                  NB. Knuth (n) for n>:1
n =. y
all =. i.0,n
label_P1.                       NB. Initialize
a =. n#0
m =.0         NB. origin 0
label_P2.                       NB. Store final part
a =. n m}a
q =. m - (n=1)
label_P3.                       NB. Visit a partition
all =. all,(m+1){.a
if. (q{a)~:2 do. goto_P5. end.
label_P4.                       NB. Change 2 to 1,1
a =. 1 q}a
q =. q-1
m =. m+1
a =. 1 m}a
goto_P3.
label_P5.                       NB. Decrease q{a
if. q<0 do. all return. end.
x =. (q{a)-1
a =. x q}a
n =. (m-q)+1
m =. q+1
label_P6.                       NB. Copy x
if. n<:x do. goto_P2. end.
a =. x m}a
m =. m+1
n =. n-x
goto_P6.
)
```

For example, all partitions of 6 is the same result as 6 Skiena 6:

```
   Knuth 6
6 0 0 0 0 0
5 1 0 0 0 0
4 2 0 0 0 0
4 1 1 0 0 0
3 3 0 0 0 0
3 2 1 0 0 0
3 1 1 1 0 0
2 2 2 0 0 0
2 2 1 1 0 0
```

```
2 1 1 1 1 0
1 1 1 1 1 1
```

The program can be shortened drastically, get the remainder neatly, and handle `n=0` as a bonus:

```
Knuthx =: 3 : 0    NB. Knuthx (n)
all =. ,:p =. ,y
while. 1 < {.p do.    i =. <: p i. 1
            x =. - <: i { p
            a =. i {. p
            s =. y - +/a
            p =. a , x +/\ s#1
            all =. all , p
end.
)
```

Look at intermediate steps during a loop of `Knuthx 10` to produce the next partition after `4 3 1 1 1`:

```
    y =. 10
    p =. 4 3 1 1 1
   ]i =. <: p i. 1
1
   ]x =. - <: i { p
_2
   ]a =. i {. p
4
   ]s =. y - +/a
6
   ]p =. a , x +/\ s#1
4 2 2 2
```

Unfortunately, `Knuthx` uses twice the space and is (increasingly) slower than `Knuth`.

A tacit translation is even slower and fatter in space:

```
Knutht =: Parts ^: While ^:_ @ N
    N =: ,:@,
    While =: 1 < {.@{:
    Parts =: , Next@Last
        Last =: {: -. 0:
        Next =: A , X +/\ S#1:
            A =: I {. ]
                 I =: <:@i. 1:
            X =: I -.@{ ]
            S =: ] -amp;(+/) A
```

Changing `Knuthx` to use `s=.x+(#p)-i` for the sum of a suffix and including a condition to change a 2 to 1 1 will improve its speed – especially for large `n` with growing percentage of 2s:

```
Knuth2 =: 3 : 0
all =.,:p=.,y
while. 1 < {.p do.    i =. <: p i. 1
```

```
            x =. i { p
            if. x=2
            do. p =. 1 i} p,1
            else.
            a =. i {. p
            x =. <:x
            s =. x + (#p) - i
            p =. a , x Next s
            end.
            all =. all , p
end.
)

    Next =: -@[ +/\ ] # 1:
```

Better yet, use a sub-program to group partitions by their leading part and append an array of all leading 2s separately, with a final row of 1s:

```
Knuth2s =: 3 : 0
all =. i.0,n=.y
while. n>1
   do.     nexts =. n Nexts y-n
    all =. all , n ,. nexts
    n =. n-1
  end.
all,1
)

Nexts =: 3 : 0
:
if. y<2 do. ,:,y return. end.
nexts =. ,:next =. x New y
while. 2 < {.next
   do.    i =. <: next i. 1
    x =. i { next
    if. x=2
    do. next =. 1 i} next,1
    else.
    a =. i {. next
    x =. <:x
    s =. x + (#next) - i
    next =. a , x New s
    end.
    nexts =. nexts , next
  end.    NB. next is 2…(1)
repeats =. >:i.#next-.1
twos =. Two ^:repeats next
nexts,twos
)

    Two =: }. , 1 1"_
```

This is faster and much slimmer, but longer.

Nevertheless, these revisions cannot compete with `Knuth` in speed or space.

Hindenburg

Knuth’s algorithm H (attributed to Hindenburg [4]) computes partitions of `n` with m parts -- that is, m-tuples that sum to n. “The basic idea is that colex order goes from one partition $a_1…a_m$ to the next by finding the smallest j such that a_j can be increased without changing $a_{j+1}…a_m$.” [3 page 38] The new partition will have j leading parts = a_j+1 and the same sum if $a_j<a_1-1$. Note that this algorithm does not work for 0 or 1 parts. Later, see program AP that produces m-tuples robustly.

Assuming n≥m≥2, code H in J:

```
H =: 3 : 0                           NB. (n) H (m)
:
n =. x              NB. n>:m
m =. y              NB. m>:2
ps =. i.0,m
label_H1.                            NB. Initialize
a =. m#0
a =. (1+n-m) 0} a   NB. origin 0
a =. 1 (}.i.m)} a
a =. a , _1
label_H2.                            NB. Visit partition
ps =. ps , (i.m){a
if. (1{a)>:(0{a)-1 do. goto_H4. end.
label_H3.                            NB. Tweak 0{a and 1{a
a =. ((0{a)-1) 0}a
a =. ((1{a)+1) 1}a
goto_H2.
label_H4.                            NB. Find j
j =. 2              NB. origin 0
s =. (0{a)+(1{a)-1
while. (j{a)>:(0{a)-1 do.  s =. s + j{a
                j =. j+1
end.
label_H5.                            NB. Increase j{a
if. j=m do. ps return. end.
x =. (j{a)+1
a =. x j} a
j =. j-1
label_H6.                            NB. Tweak (i.j){a
while. j>0 do. a =. x j}a
            s =. s-x
            j =. j-1
end.
a =. s 0} a
goto_H2.
)
```

For example, partitions of 6 with 2, 3, and 4 parts:

```
   6 H 2
5 1
4 2
3 3
   6 H 3
4 1 1
3 2 1
2 2 2
   6 H 4
3 1 1 1
2 2 1 1
```

By appending such results, `H` can be used in a supra-program to generate all partitions:

```
Hindenburg =: 3 : 0            NB. Hindenburg (n)
n =. y
m =. 2
all =. ,.n
while. m<:n do. all =. all , n H m
           m =. m+1
end.
all
)
```

Notice that the order of all partitions is not the same as in `Knuth 6`.

`Hindenburg` is increasingly slower than `Knuth` and uses about twice the space, so it will be omitted from further comparisons.

Hui’s Algorithm

part

Roger Hui presented a concise program for all partitions [5]:

```
pu =: ] <@:+"1 [ * </\"1@=@]
pext =: [: ~. [: /:~&.> ,&.> , ;@:(pu&.>)
part =: ,&.>`(pext/)@.(1:<#) @ ($&1)
```

An example of partitions of 6 in a list of 11 boxes with ascending parts:

```
   part 6
┌───────────┬─────────┬───────┬───────┬─────┬─────┬───┬─────┬───┬───┬─┐
│1 1 1 1 1 1│1 1 1 1 2│1 1 1 3│1 1 2 2│1 1 4│1 2 3│1 5│2 2 2│2 4│3 3│6│
└───────────┴─────────┴───────┴───────┴─────┴─────┴───┴─────┴───┴───┴─┘
```

To produce a numeric table (padded with 0s), open the list:

```
   >part 6
1 1 1 1 1 1
1 1 1 1 2 0
1 1 1 3 0 0
1 1 2 2 0 0
1 1 4 0 0 0
1 2 3 0 0 0
1 5 0 0 0 0
2 2 2 0 0 0
2 4 0 0 0 0
3 3 0 0 0 0
6 0 0 0 0 0
```

With or without 0s, this program is woefully slow and obese in space. It cannot compete here.

Boss's Algorithm

Boss

R. E. Boss developed an efficient algorithm [6] that sparked interest, as well as some revisions. His program computes all partitions with descending parts:

```
init =: (<@<@i.@(1 0"_)) ,~ <"0@(] , (] (- <. >:@]) i.)@<:)
pps1 =: >:@i.@[ <@;@:(([ ,. (>: {."1) # ])&.>) {.
exit =: >@{.@>
Boss =: [: exit [: (],~ pps1)&.>/ init
```

Example:

```
   Boss 6
1 1 1 1 1 1
2 1 1 1 1 0
2 2 1 1 0 0
2 2 2 0 0 0
3 1 1 1 0 0
3 2 1 0 0 0
3 3 0 0 0 0
4 1 1 0 0 0
4 2 0 0 0 0
5 1 0 0 0 0
6 0 0 0 0 0
```

This program is very fast but fat.

AP0

Boss's program can be re-coded more perspicuously:

```
AP0 =: Exit@Part@Init            NB. AP0 (n)
    Exit =: >@{.@>
    Init =: <"0@Mins , <@Empty
        Empty =: <@,:@i.@0:
        Mins =: , (<. |.)@}.@i.
    Part =: Next EACH/
        Next =: <@;@Ps , ]
            Ps =: Ns@[ Join EACH {.
                Ns =: >:@i.
                Join =: [ ,. Select # ]
                    Select =: >: {."1
```

10% shorter, `AP0` runs at about the same speed as `Boss` in a little less space.

Hui

Hui [7] recast Boss's program using Power instead of Insert:

```
part =: 3 : 'final (, new)^:y <<i.1 0'
new =: (-i.)@# <@:(cat&.>) ]
cat =: [ ;@:(,.&.>) -@(<.#) {. ]
final =: ; @: (<@-.&0"1&.>) @ > @ {:
```

`part 6` produces a list of 11 boxes of partitions with descending parts (without 0s).
This program is much slower than Boss and uses 80% more space. It could speed up 33% if 0s in boxes were not removed: `NB final =: ;@>@{:` But that would require more space.

Make it conform to other programs here that return a numeric table:

```
Hui =: >@part
```

```
   (Hui -: Boss) 6
1
```

This could be re-coded in smaller modules:

```
Hui0 =: Final@Parts
    Final =: ;@>@{:
        Inputs =: ] ` Start
        Start =: <@<@,:@i.@0:
    Parts =: Boxes^:Inputs
        Boxes =: , <@New
            New =: (-i.)@# ;@Cat EACH ]
                Cat =: [ ,.EACH Min {. ]
                    Min =: -@<. #
```

However, these programs need excessive memory and will not compete well at high `n`.

AP1

Here is my revision of Boss's program, notably without using an outer level of boxing:

```
AP1 =: ;@All ELSE Ns WHEN (< 2:)            NB. AP1 (n)
    All =: Ns ,.EACH Parts/@Mins
        Ns =: >:@i.
        Mins =: Smaller@}.@i. , 0:
            Smaller =: <. |.
        Parts =: Next ELSE Start WHEN Zero
            Zero =: 0 e. ]
            Start =: 1 ; ,@0:
            Next =: ;@New ; ]
                New =: Ns@[ Join EACH {.
                    Join =: [ ,. Select # ]
                        Select =: >: {."1

    (AP1 -: Boss) 6
1
```

This program runs much faster than `Boss` in significantly less space. Indeed, it can do n = 70 whereas `Boss` runs out of memory.

Also consider an explicit translation:

```
AP1x =: 3 : 0
ns =. >:i.y
mins =. (<. |.) }:ns
all =. < i.1 0
while. #mins do. new =. all New~ {:mins
            all =. all ;~ ;new
            mins =. }:mins
end.
all =. ns ,.EACH all
;all
)
```

Notice use of `New` and its tacit sub-programs from above.

`AP1x` is slower than `AP1`, needs a lot more space, and is longer. So it will bow out of the competition.

How about a recursive definition?

```
AP1r =: ;@All
    All =: Ns ,.EACH Mins Parts Empty
        Empty =: <@,:@i.@0:
        Ns =: >:@i.
        Mins =: Smaller@}.@i.
            Smaller =: <. |.
        Parts =: Build ELSE ] WHEN Done             NB. Parts calls Build
            Done =: 1 > #@[
            Build =: ButFirst Parts First Next ]    NB. Build calls Parts
                First =: {.@[
                ButFirst =: }.@[
                Next =: ;@New ; ]
                    New =: Ns@[ Join EACH {.
                        Join =: [ ,. Select # ]
                            Select =: >: {."1
```

This has about the same speed and space as `AP1` but is longer. So, it will defer to `AP1`.

See **Peelle's Algorithms** later for more efficient recursive programs.

AP2

In [8], I described a variant of Boss's algorithm that computes the leading part at each iteration: the smaller of the number of accumulated boxed arrays and *n* minus that number. Renamed and edited slightly:

```
AP2 =: ;@All                    NB. AP2 (n)
    All =: Ns ,.EACH Parts
        Ns =: >:@i.
        Parts =: Build^:N Start
            Start =: 1 0 <@$ ]
            N =: 0 >. <:@[
            Build =: <@;@Next , ]
                Next =: Lead Ps ]
                    Lead =: Min #
```

```
                        Min =: - <. ]
                    Ps =: Ns@[ Join EACH {.
                        Join =: [ ,. Select # ]
                            Select =: >: {."1

    (AP2 -: Boss) 6
1
```

See [8] for a tutorial. Example:

```
   AP2 6
1 1 1 1 1 1
2 1 1 1 1 0
2 2 1 1 0 0
2 2 2 0 0 0
3 1 1 1 0 0
3 2 1 0 0 0
3 3 0 0 0 0
4 1 1 0 0 0
4 2 0 0 0 0
5 1 0 0 0 0
6 0 0 0 0 0
```

This program is about 33% faster than `Boss` at n = 65 in about 25% less space. Compared to `AP1`, it has about the same speed, slightly less space, and less length.

Efficiency can be improved slightly further by separating top and bottom halves of the partitions array, by defining separate programs for odd and even input, or with other programming techniques (not shown here because those programs are much longer).

Kelleher’s Algorithm

Jerome Kelleher and Barry O’Sullivan published two algorithms in [9] for all partitions with ascending parts, superseding speed of existing programs for descending parts by Knuth [3] and by Zoghbi & Stojmenovic [10]. Kelleher’s most efficient algorithm generates lexicographic successors iteratively with embedded loops, notably a second loop to handle transitions involving only the last two parts. Coded straightforwardly in J:

```
Kelleher =: 3 : 0            NB. Kelleher (n)
n =. y
a =. (n+1)#0
k =. 1
a =. (0) 0} a        NB. redundant
y =. n-1
all =. i.0,n
while. k~:0 do.
    x =. ((k-1){a)+1
    k =. k-1
    while. (2*x)<:y do.
        a =. x k} a
        y =. y-x
        k =. k+1
      end.
    l =. k+1
    while. x<:y do.
        a =. x k}a
        a =. y l}a
        all =. all , (i.k+2){a
        x =. x+1
        y =. y-1
      end.
    a =. (x+y) k}a
    y =. x + y - 1
    all =. all , (i.k+1){a
  end.
all
)
```

Example:

```
   Kelleher 6
1 1 1 1 1 1
1 1 1 1 2 0
1 1 1 3 0 0
1 1 2 2 0 0
1 1 4 0 0 0
1 2 3 0 0 0
1 5 0 0 0 0
2 2 2 0 0 0
2 4 0 0 0 0
3 3 0 0 0 0
6 0 0 0 0 0
```

`Kelleher` is 4 times slower than `AP2` at n = 65 but uses only about 40% of the space. It is faster than `Knuth` in about the same space, and much shorter. An even faster and shorter, more J-ish version `Kelleherx` is given in the **Appendix**.

Peelle's Algorithms

Now look at how well some new recursive algorithms perform.

AP

Peelle presented a tidy ambivalent program in [8] using a '1 Plus' recursive approach to produce all partitions of `n` or partitions of `n` with up to `p` parts:

```
AP =: ;@All                        NB. AP (n) or (n) AP (p)
    All =: Parts EACH >:@i.
        Parts =: Ones ELSE Plus1 WHEN >
            Ones =: = # ] ,:@# 1:
            Plus1 =: 1 + - AP ]
```

The main idea is to add 1 to each sub-array of partitions of (`n-1` to `p`) for 1 to `p` parts. In other words, for a given number of parts, partition n-parts, then add 1 to each part (including 0s).

For example, assemble 1 plus each result below to get `6 AP 3`:

```
   5 AP 1                            6 AP 3
5                                 6 0 0
   4 AP 2                         5 1 0
4 0                               4 2 0
3 1                               3 3 0
2 2                               4 1 1
   3 AP 3                         3 2 1
3 0 0                             2 2 2
2 1 0
1 1 1
```

See [8] for a tutorial. Notice that the order of all partitions is the same as `Hindenburg`:

```
   AP 6            NB. 6 AP 6
6 0 0 0 0 0
5 1 0 0 0 0
4 2 0 0 0 0
3 3 0 0 0 0
4 1 1 0 0 0
3 2 1 0 0 0
2 2 2 0 0 0
3 1 1 1 0 0
2 2 1 1 0 0
2 1 1 1 1 0
1 1 1 1 1 1
```

`AP` can speed up greatly by exiting when either input is 0 or 1:

```
AP =: ;@All ELSE N WHEN Under2
Under2 =: 2 > <.
 N =: ,:@{.~
```

This program runs three times faster than `Skiena` but needs 30% more space at `n=65`. Yet, AP can do `n=70`, whereas `Skiena` runs out of memory.

`AP` can be improved by handling 0 and 1 parts separately:

```
   AP =: N ;@, [ All <.
 N =: <@,:@{.~
 All =: Parts EACH 2 }. i. , ]
  Parts =: 1 + ] {."1 - AP ]
```

This version will be used for comparisons and is listed in the Appendix. It is much faster and much shorter than `AP1` and `AP2` in much less space at n=70 although it is much slower at n=65. Which is better? See Comparisons.

Translation into explicit definition avoids boxing:

```
APx =: 3 : 0
:
all =. ,:y{.x
for_p. 2 }. i.>:y do. all =. all , 1 + p {."1 n APx p<.n=.x-p end.
)
```

This needs 40% less space than AP but twice the time and is longer. So it will drop out.

APr

Here is a simple recursive definition for two inputs: n and a lead part. Explicitly:

```
APrx =: 3 : 0             NB. (n) APrx (lead)
:
all =. i.0,x
while. y>1
   do. all =. all , y ,. n APrx y <. n=.x-y
       y =. y-1
  end.
all,1
)
```

Note that it skips the loop whenever `y` is 1 and just appends the last row of 1s. Example: `6 APrx 6` or `APrx~6` is the same as `Knuth 6`.

This program is much slower than `AP` yet much slimmer in space until `n=70` when it exceeds memory. So, forget it.

A tacit ambivalent version is shorter, 40% faster in 75% more space and can do `n=70`:

```
APr =: ;@All ELSE Ones WHEN Under2        NB. (n) APr (lead) or APr (n)
    Ones =: ,:@#
    Under2 =: 2 > ]
    All =: Ns Parts EACH Leads@]
        Parts =: ] ,. [ APr <.
        Ns =: - Leads
        Leads =: - i.
```

This can be improved further by generating 1s separately:

```
APr =: <@Ones ;@, Allbut1s        NB. (n) APr (lead) or APr (n)
        Ones =: ,:@# 1:
        Allbut1s =: Parts EACH Leads
            Parts =: ] ,. - APr - <. ]
            Leads =: 2 }. i. , ]
```

Now it is more competitive with `AP` in speed, but much fatter, albeit shorter. Indeed, this is the shortest program among those here that can perform high `n`.

APm

Another simple recursive definition uses previous partitions as a *memo* to produce the next:

```
APm =: 3 : 0             NB. APm (n)
all =. i.0,n=.y
while. n>1
```

```
   do. memo =: APm y-n
    leads =. {."1 memo
    drop =. leads i. n <. y-n
      all =. all , n ,. drop }. memo
    n =. n-1
  end.
all,1
)
```

Note that global memo cuts down space significantly. Still, it's way too slow.
J's built-in Memo adverb M. does better. See APdb later in **Other Programs**.

APh

This approach produces all partitions in two halves, recursively, in an ambivalent program. For the bottom half, it recurses when n > lead part; and for the top half, it recurses with n as both inputs.

```
APh =: [ ;@Parts New            NB. APh (n) or (n) APh (lead)
    New =: -@<. {. i.@[
    Parts =: ] Part EACH -
        Part =: ] ,. Recurse ELSE Empty WHEN Zero ELSE Ones WHEN One
            Recurse =: [ APh [ ELSE ] WHEN >
            Empty =: ,:@i.@0:
            Zero =: = 0:
            Ones =: ,:@#
            One =: 1 = ]
```

Examples of its subprogram for two inputs:

```
   0 Part 6
6

   1 Part 5
5 1

   2 Part 4
4 2 0
4 1 1

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

   4 Part 2
2 2 2 0 0
2 2 1 1 0
2 1 1 1 1

   5 Part 1
1 1 1 1 1 1
```

APh 6 assembles these results into one table of descending partitions (same as |.APr 6).

Include an exit for 2 and build an array with leading 2s directly:

```
Part=: ] ,. Recurse ELSE Empty WHEN Zero ELSE Twos WHEN Two ELSE Ones WHEN One
            Two =: 2 = ]
            Twos =: Build ^: Inputs
                Build =: 1 ,~ 2 ,. ] ,. 0:
Inputs =: <.@% ` Start
                    Start =: ,:@:>:@i.@|~
```

This program is now much faster in equal space but quite longer.

APn

Another approach constructs a table of partitions horizontally by nesting columns recursively:

```
APnx =: 3 : 0                NB. APnx (n)
is =. i.y
; is Nest EACH y-is
)
```

Nest =: 3 : 0 : if. y=1 do. 1,x#1 return. end. if. x=0 do. ,:,y else. min =. – x <. y is =. min {. i.x y ,. ; is Nest EACH x-is end.)

A tacit version is shorter and much faster:

```
APn =: [ ;@Parts <.                    NB. APn (n)
    Parts =: Is Nest EACH Ns@]
    Is =: i.@] + -
    Ns =: - i.
        Nest =: Join ELSE N WHEN Zero ELSE Ones WHEN One
            Join =: ] ,. APn
            N =: ,:@,@]
            Zero =: =0:
            Ones =: 1,#
            One =: 1=]
```

APapr

Finally, consider this: halve input `n` to become the largest second part, subtract it from `n` to get the first part, iterate through decrements of first and second parts respectively, and call `APr` (presented previously) to attach a sub-array of partitions for the remaining `new_n` with the leading part as the smaller of second and `new_n`:

```
APapr =: 3 : 0           NB. APapr (n)
all =. Ns1s y-i.y
for_second. 2 To <.-:y
    do.    for_first. |.second To y-second
            do. all =. all,first,.second,.n APr second<.n=.y-first+second
            end.
    end.
)
    To =: }. i. , ]
```

For efficiency, all partitions with a second part 0 and 1 are done at once separately in a table:

```
    Ns1s =: ,. (</ }:)             NB. Ns1s (n-i.n)
```

Example:

```
   APapr 6
6 0 0 0 0 0
5 1 0 0 0 0
4 1 1 0 0 0
3 1 1 1 0 0
2 1 1 1 1 0
1 1 1 1 1 1
4 2 0 0 0 0
3 2 1 0 0 0
2 2 1 1 0 0
2 2 2 0 0 0
3 3 0 0 0 0
```

Note the unorthodox order: by increasing second part.

This program is faster than APr, and much shorter than Kelleher, using only a little more space. (even counting APr). Despite relying on a sub-program that itself can compute all partitions, it competes very well at high n. Indeed, it is the fastest here at n=70.

Comparisons

Now compare the most competitive programs here for n=65 by ratios of time, space, and length – where 1.00 is best. Finally, sum the ratios for a simple composite of overall program efficiency:

```
             Time        Space      Length     Overall

Skiena      12.59        1.46        1.00        15.05

Knuth        5.15        1.00        7.94        14.09

Boss         1.38        3.00        3.00         7.38

Hui         24.31        5.41        2.38        32.11

AP1          1.01        2.34        2.79         6.14

AP2          1.00        2.34        2.44         5.78

Kelleher     4.49        1.00        6.35        11.85

AP           2.77        1.89        1.21         5.86

APr          3.53        2.34        1.15         7.02

APh          3.71        2.34        3.44         9.49

APn          4.85        2.34        1.85         9.04
```

```
APapr       3.40        1.03        3.88         8.31
```

So AP2 is the winner, with AP and AP1 close behind.

For n=70, Skiena, Boss and Hui cannot participate on my computer because they run out of memory. Timing results for most remaining programs have large variance since they are pushing space limits. Within this uncertain repeatability, APapr emerges as the fastest and AP becomes the new winner overall. In order, the top six are: AP, APr, APapr, AP2, AP1, Kelleherx. Considering only time and space, the top six are: APapr, Kelleher, AP, Knuth, AP2, APr.

No program above can do n=75 on my laptop. Timing ratios on other computers differ somewhat but indicate that AP1 and AP2 are the fastest.

Readers may be able to run higher n, may want to weight the three criteria differently, and may add other criteria, such as a measure of clarity. See Appendix for benchmark details and copies of the best programs for convenient use.

Number of Partitions

For any program above, the number of partitions can be counted perfunctorily from the resulting table of partitions. For example: #AP 6 is 11. A list of partition numbers:

```
   #@AP"0 i.20
1 1 2 3 5 7 11 15 22 30 42 56 77 101 135 176 231 297 385 490
```

There is an issue about how to count the number of partitions for n=0:

```
   $Skiena~ 0
1 0

NB. Knuth 0 does not compute

   $Boss 0
0
   $Hui 0
1 0
   $AP1 0
1
   $AP2 0
0
   $Kelleher 0
1 1
   $AP 0
1 0
   $APr 0
1 0
   $APapr 0
0 1
```

Since the shape of an empty list is 0, one can argue that there are zero partitions of 0. One can also argue that there is one partition of 0: the empty partition (a 1 by 0 table) or a 1-item list (,0). In all cases, the sum is 0. So, what should it be – an empty list, or an empty table, or just 0? See oeis.org/wiki/Partition function.

Of course, the pertinent problem is how to compute number of partitions efficiently for large n without the effort of generating them.

P

To begin with, here is a clean program to compute number of partitions of integer n with k parts, albeit inefficiently due to recursion:

```
P =: Recur ELSE = WHEN Done        NB. (n) P (k)

    Recur =: P&<: + - P ]
    Done =: <: +. 0 = ]
```

An alternative definition:

```
P =: Recur ELSE ] WHEN (2>]) ELSE = WHEN <:        NB. (n) P (k)
```

For example, number of partitions of 10 with 4 parts:

```
   10 P 4
9
```

The total number of partitions of n with k parts is +/n P"0 i.>:k .

A table of P for both n and k from 0 to 10:

```
      P"0/~ i.11
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0 0
0 1 2 1 1 0 0 0 0 0 0
0 1 2 2 1 1 0 0 0 0 0
0 1 3 3 2 1 1 0 0 0 0
0 1 3 4 3 2 1 1 0 0 0
0 1 4 5 5 3 2 1 1 0 0
0 1 4 7 6 5 3 2 1 1 0
0 1 5 8 9 7 5 3 2 1 1
```

Notice that the nth row sum is the number of all partitions of n.

Pnk

An efficient program to build this table iteratively adds the reverse diagonal to the shifted last row:

```
TP =: One , 0 ,. Row ^: (Repeat`One)        NB. TP (n)
```

```
        Repeat =: 0 >. <:
        One =: ,:@,@1:
    Row =: , Last + Diag , 0:
        Last =: 0 , {:
        Diag =: (<0 1) |: |.
```

Index the table to get number of partitions of `n` with `k` parts:

```
Pnk =: <@, { TP@[            NB. (n) Pnk (k)  or  Pnk (n)
```

Example:

```
   10 Pnk 4
9
```

NP

Since Pnk is ambivalent, Pnk(n) is the last row, and its sum is the number of all partitions:

```
NP =: +/@Pnk             NB. NP (n)
```

Example:

```
   NP 10
42
```

First 20 numbers of all partitions:

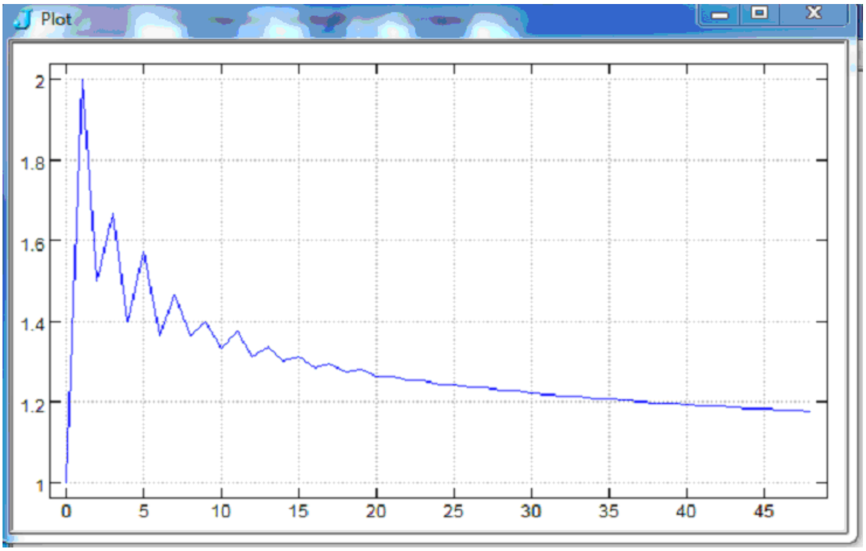
```
   NP"0 i.20
1 1 2 3 5 7 11 15 22 30 42 56 77 101 135 176 231 297 385 490
```

Larger numbers:

```
   n ,: NP"0 n =. 5*i.15
0 5 10  15  20   25   30    35    40    45     50     55     60      65      70
1 7 42 176 627 1958 5604 14883 37338 89134 204226 451276 966467 2012558 4087968
```

Growth factors:

```
   Growth =: }. % }:

   5j2 ": Growth NP"0 i.15
1.00 2.00 1.50 1.67 1.40 1.57 1.36 1.47 1.36 1.40 1.33 1.38 1.31 1.34

   load 'plot'
   plot Growth NP"0 i.50
```



```
   Growth NP"0 n=.1000+i.10
1.04035 1.04033 1.04031 1.04029 1.04027 1.04025 1.04023 1.04021 1.04019
```

For very large n, use Hui's efficient pnv program in [7] based on Euler's recurrence relation to see convergence to an asymptote approaching 1.

Other Programs

Several other programs that generate all integer partitions have not been mentioned yet because they are not nearly competitive with those above. With respect for their role in research and development, they are summarized as follows:

Reingold

Reingold et al. [11] described an algorithm for generating tuples with ascending parts. Coded explicitly in J:

```
Reingold =: 3 : 0              NB. Reingold (n)
n =. y
m =. 1
ps =. i.0 0
while. n>:m
do. ps =. ps , n Parts m
        m =. m + 1
end.
ps
)
```

```
Parts =: 3 : 0                NB. (n) Parts (m)
:
n =. x
m =. y
ps =. ,: 1 + (-m) {. n-m
while. +./1<p-~{:p=.{:ps
do. ps =. ps , Next p
end.
)
```

```
Next =: 3 : 0                 NB. Next (p)
p =. y
n =. +/p
m =. #p
i =. <: 0 i.~ 2 <: p-~{:p
p =. (1+i{p) (i }. i.m)} p
p =. (n-+/}:p) (m-1)} p
)
```

For example:

```
   Reingold 6
6 0 0 0 0 0
1 5 0 0 0 0
2 4 0 0 0 0
3 3 0 0 0 0
1 1 4 0 0 0
1 2 3 0 0 0
2 2 2 0 0 0
1 1 1 3 0 0
1 1 2 2 0 0
1 1 1 1 2 0
1 1 1 1 1 1
```

The program is lengthy, four times slower than Knuth at n=65 in twice the space, and runs for hours at n=70. Tacit translations (for both ascending and descending parts) use 30% less space but are ten times slower.

Reiter

Reiter [12] presented a recursive program in APL that uses three inputs (in one list): integer n, number of parts p and smallest part s. It splits the computation depending on whether there are partitions into one, two, or more parts. The case of two parts is not necessary to the correctness of the algorithm, but it does improve the efficiency of the algorithm. When the number of parts is three or more, then a loop catenates blocks of partitions of smaller (or equal) numbers into a smaller number of parts with the smallest number allowed being the index of the loop. [11 page 8] In J:

```
Reiter =: ;@Blocks ELSE (,@N) WHEN (P < 2:)        NB. Reiter (n,p,s)
    Blocks =: <@(S,.Reiter)"1 @ NPSs
        NPSs =: (N-Ss) ,. <:@P ,. Ss
    N =: {.
    P =: {.@}.
    S =: {:
```

```
    Ss =: S To N Q P
        Q =: <.@%
        To =: [ + i.@>:@-~
```

Example of partitions of 10 with 4 parts, starting with 1:

```
   Reiter 10 4 1
1 1 1 7
1 1 2 6
1 1 3 5
1 1 4 4
1 2 2 5
1 2 3 4
1 3 3 3
2 2 2 4
2 2 3 3
```

All partitions of `n` are produced by inputs (`n,n,0`). For example: `Reiter 6 6 0`. This program runs slower than `Skiena` but needs less space. Unfortunately, it is much longer.

Peelle

Here is my doubly recursive program based on P (above) that uses two fundamental functions to generate all partitions – appending a 1 and adding 1:

```
Peelle =: ;@All                NB. Peelle (n)
    All =: Parts EACH >:@i.
    Parts =: Recurse ELSE Empty WHEN < ELSE N WHEN One ELSE Ones WHEN =
        Recurse =: Append1 , Plus1
            Append1 =: Parts&<: ,. 1:
            Plus1 =: 1 + - Parts ]
        Empty =: 0 i.@, ]
        Ones =: # 1:
        N =: ,:@,@[
        One =: 1 = ]
```

This program is twice as fast as `Skiena`, in less space than `Reingold`, and shorter than `Reiter`. It can be improved by combining exits:

```
    Parts =: Recurse ELSE N WHEN Under2 ELSE Ones WHEN >:
        Recurse =: Append1 , 1 + - Parts ]
        N =: ,@[
        Under2 =: 2 > ]
       Ones =: = # [ ,:@# 1:
```

Now it's quite competitive at `n=70` – notably faster, slimmer, and shorter than `AP1` and `AP2` and even faster than `Kelleherx` (but with more space).

A key insight here led to the development of `AP` (in [8]): Appending 1 is equivalent to `1+(n-p) Parts (p-1)` with an appended column of 0s so that the result of `Recurse` can be produced by `Plus1` alone. `AP` is superior to `Peelle`, being faster in equal space and shorter.

Another improvement handles 0 and 1 parts separately and exits for n=2:

```
Peelle2 =:  ;@All2     NB. Peelle2 (n)
    All2 =: <@N01 , AllbutN01
    AllbutN01 =: Parts2 EACH 2 }. i. , ]
       Parts2 =: Recurse2 ELSE Pairs WHEN Two ELSE Ones WHEN <:
           Recurse2 =: Append2 , 1 + - Parts2 ]
                Append2 =: Parts2&<: ,. 1:
           N01 =: ,@[
           Ones =: = # ] ,:@# 1:
           Pairs =: [ By >:@i.@<.@%
               By =: - ,. ]
           Two =: 2 = ]
```

This program is very fast at high n in about the same space.

Using an explicit master program saves 40% space but is longer:

```
Peellex =: 3 : 0
all =. i.0,y
for_p. <i.y do. all =. all , y Parts p end.
)
```

Further, an Iterative version builds a table of boxes to index:

```
Peellei =: ;@Corner@T    NB. Peellei (n)
        EACH =: &.>
    Corner =: Top,Right ,"1 EACH Ones
        Ones =: >:@i.@# #EACH 1:
        Right =: {:"1
           Inputs =: <:@<: ` Start
                Start =: One (<0 0)} ,~@Half $ Empty
                    One =: 1 1 <@$ 1:
                    Empty =: 0 0 <@$ 0:
                    Half =: >.@-:
        Top =: First , Append1 EACH@}:@{. ,EACH Plus1 EACH@}.@Diagonal
                First =: Plus1 EACH@{.@Diagonal
                Diagonal =: (<0 1) |: ]
                Append1 =: ,.&1
                Plus1 =: >:
    T =: Build ^: Inputs
        Build =: Top }:@, ]
                                        NB. correct except for n=0 and 1
```

This program is very fast (about same speed as Boss) up to n=65 but very fat (fatter than Boss).

Now, here are several unusual (and inefficient) approaches that may be of interest to the curious:

APod

An odometer approach entails selecting unique lists of partitions with sorted parts from a table of consecutive integers represented as lists in base n+1. The program is loopless and short, but indulgent:

```
APod =: ~.@Select (Odometer >:)
    Odometer =: # #: i.@^~           NB. (n) Odometer (n+1)
    Select =: Sort"1@Parts
        Parts =: IsSum"1 # ]
        Sort =: /:~
        IsSum =: = +/
```

Lexical order, as on an odometer:

```
   APod 6
0 0 0 0 0 6
0 0 0 0 1 5
0 0 0 0 2 4
0 0 0 0 3 3
0 0 0 1 1 4
0 0 0 1 2 3
0 0 0 2 2 2
0 0 1 1 1 3
0 0 1 1 2 2
0 1 1 1 1 2
1 1 1 1 1 1
```

Revise it to be 2.5 times faster:

```
APod =: N , (~.@Select Odometer~)
    N =: - {. ]
```

Using multiple bases is 64 times faster still:

```
APod =: ~.@Select Odometer
    Odometer =: Encode@Bases           NB. Odometer (n)
        Encode =: #: i.@(*/)
        Bases =: 1 + Diagonal@i.
            Diagonal =: {"0 1 Copies@:>:
                Copies =: |. #"0 1 ]
```

6 times faster again (with left-justified result):

```
APod =: ;@(<@Parts"0 Ns)
```

```
    Parts =: 1 + - ~.@Select Odometer            NB. (n) Parts (p)
        Odometer =: Encode@Bases            NB. (n) Odometer (p)
            Encode =: #: i.@(*/)
            Bases =: (-i.) Q |.@Ns@]
                Q =: <.@%
```

Even though this is hundreds times faster than the initial definition, it is so inefficient that it runs out of memory at about `n=30`.

AP9s

Another approach encodes only multiples of 9 (instead of a full odometer) into base-10 digits, selects ascending lists, then selects lists that sum to n. How crude!

```
AP9s =: All ELSE i. WHEN (=0:)
```

```
    All =: Parts Ascend@Encode10
        Encode10 =: #&10 #: (+ 9 Multiples <:)
            Multiples =: * i.@(10&^)
        Ascend =: #~ (-: Sort)"1
            Sort =: /:~
        Parts =: IsSum"1 # ]
            IsSum =: = +/
```

This program is very slow and space-consuming – only able to do up to n=8.

APid

This next approach iterates adding an identity matrix to generate all possible new partitions: Start with an empty 1 by 0 array. Iteratively join a left column of 0s then a top row of 1s to ascending (sorted) lists in tables created by adding each row of an appropriate size identity matrix to each row in the previous array. Finally, remove duplicate partitions from the result.

```
APid =: ~.@All^:(]`Empty)
    Empty =: ,:@i.@0:
    All =: 1 , 0 ,. Parts
        Parts =: Sort"1@(,/)@:(Plus"1)
            Sort =: /:~
            Plus =: +"1 Id
                Id =: =@i.@#
```

```
   APid 6
1 1 1 1 1 1                NB. Reverse lexical order
0 1 1 1 1 2
0 0 1 1 2 2
0 0 1 1 1 3
0 0 0 2 2 2
0 0 0 1 2 3
0 0 0 1 1 4
0 0 0 0 3 3
0 0 0 0 2 4
0 0 0 0 1 5
0 0 0 0 0 6
```

This program runs out of memory at n=45.

Alternative: Use boxes and trim the identity matrix to add 1s only to the last unique part, so there is no need to sort.

```
APid =: ~.@All^:(]`Empty)
    Empty =: ,:@i.@0:
    All =: 1 , 0 ,. ;@:Parts
        Parts =:  <@Plus"1
            Plus =: +"1 Id
                Id =: (i: ~. -. 0:) =/ i.@#
```

This is almost 10 times faster and uses less than 10% space, but runs out of memory at n=60.

APpnk

Here is an approach that computes the number of partitions in advance for a given n and k in order to generate partitions iteratively in k-tuples. It uses program Pnk (from Number of Partitions earlier) to determine how many times to iterate.

```
APpnk =: ;@All                NB. APpnk (n)
    All =: Parts EACH Ns
        Ns =: >:@i.
        Parts =: Next^:(Pnks`Start)     NB. (n) Parts (k)
            Pnks =: i.@Pnk        NB. (n) Pnk (k)
            Start =: 1 + ] {. -
            Next =: Front To Body ]
                To =: Tos i. 1:
                    Tos =: 1 < (- ~{.)
                Body =: Copies , Back
                    Back =: >:@[ }. ]
                    Copies =: [ # >:@{
                Front =: (- +/) , ]
```

```
   (APpnk -: AP) 6
1
```

This program is shorter and faster than Knuth and Kelleher at n=65, but requires more space. It is even more competitive at n=70, although it seems like cheating.

APdb

Another approach updates a database every time the program is used anew. A global variable contains previously executed results for smaller n and k that can be looked up directly instead of re-computed.

First, initialize a global boxed matrix:

```
parts =: 1 1 $ <i.1 0
```

Define the program to immediately extend the matrix with empty boxes if either input n or k is larger than its shape. Then look up the nth row and kth column. If it is empty, compute the result and update the global matrix; otherwise, open it as the result.

```
APdb =: 3 : 0               NB. (n) APdb (k) or APdb~ (n)
:
nk =. x,y
if. +./ nk >: $parts do. parts =: (nk+1) {. parts end.   NB. extend
                p =. (<nk) { parts                        NB. look up
                if. p=a: do. p =. < x APrdb y             NB. compute
                parts =: p (<nk)} parts                   NB. update
end.
>p
)
```

The following co-program used above is a modification of APr that computes all partitions of n in k parts recursively but calls APdb (instead of itself) to look up a sub-result if it is already known.

```
APrdb =: ;@All ELSE Ones WHEN Under2        NB. (n) APrdb (lead)
    Ones =: ,:@#
    Under2 =: 2 > ]
    All =: Leads@] <@Parts"0 Ns
        Parts =: [ ,. ] APdb <.         NB. APdb instead of APrdb
        Leads =: - i.
        Ns =: - Leads
```

Using a database is advantageous whenever partitions below `n` and `k` must be re-computed quickly. `APdb` is also quite speedy for new `n` and `k` up to about `n=60`. For instance, `APdb~60` is about 4 times faster than `APr 60`. However, by `n=65`, it becomes sluggish and demands about 50% more space even though only about `1r6` of the database is filled up.

It is easier to use J's built-in Memo adverb `M.` to gain about the same advantage.
For example: `NB. APr =: ;@All M. ELSE Ones WHEN Under2`
Then `APr 60` would be 4 times faster, but still more than 4 times slower for `n=65`.

Other Representations

So far, a natural representation has been used for a partition – that is, a list of positive integers (without interspersed + symbols). For example, a partition of 15: 5 3 3 1 1 1 1 or 1 1 1 1 3 3 5. Other possible representations include: *base* (or *standard*), *frequency* (and *multiplicity*), and *Ferrers* dot diagram.

Base

The numerals of a partition can be compressed into digits of a single number in base 10, as in 5331111 or 1111335. This might be advantageous for certain algorithms, such as selecting multiples of 9 (see `AP9s` above). However, if any part exceeds 9, this representation becomes awkward, necessitating pairs of digits, or triples for `n>99`, etc.

To convert from natural to base representation, use J's Decode with base 10 for partitions such as:

```
   10 #. 5 3 3 1 1 1 1                      10 #. 1 1 1 1 3 3 5
5331111                                1111335
```

To convert from *base* to *natural*, use its inverse Encode with the appropriate base for each digit:

```
   (7#10) #: 5331111
5 3 3 1 1 1 1
```

Or use J's Power to find the inverse:

```
   10&#. ^:_1 ] 5331111
5 3 3 1 1 1 1
```

Frequency

A partition can be represented as a list of frequencies of parts from 1 to the largest.

To convert from natural to frequency representation, use this program:

```
   Freq =: +/@(=/) >:@i.@(>./)
```

Example:

```
   Freq 5 3 3 1 1 1 1               Freq 1 1 1 1 3 3 5
4 0 2 0 1                    4 0 2 0 1
```

To convert from frequency to natural:

```
   FreqInv =: # >:@i.@#

   FreqInv 4 0 2 0 1
1 1 1 1 3 3 5
```

Or include its inverse in the definition:

```
   Freqs =: Freq :. FreqInv

   Freqs^:_1 ] 4 0 2 0 1
1 1 1 1 3 3 5
```

For any partition of n (in any order), n equals the sum of frequencies times integers 1 to the largest part, respectively:

```
   p =. >:?10#10
   n =. +/p
   fs =. Freq p                 NB. fs =. Freqs p
   n = +/fs*>:i.>./p
1
```

Multiplicity

Alternatively, a partition can be represented simply by multiples of its unique parts. For example, 4 2 1 multiples of unique parts 1 3 5 represents 1 1 1 1 3 3 5.

Convert from natural to multiplicity:

```
   +/"1 =1 1 1 1 3 3 5
4 2 1
```

Convert from multiplicity to natural representation:

```
   1 3 5 #~ 4 2 1
1 1 1 1 3 3 5
```

Define a program with its inverse:

```
   Multi =: +/"1@= :. (#~)
```

Both conversions:

```
   Multi 1 1 1 1 3 3 5
4 2 1
```

```
   1 3 5 Multi^:_1 ] 4 2 1
1 1 1 1 3 3 5
```

Any sorted partition p is identical to the inverse conversion of its conversion:

```
   p -: (~. Multi^:_1 Multi) p =. /:~ >:?10#10
1
```

Ferrers diagram

Another representation devised by Norman M. Ferrers provides visualization of a partition via rows of dots (or any symbol) for each part. See [3] or [13]. For example, 5 3 3 1 1 1 1 would be:

```
    *  *  *  *  *
    *  *  *
    *  *  *
    *
    *
    *
    *
```

A program to produce Boolean indices for such a diagram:

```
   Ferrers =: #"0 1:            NB. Ferrers =: >/ i.@(>./)
```

Example:

```
   Ferrers 5 3 3 1 1 1 1                  ‘ *’ {~ Ferrers 5 3 3 1 1 1 1
1 1 1 1 1                        *****
1 1 1 0 0                        ***
1 1 1 0 0                        ***
1 0 0 0 0                        *
1 0 0 0 0                        *
1 0 0 0 0                        *
1 0 0 0 0                        *
```

Include its inverse:

```
   Ferrers =: (# 1:)"0 :. (+/"1)
```

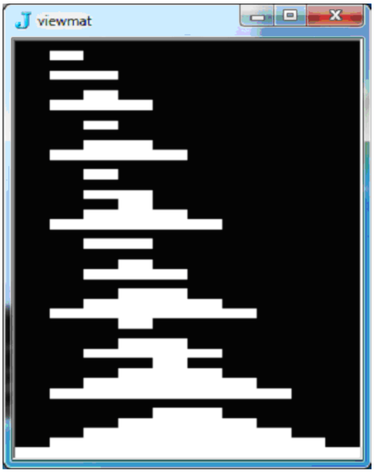
To produce a conjugate partition, simply sum the rows (vertically):

```
   +/ Ferrers 5 3 3 1 1 1 1
7 3 3 1 1
```

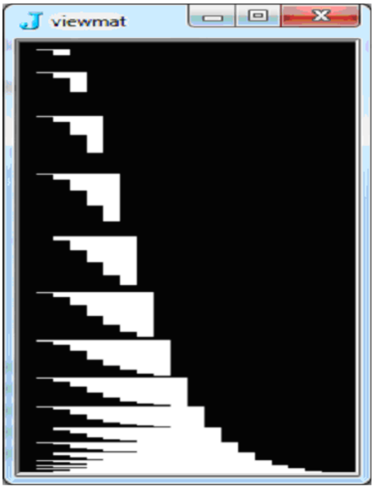
Fractal Patterns

Due to the recursive structure of partitions, it is not surprising to find fractal-like patterns. See [14].

```
   load 'graph'
   viewmat 1 = Knuth 10          NB. white 1s
```



```
   viewmat 1 = AP 20
```



Similarly, view other parts, such as `viewmat 2=AP 20` or all parts in colors: `viewmat AP 10`

Summary

J programs for generating all integer partitions were presented and compared for the first time. Most programs and some algorithms are new. The best programs were determined (within my computing constraints) for equally weighted criteria that included program length as well as speed and space. Translations between tacit and explicit definitions were often supplied and contrasted without due explication. To follow up, see References.

Acknowledgements

Thanks to Professor Emeritus Murray Eisenberg (UMass Mathematics & Statistics Department) for his helpful critique and for obtaining benchmarks on his iMac with J7.01. Thanks also to Boyko Bantchev for valuable feedback and generous contributions, to Devon McCormick (ACM, New York) for checking performance of programs, and to Professor Cliff Reiter (LaFayette College Mathematics Department) for continuing attention.

References

1. http://www.Jsoftware.com
2. S. Skiena, Implementing Discrete Mathematics … with Mathematica, Addison-Wesley,1990
3. D. E. Knuth, The Art of Computer Programming Vol.4A 7.2.1.4, Addison-Wesley, 2005-2011
4. C. F. Hindenburg, Infinitomii Dignitatum Exponentis Indeterminati (Göttingen) pp. 73-91, 1779
5. R.K.W. Hui, http://www.jsoftware.com/pipermail/general/2005-June/023191.html
6. R.E.Boss, “Partitions of numbers: An efficient algorithm in J”, Vector 23.4 pp. 121-131, 2008 http://archive.vector.org.uk/art10012080
7. R. Hui, http://www.jsoftware.com/Jwiki/Essays/Partitions , 2008-2011
8. H. A. Peelle, http://www.jsoftware.com/jwiki/Essays/AllPartitions , 2011
9. J. Kelleher & B. O’Sullivan, “Generating All Partitions: A Comparison of Two Encodings”, arxiv.org > cs > arxiv: 0909.2331 [cs.D5], 2009
10. A. Zoghbi & I. Stojmenovic, “Fast Algorithms for Generating Integer Partitions”, Int. J. Computer Math., Vol. 70 pp. 319-332, 1998
11. E. Reingold, J. Nievergelt, & N. Deo, Combinatorial Algorithms, Prentice-Hall,

1977

12. C. Reiter, “Random Markov Matrices and Partitions of Integers”, APL Quote-Quad, Vol. 22, No. 3, pp. 7-8, March 1992
13. E. W. Weisstein, http://www.Mathworld.wolfram.com/PartitionFunctionP.html
14. A. Salerno, “Partition Numbers Unveiled as Fractal”, MAA Focus, April-May 2011 maa.org/focus.html

Appendix

Benchmarks for time and space were obtained on a Dell Latitude E6410 laptop (64-bit OS M620 at 2.67 GHz with 4GB RAM) running J6.02 under Windows 7.

Length (Spread) = total number of characters in a program definition body, ignoring spaces, and counting only 1 for each name.

For a table t of benchmarks, the table of ratios (in Comparisons) is: `(,. +/"1) t %"1 <./t`

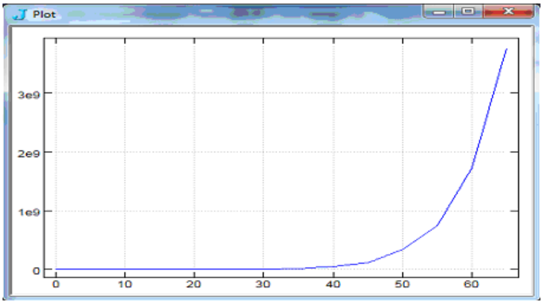
Utility programs:

```
Time =: 6 !: 2
Space =: 7 !: 2

Test =: (-: ~.)@:Sort *. +/"1 *./ . = {:@$    NB. all unique and all rows sum = n
    Sort =: /:~"1
```

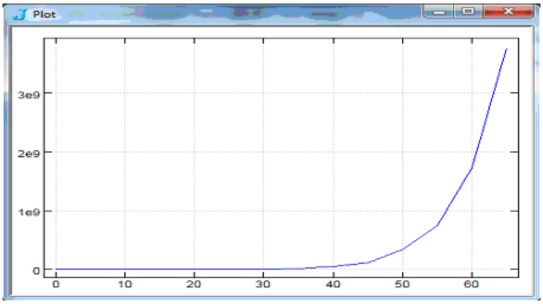
Example tests:

```
   *./Test@Knuth"0 }.i.66
1
   *./Test@Boss"0 i.66
1
   *./Test@Hui"0 i.66
1
   *./Test@AP"0 i.66
1
```

Example plots of Time and Space:



```
   load 'plot'
   plot n ; Time"1 'AP2 ' ,"1 ":,.n=.5*i.14
```



```
   plot n ; Space"1 'AP2 ' ,"1 ":,.n=.5*i.14
```

Program Definitions:

Definitions of the best programs in Comparisons are shown below for convenient copying and pasting. Note: Use *separate* scripts:

```
AP1 =: ;@All ` Ns @. (< 2:)                NB. AP1 (n)
    All =: Ns ,.&.> Parts/@Mins
        Ns =: >:@i.
        Mins =: Smaller@}.@i. , 0:
            Smaller =: <. |.
        Parts =: Next ` Start @. Zero
            Zero =: 0 e. ]
            Start =: 1 ; ,@0:
            Next =: ;@New ; ]
                New =: Ns@[ Join&.> {.
                    Join =: [ ,. Select # ]
                        Select =: >: {."1
```

```
AP2 =: ;@All                               NB. AP2 (n)
    All =: Ns ,.&.> Parts
        Ns =: >:@i.
        Parts =: Build^:N Start
            Start =: 1 0 <@$ ]
            N =: 0 >. <:@[
            Build =: <@;@Next , ]
                Next =: Lead Ps ]
                    Lead =: Min #
                        Min =: - <. ]
                    Ps =: Ns@[ Join&.> {.
                        Join =: [ ,. Select # ]
                            Select =: >: {."1
```

```
AP =: N ;@, [ All <.                       NB. AP (n) or (n) AP (p)
 N =: <@,:@{.~
 All =: Parts &.> 2 }. i. , ]
  Parts =: 1 + ] {."1 - AP ]
```

```
APr =: <@Ones ;@, Allbut1s                      NB. APr (n) or (n) APr (lead)
        Ones =: ,:@# 1:
        Allbut1s =: Parts &.> Leads
            Parts =: ] ,. - APr - <. ]
            Leads =: 2 }. i. , ]
```

```
Kelleherx =: 3 : 0                              NB. Kelleherx (n)
a =. 0,y#0
k =. 1
y =. y-1
all =. i.0 0
while. k do.
    k =. k-1
    x =. 1+k{a
    while. y>:2*x do.
        a =. x k}a
        y =. y-x
        k =. k+1
      end.
    l =. k+1
    while. x<:y do.
        a =. x k}a
        a =. y l}a
        all =. all,a{.~k+2
        x =. x+1
        y =. y-1
      end.
    a =. a k}~x+y
    y =. x+y-1
    all =. all,a{.~k+1
  end.
)
```

```
APapr =: 3 : 0                        NB. APapr (n)
all =. Ns1s -i.y
for_second. 2 To <.-:y
     do. for_first. |. second To y-second
             do. all =. all,first,.second,.n APr second<.n=.y-first+second
            end.
    end.
)

    Ns1s =: ,. (</ }:)
To =: }. i.@>:

NB. uses APr (above)
```