APL2000 Users Conference ...

BCS®

The Journal of the British APL Association

A specialist Group of the British Computer Society

ISSN 0955-1433
www.vector.org.uk
Vol.20 No.3 January 2004

Contributions

All contributions to VECTOR may be sent to the Journal Editor at the address on the inside back cover. Letters and articles are welcome on any topic of interest to the APL community. These do not need to be limited to APL themes, nor must they be supportive of the language. Articles should be accompanied by as much visual material as possible (b/w or colour prints welcome). Unless otherwise specified, each item will be considered for publication as a personal statement by the author. The Editor accepts no responsibility for the contents of sustaining members' news, or advertising.

Please supply as much material as possible in machine-readable form, ideally as a simple ASCII text file or an HTML document. APL code can be accepted in workspaces from I-APL, APL+Win, IBM APL2 or Dyalog APL/W, or in documents from Windows Write (use the APL2741 TrueType font, available free from the website), and MS Word (any version, ideally using the Vector template which is also available from the website).

Membership Rates 2003–2004

Category	Fee	Vectors	Passes
UK Private	£20	1	1
Overseas Private	£22	1	1
(Supplement for Airmail, not needed for Europe)	£4		
UK Corporate Membership	£100	5	5
Overseas Corporate	£110	5	
Sustaining	£500	10	5
Non-voting Member (Student, OAP, unemployed)	£10	1	1

The membership year normally runs from 1st May to 30th April. Applications for membership should be made to the Administrator using the form on the inside back page of VECTOR. Passes are required for entry to some association events, and for voting at the Annual General Meeting. Applications for student membership will be accepted on a recommendation from the course supervisor. Overseas membership rates cover VECTOR surface mail, and may be paid in sterling, or by Visa, Mastercard or JCB, at the prevailing exchange rate.

Corporate membership is offered to organisations where APL is in professional use. Corporate members receive 10 copies of VECTOR, and are offered group attendance at association meetings. A contact person must be identified for all communications.

Sustaining membership is offered to companies trading in APL products; this is seen as a method of promoting the growth of APL interest and activity. As well as receiving public acknowledgement for their sponsorship, sustaining members receive bulk copies of VECTOR, and are offered news listings in each issue.

Advertising

Advertisements in VECTOR should be submitted in typeset camera-ready format (A4 or A5) with a 20mm blank border after reduction. Illustrations should be photographs (b/w or colour prints) or line drawings. Rates (excl VAT) are £250 per full page, £125 for half-page or less (there is a £75 surcharge per page if spot colour is required).

Deadlines for bookings and copy are given under the Quick Reference Diary. Advertisements should be booked with, and sent to Gill Smith, Vector Production, Brook House, Gilling East, YORK YO62 4JJ. Tel: 01439-788385.

Email: apl385@compuserve.com.

Contents

Editorial:

by Stefano Lanzavecchia

> «I'm wasted on cross-country. We Dwarves are natural sprinters. Very dangerous over short distances.»
>
> The Two Towers – *Peter Jackson's Film Version*

Many computer related meetings are hosted world-wide over the span of a year. We have recently reported on the first Dyalog User Group meeting, a welcome addition to the list and while the excitement had yet to subside, two more conferences took place, almost at the same time, in two different locations of the United States: the APL2000 User Group conference and Microsoft's renowned Professional Developer's Conference (though everybody knows it simply as the PDC) 2003. In this issue our readers will find Adrian Smith's notes on the former and will be delighted by the presence of a lot of interesting content showing, in particular, that the APL community is more and more opening to the outside world, trying to integrate a lot of useful concepts, some relatively new (Web Services), some old and well-established (regular expressions). But while APL used to lead the pack back in the days when the technology it used was so advanced to be hardly understood by those who struggled with long and boring Fortran source sheets, nowadays it seems that APL and APLers have pampered themselves for too long in their happy corner and have failed to notice that the corner was shrinking and the others were making progress while they stood still.

A critical mind will probably argue that there has been very little that can be defined revolutionary or even only *new* in the last twenty years of software and programming languages development. I have no reason to disagree and I'd rather let the wise lead me, but I don't think that the "been there, done that" approach is a useful one. Maybe APLers were so advanced back then that they could fruitfully use regular expressions before the other software developers could even get a compiled program to work. But they were also so advanced that they had the time to forget how useful regular expressions are and so advanced that today half of the APL community is hardly aware of their existence. Surely there is something wrong here. We have all written our own little remote procedure call library, so why should we bother with Web Services and SOAP? Yes, indeed: what good is there in a cross-platform standard, when all we are interested in is our own little

garden? Don't misunderstand me: I am not very fond of SOAP (yes, I do shower regularly) but its advocates have many good points there.

Maybe the world of computing is not going in the direction we would have chosen, but that is not a good reason to ignore all of what the others are doing. Some of it might turn to be quite useful. And this takes me to the PDC 2003: the Microsofties showed some skills in building a lot of expectation for what they were going to reveal and I don't think anybody was really disappointed when the veil was lifted. Instead of re-running an old consolidated show, they boldly stepped in and presented ideas for technological platforms in their early stages of development. While the new release of .NET may be shipped in the late 2004, the new flagship operating system, codenamed Longhorn, will not see the light for at least another 2 to 3 years. Of course, a developers' conference cannot be run only on PowerPoint presentations and all the attendees were given developer's previews of the various bits presented, to whet their appetites, but also to give them the possibility to prepare for what is going to happen, eventually. I don't think it's a good idea to get into the technical details but let me say that, once again, I am not so blinded by my enthusiasm for new technologies to fail to notice that most of them are just *old* ideas, refitted to live in our modern times. Yet, I don't see anything bad in this: if the idea is a good one and if so far it had been neglected, because of the lack of computing power (read: raw hardware performance), good algorithms or simply public exposure, I will welcome any time its revival in new clothes.

I am looking forward to the Longhorn wave because I can see where it's going and I like the look of it, but let me tell you about some recent development experiences of mine. In a couple of very small projects I decided to try my still improving .NET skills on the field. Even if .NET is a multilanguage platform, I had no doubts when it came to make choices: the best development tool available, namely the Microsoft Visual Studio.NET, and C# as its flagship language. The problems I had to solve were mostly scalar-oriented so I didn't feel too bad about forsaking APL for once. What I lost in conciseness, I gained a hundred-fold in completeness of the libraries I had at my disposal, in the advanced features of the source code editor (first in the list, a very well-thought auto-complete mechanism called IntelliSense), in the excellent debugging facilities, which almost made me forget I was working with a compiler and not an interpreter. I repeat: the tasks I solved were extremely simple, but I didn't fail to notice that the productivity boost over an APL solution of the same kind of quality was due to the vast amount of code I could borrow from public repositories in addition to the already extensive .NET framework, coupled with the quality of the development environment, cleared geared towards a pleasant developer's experience. Here's a list of the noteworthy features of my experiment: slick GUI, auto-update of the code from a

central repository, reading of tabular Excel spreadsheets without having Excel installed, disconnected cache of the data kept in a ZIP file without having any external ZIP manager installed, installation done copying all the files in a folder and nothing more, less than one week of overall development time for the first three releases, distributed seamlessly to the users via the auto-update mechanism. If not being able to do the same thing with my favourite APL interpreter means that I am not a good APL developer, then I am afraid I'll have to publicly declare that I am not a good APL developer. Still, I think that the conclusion that can be drawn from my experience remains valid: there is room for improvement in the APL world. Amazingly enough, the path traced by the recent User Group meetings is the right one. Let's stride forward.

Quick Reference Diary

Date	Venue	Event
7–9 May 2004	Stansted	APL-Moot at Great Hallingbury Entry for all or any part of the weekend: £20 on the door

An informal weekend of programming and socialising to enjoy ourselves and learn from each other. The people are the programme!

21st May 2004	London	BAA AGM and spring meeting
Nov 2004	Florida	APL2000 User's Meeting. Naples Beach Hotel, Florida.

Dates for Future Issues of VECTOR

	Vol.20 No.4	Vol.21 No.1	Vol.21 No.2
Copy date	5th March	*	10th Sept
Ad booking	12th March	*	17th Sept
Ad Copy	19th March	*	24th Sept
Distribution	April 2004	July 2004	October 2004

CORRESPONDENCE

Claude Henriod

December 2003

Dans mon enfance j'ai fait du morse (à bras puis de la radio-télégraphie militaire), de la cartographie, j'ai aussi étudié la météorologie pour passer 11 ans sur un aéroport.

Dès 1961 je me suis intéressé à la programmation, d'abord manuelle (KeySort) pour rentrer en 1963 chez IBM comme spécialiste assembleur et des systèmes (1401, 1440, 1130, 5110, s/360, etc...).

Ce n'est que depuis 1972 que j'ai fait la connaissance de l'APL (CMS), et ce qui m'a spécialement attiré c'est la concision du langage. C'est pourquoi je ne me suis jamais intéressé à J, j'avais assez à faire à offrir mes connaissances d'APL2 pour des applications (Assurances, Statistiques, etc...).

En 1982 nous avons créé un groupe APL en Suisse (SAUG) avant même l'arrivée de Vector. Nous avions un Journal et, en tant que Président pendant quelques années, j'ai conscience de l'effort (en hommes dévoués) qu'il faut avoir pour maintenir une revue.

Aujourd'hui je voudrais *remercier* toute l'équipe actuelle et passée du Comité BAA et du Groupe de travail lesquels nous procurent régulièrement (depuis 20 ans) un VECTOR de qualité.

Claude Henriod, chenriod@vtx.ch

[translated by Sylvia Camacho and Richard Smith]

In my youth I studied Morse (by hand and then using military radio-telegraphy), cartography and meteorology, over 11 years spent at an airport.

Since 1961 I have been interested in programming, first of all using KeySort cards and later returning to IBM in 1963 as a specialist in assembler and systems (1401, 1440, 1130, 5110, s/360, etc.). It was only in 1972 that I met APL (CMS), and what

particularly attracted me was the conciseness of the language. I never became interested in J, because my knowledge of APL2 was sufficient for applications (Insurance, Statistics etc.).

In 1982 we set up an APL group in Switzerland (SAUG), even before the arrival of Vector. We had a journal and, as President for some years, I am conscious of the effort (of dedicated people) necessary to keep a journal going.

So today I would like to *thank* all the members of the BAA Committee and the Working Group, past and present, who have regularly (for 20 years) brought us a VECTOR of quality Vector.

British APL Association AGM 2004 to be held at the Royal Statistical Society starting at 2 pm on Friday 21st May 2004

Agenda

1 Minutes of AGM 2003
2 Report from the Chairman (Adrian Smith)
3 Report from the Treasurer (Nicholas Small)
(including report from membership secretary)
4 Committee for 2004-2005
5 Appointment of Auditor
6 Questions
7 Any other business

Anthony Camacho, Hon Sec

News from Sustaining Members

MicroAPL Ltd

The latest release of APLX for Windows, Linux, MacOS, and AIX (*server edition only*) is Version 2.0. The highlights of the new release are built-in APL multi-tasking, a new Grid object for spreadsheet-like display and editing, new system functions and ⎕FTIE-style component files to assist migration from other APL interpreters, and support for Unicode. The new facilities include:

- Multiple sessions can now be open simultaneously, allowing you to run several APL tasks or browse one workspace whilst editing a function in another.
- Your APLX applications can now take advantage of multi-tasking. APL tasks can be run either as ordinary tasks with their own session windows, or as background tasks:

```
'Task2' ⎕WI 'New' 'APL' ('wssize' 100000) ('visible' 1)
```

- Tasks can share variables using the traditional IBM-style shared variable mechanism, or can share data through 'delta' properties of ⎕WI objects.
- Tasks can send signals to each other, for example to start a calculation or to indicate that a result is available. Parent tasks can also read or modify the session-window contents of a child tasks, and be signalled when the child task is awaiting input, about to execute, or finished executing.
- You can dynamically change workspace size using)CLEAR N, where N can be expressed in bytes, KB, MB or (if you have enough memory) GB.
- The interpreter has been enhanced to increase the maximum rank of an array to 63 and to remove limits on the size of executed token strings.
- New system functions include ⎕FI and ⎕VI for numeric to character conversion, ⎕PFKEY to allow programming the function keys, and ⎕TCxxx for compatibility with APL+Win.
- To supplement the existing component file system, APLX now also supports ⎕FTIE-style component files, for compatibility with other APL interpreters. As well as the standard features and multi-user/multi-tasking support, APLX also allows insertion or deletion of components anywhere in the file. Unique tie

numbers can be allocated automatically by APLX if required. APLX component files are of course 100% compatible across Windows, MacOS, AIX and Linux.

- Unicode support includes Cut/Paste as Unicode, ⎕UCS to convert APLX character arrays to Unicode and vice versa, and the `unitext` property of the System object to access the Clipboard as Unicode. The native file system has also been enhanced to read or write Unicode text and byte-reversed Unicode text.

- Full path names are now allowed in system commands.

- The new built-in ⎕WI Grid object is a spreadsheet-like control with text or numeric cells. It can be used for display, user-editing, or both, and is programmed using a natural syntax, compatible with other ⎕WI controls. You can set and read cell arrays directly as APL arrays.

- A number of other new ⎕WI facilities are provided, including the `fonts` property of the System and Printer objects to list available fonts. There are also enhancements to the `Draw` method, and to the Windows OCX/ActiveX/OLE interface.

For full details, visit our website. And please keep the suggestions coming - we are ready to hear what you would like to see in the next release!

HMW Systems Ltd

Another year, ho hum. Not a lot has changed for us: we're still plodding on with assorted maintenance agreements; we're still selling our dealing systems; we're still developing our range of APL products. We are, of course, still trying to find more customers!

We didn't get many takers for our offer to sell your software, so I suppose that most of your applications aren't really suitable for the "stack 'em high and sell 'em cheap" approach of an Internet software shop. HMW has branched out into the area of web site hosting (and yes, we too are Wiki fans, although a different flavour to that found on the Vector web site), so in a spirit of generosity we'll make another offer – does anyone want or need reasonably priced web hosting? This isn't intended as a commercial offer; we'll pass on our costs, but won't attempt to make swinging profits out of anyone. It's intended to be in the same air of cooperation that seemed to flow from the Dyalog conference, to raise the profile of APL and APLers on the Internet.

If anybody is interested, please contact us.

The Vector Product Guide

compiled by Gill Smith

VECTOR's exclusive Product Guide aims to provide readers with useful information about sources of APL hardware, software and services. We welcome any comments readers may have on its usefulness and any suggestions for improvements.

We reserve the right to edit material supplied for reasons of space or to ensure a fair market coverage. The listings are not restricted to UK companies and international suppliers are welcome to take advantage of these pages.

For convenience to readers, the product list has been divided into the following groups ('poa' indicates 'price on application'):

- Complete Systems (Hardware & Software)
- APL and J Interpreters
- APL-based Packages
- Consultancy
- Other Products
- Overseas Associations
- Vendor Addresses
- World Wide Web and FTP Sites

Every effort has been made to avoid errors in these listings but no responsibility can be taken by the working group for mistakes or omissions.

We also welcome information on APL clubs and groups throughout the world.

Your listing here is absolutely free, will be updated on request, and is also carried on the Vector web site, with a hotlink to your own site. It is the most complete and most used APL address book in the world.
Please help us keep it up to date!

All contributions and updates to the Vector Product Guide should be sent to: Gill Smith, Brook House, Gilling East, York, YO62 4JJ. Tel: 01439-788385, Email: apl385@compuserve.com

APL INTERPRETERS

COMPANY	PRODUCT	PRICES(£)	DETAILS
APL Borealis Inc.	Dyalog APL	poa	Distributor of Dyalog APL products from Dyadic
	APL2000	poa	Distributor of APL2000 products
APL Systems IDC SL	APL2000 APL interpreters and toolspoa		Distributor of APL2000 products for the UK Consulting and maintenance for APL applications
Beautiful Systems	Dyalog APL/W for Windows	poa	US Distributor of Dyalog APL products from Dyadic.
	Dyalog APL for Unix	poa	See Dyadic listing for product details.
Dinosoft Oy	Dyalog APL/W for Windows	poa	Finnish distributor of Dyalog APL products.
	Dyalog APL for Unix	poa	See Dyadic's listing for product details.
Dittrich & Partner	APL+Win	poa	Cognos/APL2000 Inc products
	Dyalog APL	poa	Dyadic Systems Ltd. products
	IBM APL products	poa	
Dyalog	Dyalog APL for DOS/386	995	Second generation APL for DOS.Runs in 32-bit mode, supports very large workspaces. Unique "window-based" APL Development Environment and Screen Manager. Requires 386/486 based PC or PS/2, at least 2Mb RAM, EGA or VGA, DOS 3.3 or later.
	Dyalog APL/W for Windows	995	As above, plus object-based GUI development tools. Requires Windows 3.0 or later.
	Dyalog APL for Unix	995-12,000	Second generation APL for Unix systems. Available for Altos, Apollo, Bull, Dec, HP, IBM 6150, IBM RS/6000, Masscomp, Pyramid, NCR, Sun and Unisys machines, and for PCs and PC/2s running Xenix or AIX. Oracle interface available for IBM, Sun and Xenix versions.
DynArray	DICE for Windows	poa	Software development kit which includes an APL interpreter as a DLL and the ability to run and link existing and new APL code to non APL code such as VB, C/C++, Java and integration with various Windows software applications and database packages such as MS Office.
I-APL Ltd	I-APL/PC or clones	8	ISO conforming interpreter. Supplied only with manual (see 'Other Products' for accompanying books).
	I-APL/BBC Master	8	As above
	I-APL/Archimedes	8	As above
IBM APL Products	TryAPL2	free	APL2 for educational or demonstration use. Write, fax or Email to APL Products; specify disk size desired.
	Workstation APL2 V2 Version 2	$1500	AIX, Linux, Solaris, Windows Product 5724-B74, Part Number 45P7514
	APL2 Version 2	poa	Product No. 5688-228. Full APL2 system for S/370 and S/390
	APL2 Application Envt Vn2	poa	Product No. 5688-229. Runtime environment for APL2 packages
Insight Systems	Cognos/APL2000 Inc	poa	Leading distributor of APL2000 products in Denmark
	Dyalog Ltd.	poa	Leading distributor of Dyalog APL products in Denmark
	IBM	poa	Leading distributor of IBM APL & GraphX products in Denmark
J Austria	J	poa	Distributor for Austria and Switzerland
	Dyalog APL	poa	Distributor
	Causeway Products	poa	Distributor
	Structural Analysis Software	poa	Complete package by IG Zenkner&Handel to perform structural analysis/engineering calculations. Also suitable for dynamic problems, e.g. earthquake simulation.
JSoftware Inc.	J on the Web online registration ...		
	J Professional (online reg.)	$895	includes manual set and one year of updates
	J Standard (online reg.)	Free	Free for download only

	Books and accessories		
	J Dictionary	$50	
	J Phrases	$50	
	J Primer	$50	
	Fractals, Visualization and J	$80	
	Concrete Math	$40	
	Exploring Math	$50	
Lescasse Consulting	APL+PC	poa	Lescasse Consulting is the exclusive APL2000 distributor in France and also distributes in Switzerland and Belgium. Call for price quotes.
	APL+Unix	poa	
	APL+DOS	poa	
	APL+Win	poa	
	Dyalog APL/W	poa	French distributor for Dyalog
MasterWork Software	Manugistics Products and ISI	poa	New Zealand distributor
MicroAPL	APLX for Windows/MacOS	499	Cross-platform APL development environment with GUI programming facilities. Interpreter modelled on APL2. Available for Windows 95/98/ME/NT/2000/XP, Mac OS 9 and Mac OS X.
	APLX Server Edition	poa	For running large multi-user APL applications on x86 Linux, RS/6000 AIX, and Windows NT/2000.
	APLX for Linux (commercial)	499	Linux desktop edition of APLX, compatible with Windows and MacOS versions, with full development environment and GUI programming facilities. Runs under RedHat, Debian, Mandrake and SuSe Linux distributions.
	APLX for Linux (personal)	Free	Full version of APLX, can be downloaded free for personal use.
Oasis	Dyalog APL	poa	Dyalog Ltd
	APL*PLUS	poa	Manugistics
	APL.68000	poa	MicroAPL Ltd
	APL2	poa	IBM
Omega	Zero	poa	A "small simple and fast" alternative to APL
Optima	Dyalog APL/W	995	Dyalog's best, leading edge APL interpreter
RE Time Tracker Oy	APL+PC (APL*PLUS/PC)	poa	Complete APL+ and Statgraphics product range and links to various 3rd party products.
	APL+DOS (APL*PLUS II)		
	APL+Win (APL*PLUS III), APL+Link		
	APL+UNIX		
	APL*PLUS Sharefile		
Soliton Associates	SHARP APL for OS/390	poa	for IBM OS/390 mainframes
	SHARP APL for UNIX	poa	for SunOS and IBM AIX
	SHARP APL for Linux	poa	for Intel Linux
Strand Software	Canada		
	All APL*PLUS products	poa	All APL*PLUS products including upgrades and educational.
	Dyalog and JSoftware products	poa	
	USA		
	Dyalog and JSoftware products	poa	

APL PACKAGES

COMPANY	PRODUCT	PRICES(£)	DETAILS
ADAPTA Software	MPS	poa	Master Production Scheduling
	FBS	poa	Forecasting and Budgeting System
	DRP	poa	Distribution Requirements Planning
Adaptable Systems	FLAIR	poa	Finite loader and interactive rescheduler. Customisable full-function scheduling system. (Available outside Australia by special arrangement only.)
Adaytum	Adaytum e.Planning	poa	Adaytum e.Planning offers a Web-based solution that combines planning, forecasting, budgeting, modelling and reporting in a single, integrated application.
APL Software\Services	APL Utilities	poa	Software: mostly .AWS for DOS, utilities for most APL interpreters. Public domain APL*Plus v10 with on-screen documentation and interactive tutorials. APL Conference Software. Books: APL user manuals for STSC, IBM, and Sharp. Request email catalog from dick.holt@juno.com.
Beautiful Systems	ASF_FILE	$399	Dyalog APL/W auxiliary processor for access to APL*PLUS/PC APL component files (*.ASF).
	NAT_FILE	$299	Dyalog APL/W auxiliary processor which emulates the APL*PLUS/PC quad-N native file subsystem for access to the DOS file system.
	DBF_FILE	$299	Dyalog APL/W auxiliary processor for efficient block mode access to dBASE format files. Designed to get large amounts of data in and out of dBASE. Not suited for random access to small amounts of data (it does not handle keys).
	SF_READ	poa	Dyalog APL/W functions to read APL*PLUS data objects of any type or structure from *.SF style component files created by APL*PLUS II or III.
Causeway	*CausewayPro* for Dyalog/W	400	Causeway application development platform for Dyalog APL/W.
	RainPro Business Graphics	250	The ultimate graphics toolkit for the APL developer. Adds 3D charting capability, Web publishing and clipboard support to the shareware product. Charts can be included in *NewLeaf* reports. Functionally compatible across Dyalog/W and APL+Win.
	NewLeaf for Dyalog and +Win	400	Frame-based reporting tool with comprehensive table-generation and text-flow support. Offers multiple master-page capability, bitmap wrap-around and on-screen preview with pan and zoom. Fully supported on Dyalog/W and APL+Win
Cinerea AB	ORCHART	250	Organization chart package for IBM APL2/PC. Full & heavily commented source code included - free integration into other applications. NB: ASCII output with line-drawing (semi-graphic) characters for boxes.
CODEWORK	3-way TANGRAM	poa	3-way TANGRAM is a DSS-OLAP product, basically a powerful and versatile handler of multi-way tables (also known as hypercubes). It entails 46 analysis modules, including computations, cube merging, sensitivity analysis, time intelligence, free format queries, HTML and LaTeX outputs. Current version is 7.0 At the time of writing, the product is available on Dyalog APL 8.3.1 for Windows 95/98/NT. Future plans include an APL+WIN version and later a LINUX version.

CoSy	K.CoSy	$30 % yr sub	K.CoSy is a general purpose computing and programming environment constructed, all in open code, in Arthur Whitney's very high level, yet structurally transparent Array Programming Language, K , and its tightly coupled User Interface. K.CoSy is an extremely productive environment in one of the most powerful and fastest of APL's progeny, and therefore, likewise, of all languages. K.CoSy provides a workspace-like interactive development environment previously impossible in K. Because of its unique open construction within the language itself, this environment is clearly competitive in a large domain with the APLs from the other vendors. K.CoSy notepad nature, interactivity, and open K code vocabulary make learning Arthur Whitney's K far less daunting and far more productive than its raw console, or any external scripting method. If you are a client of Kx Systems , or are investigating the possibilities, Contact us. See CoSy/K/CoSy for more information.
DynArray	DynaWeb Server	poa	A web server providing web based access to applications running on the DICE interpreter from DynArray, or on an IBM mainframe running APL2.
	DynaHarry	poa	A DSS system which offers the next generation capabilities for current APLDI, IC/E and IC/1 users. It comes with ROLAP capabilities, multisystem access to a wide variety of databases and data warehouses.
	DynaLink	poa	An ODBC client interface for DICE and IBM APL2 programs.
Entropy Software	Personal Information Management	poa	Entropy specialises in Personal Information Management Systems (PIMS) which run on a PC. Most systems are inadequte in some way. This is an attempt to fulfil all the needs of George MacLeod who has been an unhappy user of a variety of systems for many years. The product is being developed in modules (Calendar, Address List, To Do List, etc) which will be available at no charge. When all the modules are complete they will be integrated together. There will be a charge for the integrated product.
HMW	4XTRA	poa	Networked, Windows/Unix based Front End and Middle Office Foreign Exchange and Money Market Dealing System. Scalable from 1 user to 120+.
	Inca	poa	Software Change Management System. Enables the user to co-ordinate development work from several sources, resolve clashes, promote work items for testing and configure releases to a live environment.
	Maya	poa	APL code file manager. A comprehensive suite of tools giving a multi-window IDE style interface to file based APL code. Offers features such as copying from file to file, object comparison, string search, style formatting, hot-spot editor for filed objects (including variables), etc.
	Aztec	poa	System shell for APL development. Manages real-time and batch applications across multiple platforms. Offers standardised error trapping, job scheduling, task communication and recovery/restart features.
	Olmec	poa	APL GUI environment, providing menu bar, tool bar, status area, navigation sidebar (with treeviews & listviews)and client area. All are configured by simple text files and require no programming. Client area has a "tab wizard" option to provide ordered transaction processing
	Nazca	poa	fast, flexible and reliable static database and editor.
I-APL Ltd	Educational workspaces	5	PC format disks with the examples from: Thomson. Espinasse (Kits 1-4), Kromberg, Jizba & FinnAPL. All the examples to save your fingers!
IBM APL Products	A Graphical Statistical System	$250	for DOS, Product Number 5764-009
Insight Systems	Causeway	poa	Leading distributor of Causeway products in Denmark

All our old products are now either OEM'd, in the public domain, out of date, or all of the above. We'll be back!

JAD Software	JAD SMS	poa	JAD SMS is a multi-user software management system for Dyalog APL™ based on shared, hierarchical databases. JAD SMS databases let you keep historical versions of apl items as well as attributes such as timestamp, user name and documentation. The software includes a graphical user interface as well as specialized functions for inclusion in applications. No charge for single-user version; $100/user for multiple users
Lescasse Consulting	APL+Win Monthly Training	$600	Download 50+ page document about APL+ programming each month. You also get one or more workspaces full of re-usable APL code and sometimes additional files or products.
	Advanced Windows Programming	$95	200-page book plus companion disk on interfacing APL and Delphi. Contains full coverage of Delphi-2, +Win and Dyalog.
	DLL parser for APL	$250	Parse any Visual Basic DLL declaration file into a set of quadNA definitions. Turn constants and structures into APL variables. Available for APL+Win and Dyalog/W.
	Delphi Forms Translator	$195	Design forms with Delphi and turn them automatically into APL programs which recreate the same form (+Win and Dyalog/W).
	APL+Link Pro	poa	ODBC interface for APL+Win
	SQAPL Pro	poa	ODBC interface for Dyalog APL/W
	RainPro	poa	Highly customisable 2D and 3D publication graphics for APL+Win and Dyalog APL/W
	NewLeaf	poa	Page layout and printing tools for APL+Win and Dyalog
	GraphX and ChartFX	poa	High-quality business graphics for APL+Win
	Formula One and Dyalog APL	$95	100-page book + companion disk on how to use the Formula One VBX with Dyalog APL/W
Lingo Allegro	FACS	poa	EMMA-like interface to DB2 or ODBC databases
	QWIN	poa	Legacy DOS Windowing support for APL+Win
	ODBC/127	poa	IBM AP127-like ODBC Interface for APL+Win and Dyalog APL/W
Optima	ServiceLine	poa	A property management system which keeps a record of all outstanding tasks, produces an up to date list of work to-do and Scheduled reminders plus automated standard letters and basic financial control.
	TravelLine	poa	A system designed to control the workload allocation for a fleet of chauffeur vehicles plus a reminder system for fleet management and Local Authority requirements.
	BPA	poa	"Brand Performance Analysis" allows for the modelling of product/brand performance over time and comparison with competitor products.
	DBI	poa	"Database Interrogator" allows for the non technical user to ask sensible questions of a large database such as a questionaire and obtain results tables and graphs quickly, easily and accurately.
Qualedi	Qualedi	$850-$5,500	Electronic Data Interchange (EDI) translation software for the PC, with strict compliance checking.
RE Time Tracker Oy	UIT/W	poa	Comprehensive high-level Windows User Interface library for APL+Win and +II v 5.1. Comprehensive spreadsheets, replicated fields, special field types, etc. 16 and 32 bit versions available.
	AJGRAPH	poa	Graphpak-compatible 2D graphics package for +Win and +DOS. Includes multi-window support, print and metafile support. No DLLs required.
	ECCO PRO with APL	poa	Leading group and personal information management system with comprehensive customising. Supplied with sample +Win workspace to interface to ECCO databases via DDE.
	NEWT TCP/IP SDK with APL	poa	Lead TCP/IP SDK with interfaces to all protocols. Supplied on 3 CD ROMS together with a sample +Win workspace.

	DB+	poa	Database interface for APL+DOS under Windows. Allows combining character-based APL applications with ODBC-compliant databases such as Oracle and SQL-server.
Warwick University	BATS	250	Menu driven system for time series analysis and forecasting using Bayesian Dynamic modelling. Price is reduced to £35 for academic institutions.
	FAB	free	Training program for the above.
Weighahead Systems	Weighahead Windows Weighing System (3WS)	poa	Recipe Weighing System for Manufacturing Industries. Pharmaceutical, Cosmetics, Foods etc. Works without keyboard or mouse. Uses Electronic Balances, Laser scanners, bar codes and label printers.
Zark	APL Tutor (PC)	$299	APL computer-based training. Available for APL*PLUS PC & APL*PLUS II. Demo disk $10.
	APL Tutor (MF)	$5000	Mainframe version.
	Zark ACE	$99	APL continuing education. APL tutor news and hotline phone support.
	APL Advanced Techniques....	$59.95	488pp. book, (ISBN 0-9619067-07) including 2-disk set of utility functions (APL*PLUS PC format).
	Communications	$200 pc, $500 mf	Move workspaces or files between APL environments.

APL CONSULTANCY AND DEVELOPMENT

COMPANY	PRODUCT	PRICES(£)	DETAILS
Adfee	Consultancy	poa	Development, maintenance, conversion, migration, documentation, of APL products in all APL environments
Andrews	Consultancy	poa	APL programming and analysis, algorithms, tree processing and design programs for craft work.
APL Borealis Inc.	Support and Development	poa	APL Software Support and Development. Specialists since 1979 in Sharp APL, APL*Plus, APL+Win, Dyalog APL
APL Solutions Inc	Consultancy	poa	APL systems design, development, maintenance, documentation, testing and training. Providing APL solutions since 1969.
APL Systems IDC SL	Consultancy	poa	Consulting and maintenance for APL applications.
AUSCAN Software	Consultancy and Training	poa	APL software development, training
Camacho	Consultancy	poa	Manuals; feasibility reports and estimates; analysis and programming; APL and MS Windows applications; Sharp, ISI APL, APL*PLUS, APL2/PC and other APLs spoken. Fixed price systems a speciality
Ian Clark	Consultancy	poa	Interfacing APL, VB, C/C++, ActiveX/COM. Screen design and documentation. National language porting.
Ray Cannon	Consultancy	poa	APL, C, Assembler, Windows, Graphics: PC and mainframe
Causeway	Consultancy and Training	poa	On-site training for Causeway, RainPro and NewLeaf. Customisation and enhancement to meet local needs. Code review and pre-implementation check of Causeway applications.
Paul Chapman	Consultancy	250-500	24-hour programmer: APL, Smalltalk, C; Windows front end design a speciality.
CODEWORK	Consultancy	poa	Development, maintenance, migration, documentation of APL applications. Speciality: info systems for top executives, internet applications.
CoSy	Consultancy	poa	CoSy.com, Coherent Systems, provides rapid development in the K language and associated data base products, with a particular interest in quantitative (financial) modelling.
David Crossley	Consultancy	poa	Experienced in large APL system developments since 1969 for PC or mainframe.
Dinosoft Oy	Consultancy	poa	Specialised in very large databases.

Dittrich & Partner	Consultancy	poa	APL programming and analysis; APL workshops and training on the job
Dyalog	Consultancy	poa	APL and Unix system design, consultancy, programming and training.
DynArray	Consultancy	poa	DynArray offers consulting in the areas of DSS, Y2K and APL programs upgrade/conversion to modern Web enabled platforms.
Evestic AB	Consultancy	poa	Excellent track record from 15+ years of APL applications in banking, insurance, and education services. All dialects, platforms and project phases. SQL expertise.
First Derivative Analytics Ltd.	Consultancy	poa	Analysis, design, prototyping, development & testing of APL (especially financial) applications: Sharp, Dyalog APL/W.
General Software	Consultancy	from 200	Over 20 years experience with every version of APL, large mainframe systems and small PC based programmes.
Godin London Inc	Software Development	poa	We have applications in the food manufacturing field, travel agency and airline bookings field and in product lease management.
HMW	Consultancy	poa	System design consultancy, programming. HMW specialize in banking and prototyping work.
Hoekstra Systems	Consultancy	poa	APL consultancy, programming, etc. Also UNIX system administration
Michael Hughes	Consultancy	poa	APL consultant with 20 years experience with all versions of APL. I can create your dynamic Web sites using the full power of APL working with Microsoft IIS (Internet Information Service) on Windows NT or 2000. I also undertake System design, Programming and Maintenance on all platforms, particularly MS Windows.
INFOSTROY	Consultancy	poa, competitive	Broad experience in various APL platforms. Special skills and knowledge in developing complex applications for investment, financial and construction markets. Implementation of hybrid solutions based on APL, Delphi, C++, VBA, SQL servers.
Insight Systems	Consultancy	poa	We have experience with just about every APL system and platform in common use during the last 20 years, from SHARP APL under MVS or Linux to APL+Win and in particular Dyalog APL under Windows 9x, NT or 2000. If you have decisions to take about adapting your APL application to take advantage of emerging technologies, or would like your strategy reviewed, give us a call. We have extensive experience in all areas of APL development, from legacy systems, up, down and sideways migrations, to the development and support of shrink wrapped solutions based on APL. Even if we don't have time to do the work ourselves, we will know where to find someone who is an expert in your version of APL and your application area, on your continent.
JAD Software	Consultancy	poa	Systems design and development, project management, technical manuals, financial and actuarial expertise in APL.
KJK	Consultancy and software development	poa	APL-based data management: conversions, ad hoc-analyzing tools, well-interfaced methods for defining, processing and browsing of multi-dimentional reports. Rapid custom software development based on proven modular toolset approach.
Lambent Technolgy	Consultancy	poa	APL programming, consulting & training; web design and construction.
Phil Last	Consultancy	poa	APL consultancy, modelling and programming.
Lescasse Consulting	Consultancy	poa	A range of consultants, experts in Windows programming, with APL+Win and Dyalog APL/W. More than 100 major APL applications already developed. We all have additional expertise in Formula One and Delphi.
Lingo Allegro	Consultancy	poa	General APL consulting, internet website development, migration and downsizing, performance tuning, education and training.
Lucas Solutions	Consultancy	poa	Rates depend on task and location.

Mackay Kinloch Ltd	Consultancy	poa	Design, analysis and programming for banking, insurance and pensions, financial planning and modelling, corporate performance and legal reporting
MasterWork Software	Consultancy	poa	Consulting and J programming for econometrics and statistics in public policy, health and food industries.
Milinta Inc	Consultancy	poa	Design, development, maintenance, conversion, documentation in all APLs, most APs and some specific Sharp products (LOGOS, ViewPoint, Retrieve). Experience in multi-user, multi-task systems, databases, Windows programming.
Ellis Morgan	Consultancy	poa	Business Forecasting & APL Systems.
Oasis	Consultancy	poa	Expertise in APL system design, Project management, conversion, migration, tuning; for all APL versions (10+ years experience)
Omega Computing	Consultancy	poa	APL consultancy, programming, etc.
Optima	Consultancy, Training & Development	poa	We have been in business since 1990 and have a team of APL professionals with many years experience in pharmaceutical, industrial and financial systems on both PC and Mainframe platforms. In addition to pure APL we can also offer assistance with network integration, ACT! customisation and issues concerning the Internet and/or web design.
RadSys Technologies	Consultancy	poa	Areas of expertise: financial systems, risk analysis systems, healthcare systems.
Resources & Results	Consultancy	poa	Knowledge management company builds decision support, data warehouse, data mining and strategic planning systems for CFO's, CEO's, and senior management, using our Rapid Application Development (RAD) methods and tools. Extensive experience in large-scale system development and ad hoc executive support for Fortune 500 clients.
RE Time Tracker Oy	Consultancy	poa	APL application conversions, APL Windows interfaces, APL to API-level interfacing to any system under Windows, TCP/IP network and database connectivity.
Rex Swain	Consultancy	poa	Independent consultant, 25 years experience. Custom software development, PC and/or mainframe.
Graeme Robertson	Consultancy & Training	250-500	Mathematical, statistical and financial APL runtime systems designed, written from scratch and documented. Legacy systems upgraded. 20 years experience covering most APLs (esp. Dyalog) on most platforms (esp. Windows). Particularly interested in dynamic-web or Mathematica applications.
Rochester Group	Consultancy	poa	Specialise in MIS using Sharp APL
Shepp & Associates	Consultancy	poa	APL applications development and consulting, especially in the travel industry, especially on small computers. 25 years experience in APL programming.
Snake Island Research Inc	Consultancy	poa	APL interpreter and compiler enhancements, intrinsic functions, performance consulting. APL parallel compiler APEX is giving very good initial performance tests with convolution somewhat faster than FORTRAN.
SovAPL	Consultancy	poa	Offshore APL development service.
Strand Software	Consultancy	poa	Advice on migrating to and from all flavours of APL and hardware platforms. Full-screen interface implementation, APL utilities, benchmarking, efficiency analysis, actuarial software, system development tools, valuation, pricing and modelling systems.
Sykes Systems Inc	Consultancy	poa	Complete APL services specialising in audit, optimisation and conversion of APL systems. Excellent design skills. All dialects and platforms. 17-23 years experience.
Weighahead Systems	Consultancy	poa	Specialising in industrial systems. Links to PLCs, laser scanners, bar codes, weigh scales, label printes etc. Also programmable hand held scanners.
Stephen Wynn	Consultancy	poa	Most experience of financial planning, and mathematical areas: operational research, quality control, experimental design.

OTHER PRODUCTS

COMPANY	PRODUCT	PRICES(£)	DETAILS
Adfee	Employment	poa	Contractors and permanent employees
APL-385	Typefaces	poa	Variants of the APL2741 typeface available to specification.
APL Borealis Inc.	APL Training	poa	Hands-on courses in Introductory, Intermediate, Advanced and Windows APL. Courses are customized and flexible, and may be delivered on-site, with strong emphasis on methods for efficient and maintainable APL systems development.
Atomic Vector	RELCELL	poa	Details: Report Writer, Data Mining, Database Visualization, and Export to Excel from relational databases. Offered in English, Spanish, Italian, Portugese, French and German. With this product, you don't have to know English in order to use SQL. Point and click with easy to use Wizards.
	APL Games Volume 1	$15	Details: Volume 1 - A collection of four APL games on cd, saved from past yearly issues of the APL Planet. Great way to demonstrate the capabilities of APL, and to entice the younger generations into APL. Written entirely in Dyalog APL.
ComLog	Comic-Logger	$25.95+p&p	APL*PLUS II comic-book inventory system. Shareware version available on America OnLine.
HMW	Employment	poa	Contractors and permanent employees placed.
I-APL Ltd	Books	poa	I-APL stocks books written to go with the I-APL interpreter and some APL Press books. For a list write to 11 Auburn Road, Bristol BS6 6LS, ring 0117 973 0036 or email acam@blueyonder.co.uk.
Oasis	Training	poa	Introductory courses in APL Advanced courses for different APL versions
Optima	IBM Compatible	poa	Complete business solutions including hardware, software and support plus additional network installation and maintenance if required.
Renaissance Data Systems	Booksellers		The widest range of APL books available anywhere. See Vector advertisements.
Right Seat Software	Vox Proxy	$199 (comm) $69.95 (edu)	Vox Proxy is authorware for PowerPoint(r) 2000 or 2002 which allows the use of Microsoft Agent Technology (3D talking animated characters) within slide shows. VP appears on PowerPoint's main menu and provides editing side-by-side with slides. Automated script-writing provides control of PowerPoint, allowing the use of characters for live presentations or fully-automated tutorials, demos, or training programs. Optional CD Prep program allows the user to create auto-starting CD's that will play on any version of PowerPoint or without PowerPoint.

OVERSEAS ASSOCIATIONS

GROUP	LOCATION	JOURNAL	OTHER SERVICES	Ann.Sub.
ACM SigAPL	International	APL QuoteQuad	Conferences; APL white pages; web site	$30
APL Bay Area	USA N. California	APLBUG	Monthly Meetings (2nd Monday)	$20
APL Club Austria	Austria	-	Quarterly Meetings	200AS(indiv), 1000AS(corp)
APL Germany e.V.	Germany	APL Journal	Semi-annual meetings	DM60
Ass. Francophone pour la promotion d'APL	France	Les Nouvelles d'APL		FF350 (private) FF2800 (Company)
BACUS	Belgium	APL-CAM	Conferences & Seminars	£18 ($30)
Danish SIG	Denmark			
Dutch APL Assoc.	Holland	Vector provided	Mini-congress, APL ShareWare Initiative	
FinnAPL	Helsinki, Finland	FinnAPL Newsletter	Seminars on APL	100FIM(private), 30(student), 1000 (Co)
Japan APL Assoc	Tokyo	APL Journal	Monthly meetings (4th Sat)	10,000yen to join
NY SigAPL	New York, USA	Big Apple APL	Monthly meetings	$35/$25(ACM)

Rome/Italy SIG	Roma, Italy			
SE APL Users Grp	Atlanta, Georgia	The APL Planet	Semi Annual meetings	$13 (US), $25 (non-US)
SovAPL	Moscow, Russia	-	Seminars and Annual Meeting	
SwedAPL	Sweden	SwedAPL Nytt	Semi-annual meetings, seminars	SEK 75
SWAPL	Texas, USA	SWAPL		$18
Swiss APL (SAUG)	Bern	Part of Qtly SI-Info		SF60 (SI) + SF20 (SAUG)
Toronto SIG	Toronto, Canada		Occasional meetings, APL Skills Database, Toronto Toolkit	

ADDRESSES

ORGANISATION	CONTACT	ADDRESS, TELEPHONE, FAX, EMAIL etc.
ACM SigAPL	David Siegel	ACM, 1515 Broadway, 17th Floor, New York, NY 10036, USA (Subs only)
ADAPTA Software GmbH	Michael Baas	Stellinger Weg 19, 20255 Hamburg, Germany. Tel: +49 40 40170951 Fax: +49 40 40170952. Email: info@adapta.de
Adaptable Systems	Lois & Richard Hill	49 First Street, Black Rock 3193, Australia. Tel: +61 3 9589 5578 Fax: +61 3 9589 3220 Email: hillrj@melbpc.org.au
Adfee	Bernard Smoor	Dorpsstraat 50, 4128 BZ Lexmond, Netherlands. Tel +31 347 342 337 Fax: +31 347 342 342 Email: adfee@concepts.nl
Andrews	Dr Anne D Wilson	12 Thorny Hills, Kendal, Cumbria LA9 7AL, UK. Tel: 01539-731205 Email: ADWilson@kencomp.net
APL-385	Adrian Smith	Brook House, Gilling East, York YO62 4JJ, UK. Tel: 01439-788385 Email: apl385@compuserve.com
APL2000 (Europe)	Fred Honea	see APL Systems IDC SL.
APL Bay Area APLBUG	Curtis Jones (Sec)	228 South 15th Street, San Jose, CA 95112-2150, USA Tel: +1 (408) 292-4060 Email: jonesca@vnet.ibm.com
APL Borealis Inc.	Richard Procter	381 Manor Road East, Toronto, Ontario M4S 1S7, Canada. Tel: (416) 457-7828. Fax: (416) 482-6582 Email: info@apiborealis.com
APL Club Austria	Harald F. Nelson	c/o N-TECH, Siebenbrunnenfeldg. 4-6, A-1050 Wien, Austria. Tel: +43 1 5458063 Fax: +43 1 5458063-17
APL Germany e.V.	Dieter Lattermann	Rheinstraße 23, D-69190 Walldorf, Germany. Tel: +49 6227-63469. Email: Dieter_Lattermann@t-online.de
APL Software\Services	Dick Holt	3802 N Richmond St, Suite 271, Arlington, VA 22207 USA Tel: +1 (703) 528-7624; Fax: +1 (703) 528-7617; Email: dick.holt@juno.org
APL Solutions Inc	Eric Landau	1107 Dale Drive, Silver Spring, MD 20910-1607 USA Tel: +1 (301) 589-4621 Fax: +1 (301) 589-4618 Email: aplsi@starpower.net
APL Systems IDC SL	Fred Honea	Alfredo Marquerie, 12 - 2 F, 28034 Madrid, Spain. Tel: +34 91 730 7008 (Office) +34 60 680 5949 (Mobile). Fax: +1 775 743 6131. Email: uksales@apl2000.net
Association Francophone pour la promotion d'APL	Ludmila Lemagnen	174 Boulevard de Charonne, F-75020 Paris, FRANCE Email: lemagnen@aol.com
AtomicVector Productions Inc.	John Manges	413 Comanche Trail, Lawrenceville, GA 30044, USA. Tel: +1 (770) 972-3755 Email: E-mail: mailto:seapldoc@aol.com
AUSCAN Software Ltd	Richard Procter	PO Box 39, Mansfield, Ontario L0N 1M0 Canada Tel: +1-705-434-1239 Email: rjp@ca.inter.net
BACUS	Joseph De Kerf	Rooinberg 72, B-2570 Duffel, Belgium. Tel: +32 15 31 47 24
Beautiful Systems, Inc.	Jim Goff	308 Old York Road, Suite 5, Jenkintown, PA 19046, USA Tel: +1 (215) 886-2636; Fax: +1 (215) 886-4888 Email: BeautifulSystems@goffs.net
Camacho	Anthony Camacho	11 Auburn Road, Redland, Bristol BS6 6LS, UK. Tel: 0117-973 0036. email: acam@blueyonder.co.uk
Ray Cannon		21 Woodbridge Rd, Blackwater, Camberley, Surrey GU17 0BS, UK. Tel: 01252-874697 Email: ray_cannon@compuserve.com
Causeway Graphical Systems Ltd	Adrian Smith	The Maltings, Castlegate, MALTON, North Yorks YO17 7DP, UK. Tel: 01653-696760 Fax: 01653-697719 Email: adrian@causeway.co.uk

Paul Chapman		51B Lambs Conduit Street, London WC1N 3NB, UK. Tel: 020 7404 5401. Compuserve: 100343,3210
Cinerea AB	Rolf Kornemark	Box 61, S-193 00 Sigtuna, Sweden. Tel/Fax: +46 859 255 421 Email rolf@cinerea.se
Ian Clark	Ian Clark	1205 Deer Creek Drive, Plainsboro, NJ 08536, USA. Tel: +1 609 716 8832 Email: earthspot2000@hotmail.com
CODEWORK	Mauro Guazzo	Corso Lanza 105, 10133 Torino, Italy. Tel: +39 11 6601560 Fax: +39 11 6601560 Email: codework@codework-it.com
ComLog Software	Jeff Pedneau	18728 Bloomfield Road, Olney, MD 20832 USA Tel: +1 (301) 260-1435 Email: jeff@softmed.com
CoSy.com	Bob Armstrong	42 Peck Slip #4B, New York, NY 10038-1725, USA. Tel: +1 212-285-1864 Fax: +1 212-285-1864. E-mail: bob@CoSy.com.
David Crossley	David Crossley	187 Le Tour du Pont, 84210 ST DIDIER, France. Tel: +33.4.90.66.08.87 Email: crossley@au-village.com
Danish User Group	Helene Boesen	c/o Insight Systems ApS, Nordre Strandvej 119G, Hellebæk, Denmark
Dinosoft Oy	Pertti Kalliojärvi	Lönnrotinkatu 21C, 00120 Helsinki, FINLAND. Tel: +358 9 70028820 Fax: +358 9 70028824 Email: dinosoft@dinosoft.fi
Dittrich & Partner Consulting GmbH	Axel Holzmüller	Kieler Strasse 17, D-42697 Solingen, Germany. Tel: +49 212-260 660 Fax: +49 212-260 6666; Email: info@dpc.de
Dutch APL Association	Bernard Smoor (Sec)	Postbus 1341, 3430BH Nieuwegein, Netherlands. Tel: +31 347 342 337 Fax: +31 347 342 342
Dyalog Ltd.	Peter Donnelly	Grove House, Lutyens Close, Chineham Court, Basingstoke, Hampshire RG24 8AG, UK. Tel: 01256-338461 Fax: 01256-316559 Email: sales@dyalog.com; support@dyalog.com
DynArray Corporation	Dr James Brown	16360 Monterey Rd. Suite 260, Morgan Hill, CA 95037, USA Tel: +1 (408)-782-6648 Fax: +1 (408)-782-6627 Email:info@DynArray.com
Entropy Software Limited	George MacLeod	Calle Viena 13, Residencia Larrosa, Guardamar del Segura, 03140 Alicante, Spain. Tel: +34 61 00 41 940 Email: george.macleod2003@yahoo.co.uk
Evestic AB	Olle Evero	Berteliusvagen 12A, S-146 38 Tullinge, Sweden Tel & Fax: +46 778 4410 Email: olle.evero@mail.com
FinnAPL	Olli Paavola	Suomen APL-Yhdistys RY, FinnAPL RF, PL 1005, 00101 Helsinki 10, Finland Email: olli.paavola@pyr.fi
First Derivative Analytics Ltd.	Ken Chakahwata	114 Lemsford Lane, Welwyn Garden City, Herts AL8 6YP, UK Tel/Fax: 01707-339620. Email: KenChakahwata@compuserve.com
General Software Ltd	M.E. Martin	Little Wester House, Westerhill Road, LINTON, Kent ME17 4BS Tel: 01622 832463 E-mail: martin@gsoft.plus.com
Godin London Incorporated	Gaëtan Godin	12 Gerrard St., London, Ontario, Canada N6C 4C5 Tel: +1 (519) 679-8290 Fax: +1 (519) 438-6381 Email: info@godin.on.ca
HMW Computing	Chris Hogan	Hamilton House, 1 Temple Avenue, London EC4Y 0HA, UK. Tel: 0870-1010-469; Email:HMW@4xtra.com
Hoekstra Systems Ltd	Bob Hoekstra	Dominique, Salisbury Road, Woking, Surrey, GU22 7UR, UK. Tel: 01483-771028. Fax: 01483-837324 Email: Bob.Hoekstra@HoekstraSystems.ltd.uk
Michael Hughes		28 Rushton Road, Wilbarston, Market Harborough, Leics. LE16 8QL, UK. Tel: 01536-770998 Email: Michael@Hughes.uk.com
I-APL Ltd	Anthony Camacho	11 Auburn Road, Redland, Bristol BS6 6LS, UK. Tel: 0117-973 0036. Email: acam@blueyonder.co.uk
IBM APL Products	Nancy Wheeler	APL Products, IBM Silicon Valley Lab, Dept H36/F40, 555 Bailey Avenue, San Jose CA 95141, USA. Tel: +1 (408) 463-APL2 [+1 (408) 463-2752] Fax: +1 (408) 463-4488 Email: APL2@vnet.ibm.com
INFOSTROY	Alexey Miroshnikov	3 S. Tulenin Lane, St. Petersburg 191186 Russia. Tel:+7 812 325-9797 Fax:+7 812 311-2184 Email:aim@infostroy.ru
Insight Systems ApS	Helene Boesen	Nordre Strandvej 119G, DK-3150 Hellebæk, Denmark Tel:+45 70 26 13 26 Fax: +45 70 26 13 25 Email: info@insight.dk
JAD Software	David Crossley	175 East 96th St., Apt. 17G, New York, NY 10128 Country: USA Tel: +1 (212) 369-6713 Fax: +1 (212) 761-0124 Email: jadsms@usa.net
Japan APL Assoc	Toshio Nishikawa	1-8-13 Masujima Buld.6F Higashi Gotanda Shinagawa-ku, Tokyo Japan 141-0022. Tel: +81 (03) 3280-0411 Fax: +81 (03) 3280-0418 Email: KYY00361@niftyserve.or.jp

J Austria	Joachim Hoffmann	Hochsteingasse 13/26, 8010 Graz, Austria. Tel:0043 (0)316 91 42 51 Mobile: 0043 (0)699 1 91 42 51 2. Email: joachim.hoffmann@gmx.at
JSoftware Inc.	Eric Iverson	33 Major Street, Toronto, Ontario, Canada M5S 2K9. Tel: +1 (952) 470-7345 Fax: +1 (952) 470-9202 Email: info@jsoftware.com
KJK-tieto Oy	Kimmo Kekäläinen	Merikasarminkatu 10 B 56,00160 Helsinki, Finland. Tel: +358 50 55 27 207; Email: Kimmo.Kekalainen@pp.htv.fi
Kx Systems	Simon Garland [Europe]	555 Bryant St., No. 375, Palo Alto CA 94301 USA. Tel: +1 866 kdb fast email: info@kx.com (in Europe simon@kx.com)
Lambent Technology Ltd	Stephen Taylor	81 South Hill Park, London NW3 2SS, UK. Tel: +44(0)20 7813 3786. Email: sjt@lambenttechnology.com
Phil Last Ltd	Phil Last	146 Crossbrook Street, Cheshunt, Herts, EN8 8JY, UK. Tel: 01992-633807 Fax: 0121-359 0375 Email: phil.last@net.ntl.com
Lescasse Consulting	Eric Lescasse	18 rue de la Belle Feuille, 92100 Boulogne, France. Tel: +33.1.46.05.10.76 Fax: +33.1.46.04.60.23 Email: eric@lescasse.com
Lingo Allegro USA, Inc	Walter G. Fil	203 N. LaSalle Street, Suite 2100, Chicago, Illinois 60601, USA. Tel:+1 (800) 546 4621 E-mail: lingo-allegro@visto.com
Lucas Solutions	Jim Lucas	Stubbedamsvej 9C, 3.tv., 3000 Helsingør, Denmark Tel: +45 49 26 52 42. Email: jel@danbbs.dk
Mackay Kinloch Ltd	Alastair Kinloch	619 Webster's Land, Edinburgh EH1 2RX, Scotland, UK. Tel: +44 (0)131 228 5235 Email: alastair.kinloch@btinternet.com Voicemail/Pager 07626 98 3858
MasterWork Software Ltd	Fraser Jackson	PO Box 56-036, Tawa, Wellington, New Zealand. Tel: +64 (4) 232-4440 Fax: +64 (4) 232-4452 Email: fraser.jackson@xtra.co.nz
Mercia Software Ltd.	Gareth Brentnall	Mercia House, Ashted Lock, Aston Science Park, Birmingham, B7 4AZ, UK. Tel: 0121-359 5096. Fax: 0121-359 0375
MicroAPL Ltd.	Richard Nabavi	The Roller Mill, Mill Lane, Uckfield, E.Sussex TN22 5AA Tel: 01825 768050. Fax: 01825 749472 Email: MicroAPL@microapl.demon.co.uk
Milinta Inc.	Dan Baronet	Contact Dan Baronet at danb@milinta.com
Ellis Morgan	Ellis Morgan	Myrtle Farm, Winchester Road, Stroud, Petersfield, Hants GU32 3PE, UK. Tel: 01730-263843 Email: apl@ellismorgan.co.uk
NY Sig	David Siegel	PO Box 2697; New York, NY10163-2697, USA. Email: NYSIGAPL@ACM.ORG
Oasis b.v.	Louis Rijkse	Lekstraat 4, 3433 ZB Nieuwegein, Holland. Tel: +31 30 60 66 336 Fax: +31 30 60 65 844 Email: rijkse@oasis.nl
Omega Computing Inc	Alan Graham, Andrew Chou	3 Columbus Avenue, Edison, NJ 08817, USA. Tel: +1 (732) 985 9519 Email: alangraham@mindspring.com
Optima Systems Ltd	Paul Grosvenor	Optima House, Mill Court, Spindle Way, Crawley, West Sussex, RH10 1TT, UK. Tel: 01293 562700 Fax: 01293 562699 Email:paul@optima-systems.co.uk
Qualedi Inc.	Nicole Schless Georges Brigham	121 West Main Street, Milford, CT 06460, USA. Tel: +1 (203) 874-4334 Fax: +1 (203) 876-9083. Email: sales@qualedi.com; info@qualedi.com
RadSys Technologies AB	Randolph Schrab	Lovsangarv. 18, S-756 52 Uppsala, Sweden. Tel: +46 18 32 41 53 Fax: +46 708 1996 11 Email: randolph.schrab@radsys.se
Renaissance Data Systems	Ed Shaw	P.O. Box 511, Botsford, CT 06404, USA. Tel: +1 (203) 270-9729 Email: aplbooks@earthlink.com
Resources & Results	Frank Rhodes	342 Glen Oaks Blvd. Dallas, TX 75232, USA. Tel: +1 (817) 501-9825 Email: frank21@decision-support.net
RE Time Tracker Oy	Richard Eller	Mikonkatu 8 A, 2.krs, PL 363, 00101 Helsinki, Finland. Tel: +358 9-621 3300 Fax: +358 9-621 3378 Email: re@rett.fi
Right Seat Software, Inc.	Tom Atkins	1110 12th Street, Golden, CO 80401, USA. Tel:+1 303 278 2244. Fax: +1 303 278 6967. Email: info@voxproxy.com
Graeme Robertson	Dr. Graeme Robertson	15 Little Basing, Basingstoke, Hants RG24 8AX, UK. Tel: +44 0 1256-364071 Email: GraemeDR@aol.com
The Rochester Group Inc.	Robert Miller	600 Park Avenue, Rochester, NY 14607-2926, USA. Tel: +1 (716) 271-1110. Fax: +1 (716) 271-1230
Rome/Italy SIG	Mario Sacco	Casella Postale 14343, 00149-Roma Trullo, Italy Email: mario.sacco@tin.it
SE APL Users Group	John Manges	413 Comanche Trail, Lawrenceville, GA 30044, USA Tel: +1 (770) 972-3755 Email: seapldoc@aol.com

Shepp & Associates LLC	Andrew Shepp	1312 Washington Avenue, 6th Floor St. Louis MO 63103, USA Tel: +1 (314) 621-3272 Fax: +1 (314) 621-4267 Email: ashepp@compuserve.com
Snake Island Research Inc	Bob Bernecky	18 Fifth Street, Ward's Island, Toronto, Ontario M5J 2B9 Canada Tel: +1 (416) 203-0854 Fax: +1 (416) 203-6999 Email: bernecky@interlog.com
SOCAL (South California)	Roy Sykes Jr	Sykes Systems Inc, 4649 Willens Ave, Woodland Hills, CA 91364-3812 USA. Tel: +1 (818) 222-2759 Fax: +1 (818) 222-9250
Soliton Associates	Benoit Paquin	Soliton Denmark, Havsgårdsvej 4, 2900 Hellerup, Denmark Email: sales@soliton.com
SovAPL Russian Chapter of SIGAPL	Alexander Skomorokhov	PO Box 5061, Obninsk-5, Kaluga Region 249020, Russia Tel: +7(08439)47109 Fax: +1 (530) 6885510 Email:askom@obninsk.com
Strand Software Inc	Anne Faust	P.O. Box 330, Excelsior, MN 55331 USA Tel: +1 (952) 470-7345 Fax: +1 (952) 470-9202 Email: info@strandsoft.com
Rex Swain	Rex Swain	8 South Street, Washington, CT 06793 USA. Tel: +1 (860) 868-0131 Fax: +1 (860) 868-9970 Email: rex@rexswain.com
SwedAPL	Christer Ulfhielm	Novator Consulting Group AB, Svärdvägen 11C, S-182 33 Danderyd Sweden. Tel: +46 8 622 63 50 Fax: +46 8 622 63 51 CServe: 100341,404
Swiss APL User Group		Swiss APL User Group, CH-3001, Bern 1, Switzerland Email: si@ifi.unizh.ch
Sykes Systems Inc	Roy Sykes Jr	4649 Willens Ave., Woodland Hills, CA 91364, USA Tel: +1 (818) 222-2759 Fax: +1 (818) 222-9250
Toronto SIG	Richard Procter	PO Box 55, Adelaide St. Post Office, Toronto Ontario M5C 2H8, Canada Email: info@torontoapl.org
Welghahead Systems	Philip Bulmer	Camberley House, 1 Portesbery Road, Camberley, GU15 3RB, UK. Tel +44 1276 20789 Email: sales@welghahead.com
Stephen Wynn		8 Clarence Gardens, Brighton, Sussex BN1 2EG, UK. Tel: 01273-327238 Email: centre@boltblue.com
Zark Incorporated	Gary A. Bergquist	23 Ketchbrook Lane, Ellington CT 06029, USA. Tel: +1 (860) 872-7806

FTP SITES

IBM APL2	ftp.software.ibm.com/ps/products/apl2
Waterloo Archive	archive.uwaterloo.ca/ftparch/languages/apl
APL-to-ASCII	archive.uwaterloo.ca/languages/apl/workspaces/aplascii

WORLD WIDE WEB SITES

ACM SigAPL	www.acm.org/sigapl/
Adapta Software	www.adapta.de/
Adaptable Systems	www.assuredsystems.com.au/
AFAPL	www.afapl.asso.fr/ (Journal available on line)
APL2000	www.APL2000.com/
APL-385	www.demon.co.uk/apl385/
APL Forum	www.aplforum.com
APL Journal, Germany	www.rhombos.de/apljourn.htm
APL Borealis Inc.	www.aplborealis.com
APL Systems IDC SL	www.apl2000.net
AUSCAN	http://home.ca.inter.net/rjp/auscan/
Eke van Batenburg	wwwbio.LeidenUniv.nl/~Batenburg/index.html
Carlisle Group	http://carlislegroup.com/
Causeway	www.causeway.co.uk/
CODEWORK	www.codework-it.com/tangram/eng/
CoSy (Bob Armstrong)	CoSy.com/
Dinosoft Oy	www.dinosoft.fi/

Dittrich & Partner	www.dpc.de; www.apl-online.de
DMOZ - Open Directory	http://dmoz.org/Computers/Programming/Languages/APL/
Dyalog Ltd	www.dyalog.com/
DynArray	www.dynarray.com/
FinnAPL	www.pyr.fi/apl/
Godin London Inc	www.godin.com/
Houben (IQL)	www.apl.olap.club.tip.nl
Hoekstra Systems	www.HoekstraSystems.ltd.uk/
IBM APL2	www.ibm.com/software/ad/apl
Infostroy	www.infostroy.ru
Insight Systems ApS	www.insight.dk/
Japan APL Association	www.naska.co.jp/JAPLA/
JSoftware Inc	www.jsoftware.com/
KJK-tieto Oy	www.kjk-tieto.com
kx systems	www.kx.com
Lambent Technology	www.lambenttechnology.com
Lescasse Consulting	www.lescasse.com/
Lingo Allegro USA Inc	www.lingo.com/
Mackay Kinloch	http://mackaykinloch.ltd.uk/
MicroAPL Ltd	www.microapl.co.uk/apl
Milinta Inc	www.milinta.com
Oasis b.v.	www.oasis.nl/
Optima Systems Ltd	www.optima-systems.co.uk
Qualedi, Inc.	www.qualedi.com
Renaissance Data	www.aplbooks.com/
Resources & Results	www.decision-support.net
RE Time Tracker Oy	www.rett.fi/
Right Seat Software, Inc.	www.voxproxy.com
The Rochester Group Inc.	www.rochgrp.com/
Shepp & Associates	www.digitravel.com/
SigAPL	www.acm.org/sigapl/
Soliton	www.soliton.com/
Snake Island Research Inc.	www.snakeisland.com
Strand Software Inc.	www.strandsoft.com/
Rex Swain	www.rexswain.com/
Toronto SIG (for Toolkit)	www.torontoapl.org/
Jim Weigang	www.chilton.com/~jimw/
Weighahead Systems	www.weighahead.com

Zark Newsletter Extracts

Introduced by Jon Sandles

Here is another Zark newsletter extract. As always, don't hesitate to send in any solutions you may have that you may be inspired to investigate. We have had some interesting responses in recent issues which prove that however old the problem new approaches are always worth considering.

Utility Corner: Execute Inverse

(The purpose of this column is to make you more productive by introducing you to utility functions. Think of utility functions as APL functions that have names instead of symbols. By expanding your function vocabulary, you'll be able to write APL code that's more concise, more efficient, and more readable.)

In last issue's *Limbering Up* column, you were asked to define a monadic utility function `EI` (Execute Inverse) that converts any array `A` to a character vector such that `A≡⍎EI A`. For example:

```
      EI 'TEST'
'TEST'
      EI 2 3⍴6 1 0 2 9 8
2 3⍴6 1 0 2 9 8
```

The ability to convert any array to a simple character vector has some useful applications. For example, suppose your workspace contains a function named `MODEL` that requires a global variable named `TABLE`. Suppose you want to incorporate the assignment and localisation of `TABLE` into the `MODEL` function so that `MODEL` becomes a self-contained function, requiring no global variables. `TABLE` will exist only while `MODEL` is running. Finally, suppose line `[17]` of `MODEL` is the line on which you want `TABLE` to be assigned. Here's how you can use `EI` in APL*PLUS to eliminate `TABLE` as a global variable:

1. `⎕DEFL 'MODEL[17] TABLE←',EI TABLE`
2. Localise `TABLE` in line `[0]` of `MODEL`
3. Erase `TABLE` from the workspace

The system function ⎕DEFL (DEFine Line) modifies a single line of a function when given a character vector in the format

```
⎕DEFL 'Fname[Line no.] New line'
```

A second application of EI is to transfer any arbitrary array from one APL system to another. Here are the steps:

1. On the originating system,

   ```
   CVEC←EI ARRAY
   ```

2. Transmit the character vector to the recipient system, via a transmission function such as ⎕ARBIN (APL*PLUS), or by writing it to an operating system file on the originating system, transferring it to an operating system file on the recipient system, and reading it into the active workspace on the recipient system. Take care to perform translations if the APL characters have different locations in the atomic vectors (⎕AV) of the originating and recipient systems.

3. On the recipient system,

   ```
   ARRAY←⍎CVEC
   ```

At first glance, EI seems an almost trivial function to write. Basically, you pass numbers through ⍕, and you enclose characters within a set of quotes. Hoever, the more you look at it, the more exceptions you realise have to be handled. Here is a list of some different cases:

	Example
Numeric scalar	`5`
Empty numeric vector	`⍳0`
One-element numeric vector	`,5`
Numeric vector	`2 3 4`
Empty numeric array	`0 3⍴0`
Numeric array	`2 1⍴6 7`
Character scalar	`'A'`
Empty character vector	`''`
One-element character vector	`,'A'`
Character vector	`'ABC'`
... with embedded quotes	`'CAN''T'`
... with nondisplay elements	`'AB',⎕TCNL,'C'`
Empty character array	`0 3⍴''`
Character array	`2 1⍴'AB'`

Nested scalar	⊂'ABC'
Empty nested array (with prototype)	0 3⍴1 1⍴0
Nested vector	'ABC'(2 3)0
Nested array	1 3⍴'ABC'(2 3)0
Repetitive numeric array	50⍴0
Repetitive character array	50⍴' '
Repetitive nested array	50⍴⊂⍳0

Evan Jennings pointed out that APL2 contains a system function ⎕TF (Transfer Form) that performs this very task. Here is his submission:

```
      ∇ Z←EI A
[1]     Z←2↓2 ⎕TF 'A'
      ∇
```

The right argument of ⎕TF ('A') is the name of the array to be converted (in this case, the argument variable). The left argument (2) tells ⎕TF that the "extended" (executable) transfer form is desired. Since the result begins with the name of the array and an assignment arrow (A←), the 2↓ eliminates them.

Rex Swain indicated that the logic of the EI function depends on the implementation of APL on which it runs. He submitted two versions, one for conventional (non-nested) APL systems, and one for nested APL systems. Here is the conventional one:

```
     ∇ R←EI A;S;⎕PP
[1]   ⍝ Execute Inverse: A≡⍎EI A
[2]   ⍝ Returns character vector that can
[3]   ⍝ be executed to re-crete the
[4]   ⍝ array (not nested or hetero).
[5]   ⍝
[6]    S←⍴A ⍝ Record original shape
[7]    A←,A ⍝ Then ravel
[8]   ⍝ Need explicit reshape if rank≥2:
[9]    R←(1<⍴S)/(⍕S),'⍴'
[10]  ⍝ Need to ravel if a one-element vector:
[11]   R←R,(S≡,1)/',⍳'
[12]   →(0=1↑0⍴A)⍴L1 ⍝ Brance if numeric
[13]  ⍝ Wrap/double quotes on characters:
[14]   R←R,'''',((1+A='''')/A),''''
[15]   →0
[16]  ⍝ Use maximum print precision for
[17]  ⍝ floating point numbers:
[18]  L1:⎕PP←17
[19]  ⍝ Format numbers; handle empty case:
[20]   R←R,(((S=,0)/'⍳'),(0∈S)/'0'),⍕A
     ∇
```

The use of ⎕PP (Print Precision) is needed in the event that floating point numbers (such as those from 1÷3 7) are involved. The format function (⍕) is sensitive to the setting of ⎕PP, and normally formats numbers to just 10 decimal places. To avoid loss of precision, ⎕PP is set to its largest (most decimal places) setting. On some versions of APL, the largest setting is 16.

Here is Rex's version that supports nested and heterogeneous arrays. Heterogeneous arrays are arrays that are not nested but that contain both numbers and characters. (We modified Rex's function somewhat to work in other nested versions of APL than just Dyalog APL, such as the original "Evolution Level 1" version of nested APL*PLUS.)

```
     ∇ R←EI A;B;C;D;E;I;J;K;L;N;S;T;U;X;⎕IO;⎕PP
[1]   ⍝ Execute Inverse: A≡⍎EI A
[2]   ⍝ Returns character vector that can
[3]   ⍝ be executed to re-crete the
[4]   ⍝ array. Recursive
[5]   ⍝
[6]    ⎕IO←1
[7]    K←⍴S←⍴A ⍝ Original shape and rank
[8]    A←,A ⍝ Then ravel
[9]   ⍝ Branch if fewer than 2 elements
[10]   →(2>⍴A)/L1
[11]  ⍝ ... or if not all the same:
[12]   →(A≡(⍴A)⍴A[1])⍳L1
[13]  ⍝ Else express as SpA[1]:
[14]   R←(⍕S),'⍴',EI A[1]
[15]   →0
[16]  L1:N←1<≡A ⍝ Nested?
[17]  ⍝ Need explicit reshape if rank≥2 or
[18]  ⍝ if nested empty:
[19]   R←((K>1)∨N∧0∈S)/(⍕S),'⍴'
[20]  ⍝ Need ravel if one-element vector:
[21]   R←R,(S≡,1)/',⍬'
[22]   →N/L4 ⍝ Branch if nested ...
[23]   →(∧/0 ' '∈1↑0⍴<A)/L4 ⍝ or hetero
[24]   →(' '=1↑0⍴A)/L2 ⍝ or character
[25]  ⍝ Else numeric; use maximum print
[26]  ⍝ precision for floating point no.s:
[27]   ⎕PP←16
[28]  ⍝ Format numbers; handle empty case:
[29]   R←R,(((S≡,0)/'⍳'),(0∈S)/'0'),⍕A
[30]   →0
[31]  ⍝ Wrap/double quotes on characters:
[32]  L2:X←'''',((1+A='''')/A),''''
[33]  ⍝ Branch if no terminal ctrl chars
[34]  ⍝ (Use ⎕TC[⎕IO+i] of ⎕AV[⎕IO+i] for
[35]  ⍝ non-APL*PLUS versions of APL.)
[36]  T←⎕TCBS,⎕TCNL,⎕TCLF,⎕TCFF,⎕TCHT,⎕TCESC,⎕TCBEL,⎕TCDEL,
          ⎕TCNUL
[37]   →(1∈U←X∈T)⍳L3
```

```
[38]   I←U/⍳⍴U ⍝ Indices of these chars
[39]   D←T⍳X[I] ⍝ Which one is each?
[40]   C←'',⎕TCBS,'''',⎕TCNL,'''',⎕TCLF,'''',⎕TCFF,'''',⎕TCHT,'''',⎕TCESC,
         '''',⎕TCBEL,'''',⎕TCDEL,'''',⎕TCNUL,''''
[41]   ⍝ Lengths and beginnings:
[42]    L←(9 9 9 9 9 10 10 10 10 10)[D]
[43]    B←(0 9 18 27 36 45 55 65 75)[D]
[44]   ⍝ Shorten if not after normal char:
[45]    L←L-T←(2,U)[I]
[46]    B←B+T
[47]   ⍝ ... or before normal char:
[48]    L←L-2×(1↓U,1)[I]
[49]    E←(⍴X)⍴1 ⍝ Expansion (repl) vector
[50]    E[I]←L
[51]    X←E/X ⍝ Expand it
[52]   ⍝ Adjust inds for expanded version:
[53]    I←(+\1,E)[I]
[54]   ⍝ J←(⍳L[1]),(⍳L[2]),(⍳L[3]),... :
[55]    J←L/¯1↓0,+\L
[56]    J←(⍳⍴J)-J
[57]   ⍝ Stuff them in:
[58]    X[J+L/I-1]←C[J+L/B]
[59]   ⍝ Drop 1st/last 3 chars if needed:
[60]    X←(3×0 1≡2↑U)↓(¯3×1 0≡¯2↑U)↓X
[61]   L3:R←R,X
[62]    →0
[63]   ⍝ Nested or hetero; branch if more
[64]   ⍝ than one item
[65]   L4:→(1<S←⍴A)/L5
[66]   ⍝ Use 1↑ to get prototype if empty:
[67]    R←R,'⊂',EI 1⊃1↑A
[68]    →0
[69]   ⍝ Nested array with >1 item. One
[70]   ⍝ algorithm is:
[71]   ⍝     R←R,⊃,/'(',¨(EI¨A),¨')'
[72]   ⍝ but it gives more parentheses than
[73]   ⍝ needed, so we loop instead:
[74]   L5:I←0
[75]   L6:I←I+1
[76]    →(I>S)/0
[77]   ⍝ Get item and convert it:
[78]    U←EI D←I⊃A
[79]   ⍝ Need parentheses if ...
[80]    →(1<≡D)/L7 ⍝ Nested
[81]    →(∧/0 ' '∈1⊃1↑0⍴⊂D)/L7 ⍝ or hetero
[82]   ⍝ ...or if numeric nonscalar:
[83]    K←⍴⍴D
[84]    →((0=1↑0⍴D)∧(K≠0),1)/L7,L8
[85]   ⍝ But not if a char scalar:
[86]    →(K=0)/L8
[87]   ⍝ Need parentheses if rank >1 or if
[88]   ⍝ a one-element vector:
[89]    →((K>1)∨1∈⍴D)/L7
[90]   ⍝ or if some terminal ctrl chars
[91]    →(1∈D∈⎕TC8S,⎕TCNL,⎕TCLF,⎕TCFF,⎕TCHT,⎕TCESC,⎕TCBEL,⎕TCDEL,⎕TCNUL)↓L8
```

```
[92]  L7:U←'(',U,')' ⍝ Provide the parens
[93]  ⍝ Incl space betw items if needed:
[94]  L8:T←('('=1↑U)∨(¯1↑R)∈' )'
[95]   R←R,(T↓' '),U
[96]   →L6
      ∇
```

Lines [9] to [15] handle arrays whose items are all the same. Because of this logic, the result can be much smaller. For example, if the argument is a matrix of 500 zeros the result is '50 10⍴0' instead of '50 10⍴0 0 0 0 0 ... 0'.

To handle nested arrays, the function recurses (calls itself) to format the items of the array. See lines [14], [67], and [78].

To handle terminal control characters, such as new lines (carriage returns), linefeeds, formfeeds, and so on, the function replaces those characters by system functions that return them. See lines [33] to [60]. The following illustrates this replacement logic. Suppose the right argument of the EI function is ⎕TCFF,'ABC',⎕TCNL,'DE',⎕TCNL,⎕TCLF,'GE',⎕TCESC. Here's what happens:

```
[36-37]  Branch if no terminal chars
 [32]    X:   '_ABC_DE__GE_'
 [37]    U:   01000100110010
 [38]    I:   2 6 9 10 13
 [39]    D:   4 2 2 3 6
 [42]    L:   9 9 9 9 10
 [43]    B:   27 9 9 18 45
 [45]    L:   9 9 9 8 10
 [46]    B:   27 9 9 19 45
 [48]    L:   9 9 7 8 10
[49-50]  E:   1 9 1 1 1 9 1 1 7 8 1 1 10 1
 [51]    X:   '_________ABC_________DE_________
              GE_________'
 [53]    I:   2 14 25 32 42
[55-56]  J:   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
              1 2 3 4 5 6 7 8
              1 2 3 4 5 6 7 8 9 10
 [58]    X:   '',⎕TCFF,'ABC',⎕TCNL,'DE',⎕TCNL,⎕TCLF,
              'GE',⎕TCESC,''
 [60]    X:   ⎕TCFF,'ABC',⎕TCNL,'DE',⎕TCNL,⎕TCLF,'GE',⎕TCESC
```

If your version of APL is not APL*PLUS use the terminal control system functions available (e.g. ⎕TC and ⎕AV). For example, make the following modifications:

```
[36]    T←⎕TC,⎕AV[1]
[40]    C←'''',⎕TC[⎕IO],'''',⎕TC[⎕IO+1],'''',⎕TC[⎕IO+2],''''
        ,⎕AV[⎕IO],'''
[41]  ⍝ Lengths and beginnings:
[42]    L←(12 14 14 12)[D]
[43]    B←(0 12 26 40)[D]
```

On line [66] of the above EI function, the word "prototype" is used. The "prototype" of an empty array is the array return when you "pad" it via take (1↑A) or expansion (0\A). For simple empty arrays such as ⍳0 or '', the prototype is obvious. You get back 0 when do 1↑⍳0 and get back a blank when you do 1↑''.

However, for an empty nested array, what is the prototype? Zero? Blank? Neither?

The prototype depends on how the empty nested array was created. For example, the prototype of 0⍴(3 4⍴⍳12) 'HELLO' is 3 4⍴0. The prototype of 0⍴'HELLO' (3 4⍴⍳12) is 5⍴' '. In general, the prototype is the same as the first item of the array from which the empty array is derived, with all numbers replaced by zeros and all characters replaced by blanks.

It is the desire to accurately reproduce the prototype of the empty nested array that accounts for the logic on line [67]: R←R,'⊂',EI 1⊃1↑A.

1⊃1↑A returns the prototype, EI formats it, '⊂' encloses it, and R (e.g. '0⍴') reshapes it back to its empty shape.

Limbering Up: Word Wrap

(The purpose of this column is to work some flab off your APL midsection. Like muscles, your APL skills can atrophy if not exercised with adequate frequency and variety. This column presents a task for you to perform. Set aside a few minutes from your busy schedule and work the task. Mail in your solution and stay tuned for the results.)

Given a lengthy piece of text, it is sometimes necessary to "wrap" it so it fits within a specified width. The piece of text is broken into subsets, without breaking any words, so that each subset is no longer than the specified width. The following sample has been wrapped within a width of 17:

```
This is a sample.
The last nonblank
character in each
line fits within
the specified
width.
```

Your task is to write a dyadic function WRAP that does wrapping. Its right argument is the character vector to be wrapped. Its left argument is the width within which the text is to be wrapped. Its result is a character vector with newline (carriage return) characters inserted in such a way that the text wraps appropriately when displayed. For example:

```
      17 WRAP 'This is a sample. The ...'
This is a sample.
The last nonblank
     (etc.)
```

Some notes:

1. The result has the same length as the right argument. The newline character (⎕TCNL or ⎕TC[2]) *replaces* the first blank after the last word in each line.

2. If the right argument already contains newline characters, don't ignore them and don't remove them. Each one begins a new line. In a sense, existing new lines are "paragraph" delimiters.

3. Consider each group of contiguous non-blank characters a word. The aim is to wrap the text without breaking any words. However, if any "word" is longer than the specified width, you should insert one or more newline characters within it to break it into pieces whose length equals the width. If this is the case the result will naturally be longer than the right argument.

Your primary objective is to write a function that works. You'll receive extra credit (a pat on the back) if the function is efficient, clever, simple, elegant, or can leap tall buildings in a single bound. Send your solution to:

Vector Production, Brook House, Gilling East, York YO62 4JJ
Email: apl385@compuserve.com

The notable functions and their authors' names will be printed in the next issue of *Vector*. Good luck and happy limbering.

Reprinted with kind permission from Zark APL Tutor News, a quarterly publication of Zark Incorporated, 23 Ketchbrook Lane, Ellington, CT06029, USA

Notes about Execute Inverse Functions

by David Crossley

I submit several variations as my solutions to the Zark challenge in Vector Vol 20 No 2 to write an Execute Inverse function. I have frequently used functions with this specification in the past (often known as UnExecute) for various purposes, for example, to transfer variables between different APLs. More recently, I have developed a version for my Variable Editor to present an editable expression to the user, though this exercise has proven useful in bringing to light one or two deficiencies.

My first solution, the fastest, is a Dyalog dynamic function name EI1:

```
EI1←{⎕PP←15 ⍝ String that reproduces ⍵ when executed - yukky display
     sh←{⍺≠0:(⍕⍺),'⍴',⍵ ⋄ ⍵}                    ⍝ reshape
     ad←{⍺:1↓'),(⊂',⍵ ⋄ ⍵}                     ⍝ adorn with ,(⊂...)
     ei←{⍺←0                                   ⍝ ⍺=1 when recursing
         s←⍴⍵                                  ⍝ shape
         0≡,⍵:⍺ ad s sh'0'                     ⍝ empty numeric
         ''≡,⍵:⍺ ad s sh''''''                 ⍝ empty text
         326=⎕DR ⍵:⍺ ad s sh{                  ⍝ nested or heterogeneous...
             0=⍴⍵:1↓⊃,/1 ei¨1⍴⍵               ⍝ empty - assemble prototype
             1↓⊃,/1 ei¨⍵                       ⍝ recurse on nested elements
         },⍵
         0≡⊃0⍴⍵:⍺ ad s sh⍕,⍵                   ⍝ numeric
         ⍺ ad s sh 1⍴''''''',(1+''''=,⍵)/,⍵     ⍝ quotes round text/double quote
     }
     ei ⍵
}
```

The code could be 3 lines shorter by including the empty cases with the non-empty cases and modifying ⍵ to (1⌈⍴,⍵)⍴⍵ (see next function). However, my time tests showed a significant time improvement with specific handling of the empty cases.

You will note that I have set ⎕PP to 15 rather than 16. This is done to reduce the output of real numbers. The number 6.6, for example, would otherwise be displayed as 6.599999999999999 rather than 6.6.

I use ⎕DR, available in both Dyalog and APL2000, to determine data representation (type) rather than pure, but slower, APL idioms. Note in particular that simple heterogeneous arrays are handled through recursive calls. The ⎕DR of both nested and heterogeneous arrays is 326. By testing for these structures first, the identification of simple numeric or text arrays is simplified.

I do not cater for ⎕OR elements and References in arrays. I felt that handling these items is beyond the scope of this exercise, and indeed I wonder if there is a way to reliably reproduce a Reference! Come to that, the ⎕OR of a namespace would not be too easy to reproduce.

My second submission is a normal user-defined function, though I do use Dyalog control structures which are implemented in other versions of APL. It is of course a simple exercise to convert the control structures to good old-fashioned branch and label equivalents. The function is named EI2:

```
     ∇ r←{a}EI2 w;⎕PP;x
[1]   ⍝ String that reproduces ω when executed - conventional fn/yukky display
[2]   ⍝ a ↔ 1:indicates recursion
[3]    ⎕PP←15
[4]    a←1≡⍬⍴⊃(0=⎕NC'a')⌽'a0'                    ⍝ a=1 when recursing
[5]    :If 326=⎕DR w ⋄ r←1↓⊃,/1 EI2¨(1⌈⍴,w)⍴w     ⍝ nested or heterogeneous
[6]    :ElseIf 0≡⊃0⍴w ⋄ r←⍕(1⌈⍴,w)⍴,w            ⍝ numeric
[7]    :Else ⋄ r←1⌽''''',(1+''''=x)/x←(1⌈⍴,w)⍴w ⍝ quotes around/double quote
[8]    :EndIf
[9]    :If 0≠⍴w ⋄ r←(⍕⍴w),'⍴',r ⋄ :EndIf          ⍝ reshape
[10]   :If a ⋄ r←1⌽'),(⊂',r ⋄ :EndIf              ⍝ addorn with ,(⊂...)
     ∇
```

This function is about 50% slower than the Dfn, though there was no consistent timing difference with or without special-casing of empty items so I have opted for the shorter form.

What I do not like about the above functions, which produce identical results, is that though they conform to the requirement that $\omega\equiv\text{⍎}EI\omega$ (except that they will fail if a variable includes ⎕OR'ed or Reference elements), the displayed result is awful, lots of parentheses and reshapes, and the long-winded form for arrays of text vectors.

My third submission, another Dyalog dynamic function, returns a result that is far more readable, with no redundancy, and close to the preferred way in which most people would enter the expressions at the desktop, or in applications where the executable form may be presented for user editing, e.g. as in my Variable Editor.

As this is my favourite, I call it EI. It is slower than the first Dfn, but is comparable to the second conventional function in my timing tests.

```
EI←{⎕ML←0 ⋄ ⎕PP←15 ⍝ String that reproduces ω when executed - friendly display version
     sh←{⍝ reshape ω according to shape α
         α≡,1:',',ω                                ⍝ ravel rank-1 singltn (shorter)
         1∊(1=ρρα)^(0 1=⊃ρα)^1,0<⊃α:ω              ⍝ vector, null or length>1
         (⍕α),'⍴',ω                                ⍝ include explicit reshape
     }
```

```
      ad←{⍝ adornments on ⍵ given by ⍺
          ⍺=1:'(',⍵,')'                                    ⍝ in parentheses
          ⍺=2:' ',⍵                                        ⍝ precede with space
          ⍵                                                ⍝ no change
      }
      ei←{⍝ main dfn: recursive call if ⍺
          ⍺←0                                              ⍝ default not a recursive call
          s←⍴⍵                                             ⍝ shape
          θ≡,⍵:(⍺×1<⍴s)ad⊃(1<⍴s)⌽'θ'((⍕s),'⍴0')            ⍝ is θ
          ''≡,⍵:(⍺×1<⍴s)ad((1<⍴s)/(⍕s),'⍴'),''''''         ⍝ is ''
          (s≡θ)∧1=≡⍵:2 ad'''***⎕OR***'''                   ⍝ ⎕OR: cannot handle
          (s≡θ)∧9=⎕NC'⍵':2 ad'''***Ref:',(⍕⍵),'***'''      ⍝ Reference: cannot handle
          326=⎕DR ⍵:⍺ ad s sh s{                           ⍝ nested or heterogeneous...
              0=⍴⍵:'c',i↓(-i←'('=⊃x)↓x←1 ei⊃⍵               ⍝ empty
              x←(' '=⊃x)↓x←⊃,/1 ei¨⍵                       ⍝ recurse on nested elements
              (0=⍴⍺)∨(1=⍴⍺)∧1=⊃⍺:'c',i↓(-i←'('=⊃x)↓x       ⍝ strip parens and enclose
              x                                            ⍝ unmodified
          },⍵
          0≡⊃∈⍵:(⍺×1+0=⍴s)ad s sh⍕,⍵                       ⍝ numeric
          (⍺×1+1≥⍴s)ad s sh 1⌽''''''',(1+''''=,⍵)/,⍵       ⍝ quotes round text/double quote
      }
      ei ⍵
  }
```

And that is it.

David Crossley
20th November 2003
St Didier

Limbering up – Execute Inverse

from Nicholas Small

Acknowledgement

I can't claim much credit for the attached - it was mainly the work of Allan Gay and as far as I recall my contribution was limited to removing unnecessary parentheses from the result. As you will see, all I have done is write a cover function to call the function that emulates APL2's ⎕TF.

Example Usage

```
      qq←(2 3⍴⍳6)(3 4⍴⍳12)'Hello, world'
      iq←EI qq
      iq
3⍴(2 3⍴1 2 3 4 5 6)(3 4⍴1 2 3 4 5 6 7 8 9 10 11 12)(12⍴'Hello, world')
      ⍪iq
  1 2 3    1  2  3  4   Hello, world
  4 5 6    5  6  7  8
           9 10 11 12

      ⎕vr 'EI'
    ∇ Z←EI Y
[1]   Z←2+2 quadTF 'Y'
    ∇
```

Listings (for extended transfer form only)

```
    ∇ ΔZ←ΔA quadTF ΔW;⎕IO
[1]   ⍝ Limited simulation of APL2's ⎕TF for APL*PlusII/386.
[2]   ⍝ (System ⎕ functions and variables are not supported).
[3]   ⍝  ⍺  is integer scalar or 1-integer vector:
[4]   ⍝       1  declares migration transfer form
[5]   ⍝       2  declares extended transfer form.
[6]   ⍝  ⍵  is character scalar or vector, which is one of:
[7]   ⍝       i)   the name of an APL item to represent
[8]   ⍝       ii)  the representation of an APL item to be reconstituted.
[9]   ⍝  ←  is character vector containing, for the two cases of ⍵:
[10]  ⍝       i)   an ⍺-type representation of the item named in ⍵
[11]  ⍝       ii)  the name of the item reconstituted from ⍵.
[12]
[13]  ⎕ERROR (~ΔA ∈ 1 2)/'DOMAIN ERROR'       ⍝  ⍺ must be 1 or 2.
[14]  ⎕ERROR ((ΔA∈1)^1<≡ΔW)/'DOMAIN ERROR'   ⍝  ⍵ must be simple when ⍺ is 1.
[15]  ⎕ERROR (' '≠⎕TYPE ↑ΔW)/'DOMAIN ERROR'  ⍝  ⍵ must be text.
[16]  ⎕ERROR (0∈⍴,ΔW)/'DOMAIN ERROR'         ⍝  ⍵ must not be empty.
[17]
```

```
[18]   →(' 'ϵΔW)/RRR                          ⍝ Go reconstitute from ⍵.
[19]
[20]   →(AAA,BBB)[ΔA]
[21]   AAA:                                   ⍝ MAKE MIGRATION TRANSFER FORM:
[22]    ΔZ←quadTF_out1 ΔW                     ⍝  Make representation.
[23]    ⎕ERROR (~1≡ΔZ)/'DOMAIN ERROR'         ⍝  Mixed datatypes unsupported.
[24]    →ZZZ
[25]   BBB:                                   ⍝ MAKE EXTENDED TRANSFER FORM:
[26]    ΔZ←quadTF_out2 ΔW                     ⍝  Make representation.
[27]    →ZZZ
[28]
[29]   RRR:
[30]    →(TTT,UUU)[ΔA]
[31]   TTT:                                    ⍝ USE MIGRATION TRANSFER FORM:
[32]    ΔZ←quadTF_in1 ΔW                      ⍝  Reconstitute APL object.
[33]    →ZZZ
[34]   UUU:                                    ⍝ USE EXTENDED TRANSFER FORM:
[35]    ΔZ←quadTF_in2 ΔW                      ⍝  Reconstitute APL object.
[36]    →ZZZ
[37]
[38]   ZZZ:
    ∇
```

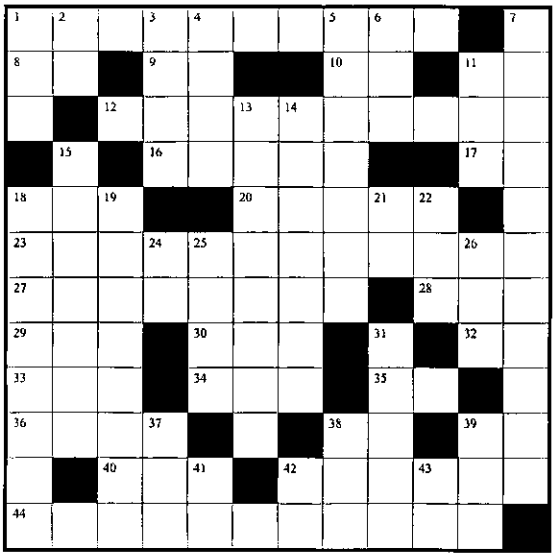
```
    ∇ Δz←quadTF_out2 Δw;ΔV;⎕IO
[1]    ⍝ Represent APL object named ⍵ in extended transfer form
[2]    ⍝   ⍵  is character scalar or vector name of an APL item to represent.
[3]    ⍝   ←  is character vector α-type representation of the item named in ⍵.
[4]
[5]    ⎕IO←1
[6]
[7]    Δz←''                                  ⍝ Default result.
[8]    →(~(⎕NC Δw)ϵ2 3)/ZZZ                   ⍝ Ignore unsupported classes.
[9]    →(3=⎕NC Δw)/XXX                        ⍝ Go represent fn ⍵.
[10]
[11]   ΔV←⍴Δw                                 ⍝ Determine array type:
[12]   →(1<≡ΔV)/NNN                           ⍝ . go do nested array.
[13]   →(1<+/0 1ϵϵ0=1↑¨0⍴¨ΔV)/MMM             ⍝ . go do mixed array.
[14]   →SSS                                   ⍝ . go do simple array.
[15]
[16]   MMM: Δz←quadTF_out2_mixed ΔV ⋄ →VVV    ⍝ Represent mixed array.
[17]   NNN: Δz←quadTF_out2_nested ΔV ⋄ →VVV   ⍝ Represent nested array.
[18]   SSS: Δz←quadTF_out2_simple ΔV ⋄ →VVV   ⍝ Represent simple array.
[19]
[20]   VVV:
[21]   →(1≥≡ΔV)/WWW
[22]   Δz←1↓¯1↓Δz                        ⍝ Drop superfluous parentheses
[23]   WWW:
[24]   Δz←Δw,'←',Δz,' ' ⋄ →ZZZ           ⍝ Make an assignment statement.
[25]
[26]
[27]   XXX:                                   ⍝ REPRESENT FUNCTION ⍵:
[28]    Δz←,'⍞',⎕CR Δw                        ⍝ Get the ⎕CR and double all
[29]    Δz←(1+Δzϵ'''')/Δz                     ⍝  quotes first.
[30]    Δz←DLTB ¨⎕SPLIT SStoMAT Δz            ⍝ Convert to vectors, omitting
[31]    Δz←'⎕FX disclose', ⍕'''',¨Δz,¨''''     ⍝  leading/trailing blanks.
```

```
[32]    →ZZZ
[33]
[34]   ZZZ:
    ∇


    ∇ Δz←quadTF_out2_mixed Δw
[1]    ⍝  Represent simple mixed APL array ω
[2]    ⍝    in extended transfer form.
[3]    ⍝     ω  is the mixed array
[4]    ⍝     ←  is text vector representation of the named array
[5]
[6]     Δz←(1,¯1↓(Δw∈⎕AV)≠1↓(Δw∈⎕AV),0) ⎕PENCLOSE ,Δw
[7]     Δz← 1↓∈ ',' ,¨ quadTF_out2_enquote ¨Δz
[8]     Δz←(⍕⍴Δw),'⍴',Δz,' '
    ∇


    ∇ Δz←Δa quadTF_out2_nested Δw;ΔS;ΔX;⎕IO
[1]    ⍝  Represent nested APL array ω
[2]    ⍝    in extended transfer form.
[3]    ⍝     α  is 1 if parent array was scalar, else 0
[4]    ⍝     ω  is the array
[5]    ⍝     ←  is character vector representation of ω content
[6]
[7]     ⎕IO←1
[8]    ⍎(0=⎕NC 'Δa')/'Δa←0'
[9]
[10]    →(ΔX←0=⍴ΔS←⍴Δw)/AAA                    ⍝  Note if ω scalar.
[11]    →(~0∈ΔS)/AAA                           ⍝  Go do non-empty
[12]    ΔΔz←⊂⎕TYPE ↑Δw                         ⍝  Approximation when empty
[13]    Δz←4↓2 quadTF 'ΔΔz'
[14]    →OOO
[15]
[16]   AAA:
[17]    →(1<≡Δw)/NNN                           ⍝  Go do nested array.
[18]    →(1<+/0 1∈∈0=⎕TYPE¨Δw)/MMM             ⍝  Go do mixed array.
[19]    →SSS                                   ⍝  Go do simple array.
[20]
[21]   NNN:
[22]    Δz←∈(⊂ΔX)quadTF_out2_nested ¨Δw         ⍝  Represent nested array.
[23]   OOO:
[24]    Δz←((-ΔX)/(⍕ΔS),'⍴'),Δz                ⍝  Make a representation.
[25]    →WWW
[26]
[27]   MMM: Δz←quadTF_out2_mixed Δw ⋄ →WWW      ⍝  Represent mixed array.
[28]
[29]   SSS: Δz←quadTF_out2_simple Δw ⋄ →WWW     ⍝  Represent simple array.
[30]
[31]   WWW:
[32]   →Δa↓XXX
[33]    Δz←'⊂',Δz                              ⍝  Enclose if parent scalar
[34]   XXX:→ΔX/YYY
[35]    Δz←'(',Δz,')'                          ⍝  Parenthesise result.
[36]       →ZZZ
[37]   YYY: Δz←Δz,(')'≠¯1↑Δz)/' '
```

```
[38]
[39]  ZZZ:
    ∇


    ∇ Δz←quadTF_out2_simple Δw
[1]   ⍝  Represent simple unmixed APL array named ⍵
[2]   ⍝   in extended transfer form.
[3]   ⍝    ⍵  is the simple array
[4]   ⍝    ←  is text vector representation of ⍵ content
[5]
[6]    Δz←quadTF_out2_enquote Δw              ⍝  Add/double quotes as needed
[7]    →(0=ρρΔw)/ZZZ                          ⍝  If nonscalar,
[8]    Δz←((⍕ρΔw),'ρ'),Δz                     ⍝   apply shape.
[9]
[10]  ZZZ:
    ∇


    ∇ Δz←quadTF_out2_enquote Δw
[1]   ⍝  Represent simple unmixed APL array ⍵ as text.
[2]   ⍝  If ⍵ is text, enquote it and double all embedded quotes.
[3]   ⍝    ⍵  is the array
[4]   ⍝    ←  is character vector representation of ⍵ content
[5]
[6]    Δz←⍕(,Δw),((0=⎕TYPE↑Δw)^0∈ρΔw)/0 ⍝  If ⍵ is not text,
[7]    →(' '≠⎕TYPE ↑Δw)/ZZZ              ⍝   no quotes are needed.
[8]    Δz←(1+Δz∈'''')/Δz                 ⍝  Double any single quotes
[9]    Δz←'''',Δz,''''                   ⍝  Enquote the result.
[10]
[11]  ZZZ:
    ∇


    ∇ ΔZ←quadTF_in2 ΔW;ΔN;ΔQ;ΔR;ΔS;⎕IO
[1]   ⍝  Reconstitute an APL object from extended transfer form in ⍵
[2]   ⍝   ⍵  is character vector representation of an APL object.
[3]   ⍝   ←  is character vector name of reconstituted object.
[4]
[5]    ⎕IO←1
[6]
[7]    →('⎕FX'^.=3↑ΔW)/PPP                   ⍝  Go handle fn.
[8]    ΔZ←(¯1+ΔW⍳'←')↑ΔW                     ⍝  Get name to reconstitute.
[9]    ⍎ΔW                                   ⍝  Reconstitute the var.
[10]   →ZZZ
[11]  PPP:
[12]   ⍎'ΔZ←',ΔW                            ⍝  Reconstitute a fn.
[13]   →ZZZ
[14]
[15]  ZZZ:
    ∇
```

New Crossword and 20.2 Solution

New Crossword



Across

1. Character matrix CL in sorted order according to collating sequence CS
8. A single random number from ⍳X
9. →⎕__ to restart your program
10. ⎕__ to see how much space remains
11. ⎕__ to show number of significant digits being displayed
12. Add vector P to each column of M
16. Index in vector V[⍋V] of V[1]
17. ⎕__ lets you make lightning strike twice
18. Branch to the line labelled LP
20. Length of tape to encircle V cylinders of diameter B, rounding each piece up to the nearest integer
23. First differences of vector NC
27. R5 followed by 1 if LS and V2 are identical, otherwise followed by 0
28. *(⊗V)-⊗A
29. B≡(-ρB)↑1 for bit vector B

30. (1+E)-⍳E in origin 1
32. ⎕__ to change the sensitivity of =
33. (((1↑⍴R),1)⍴5),[2]R for matrix R
35. +/⍳7
34. C+0×⍳V for numeric scalar C
36.)RESET
38. Alternating sum
39. ⎕__ to use either whole numbers (0, 1, 2, ...) or natural numbers (1, 2, 3, ...)
40. (⍳0)⍴C[⍋C]
42. ,+/NV≠(N*2)*0.5 vector NV, scalar N
44. A_L

Down

1. C different random numbers from ⍳X
2. ⎕__ to get things going
3. CL←____ to increment CL
4. ('SCA',⍕'''⍎P↑L''')~'APL'
5. V floored at ¯2 and repeated to shape LW
6. Localize AM
7. The value of a programming language lies in its _____
11. P/R for P and R scalars
13. Is vector V in ascending order?
14. (⍬⍴(VEC<0)/⍳⍴VEC),(⍬⍴(VEC=0)/⍳⍴VEC),(⍬⍴(VEC>0)/⍳⍴VEC)
15. If BL15←,S<15 for matrix S, return flagged elements of S
18. Branch to the line labelled P if R is less than 5
19. Return all but the first P elements of matrix BRI, but capped at 0
21. 3.14159...
22. V[B+⍳(⍴V)-B]
24. ⎕__ to get list of fns or vars
25. (CS↓V),CS↑V for positive CS
26. ÷C÷N
31. ,⍉(12,⍴VI)⍴VI for vector VI
37. (+/C)⍴G for bit vector C, scalar G
38. O-NM
39. G-I×⌊G÷I
41. ⎕__ to express a user-defined function as a character matrix
42. Number of rows and columns in M
43. 1-N for Boolean N

Solution to Crossword in 20.2

J-ottings 39: All Verbs are Monadic

says Norman Thomson

Read on – you will find this article is full of interest! And no, its title is not an aberration like signing up to the Flat Earth Society – what I contend is that the concept of valence in J, and indeed in APL, is necessary only at the surface level of language. At interpretation time, one argument of a dyadic verb must be fetched before the other, which means that argument marshalling is necessarily non-symmetric, i.e. 2+3 is not the same as 3+2. In the case of 2+3 an intermediate verb, call it *two-add*, is created, which is then presented with 3 as argument. This idea arises more generally in computing science, where the idea of argument bonding is known as "currying" after the American logician H. Curry. Indeed the J documentation explicitly offers "curry" as an alternative name for bond (&).

At a surface level J-users do not need to be aware of this, and rightly perceive an inherent symmetry between the left and right arguments of dyadic verbs. However, the inadequacy of this view becomes apparent in applying the power conjunction, as in, for example, `2(+^:4)3`. Is *two-add* to be applied four times to 3 to give 11, or is *three-add* to be applied four times to 2 to give 14? The answer is the former, illustrating that dyadic verbs are in reality a cover for a two-stage process of "bond left, then apply right".

Sooner or later every J user becomes frustrated by the fact that verbs must be either monadic or dyadic – how nice it would be to have triadic or even *n*-adic verb forms to use like argument lists in C++ or Java! But if all verbs are monadic, and thus describable as word groups of the form NV (noun plus verb), then, since the physical placement of the arguments is irrelevant to valence, a dyadic verb can be represented as N(NV) A triadic verb in this notation would be represented as N(N(NV)), and verbs with multiple arguments would have the description N^nV. In theory at least, universal monadicity is a liberation which gives J users the power of *n*-adic verbs!

At this point I return to earth (flat or otherwise) to consider some expressions with superficially triadic verbs. The first evaluates a linear function such as px + y where x and y are lists of equal length, and the calculation is repeated over a range of values of p. To be more specific, think about simple interest calculations and make x = 4 40 (periodic increments), y = 100 1000 (capital sums) and p = 1 2 3 (periods), so that there are three result lists x + y, 2x + y, and 3x + y.

One way to do this which is closely related to the algebraic statement and uses verb rank, is

```
   (1 2 3*/4 40)+"1(100 1000)     NB. basically p*x + y
104 1040
108 1080
112 1120
```

However explicit verb rank can be avoided by using the equivalent expression

```
    1 2 3(+&4 40)100 1000          NB. basically p(+&x)y
```

This second version is effectively triadic *plus*, and, interestingly, contains no multiplication symbol. Bonding the verb + with the noun `4 40` creates a superficially dyadic verb `+&4 40`, call it *add-four-forty-to*, which becomes bound to `1 2 3` to form a monadic verb, call it *repeat3-add-four-forty-to*, which is applied to the argument `100 1000`. The notion of *repeat3-add-four-forty-to* may be a little harder to grasp than that of *two-add*, although in the context of simple interest it can be given a more meaningful name such as *repeat3-add-increments-of-interest-to*.

The J Dictionary definition of *bond* (&) shows why this second version works, but first, it is worth commenting on the apparent paradox whereby although + is the only verb, the effect of *repeat3-add-four-forty-to* is to **multiply** the list `1 2 3` by `4` and `40` separately and then add the results to `100` and `1000` respectively. What would happen if `4 40` and + were reversed to give a verb which would logically be called *four-forty-add-to*? The result of `1 2 3(4 40&+)100 1000` is no different from that of
`1 2 3(+&4 40)100 1000`, but if the argument of the opening paragraph is correct the verbs *four-forty-add-to* and *add-four-forty-to* are functionally different. To investigate this question change + to a non-commutative verb:

```
   (1 2 3 4 5(-&4 40)100 1000);1 2 3 4 5(4 40&-)100 1000
+------+--------+
|96 960|_96 _960|
|92 920|100 1000|
|88 880|_96 _960|
|84 840|100 1000|
|80 800|_96 _960|
+------+--------+
```

The first box contains in its columns the results of subtracting `4 40` once, twice, thrice, etc. in succession, which is what might be expected, but what about the second box? It shows that the lists `100 1000` and `_96 _960` alternate, that is, either there is a single subtraction of `100 1000` from `4 40` or else nothing at all

happens, depending on whether the corresponding left argument item is odd or even. So what is going on?

At this point return to the formal definition of compose as given in the J Help file :

```
x m&v y ↔ m&v^:x y
```

Take `m&v` to be `4 40&+` so that `1 2 3(4 40&+)100 1000` means `(4 40&+)^:1 2 3(100 1000)`, that is add `4 40` to `100 1000` once, twice and three times, which is equivalent to multiplying `1 2 3` by `4` and `40` respectively and then adding `100 1000`. `(4 40&-)` means that `4 40` is subtracted-from a given number of times; applying the bonded verb a second time means combining *four-forty-subtract-from* and *four-forty-subtract-from* **before** any reference to any other data is made. Now *four-forty-subtract-from* is an idempotent operation, that is, successive executions cancel each out like a toggle switch, and so the only two final possibilities are to subtract `4 40` once or not at all – precisely what was observed. On the other hand, changing the composed verb to `-&4 40` means that the verb `-&4 40` is bonded with `1 2 3` so that *subtract-four-forty-from* is subtracted once, twice, thrice etc. from the right argument, which is always the object to which a complex verb is finally **applied**.

The logic can be seen even more clearly by comparing the expressions

```
   5(2 3&%)6 NB. (two-and-three-divide-by 5 times)6
0.333333 0.5 NB.  .. equivalent to (2 3&%)6

   5(%&2 3)6 NB. (divide-by-two-and-three 6) 5 times
0.1875 0.0246914
```

or, to underline these points :

```
   2 3% 2 3% 2 3% 2 3% 2 3% 6
0.333333 0.5

   ((((6%2 3)%2 3)%2 3)%2 3)%2 3
0.1875 0.0246914
```

Now replace + with * to investigate triadic times :

```
   1 2 3(*&1.04 1.05)100 1000    NB. basically p(*&x)y
    104    1050
 108.16  1102.5
112.486 1157.63
```

These results are compound interest calculations, and since `*`, like +, is commutative they are unaffected by replacing `(*&1.04 1.05)` with `(1.04 1.05&*)`. Now, instead of $p\underline{x} + y$ over a range of values of x and y, the result is yx^p, this in spite of there being no explicit exponentiation present.

With division, the results are analogous to those with minus, viz. either just two alternating results arise if the constants are on the left of the bond, or a list $y_1/1.04^p, y_2/1.05^p$ if they are on the right. The latter case evaluates net present values as in :

```
     1 2 3(%&1.04 1.05)100 1000
96.1538 952.381
92.4556 907.029
88.8996 863.838
```

Do not confuse these problems with the somewhat similar problem of evaluating a linear function, say 2x + 3y, at all points on a grid defined by a list of x values, say `0 10`, and a list of y values such as `0 1 2 3 4 5`. The grid points are generated by `,"0/` and a dot product generates the function values :

```
   (+/ .*&2 3)0 10,"0/0 1 2 3 4 5
 0  3  6  9 12 15
20 23 26 29 32 35
```

When triadic expressions are consolidated into user-defined verbs, the surface constraint of "not more than two arguments" has to be observed, and subsequent arguments must be built in as illustrated in the examples below, which obtain the same compound interest and net present value information obtained earlier :

```
   ci=.*&100 1000@:(^/)   NB. build in the capital sums
                          NB. call is 1.04 1.05 ci 1 2 3
   npv=:%&1.04 1.05       NB. build in the discount rates  NB.
call is 1 2 3 npv 100 1000
```

To summarise so far, provided that x and y are equal in length `p(+&x)y` means add x to y p times with feedback to give y+px, and `p(*&x)y` means multiply x by y p times with feedback to give values of yx^p. Similarly `p(-&x)y` evaluates y-px and `p(%&x)y` evaluates y/x^p. The paradox observed earlier can be explained because, with feedback, multiplication is repeated addition and exponentiation is repeated multiplication.

Now look at a four parameter problem, for example valuing a savings plan where there is a mixture of compound interest at 1.05% and periodic additions of 4 are applied to an initial capital of 100. After the first year the value is .

```
      1((+&1.05)&4)100  NB. basically p((+&k)&x)y
109
```

and in principle the values for the next two years are `2 3((+&1.05)&4)100`. However, the interpreter requires that powers higher than 1 must be integers (or at least multiples of 0.5), so this expression gives a domain error.

Nevertheless 4-adic *plus* should not be discounted. Consider

```
    0 1 2 3 4((3&+)&2)0 1 2 3 4 5 6 7 8 9
 0   1   2   3   4   5   6   7   8   9
 2   5   8  11  14  17  20  23  26  29
 8  17  26  35  44  53  62  71  80  89
26  53  80 107 134 161 188 215 242 269
80 161 242 323 404 485 566 647 728 809
```

The first row in this table is just `i.10` since according to the power definition of *bond* applying something 0 times means returning the right argument unchanged. Call the first row values n, then the second row is the series 3n+2 (that is using the middle arguments), the third row is the series 3(3n+2)+2, that is 9n+8, the third row is 27n+26 and so on, whereas each column is a series 3n+2 with feedback, using the first row as start values.

If the arguments of 4-adic *plus* are lists, the scalar rule that repeated application of addition is equivalent to multiplication has to take into account that repeated operations on lists generate lists of lists. Inspection of the following expression shows that it returns the results of applying 2x+4 and 3x+40 twice with feedback to initial values `100 1000`

```
    2((2 3&+)&4 40)100 1000
 412  652
 684 1060

4012 6052
6084 9160
```

Returning to the savings plan problem which was left in limbo, this can be addressed using the verb *base* (`#.`). For example, if an initial capital sum of 100 attracts 5% compound interest and also 4 units are added every period, the values in the first three years are :

```
    1 2 3(#.&1.05 4)100  NB. basically p #.x y (y scalar)
109 118.45 128.373
```

Because *base* is not a scalar verb, a modest amount of adjustment has to made in order to replicate the calculation over several capital sums :

```
   1 2 3 (#.&1.05 4)"(1)0,.,100 1000
 109 118.45 128.373
1054 1110.7 1170.24
```

The next question is whether there are primitive verbs other than + and * for which triadic application leads to useful results. One such is triadic *shift:*

```
   0 1 2(1&|.)'ABCDEFGHIJK'
ABCDEFGHIJK
BCDEFGHIJKA
CDEFGHIJKAB
   (+/ .*&2 3)1 2 3,"0/6 8
20 26
22 28
24 30
```

which does progressive 1-shifts, and generalises to n-shifts :

```
   a=.'ABCDEFGHIJK'
   0 1 2(3&|.)a
ABCDEFGHIJK
DEFGHIJKABC
GHIJKABCDEF
```

This could also be written `0 3 6 |."0 1 a` using explicit rank. Neither of these alternatives is inherently superior to the other. As is often the case, J provides different ways of expressing ideas, and the choice between a bonding or a verb rank approach here is purely a matter of user preference.

Another example is the progressive reduction of a list, n items at a time (triadic *drop*) :

```
   t=.'parallelogram'
   0 1 2(2&}.)  t
parallelogram
rallelogram
llelogram
```

which again has an alternative explicit rank formulation
`0 2 4 }."0 1 t .`

A matching verb for progressively building up a list is triadic *append* :

```
   1 2 3 ('ab'&,)''
ab
abab
ababab
```

which is an alternative to `'ab',^:1 2 3 ''`

In summary, the conjunction *bond* allows extension to surface triadicity, thereby adding meaningfully to the algorithmic repertoire. And because beneath the surface all verbs are *actually* monadic, extension with further explicit bonds allows 4-adicity and beyond. Thereafter the number of **useful** applications begins to thin out, and also use with lists can lead to rank explosion if care is not exercised – I wonder if the Flat Earth Society has some interesting *n*-adic problems waiting in the wings ...

Double dyadic verbs

```
   f=.2 3&+

   f 6
8 9
   5 f 6      NB. (2*5) + 6 , (3*5) + 6
16 21
   f/4 5 6    NB. (2*(4+5))+6 , (3*(4+5))+6
24 33

   g=.2 3&*

   g 6
12 18
   5 g 6      NB. (2^5) * 6 , (3^5) * 6
192 1458
   g/4 5 6    NB. (2^(4+5))*6 , ((3^(4+5))*6
3072 118098

   j=.2 3&$

   $5 j 6          NB. (2,(5$3))$6
2 3 3 3 3 3
   $j/4 5 6   NB. (2,(4+5)$3)$6
2 3 3 3 3 3 3 3 3 3
```

To calculate 2x + 3y for x = 1 2 3 and y = 6 8, a rank based solution is

```
    f=.+/@:(2 3&*)
    f"(1)1 2 3,"0/6 8
    (+/@:(2 3&*)"1)1 2 3,"0/6 8   NB. (..p..)x,"0/y
20 26
22 28
24 30
```

Common Mean

```
    am=: +/ %   #          NB. arithmetic  mean
    gm=: */ %:~ #            NB. geometric  mean
    cm=: [: {. (am,gm)^:_    NB. common mean
    c=.cm@:(1&,@%@%:)
    c 2
0.847213
ser=.3 :0
t=.(am,gm)^:(i.10)1,%%:y.
'a b'=.|:t
r=.+/\ (2^i.#a) * a -&*: b
)
conv=.(*:@c)%-.@{:@ser
   conv 3
1.5708
```

Conference Report: APL2000 at Naples Beach in 2003

reported by Adrian Smith

Location (see last year) and General Impressions

Really, I should just refer readers back to the January 2002 edition of Vector. This time, Causeway did take advantage of a Virgin Atlantic 7-day golfing holiday which flew into Miami just in time to let us clear customs, get to the Dollar car-rental desk and drive over. The weather in Miami was pretty horrid (mid 80's, relative humidity 110%, steady rain) but it cleared nicely as we drove across Florida, and for the first 4 days, Naples was its usual sunny self. Then the wind shifted on-shore and oh-boy did it rain!

But mostly, it was fair, and getting trapped in the poolside bar by a 2-hour downpour is not such a terrible thing. There were 92 registered attendees (so a slightly smaller gathering than last year) of whom around a dozen were from APL2000. Sunday was a well-attended training day, and there were 2 full days of papers on Monday and Tuesday. Wednesday offered additional training time, although by then many delegates had drifted home. We had an excellent informal

reception and meal on Monday (with live entertainment Boogie-style from The Revd Billy Wirtz - I made sure to buy his CD so the family could suffer in sympathy) with the banquet back at the excellent Chinese restaurant which we visited last year.

Conference Programme - Day-1

Opening Remarks, from Eric Baelen

Eric opened with the usual excellent pep-talk, focussing on APL's reliability and performance. He noted that the APL components always passed the stringent Cognos pre-release trials, and stressed the excellent statistics you get on comparing the APL+Win webserver technology with IIS.

> "Everything has been said before, but you guys weren't listening!"
>
> What's good about APL? It has a low TCO (Total Cost of Ownership) - competitively priced, and allows a small group to develop systems fast. Look at Vox Proxy! It turned a daft idea from an APL conference into a business reality.
>
> APL has a large sweet-spot. It can take on OLAP problems, detailed calculations, specialised file I/O. You can write scripting tools with it, or create a web service. The performance will typically be 10 times faster than the same apps written in VB.
>
> APLers are smarter, and care more about their users. We spend longer thinking about the problem and we make sure the solution works as well as it can.

Eric also introduced us to the excellent conference freebie - instead of a jacket (shirt, fleece ...) we all got a 128Mb USB Jumpdrive pre-loaded with the proceedings and workspaces. What's more, early registrants all had their name very nicely engraved on the back! This was a far more practical way to deliver the supporting workspace than the usual CD ROM, and many more delegates than usual were able to 'type along' with the presenters as a result. The drives also came in very handy for ad hoc workspace swapping - they were automatically recognised by any laptop with Win2000 or above, and drivers are available for Windows 98 from the APL2000 website if you want to use it on an earlier OS.

What's New in Version-5 and Version-6, by Michael Steiner

Version 5 has been a year in the making (we all got the pre-beta last year) and it is one of those releases which changes very little on the outside, and quite a lot on the inside. Bill Rutiser has been quietly migrating the 'COMIC' memory management into the production interpreter so that it now grabs and releases memory on demand, rather than allocating a fixed slab of workspace and sitting on all of it regardless. There were lots of other small improvements (mostly session colours and logging options) which you can read about on the APL2000 website.

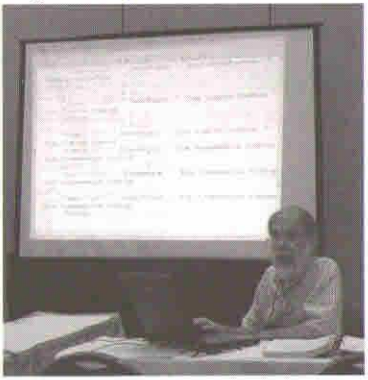
Web Services and the Google API from William Parke

I was torn between this paper and the concurrent session from Bill Rutiser on RegExp. In the end I tossed a mental coin and joined the Rutiser stream, so here are a few comments from the author:

> "Regarding my own presentation, unless I misjudged the mostly expressionless faces of my audience, I think most of the attendees were lost or bored midway through the talk. In my defense, it would have flowed more smoothly and made a little more sense if I had had Internet access, but I should have been able to demonstrate all the necessary steps and code anyway. In retrospect, I had prepared too many examples demonstrating different approaches to application implementation *and* tried to touch on them all, in the process failing to explain some of the important basics along the way. The sequence of steps for accessing the Google API is quite simple, but some of the steps themselves are complex. The main aspects needing more emphasis and detail were the general methodologies for manipulating XML directly in APL, although in truth that should be a workshop or two on its own.
>
> Nevertheless I do believe there is some value in the workspaces I included on the conference disk, so anyone who wants to pursue this topic should have enough material to get started. On the other hand, the usefulness of the Google API may be short-lived if Fred Waid's predictions about the search engine in the next version of Windows are fulfilled."

It looks like really useful stuff, and I know how good and solid his utilities normally are (I rely on his JPEG code in RainPro and NewLeaf) so I am betting that there will be some high-value free stuff ready to take away and use. I don't think Google will go out of fashion for a year or two. Look out for a comprehensive technical article from William in the next Vector.

Regular Expression Pattern Matching by Bill Rutiser



It is probably best to refer you direct to Bill's notes here on page 64. Regular expressions are a little programming language in their own right, and for some very specific search-and-replace operations knowing the right regexp can save you masses of work. The challenge is partly in knowing when you have the right problem, and then finding the regexp to crack it.

Bill left us with all the tools we need to get at a free regexp engine from APL, but I think many in the audience went away wearing slightly puzzled frowns and wondering if they would ever have the courage to use it.

A Layman's Tour of .Net for APL Developers by Adrian Smith

When I wrote the notes for this paper (included in slightly revised form on page 82) I was very unsure just what colour of rabbit Fred Waid would be pulling out of his hat immediately afterwards. In fact, we were both doing much the same thing – running the +Win COM server though a .Net facility caller COM InterOp – only he was hiding it better. I was very encouraged how many people came to my talk, and at the level of awareness of the .Net technologies in the +Win community. I think the pressure on APL2000 to 'do it right' (cut out the InterOp step) will be coming from all sorts of directions over the next few months as key users take home the message that this technology is now very imminent and they are going to have to live with it, come what may.

Tool Talk by Rex Swain [notes from Bill Parke]

Rex shared some of his favourite user-command development tools, including:]filedoc and]pkgdoc, GUI tools to inspect .sf file components and user command packages;]notlocal, a tool to identify all workspace functions in which a given variable is used as a global;]varss and]fnss, with case-insensitivity, multi-word and whole-word extensions to the original like-named search functions;]varfind, that searches variables for strings and numbers similar to the way]wsloc searches functions;]si, an enhanced display of)si that includes the relevant line of code with each level in the stack; and]symfind, another GUI tool that examines and displays the symbol tables in every workspace (using your library definitions) and user-command file it can find. Finally he demonstrated]diff, a cover for a 3rd-party, side-by-side code-comparison utility called Araxis Merge

(www.araxis.com/merge/ – $129 US.) Rex provided his own, personal money-back guarantee that we'd find this tool indispensable.

It's always a treat to listen to one of the masters, and Rex has a relaxed style that is easy to follow. These are all handy tools to have available, and as with most of the workshops, his work serves to spark more new ideas among the audience. He said he usually develops his tools to about 90% functionality, stopping at the point that meets his own needs, but that just means there's room to add whatever custom features we can dream up. No doubt we'll take at least one example and use it as a starting point to create our own next favourite tools.

Bill Parke 21 Nov 2004

.Net, Managed Code and your Future by Fred Waid

In his talk at Madrid, Fred left us with the feeling that he regarded .net as mostly marketing hype, and not really worth being interested in. To be fair to him, what he was mostly saying was that the Microsoft marketing guys had gone way too far in promoting .net as the future of everything and this needed some serious debunking. Be that as it may, Fred has now firmly fallen in behind the technical stuff in .net (basically the Framework, the Runtime, and the SDK) which is really all that is left, now that Microsoft have virtually abandoned the 'brand' and released *Windows 2003 Server* with no mention of .net in its name.

He ran us through the usual 'Fred' warm-up which included some pretty interesting statistics (some 35% of recent hits to the OpenHere website are from clients with the .net runtime installed) and some useful pointers to where the market is headed. Then he showed a simple cover on the +Win COM server, called *APL+ Inter* which effectively automates both the generation of the .net assembly from your APL code, but additionally registers that as a valid COM server itself, and as a bonus uses the tools from the SDK to build a very standard help file for the DLL you constructed. This is a nice mechanisation of the 'fully manual' approach I had showed in the previous session, and I like Fred's syntax of 'triple-comment' as the equivalent of '///' for marking out the meta-data needed to create the help information. I probably won't use all his toolkit here, but I will fall into line with the basic approach.

The DHTML Object Revisited by Warren Baelen and Andy Weiss (pictured)

Now this one looks to have some really useful stuff hidden in there. It is probably best if Warren and Andy write this one up themselves, but my impression was that the DHTML object could be used to create a 'web-style' interface in a stand-alone desktop application. Why do this? Well one reason is that you can allow your user to make radical changes to the look and feel very easily simply by having them edit the CSS style sheet for your interface. Microsoft Money is an application which plays just these games.

One thing I may be investigating very soon is the possibility of making an HTML Edit control which behaves very much like a RichEdit (from the user's perspective). As outlined in Vector 20.2, parsing RTF is a dog of a job, while parsing HTML is very straightforward. For GraPL, I should allow my users to make 'rich' notes for their charts – far better to have them create these in HTML directly, then formats like VML and SVG are 'fall-through' and PDF/EPS are much easier to produce from the marked-up text.

Plenary Session – APL in MetLife by Bob Shalack and George Weiss

This felt like one of those many APL success stories that it is hard to get excited about. MetLife have 12 marketing executives working on highly tailored personal insurance plans, mostly for the seriously rich who want to move money to the children without it getting taxed. The rules are complex, and change frequently as the government stops up one loophole but leaves another one open. So APL scores highly, for all the reasons Eric gave in his opening session. A small team (basically the two presenters, with George on the nth year of his 3-month contract) can keep the calculations in line with the legislation. It is a low-cost, fast moving, calculation-rich, business-critical application. Right in APL's sweet-spot.

Probably this was not really a full 1-hour presentation – there were rather too many PowerPoint fill-ins and too little real meat. We did glean that they used 2 add-in grids (*FarPoint* for data input and *Formula-1* for reporting) but were looking hard at the native APL2000 grid control, particularly in the light of the change to the deployment rules on Formula-1. Some kind of grid is essential, as Excel is the prime target for all the output from the system.

Conference Programme - Day-2

The Plain Vanilla Player from Carl House and Davin Church

From the feedback I heard afterwards, Carl and Davin left the audience feeling a little confused over what they had been shown. I think this was a pity, because Carl has some good ideas and Davin has been working really hard to make them happen. In the end, they probably left the construction of the presentation too late in the day, and a few pieces went missing at the last minute.

Carl's main business is in modelling social and strategic infrastructure for major city developments (often green-field sites like the new capital of Tanzania). The basic problem is very stable (a city is a city) but the detailed requirements for data management are different every time. Carl (like the rest of us) hates writing Windows forms, programming lots of reports, and building multiple HTML tables. "I don't want to write forms - I want specifications for forms".

Which is where Davin comes in. He has been working with Carl on an application called "Phoenix" which can take a data-driven specification and generate input documents, HTML tables, and NewLeaf reports (for printing or mailing as PDF). It uses an MDI interface so that each form handles a single set of data. The user can select from the views which the developer has specified, and it is easy to update on the fly simply by mailing out new view definitions. Davin showed a very nice HTML implementation with a 2-level menu structure organised to fit in with the style of the Microsoft website. Carl has also been making good use of the APLDraw OCX (now generally available with version 5) to watermark photographs for his 'hobby' project of managing a major website of swimming information. Unfortunately many of the best examples didn't make it across to his laptop, so we will have to check them out on the web for ourselves.

Nationwide Delivery of Pension Plan Valuations from Joe Blaze

For me (and for many others) this talk was the highlight of the conference. Was it a co-incidence that Joe simply sat and talked for an hour with no foils, no jokey videos, no PowerPoint strait-jacket? Probably not. He had good material and he delivered it in a simple and logical way. An APL audience appreciates this.

"I'm actually on a 25-year sabbatical as a maths professor - we had more tenured professors than we had students!"

Joe started out with a pension valuation product for 'smaller machines' in the days when all the competition had only expensive timesharing. He was running with 50 employees within 2 years, the vast majority were programmers as the work was very tedious. By comparison, his company today operates with 25 employees, of whom only 5 are programmers, and supports a very much bigger business operation. His main site is in New Jersey, but a group of 5 support folk are based in San Diego to be on west-coast time. Most of the current employees have been with the company for over 20 years.

Joe went on to describe a number of his products, many of which are gradually converting to a web-based model, but which must still be maintained in parallel as stand-alone applications.

1. **The Document System.** Secure and proper distribution of documents for employee benefit plans. Some of these are required to be filed with the federal government, so the government people also use the system to validate return dates and other critical details on the forms

2. **Risk factors for coronary diseases.** This system uses a predictive formula to estimate the probability of coronary disease based on various known risk factors. The system is web-based and can be used on the Blaze server, added to the websites of drug companies, doctors and hospitals, or offered by personnel departments in a 'kiosk' at the employees workplace. Individuals can use the model to get a prediction, and companies offer it in the hope of changing

employee behaviour. The same system (with a different interface) is available to the underwriters of healthcare cover. They have partial information on the whole of an employee group, so a risk-estimate helps to set the level of premium. Finally, it can be used by employers to contest the group premiums they have been offered!

3. **Tax code modelling.** This is a classic 'what-if' system used by the IRS to determine the effect on revenues of changes to the tax codes. The same system can be used by pension plan administrators and the plan sponsors.

4. **Specialised pension valuation.** This system is used in countries where the normal rules fail due to high inflation. Joe commented that these may be very nice countries for his staff to visit, but that this is a very good case for pure web delivery!

5. **Very standard (so lots of competition) plan valuation.** This was a fine example of using the web to lower the transaction cost for low throughput, and so make the system very competitive for smaller users.

Of the 87 systems that BlazeSSI supports, 20 have more than 1000 users, one has 100,000 users, some have 5 users, and one has 1 user.

Joe made some excellent points on methodology, which I hope I have accurately quoted here:

- Concentrate on systems that have a very large user-base. Smaller systems are hard to make profitable.
- We do a lot more on market research than on systems work (15 people out in the field, 5 back at base).
- We always find out what the customers really want. Never just duplicate what they have.
- Documentation is very lightweight now – just training guides with references to primary sources (actuarial regulations and so on).

Moving from conventional contracts to licensing via the web was a 6–8 month redesign after long discussions with customers. The model changed from an annual fee to a cost per transaction, but Blaze shared the risk by capping the annual charge for the first year to give existing customers a "comfort zone". Often the initial cost did not trip the customer's 'ceiling' making the contracts much easier to negotiate. Sometimes it was possible to reach really tiny outfits who might do only 1–2 transactions per month – they would never have taken on the up-front cost.

Joe pointed out the value of choosing systems that require regular updates. Security is not an issue, as the regulations change every year, so there is no danger of anyone stealing your source code as it goes out of spec very quickly. Blaze take care to mail out descriptions of the updates very early, giving 'sunset' dates for expiring features and systems. They keep the customers informed well ahead of time of new software and hardware requirements (for example "we need you to move to 800 by 600 screens" could be a big issue for a site with 1000 users). Although updates are usually a progressive change, Joe noted that an interface with 200 fields could drift over time to having 5000 fields, and sooner or later you must make a major rebuild. Joe was hopeful that the present monthly mailing of over 3000 CDs would start to come down as more web-applications are deployed.

He made some interesting observations on the Adobe PDF tools. Blaze use Acrobat to add 'fields' to dumb government forms, with a [Submit] button to get the completed form sent over the web and hence to have the information captured by an ASP page. They have incorporated some validation into the form fields, but Joe noted that the JavaScript engine inside Acrobat Reader is very poor and it was hard to write good validation code. Someone from the audience commented that after 'upgrading' to Acrobat Distiller 6 their PDF documents had mushroomed in size for no apparent reason. I have been working with Joe over the summer to update Causeway's NewLeaf engine to merge multiple reports into a single PDF (and get the image handling up to speed with JPEGs) so hopefully he will not have this problem, as NewLeaf creates very lean and mean PDFs which run any anything from Reader 2.0 upwards (see *A PDF Primer for APLers*, Vector Vol. 18 No.4 page 77 for the details). Joe noted "The use of .pdf-format output created by NewLeaf from Causeway, and compatible with a broad range of Adobe Acrobat Reader versions, was important in converting systems to the web-services environment" which is nice to know.

The final section of Joe's talk was all about efficient internet delivery, and the value of APL web services. On switching from IIS+VB to IIS+APLWCO certain calculations went from needing 9 servers to just 2 (of which one is just there for backup). The problem with this technology is debugging (you cannot make the server visible in the live environment) and the overhead of loading the workspace for every call. The next stage is a move to web services which already have the workspace loaded and initialised. The welfare plans system dropped from 4 servers (APLWCO) to partial use of one server, and the most heavily loaded system dropped from 21 servers to 2 (each with 5 instances of APL loaded).

Creating COM Servers with APL+DLL by Andy Weiss

Best to read Andy's own notes on this one – see next Vector. Essentially I saw it as automating what we had done by hand when we built our GraPL COM server. We used Delphi (where Andy uses VB6) and we constructed the property table once (using definitions from RainPro, the clipboard, and some clever Search/ Replace in Notepad) and now maintain the Delphi code in parallel. I think that automating the process makes a lot of sense, but it is a pity that no-one did this two years ago! If Causeway's experience (and Joe Blaze's comments) are anything to go by, COM is going rapidly out of fashion as the server folk move to web services or to ASP.net for the applications they are building today. However the principle is fine – use structured comments or definition variables in your workspace to generate source code you can compile. The target language is easy to change, once you have the infrastructure in place.

Using the New Unicode Controls by Marvin Renich and Colyn Phillips

This one caused some interesting spluttering noises among the delegates! The idea of a text field that takes *integers* – what can they be thinking of? Actually what APL2000 have done here is extremely sensible, and is not at all hard to program for. Moving your APL to a full Unicode environment is a tough call (do you really want 65000 elements in quad-AV?) and strictly unnecessary. Handling input in Greek, Korean, or whatever is essential for many applications, and most of the standard string-handlers (deb and friends) can easily be adapted to work with integer vectors with very little loss of efficiency. As long as you keep your Unicode 'text' as numbers then everything just works as before. Web output is fine (just use the AmpersandHash notation with the decimal values) and I think it works in PDF. I am interested to know how you print it, though. I didn't see anything about the 'DRAW' method taking Unicode characters. I also have the problem of taking Unicode strings in as properties of a COM server, and as the return from Windows calls like GetDateFormat when used in Korean. There is a little way for APL2000 to go here, but this is a hopeful pointer and a very promising start.

Cryptographic Functions with APL by Warren Baelen

Warren made the mistake of writing this up much too well in the distributed papers. I enjoyed it, understood most of it, and went off to loaf by the beach. If you are interested in the techniques which work well in APL, then look out for Warren's paper in a future Vector. The sample functions were all included on the conference JumpDrive.

Version 5 Memory Management by Bill Rutiser

Again, I can only recommend reading Bill's written notes, which are an excellent summary of memory management techniques from the early days of APL to the strategies employed by Version-5 to make the workspace as elastic as the memory in your machine. See next Vector.

Animated Graphics and APL by Fred Waid IV and Gerald Waid

Microsoft have a strange habit of smuggling out really useful tools, either disguised as something else or just plain undocumented. Fred took us through the strange history of *VRML* (a C-like language for defining 3D worlds) which Microsoft promised to implement as *ActiveVRML* in IE4 (but never did) which mysteriously showed up as *Direct Animation* in IE5 and above. APL2000 discovered it as a handy way of implementing a selection of the DRAW method commands in the browser (as exported with JSave) and Fred showed us some of the nice stuff it can do with true 3D animations.

Which all emphasises the question "why is no-one using this stuff?". I suppose that the 'scripted only' interface has a lot to do with it. Web authors expect to be able to write their pages with HTML tags, but to make the DA control do anything you have to generate JavaScript. Having said this, it does have handy features like proximity detection (this object just hit that object, so fire an event) which are just what games authors need. It also looks likely to turn up in yet another disguise as Avalon (the interface layer for Longhorn) so it is well worth being interested in. You might also be interested in some work Gerald has done to scan old APL listings with *OmniPage* (this was edged out of the programme due to lack of slots, but we did get the printed material and I reckon it deserves to be more widely known). It should appear in the next Vector.

Summary and Wrap-up

Every time I go to one of these events, I have to ask myself "was it a conference or a reunion?". The SigAPL conferences were sliding towards reunion status back in 1995, and the only way to go from there is down, and eventually out. I think that the APL2000 meeting is still OK – most of the people who came were there for the content, not the beer and the beach. It would be good to get more code-heavy papers for next year – looking back through the notes there were depressingly few APL symbols in there, and one reasonably APL-heavy paper had had all the code blocks reduced to bullets. It would also be good to have more talks of the same quality as Joe Blaze gave, and I am sure the training days were good value for those with specific techniques to learn. By this time next year .net will be pervasive (at least on servers) and Longhorn will be imminent. How APL gets to play in this world is going to affect all of us for the next 10 years, maybe 20. I think they should start planning the programme now.

Oh, and if we do come again next year, please can we have Room 41 again! The view from the patio was gorgeous!

Regular Expressions and APL using the PCRE Library

by William Rutiser (APL2000 Inc.)

Following an introduction to the notation and a description of the Perl Compatible Regular Expression Library, we describe a DLL and cover functions that provide Regular Expression support for APL+Win. Based on open source code, the DLL is self-contained. There is a tabular description of much of the PCRE notation and options at the end.

Regular Expressions

The regular expression notation, invented by mathematician Stephen Kleene in the 1940's, is a powerful means to describe textual patterns; it underlies the pattern matching and searching facilities in many text-processing applications and scripting languages. While extensions to the basic notation have proliferated, the Perl language variant described here is close to a de-facto standard.

In this brief, incomplete, and simplified introduction, double quotation marks, having no special meaning in the notation, delimit both regular expressions and character strings. The contractions regex and regexes are widely used.

A regular expression defines a, possibly infinite, set of character strings. The regular expression and each of the strings it defines are said to *match*. The practical value of the regular expression notation is due to the compactness and manipulability of a regex definition relative to the set of strings it defines.

Given a regex and a subject string, a regex matching engine attempts to discover a sub-string of the subject that matches the regex. If any such sub-string exists, the match is said to *succeed;* if none exists the match is said to *fail*. When there are multiple matching sub-strings, most engines find the longest of the leftmost matches.

A regular expression is represented as a character string satisfying certain syntactic rules.

Except for 12 non-alphanumeric meta-characters that have special meaning, most characters in a regex stand for themselves.

Any character without special meaning is a regex that matches only the character itself. So the regex "q" matches only the single string "q". Normally case sensitive, a regex can be made case insensitive by a matcher option.

A backslash preceding a non-alphanumeric character is said to escape or quote the character, suppressing any special meaning. The regex "*" matches only "*" and "\\" matches only "\".

A backslash may give special meaning to a following alphanumeric. For example, "\n" is a regex that matches the newline character (hex 0C).

A catenation of regular expressions forms a regular expression that matches a catenation of strings each matching its corresponding sub-expression. So the regex "array", the catenation of five one-character expressions, matches sub-strings of "an array language", "anarrayofhope", and "array of arrays".

The period meta-character matches any single character. So the regex "s.x" matches "sax", "six", "sxx", and "s.x" but not "sx" or "sabx". The regex "s\.x" matches only "s.x".

Characters enclosed in square brackets form a *character class* regex that matches a single instance of any of its enclosed characters. The regex "s[aeiou]x" matches three-character strings such as "sox", "sax", and "six" but not "sqx", "s]x", or "s[x". If a circumflex immediately follows the left bracket, the regex matches only characters not included. The regex "[^aeiou]" matches "b", "q", "9", and "^" but not "a", "e", or "u".

Within a character class regex, a pair of non-meta-characters separated by a minus sign, represents a range of characters; so the character class regex "[A-Za-z]" matches any upper or lower case letter and "[^0-9]" matches all characters except digits.

Several predefined character classes are denoted by a backslash followed by certain letters. For example, "\d" is equivalent to "[0123456789]" and "\D" is equivalent to "[^0123456789]". So "\d\d\d" matches any string of three digits while "x\Dy" matches "xay", "x]y", as well as "x\y" and "xDy" but no string having a digit in its second position.

A regex and a following *quantifier* form a regex that matches the catenation of some number of strings each matched by the base regex. A regex quantified by the question mark meta-character matches zero or once. So "\d\d\.?" matches two digit numbers with an optional trailing period.

An asterisk quantified regex matches any number of times including zero. "10*2" matches "12", "102" and "10000002" but not "10*2" or "10abc2".

The plus sign quantifier is similar but requires at least one match. So "10+2" doesn't match "12" but does match "102" and "1002".

These quantifiers are shorthand for frequently needed cases of the min-max quantifier: "*regex*{*min*,*max*}" matches at least *min* and at most *max* instances of *regex*. "\d{1,4}\.{0,1}\d{0,3}" matches "1", "1234", "5." and "67.890" but not "12345", "0.2345", or "12..34".

Quantifiers have higher syntactic precedence than catenation but parenthesis may be used to define their scope. So "(10)*2" matches "2", "102", and "10102". "M(iss)+ippi" matches "Missippi", and "Mississississippi" as well as the correct spelling.

Alternative sub-patterns are separated by vertical bars, which have lower precedence than catenation so parenthesis may be needed to limit the first and last alternative. "Booleans are true|false values" matches "Booleans are true" and "false values" whereas "Booleans are (true|false) values" matches "Booleans are true values" and "Booleans are false values".

Modern regular expression notation incorporates many other pragmatically useful and convenient variations on the elements described above.

Parentheses, used to indicate syntactic grouping as described above, have other important uses.

Matching engines, in addition to reporting whether or not a pattern matches a subject string, report information about where and how the pattern matched. The positions of the ends of the matching sub-string are reported. PCRE reports such positions as the number of characters preceding the positions. In APL terms, these are the zero origin indices of the first character of the sub-string and character immediately following the sub-string.

Similarly, the matcher reports where each parenthesis enclosed sub-expression matched. When the sub-expression matched at several places, the last match is reported. So "abc(def|ghi)jkl" matches "123abcghijkl456" between positions 3 and 12. The sub-expression "def|ghi" will be reported to match between positions 6 and 9. The parentheses are said to *capture* the matching sub-string. Complex regexes may include many pairs, possibly nesting, of capturing parentheses. The captures are numbered left to right according to the position of

the opening parenthesis. The leftmost parenthesis is assigned the number one. The number zero is associated with the entire match.

A backslash followed by a digit refers back to the sub-string captured by the correspondingly numbered parenthesis. "`\s+((\w+)\s+\2)\s+`" matches a pair of identical words separated by white space. Capture 1 describes the pair of words and white space while capture 2 describes the first word.

A question mark immediately following a left parenthesis modifies the meaning of the rest of the enclosed text in various ways. When followed by a question mark followed by a colon, a left parenthesis becomes non-capturing.

The regular expressions described above all match one or more characters, at least potentially. The regular expression notation also provides so-called *assertions*, which never consume characters during the matching process; they succeed or fail depending on the current match position and the characters around it. The circumflex (APL's AND primitive) succeeds only at the beginning of the subject; the dollar sign succeeds only at the end. These are used to anchor a pattern at the ends. "`^\d+$`" matches only strings consisting entirely of digits. "`\D\d{5}$`" matches only strings ending with exactly five digits. Note that some complexities involving new-line characters has been skipped.

The assertion "`\b`" matches only at word boundaries, i.e. positions preceeded by a non-word character and followed by a word character or vice-versa. Neither character is consumed in the match. The assertion "`\B`" matches only at positions that are not word boundaries. So when "`(.*)\b(\w+)`" is matched to "`one two three`", the first capture is "`one two `" and the second it "`three`".

A Few Examples

The regex "`any ((T[io]m){1,3}|Di[cr]k|[HBL]arry)`" matches "`any Tom`", "`any Tim`", "`any Dick`", "`any Dirk`", "`any TomTom`", "`any TomTim`", and "`any Barry`".

"`\d\d\d-\d\d-\d\d\d\d`" matches a social security number.

The regex "`((\(\d\d\d\))|( \d\d\d [\-\.]))?\d\d\d[\-\.]\d\d\d\d`" matches telephone numbers in several styles: "`555-1212`", "`555.1212`", "`(301)208-7217`", and "`301.208.7217`". It also matches "`301-208.7217`". This can be corrected by using a back reference and conditional sub-pattern as in the regex:

```
  "^(\(\d\d\d\)|\d\d\d([\-\.]))?\d\d\d((?(2)\2|[\-
\.]))\d\d\d\d$"
```

The PCRE Library

The library is open-source software written by Philip Hazel of the University of Cambridge Computing Service, Copyright ©1997-2001 University of Cambridge.

Its author describes the Perl Compatible Regular Expression Library as a "set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl, with just a few differences. The current implementation of PCRE (release 4.x) corresponds approximately with Perl 5.8".

Its short and simple PCRE license is easily observed. Many open source software projects use the library and it is included in several Linux distributions.

The APL_PCRE DLL

This pre-built DLL contains the PCRE library functions and extra routines needed when using the library from APL.

The library has two central functions: `pcre_compile()` and `pcre_exec()`.

Essentially, `pcre_compile()` accepts a regular expression string and returns a pointer to an opaque structure known as a `pcre`, which contains a compiled representation of the regex and tables needed for matching. A `pcre` is contained in storage managed by Windows and must be de-allocated when it is no longer needed. Several ancillary arguments communicate matching options and error codes.

The `pcre_exec()` function takes a `pcre`, a subject string, and a pointer to space to store the results of matching the regex to the string. These results are offsets into the string of the beginnings and ends of the matching and captured sub-strings. Additional arguments communicate an offset into the string where matching is to begin, matching options, as well as a pointer to an optional block used for certain esoteric purposes.

One extra-library routine, `pcre_dispose()`, frees a `pcre`'s storage into the storage pool used by the library. Any subsequent use of the pointer is likely to initiate a hard crash or data corruption. Failure to free a `pcre` will leak a small amount of storage. This is benign unless the small leak is repeated many times.

These functions are called via ⎕WCALL. The active aplw.ini file must contain the quad-wcall definitions for the `pcre_...()` functions and constants.

The following steps summarize the direct use of the DLL's functions to match a regular expression and a subject string. A family of APL functions encapsulates much of this detail, making PCRE much easier to use than this outline implies.

1. Create an APL character vector containing the regular expression. This can be a quoted string or the result of other APL code.

2. Prepare a character vector containing the names of such PCRE options as may be needed.

3. Initialize three APL one item numeric vectors for the `errptr` and `erroffset` output arguments.

4. Using `QUAD-WCALL`, invoke `pcre_compile()` passing the regular expression vector, the options vector, the `errptr` variable, the `erroffset` variable, and a numeric zero for the `tableptr` argument. Capture the result of the call. This will be a three item numeric variable. The first item will be a `pcre` pointer or zero. If the first item is zero, the compilation failed. The other two result items contain the address of an error message(inside the DLL) and an offset into the regular expression related to the error. Use `W_Mem` to retrieve the text of the message.

5. Analyze the results of step 4. If it failed, using the error text and offset, figure out what went wrong and return to step 1. If it succeeded go on to step 6.

6. Obtain or construct an APL character vector containing the subject string.

7. Prepare a numeric vector to receive the starting and ending offsets of the matching substrings. PCRE uses the last third of this vector internally so it length should be more than three times the maximum expected number of matching and captured sub-strings. The length must be a multiple of three. PCRE refers to this as `ovector`.

8. Prepare an APL character vector containing the names of any required matching options. If no options are needed, pass a numeric zero instead

9. Using `QUAD-WCALL`, invoke `pcre_exec()` passing the `pcre` pointer, an integer zero, the subject vector, its length, an integer starting offset, the option vector or zero, the `ovector`, and the length of the `ovector`. Capture the result of the call. It should be a two item vector. The second item is the modified `ovector`. The first item is an integer return code. If the match succeeds, the return code will be 1 or greater and is the number of pairs of meaningful `ovector` items. If the call fails the return code is negative. A -1 indicates that the subject string didn't match the pattern. Other negative

numbers indicate serious errors. Consult the PCRE documents for their meaning.

10. If the match in step 9 succeeds continue with step 11. If the match fails take whatever action is appropriate for your application then continue at step 13.

11. It is convenient to reshape the meaningful leading elements of `ovector` into a two-column matrix. The first column contains starting offsets of matching or captured sub-strings of the subject. The second column contains their ending offsets. Offsets can be interpreted as zero origin indices of the first character and the character after the last character of a sub-string. Unused captures have offsets of -1. The first row, row zero in origin 0, describes the entire matching sub-string. Subsequent rows describe captured sub-strings in order. Their zero origin index is the same as their back-reference number. Row 1 corresponds to the string captured in Perl's $1 variable, Row 2 to $2, and so forth.

12. Use the results of the match as appropriate for your application.

13. Repeat from step 6 to reapply the regular expression to a different subject string or with a different offset. Otherwise continue to step 14.

14. This step releases the `pcre`'s storage for reuse. Using `QUAD-WCALL`, invoke `pcre_dispose⍬` passing the `pcre` pointer as the sole argument. No result is returned. **This `pcre` pointer must not be used again. Doing so will crash the interpreter or induce data corruption.**

The APL_PCRE Workspace

Functions for Pattern Matching

These functions encapsulate the complexity of direct calls to the PCRE library. Functions that directly call the library are described first, followed by functions that work with the result of a match. These are followed by simple compositions of the earlier functions. The last function displays the result of matching a regex to a string for experimentation and debugging.

`pcre` `<-` **`Compile`** *`regex options`*

`regex` should be a character vector representing a regular expression

`options` may be a character vector of the single letter option codes or a number. At least a zero or an empty string must be present.

`pcre` will be an integer containing a pointer to a `pcre` in the library's memory pool.

Syntactic errors in the regular expression are signalled as APL errors including PCRE's description and offset.

position_matrix <- *pcre offset* **Match** *subject*

pcre must be a pcre pointer.

offset is the offset into the subject where matching should begin. If not present, matching starts at the beginning, which is equivalent to specifying zero.

subject should be a character vector of text to be matched.

position_matrix will be a two-column matrix with one row for each matching or captured sub-string of the subject. If the match fails, the matrix will have zero rows. The first column contains starting offsets into the subject string; the second contains ending offsets.

The regular expression is matched against the subject string beginning at the offset specified. An APL error is signaled if the library returns an error other than a failing match.

position_matrix_vector <- *pcre offset* **MatchMultiple** *subject*

pcre must be a pcre pointer.

offset is the offset into the subject where matching should begin. If not present, matching starts at the beginning, which is equivalent to specifying zero.

subject should be a character vector of text to be matched.

position_matrix_vector will be a vector of two-column matrices as returned by Match.

The regular expression is repeatedly matched against the subject string until matching fails. The first match begins at the offset specified. Subsequent match begins at the end of the previous match. The result contains a position matrix for each successful match. An APL error is signaled if the library returns an error other than a failing match.

Dispose *pcre*

pcre must be a pcre pointer returned from the PCRE library.

The pcre is freed into the storage pool used by the library. Dispose should be called exactly once for each pcre created by PCRE.

`vector_of_strings <- position_matrix` **`SplitCaptures`** *`subject_string`*

> *`positon_matrix`* must be a matrix as returned by Match
>
> *`subject_string`* is the subject character vector from the match
>
> *`vector_of_strings`* will be a vector containing the sub-strings described by the position matrix.

An empty vector is returned for a zero-row position matrix.

`vector_of_strings <-` *`pcre`* **`MatchAndCapture`** *`subject`*

> *`pcre`* must be a `pcre` pointer.
>
> *`subject`* should be a character vector of text to be matched.
>
> *`vector_of_strings`* will be a vector containing the sub-strings matched or captured

The regular expression is matched against subject to return a vector of matched and captured sub-strings. This is a simple composition of `Match` and `SplitCaptures`.

`vector_of_strings <-` *`regex options`* **`C_M_C`** *`subject`*

> *`regex`* should be a character vector representing a regular expression
>
> *`options`* may be a character vector of the single letter option codes or a number. At least a zero or an empty string must be present.
>
> *`subject`* should be a character vector of text to be matched.
>
> *`vector_of_strings`* will be a vector containing the sub-strings matched or captured.

The regular expression is compiled and matched against the subject. A vector of matched and captured sub-strings. The pcre is disposed of internally. This is a convenient composition of Compile and MatchAndCapture.

`boolean_scalar <-` *`pcre`* **`DoesMatch`** *`subject`*

> *`pcre`* must be a `pcre` pointer.
>
> *`subject`* should be a character vector of text to be matched.

boolean_scalar will be 1 if the regular expression matches the subject and 0 if it does not.

The regular expression is matched against subject to return a vector of matched and captured sub-strings. Most of the work is done by `Match`.

result <- *regex options* **DemoRegex** *subject*

regex should be a character vector representing a regular expression

options may be a character vector of the single letter option codes or a number. At least a zero or an empty string must be present.

subject should be a character vector of text to be matched.

result is a character matrix.

The regular expression is compiled and matched against the subject. The subject and match results are formatted for display. The subject is on the top line. The capture matrix is at the right of the remaining lines. Digits mark the positions matched or captured. If the match fails, there is only one line.

```
      '^ \s* (\w*) \s* (\d*)f \s* $' 'xi' DemoRegex '  Princeton 72F  '
  Princeton 72F
00000000000000000  0 17
..111111111......  2 11
............22... 12 14

      '^ \s* (\w*) \s* (\d*)([fc]) \s* $' 'xi' DemoRegex 'Bethesda 20c'
Bethesda 20c
000000000000  0 12
11111111....  0  8
.........22.  9 11
...........3 11 12

      '^ \s* (\w*) \s* (\d*)([fc]) \s* $' 'xi' DemoRegex 'Erehwon 20R'
'Erehwon 20R'
```

Functions for Replacing Sub-strings

These functions interpolate captured sub-strings into a template. A template is a character vector. Within a template, most characters stand for themselves. A few meta-characters are used to specify points of substitution. [While only one is currently in use, several more are envisioned.]

Interpolation replaces a dollar-sign followed by a digit, *n*, with the *n*th matched or captured sub-string. The sub-strings can be interpolated multiple times and in any order.

string <- *template* InterpolateCaptures *vector_of_strings*

template is a character vector.

vector_of_strings is a vector of character vectors.

string will be a character vector.

The result is a copy of the template with the $-digit meta-sequences replaced by the corresponding item of vector_of_strings. $'s not followed by a digit are not changed.

```
    $2 is 1st then $1. $0 else comes last' InterpolateCaptures 'all' 'one' 'two'
  two is 1st then one. all else comes last
```

string <- *pcre template* **MatchAndReplace** *subject_string*

pcre is a pcre pointer.

template is a character vector.

subject_string is a character vector.

string will be a character vector.

The regular expression is matched with the subject. The matched and captured sub-strings of the subject are interpolated into the template. The result is a copy of the subject-string with the interpolated template replacing the segment that matched.

string <- *pcre template* **ReplaceAll** *subject_string*

pcre is a pcre pointer.

template is a character vector.

subject_string is a character vector.

string will be a character vector.

All matches are found for the regex in the subject. Copies of the template are interpolated with the captured sub-strings for each match. The result is a copy of the subject-string with the interpolated templates replacing the matching segments. Since all

matching and computation of replacement positions is done before any replacement, there is no risk of replacements being made in replacement text.

More Information

The documentation contained in the PCRE Library distribution is the primary reference for the regular expression engine.

Jeffrey E. F. Friedl. *Mastering Regular Expressions.* Second Edition. O'Reilly. 2002 Almost everything one would ever want to know about the use of regular expressions.

Larry Wall, Tom Christiansen & Jon Orwant. *Programming Perl. 3rd Edition.* O'Reilly. 2000. Affectionately known as the camel book, this is the standard reference for Larry Wall's Perl language. The sections on regular expressions explain the extensions from the viewpoint of their inventor.

There are many books on regular expressions and the languages and tools that use them. Those oriented to programmers working in your application domain may be helpful. Try not to be confused by examples in different regex dialects.

Try a Google search on "regular expression tutorial". This will locate Web tutorials intended for all levels and kinds of users. Those oriented to users of the Perl, Python, or Ruby languages may be most relevant.

Distribution Software

The conference digital material contains a directory that includes the following files("xx" will be replaced by a version identifier):

`Readme.txt`

Late news, installation instructions, etc.

`APL_PCRE_xx.dll`

A DLL containing the PCRE regular expression library and support functions for its use from APL. The DLL is most conveniently installed in the same directory as `aplw.exe`

`APL_PCRE_xx.w3`

This workspace contains cover functions for use with the DLL, a simple set of testing tools, and a set of test functions to exercise and validate the installation. The test function's names begin with "t_" followed by an integer followed by the test's subject. These functions are safely executable. Most produce no output unless something fails, in which case an error will be signaled. The niladic function F12 runs all the tests. The dyadic function AEM uses the APL match primitive to compare its arguments, signaling an error if they differ. The test functions use AEM extensively to compare expected to actual results. The tests are intended to be useful as examples.

`APL_PCRE_xx.ini`

This file contains QUAD-WCALL definitions for the DLL's functions and constants. The definitions should be copied into your `aplw.ini` file.

`Pcre-4.3.tar.gz`

This is an unmodified copy of the PCRE distribution. It can be uncompressed and opened with WinZip. The top-level directory contains source code for the PCRE library and several text files without suffixes. You should review the AUTHORS, COPYING, LICENSE, NEWS, and README files. The doc subdirectory contains PCRE documentation in several formats. `pcre.txt` is a plain text version. The version in the `html` directory can be opened with a web browser. Begin with `index.html`; it links to the other files. You should arrange ready access to this primary reference documentation.

`APL_PCRE_xx.zip`

This contains APL2000 written source code and MSVC project files needed to rebuild or modify the DLL.

`Apl_Regex_xx.doc`

This document, possibly corrected, extended, or updated.

Regular Expression Reference

Meta-Characters

\...	Escape or quote the next non-alphanumeric character suppressing and special meaning. Give the next alphanumeric character a special meaning if one is defined.
... | ...	Separate alternate sub-patterns
^	Assert start of string (or line, in multi-line mode)
$	Assert end of string (or line, in multi-line mode)
.	Match any one character in the subject except new-line (by default)
[	Start character class definition
(	Start sub-pattern
)	End sub-pattern
?	0 or 1 quantifier Also extends the meaning of (Also quantifier minimizer
*	0 or more quantifier
+	1 or more quantifier Also "possessive quantifier"
{	Start min/max quantifier

Pre-defined Character Classes

\d	Any decimal digit
\D	Any character that is not a decimal digit
\s	Any white-space character
\S	Any character that is not a white space character
\w	Any "word" character
\W	Any "non-word" character

Character Classes Defined Within Square Brackets

[	Begin inclusive character class. Only those characters specified.
x-*y*	Denotes characters from *x* thru *y*
[^	Begin exclusive character class. All characters except those specified.
x	The non-alphanumeric character *x* ignoring any special meaning. Used to specify] - ^ \ characters.
c	Include members of the pre-defined character class *c*. These characters are included, even within an exclusive definition(FIXME – EXAMPLE).
X	Denotes the character *x* itself except for] - ^ \
\	Quotes] - ^ \ within square brackets.
]	Ends character class

Simple Assertions

^	Matches at start of subject	Not multi-line
^	Matches at start of subject and immediately after an internal new-line.	Multi-line
$	Matches at end of subject or before a newline at the end	Not multi-line
$	Matches at end of subject and immediately before an internal new-line	Multi-line
\b	Matches at a word boundary	
\B	Matches when not at a word boundary	
\A	Matches at start of subject	
\G	Matches at first matching position in subject	
\Z	Matches at end of subject or before newline at end	
\z	Matches at end of subject	

Assertion Sub-patterns

(?=*sp*)	Matches if *sp* matches looking ahead
(?!*sp*)	Matches if *sp* does not match looking ahead
(?<!*sp*)	Matches if *sp* does not match looking behind
(?<=*sp*)	Matches if *sp* matches looking behind

Conditional Sub-patterns

(?(*condition*)*yes_pattern*)	Selects *yes_pattern* for matching next if *condition* is true.
(?(*condition*)*yes_pattern*|*no_pattern*)	Selects *yes_pattern* (or *no_pattern*) for matching next if *condition* is true (or false).

Conditional Sub-patterns Conditions

(*Number*)	Satisfied if the capturing sub-pattern of that *number* has previously matched.
(*assertion*)	Satisfied if the look-ahead or look-behind *assertion* is satisfied
(R)	Satisfied if a recursive call to the pattern has been made. The PCRE documentation describes this PCRE extension.

Quantifiers

{*min*,*max*}	At least *min* but no more than *max* repetitions.	Greedy
{*n*}	Exactly *n* repetitions.	Greedy
{*min*,}	At least *min* repetitions.	Greedy
*	Equivalent to {0,}	Greedy
+	Equivalent to {1,}	Greedy
?	Equivalent to {0,1}	Greedy

{*min*,*max*}?	At least *min* but no more than *max* repetitions.	Non-greedy
{*n*}?	Exactly *n* repetitions.	Non-greedy
{*min*,}?	At least *min* repetitions.	Non-greedy
*?	Equivalent to {0,}?	Non-greedy
+?	Equivalent to {1,}?	Non-greedy
??	Equivalent to {0,1}?	Non-greedy

Compile and Execution Time Options

PCRE_ANCHORED		C E	Require matches to be anchored
PCRE_CASELESS	i	C	Ignore alphabetic case distinctions
PCRE_DOLLAR_ENDONLY		C	
PCRE_DOTALL	s	C	Dot meta-character matches all characters including new-lines.
PCRE_EXTENDED	x	C	Extended syntax. Un-escaped white space is ignored.
PCRE_EXTRA		C	
PCRE_MULTILINE	m	C	
PCRE_NO_AUTO_CAPTURE		C	Suppresses numbered capturing parenthesis
PCRE_UNGREEDY		C	Inverts greediness of quantifiers
PCRE_UTF8		C	Controls UTF-8 support. This is not enabled in the APL2000 built DLL.
PCRE_NOTBOL		E	First char of subject is not beginning of line
PCRE_NOTEOL		E	End of subject is not the end of a line.
PCRE_NOTEMPTY		E	An empty string is not a valid match.

The names in the first column are PCRE's names for bit definitions used with the pcre functions. They have QUAD-WCALL definitions in the *.INI file.

The letters in the second column can be used in the option argument to the APL Compile function. They correspond to PERL's pattern modifier characters.

The third column indicates whether the option works at compile or execution time.

The last column briefly describes the effect of specifying the option. Consult the documentation for a more complete explanation. The options with letters in the second column are the most generally useful. The others control PCRE extensions, esoteric features, or specialized uses.

Getting Through to APL+Win from C# A Layman's Tour of .Net

by Adrian Smith (adrian@causeway.co.uk)

Background to .NET

Was it all just hype, or is there some serious meat in there? If my experience is typical, there is a lot of meat, and you should sit up and take note of where the programming world is headed. My experience has been:

- our *GraPL* customers are voting with their feet. We now have 6 existing sites switching to ASP.NET and many more enquiring how long we are going to be. Virtually all new sales enquiries are wanting .NET. Almost no-one is buying COM servers any more.
- C# is the simplest, most productive development environment I know, after APL. Jonathan is a lifelong Delphi fan, and has switched allegiance to C# almost overnight. Richard has rewritten his little R reverse polish language in C# from Java and greatly prefers it. This thing is happening faster than even Microsoft could have hoped.
- most of the best tools are free. *SharpDevelop* is a nice C# IDE and is open-source. The Microsoft compiler is free, so all you need to get going is the language spec (C# is an ECMA standard) and Notepad. Honest.

So in as few words as I can manage, what is it about .Net that has grabbed the world's programmers by the throat. Here are some thoughts:

- **The Common Type System.** All .Net languages share a common set of basic data types. This allows you to build an application from a collection of code blocks written in a variety of different languages, and have them work smoothly together.
- **Simplicity.** There are no header files, no pre-processor macros, no linker. You just compile your code, mentioning all the other modules you need on the command line. This has the huge benefit that if you make a mistake in calling someone else's code, you get to know at compile time.
- **System utilities.** All the basic stuff is there for the asking. String-handling, date-manipulation, whatever. Imagine the contents of all the best utility libraries you know properly catalogued and made available to everyone, all the time.

- **Easy deployment.** Just copy a bunch of files. No messing with OLE registrations or other nasty stuff. Because everything uses the .Net runtime classes, your application can be *REALLY* small. Just like the old days.

Enough. I think this is going to take over the development world very fast indeed, and I want to be part of this game with my APL stuff. There is a way to do it now, with nothing beyond APL 3.5 and a little ingenuity. I am sure that in the future it will get better, but let's see how far we can get with what we have today.

COM InterOp Here we Come

There is just one small piece of administration to be done before we can get to the +Win COM server from C#. We need to run a command-line utility called *tlbimp* from the framework SDK to make a .Net cover for the APL+ COM server. On my machine this looks like:

```
D:\Tools\aplc>d:\tools\microsoft.net\frameworksdk\bin\tlbimp
      d:\apl\win35\aplwco.dll
```

Which automatically creates a .Net DLL for us. In the examples that follow, I additionally specified `/out:aplnet.dll` as another parameter, so all my tests are run against a mapped COM server called *aplnet*. Of course there is the usual irritation that if you have registered the RUNTIME engine you can only load a runtimed workspace, and vice versa. I would recommend registering a development interpreter if you want to play with the examples. Then you can leave APL open in one window, hit)save and rerun your C# example with no problems.

So what can we do with it? Well let's open up a Notepad and type a bit of C# to see what happens. Here is `runapl.cs` which does a few basics:

```
using aplnet;
using System;

class TestApp
{
  public static void Main()
  {
    WSEngine ws = new WSEngine();
    Console.WriteLine("Hello World using C#!");
   // Just play with expressions - limited usefulness
    Console.WriteLine(ws.Exec("2*3").ToString());
   // Note that there is NO mapping here to []AV characters
    Console.WriteLine(ws.Exec("⍳3+4 7"));
    Console.WriteLine(ws.Exec("'Hello from APL',⍕1000'2"));
```

```
      Console.WriteLine(ws.Exec("⍳⍵FI'23 34 45'"));

    // Get some prebuilt stuff (remember to flag as runtimed)
      ws.SysCommand("load c:\\data\\W3\\demo");

      object result = ws.Call0("HelloWorld");  // Call with no arguments
      Console.WriteLine(result);

    }
}
```

Probably best to compile and run it, then stop and explain what is going on. Here we use `csc` (the Microsoft compiler is free with the .Net framework) mentioning that we need the resource `aplnet.dll` in our source code. It creates `runapl.exe` which I can try out from the command line to see the effect.

```
D:\Tools\aplc>csc /r:aplnet.dll runapl.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.


D:\Tools\aplc>runapl
Hello World using C#!
8
7 10
Hello from APL2000
23 34 45
Hello, world from APL
```

Well, it seems to work, so let's take it apart a line at a time.

```
using aplnet;
using System;
```

This just saves me some typing – effectively it sets the compiler's search path to include the *aplnet* and *System* namespaces. Otherwise I would have to type `System.Console.xxx` every time, and that gets tedious.

```
class TestApp
{
  public static void Main()
  {
```

In C# everything is a class, even the most basic `HelloWorld` program. A command line application needs a *Main* method so it knows what to run. Now it gets interesting.

```
WSEngine ws = new WSEngine();
Console.WriteLine("Hello World using C#!");
// Just play with expressions - limited usefulness
Console.WriteLine(ws.Exec("2*3").ToString());
```

The APL com server exposes a few well-known methods to allow external applications to run expressions, load workspaces and so on. Here, we create an instance of the *WSEngine* class, called *ws*, mainly so I can paste code from the APL2000 help files! We can have APL raise 2 to the power 3, and return a result (which will be of type object). Anything of type object will have a basic formatting capability built in, so I can use it here to get my number made into text. Then I write it to the console. Look back a few lines and notice the 8 that was displayed. OK, it's a start.

```
// Note that there is NO mapping here to ⎕AV characters
Console.WriteLine(ws.Exec("⍵3+4 7"));
Console.WriteLine(ws.Exec("'Hello from APL',⍵1000'2"));
Console.WriteLine(ws.Exec("⍵∘FI'23 34 45'"));
```

This took a bit more work. Those funny squiggles are the characters which occur in the Courier font at the same positions as the APL symbols occur in ⎕av. So (for example) ⍵ is the 244th character, and if you try ⎕av⍳'⍵' you will get 245 back. Remember to take one off! Really, this is just fooling around, what we really want to do is to load a workspace and have our C# run some real code.

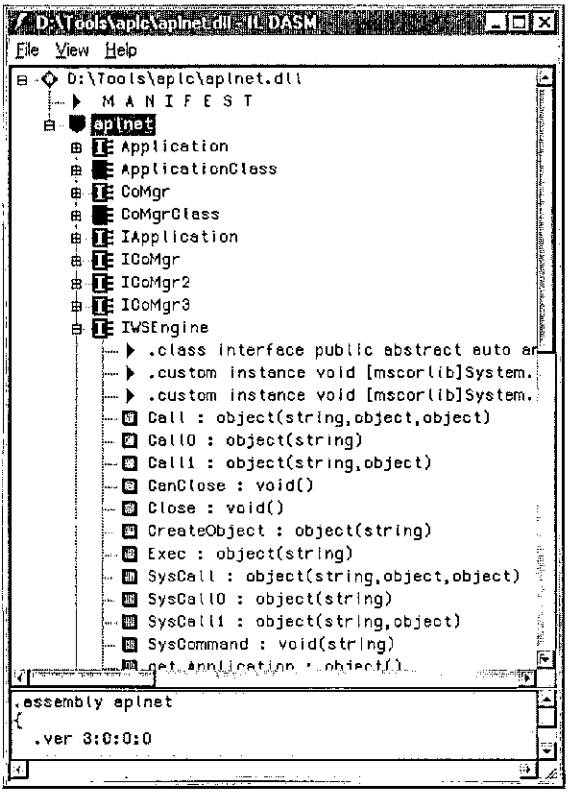
```
// Get some prebuilt stuff (remember to flag as runtimed)
    ws.SysCommand("load c:\\data\\W3\\demo");

    object result = ws.Call0("HelloWorld");  // Call with no arguments
    Console.WriteLine(result);
```

So far so good. As long as we have the right workspace name, this will go ahead and load it for us. Now, the next line deserves some more explanation. `HelloWorld` is a niladic function which returns the obvious string as a result. If I try to use `ws.Call(...)` here, it refuses to compile:

```
runapl.cs(20,18): error CS1501: No overload for method 'Call'
takes '1' arguments
```

The reason? Unlike the COM interface, .Net has no concept of optional arguments, and the `tlbimp` utility has had to cope. The way C# works is actually rather more flexible, but quite hard for an APLer to get used to. Imagine that you can have lots of different versions of `foo` in the same workspace at the same time! One might be defined as `res←foo arg` (monadic, result-returning) while another could be `res←larg foo rarg` and so on. Of course one might call another, having done

something like set a default for the left argument. Or it might do something *completely* different (think `⍳`). This is how C# works, and it is called *overloading*. You can have as many copies of a function as you like, as long as they all have different signatures. Useful, and flexible.



So this is what I expected `tlbimp` to do, but oddly it has made a set of functions with suffixed names that map to the various possible syntaxes for the *Call* method. To find out exactly what is available, we use another handy utility called `ildasm`, which is the ultimate Swiss Army knife of the .Net developer. Put a shortcut to it on you desktop, then drop `aplnet.dll` on to it, and all will be revealed

Now what have we here? There is *Call*, as expected, but we also have *Call0* and *Call1* for the niladic and monadic cases. For our `HelloWorld` function, we maybe could try *Call0*, instead of Call:

```
object result = ws.Call0("HelloWorld");
Console.WriteLine(result);
```

Works fine. It compiles, and completes our test by writing the result of the function to the console.

Now we have all the tools we need to move ahead. We can load a workspace, call code in it, execute arbitrary expressions, and get results back that our C# application can use. Let's move on to make something that might actually be useful.

Making a Function into a Method

Clearly, we want the C# programmer to see our function (and only our function) with a clearly specified set of argument types and possible results. Because APL has no need of all this 'type' information, we will need to add it in somewhere. I have chosen to do it with comments, but there are plenty of possibilities. Use whatever suits you best.

Let's do Breakback!

A little function I showed at APL Berlin 2000 will do nicely, as it can be called in several different ways, and actually does something mildly useful and array-ish.

```
    ∇ new←larg Redo ttl;holds;old;cum;held;lot
[1]   ⍝ Integer vector redistribution with optional held items.
[2]   ⍝ Result is 'integer' at the specified lotsize
[3]
[4]    :If 1≤≡larg
[5]      (old holds lot)←3↑larg,1
[6]    :Else
[7]      (old holds lot)←larg 0 1
[8]    :End
[9]
[10]   :If 0∈holds  ⍝ we can proceed
[11]     held←holds×old
[12]     cum←+\((~holds)×old)÷lot
[13]    ⍝ End-stop required if the only unheld cell is zero!
[14]     :If 0=¯1↑cum ⋄ new←old ⋄ :Return ⋄ :End
[15]     new←¯2-/0,⌊0.5+cum×((ttl-+/held)÷lot)÷¯1↑cum
[16]     new←held+lot×new
[17]   :Else  ⍝ all cells locked in
[18]     new←old
[19]   :End
    ∇

      vv
4 3 2 5
      +/vv
14
     vv Redo 17
5 4 2 6
     +/vv Redo 17
17
```

The basic call simply reworks the vector to sum to a new total. It can also take a matching vector of 'held' elements, and a final option is to set the 'lot size' of the result, forcing all the numbers to be integral multiples of this value. That is 4 C# overloads to make, each of which will just call our base function with suitable defaults. Let's add in some comment lines:

```
A:Public int[] Redo (int[] old, int newtotal)
A:Public int[] Redo (int[] old, int lotsize, int newtotal)
A:Public int[] Redo (int[] old, bool[] heldcells, int newtotal)
A:Public int[] Redo (int[] old, bool[] heldcells, int lotsize, int newtotal)
```

These could have any syntax you like, but the lazy choice seems to be to make the function signatures look just like the C# we are about to generate! Note that this is not limited to the cases with 1,2,3 arguments (as the APL was) as we can make overloads for any reasonable combinations. So, what should this look like in C#? Here is a hand-cut version to show the general idea

```
using aplnet;
using System;

public class BreakBack
{
  WSEngine ws;

  public BreakBack() {  // constructor
    ws = new WSEngine();
    ws.SysCommand("load c:\\data\\W3\\demo");
  }

  public int[] Redo(int[] old, int newtotal)
  {
    object result = ws.Call("Redo",newtotal,old);
    return (int[]) result;
  }

  public int[] Redo (int[] old, int lotsize, int newtotal)
  {
    object result = ws.Call("Redo",newtotal,new object[] {old,0,lotsize});
    return (int[]) result;
  }

  public int[] Redo (int[] old, bool[] heldcells, int newtotal)
  {
    object result = ws.Call("Redo",newtotal,new object[] {old,heldcells,1});
    return (int[]) result;
  }

  public int[] Redo (int[] old, bool[] heldcells, int lotsize, int newtotal)
  {
    object result = ws.Call("Redo",newtotal,new object[] {old,heldcells,lotsize});
    return (int[]) result;
  }
}
```

The only tricky part is to get the arguments swapped around! The Call method on the COM server takes them in the order `Rarg , Larg`, which is why `newtotal` always comes first. Apart from that, we can start to contemplate creating all this completely mechanically from the original function and a few extra comments. How does it look from the outside world? Well, lets compile it to a DLL and have a look.

```
D:\Tools\aplc>csc /r:aplnet.dll /t:library redo.cs
```

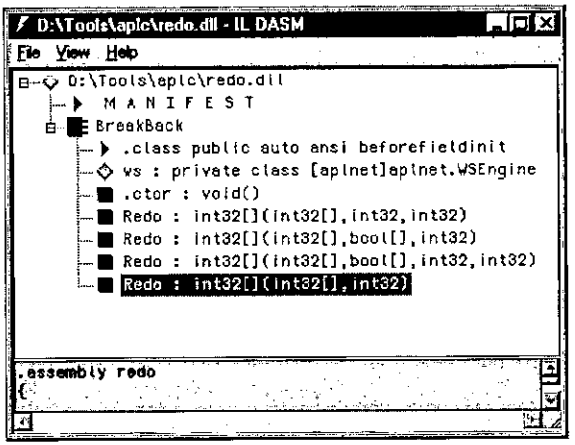
The only small addition is that we specify a library here, rather than getting the compiler to make an EXE. Hence the lack of a `Main()` function in the listing above. Now to try it out:

```
BreakBack bbk = new BreakBack();
int[] data = {4,3,2,5};
int[] res;
Console.WriteLine("Simple call to retotal to 17");
res = bbk.Redo(data,17);
foreach (int i in res) { Console.Write(i+" "); }
Console.WriteLine("");
```

And so on, for the 4 possibilities. This time we compile our code with the extra DLL mentioned on the command line, and then we can test it straight away:

```
D:\Tools\aplc>csc /r:aplnet.dll,redo.dll runapl.cs
D:\Tools\aplc>runapl
Simple call to retotal to 17
5 4 2 6
Retotal to 18 and lotsize in 2s
6 4 2 6
Retotal to 24 and hold first value
4 6 4 10
Retotal to 27 hold first value and lotsize in 3s
4 6 6 12
```

So, to recap. We can take a bunch of functions in a conventional APL +Win 3.5 workspace, add a few declarative comments, and with a bit of work create the C# source code which uses the APLW COM server (via COM InterOp) to make a completely standard .Net 'assembly' that any C# programmer will be happy to work with.



For comparison, here is the dis-assembly that the C# developer would be looking at. It is simple, and easy to work with. The redo DLL is all of 4kb in size, and even the aplnet dll is a mere 13kb. As I said at the beginning, this stuff is lean and mean and really easy to deploy.

It will work with any of the many .Net languages and can be used in a web-page created on the fly with ASP.Net.

Where Next?

That is easy to answer. What we need is for APL2000 to make the InterOp step redundant by bridging directly to the interpreter from C#. This allows the APL developer to play on a totally level field, along with the C# guys and anyone else who comes along. That legacy code could still be running in 20 years time, rebadged as .Net assemblies and fully accepted by the IT management as a good citizen of the brave new Microsoft world.

For now, we will have to fudge it, but as workarounds go, it is not too unpleasant.

Bell's Theorem and Matrix Mechanics

by Graeme D. Robertson (GraemeDR@aol.com)

An argument between Bohr and Einstein on the meaning of quantum mechanical reality is given a significant twist in Bohr's favour by Bell's inequality and its experimental refutation. I attempt to employ APL in the elucidation of quantum mechanical description of physical reality.

Two articles in Vector by Sylvia Camacho [1, 2] prompted me, eventually, to respond. After a few recent personal letters, books and emails between Sylvia and myself, I decided to write an article for Vector which, although it might raise more questions than it answers, at least demonstrates that I too am *very* interested in the whole business of reality.

Bell's Theorem

> *No theory can give (a) contingent general predictions of the individual results of measurements, (b) be compatible with the statistical predictions of quantum mechanics (even to within say 5%) and (c) satisfy local causality.*

This means that you can't rationally believe, as Einstein did, in both determinism (that every effect is uniquely caused) and locality (that things are separable) while at the same time accepting the predictions of quantum theory. Einstein believed that quantum theory is like statistical mechanics and that *hidden variables* would eventually be discovered which reveal an underlying *local deterministic structure.*

A mathematical inequality, based on some very general assumptions about the nature of reality, was derived by John Bell [3] in 1964. Yet, as Bell pointed out, this innocent-looking inequality is violated by the predictions of quantum mechanics regarding, for example, two-component spinors. As such, the inequality violation should be visible in many departments of optical, atomic, nuclear and sub-nuclear physics. A number of experiments with electrons, photons and protons have been performed to test the validity of the inequality. The first experiments were performed in 1969 by Clauser in Berkeley and Holt at Harvard using photons. The experimental and theoretical conclusions clearly favour quantum theory over all classical (*local deterministic*) versions of reality.

Bell showed that whatever hidden variables one might care to introduce, if they are constrained to be local to the environment of the object to which they refer then such a hidden variable theory will never be able to reproduce all the predictions of quantum mechanics. In particular, if two two-state (quantum

binary) systems interacted for a time and 'separated', the correlations which are predicted by quantum mechanics, and which have been observed in experiments, are stronger than can be accounted for by theories which are realistic in the sense proposed by Einstein, Podolski and Rosen [4] (who were immediately, in 1935, rebuffed by Bohr [5]).

Bell's Inequality

A simple account of Bell's inequality is given by David Harrison of Toronto University in his web page [6]. According to Harrison the essence of Bell's inequality can be summarized as follows.

> *The number of objects which have property A but not property B plus the number of objects which have property B but not property C is greater than or equal to the number of objects which have property A but not property C.*

If this appears trivially obvious then that's good because now you can see the paradox, namely: this common sense does not necessarily hold sway in quantum mechanics. (Neither does an uncritical application of Boolean algebra!) The crux of the problem revolves around the assumption that *objects have properties* independent of their observation.

If one tries to describe two systems, which have interacted in the past and have then separated, by *disjoint probability distributions* involving arbitrary *local hidden variables*, then this leads to predictions different from those of quantum mechanics. According to quantum mechanics, two (or more) systems can retain observable properties which are indefinitely outrageously strongly correlated even after the two systems have completely 'separated' in the usual sense of the word (including all requirements of relativity theory). There are correlations between observables that are stronger than can be imagined in classical physics. Such (classically independent) systems are said to be *entangled*.

Another way of expressing Bell's inequality in words may be as follows.

> *If the joint probability distributions of correlated observables are assumed to become disjoint distributions by the addition of arbitrary local hidden variables, some correlations predicted by quantum theory are greater than can be achieved by any such local hidden variable theory.*

Consider the implications of a conservation law, such as conservation of energy or momentum (or angular momentum in the case described by Bell). If an object with zero initial momentum spontaneously splits in two parts, then measurement of the momentum of one part will imply exactly an equal and opposite momentum of the other part – by conservation of momentum. This necessitates a strong

deterministic relationship between functions describing the states of the two supposedly-separate parts. In the case of the intrinsic spin angular momentum of an electron, or the polarization vector of a photon, there are only two possible outcomes, up (measures +1) or down (measures -1) which makes the product of the two results necessarily ±1.

Note that we are here, and in what follows, using units in which Planck's constant divided by 4π is equal to one unit of spin angular momentum (action).

1 Bell unit = $0.527295...10^{-27}$ erg sec

Consider two identical quantum systems labelled I and II that interacted and separated. Measure some two-component quantum observable of system I by some experimental arrangement described by *instrument vector* $\underline{a}$ (for example, the orientation of a polarizer). This measurement will be represented mathematically in a deterministic theory by some unspecified function $A(\underline{a})$, or a function $A(\underline{a}, \lambda)$ where λ represents any local hidden variables or functions, but **not** by a function $A(\underline{a}, \underline{b})$ nor a function $A(\underline{a}, \underline{b}, \lambda)$, by the locality postulate. Measure the same property of system II with instrument vector $\underline{b}$. A deterministic theory should be able to predict the result from some unspecified function $B(\underline{b})$, or a function $B(\underline{b}, \lambda)$ where λ represents any local hidden variables or functions, but **not** by a function $B(\underline{a}, \underline{b})$ nor a function $B(\underline{a}, \underline{b}, \lambda)$, by the locality postulate.

The expectation value of the product of the two results, written $P(\underline{a}, \underline{b})$, should be given in terms of a separable product of two *disjoint distributions* ($A(\underline{a}, \lambda)$ for system I and $B(\underline{b}, \lambda)$ for system II) as opposed to a single joint distribution describing I and II together (as a superposition of states in quantum theory). Any arbitrary distribution, $\rho(\lambda)$, of local hidden variable(s), λ, can be added without changing the following conclusion.

The most general local deterministic formula for the product of the results of measurements of I at $\underline{a}$ and II at $\underline{b}$ is

$$P(\underline{a}, \underline{b}) = \int \rho(\lambda) A(\underline{a}, \lambda) B(\underline{b}, \lambda) d\lambda$$

Introducing a third orientation, $\underline{c}$, Bell showed that this disjoint integral expression, plus the assumption that $P(\underline{a}, \underline{a}) = -1$, mathematically implies that

$$1 + P(\underline{b}, \underline{c}) \geq \left| P(\underline{a}, \underline{b}) - P(\underline{a}, \underline{c}) \right|$$

He then proceeded to show that the quantum mechanical prediction for the product of polarizations measured at orientation $\underline{a}$ for I and $\underline{b}$ for photon II when the pair are in the singlet state, $| \Psi_{I,II} \rangle$, is given by the joint distribution

$$P(\underline{a}, \underline{b}) = \langle \Psi_{I,II} \mid \underline{\sigma} . \underline{a} \otimes \underline{\sigma} . \underline{b} \mid \Psi_{I,II} \rangle = - \underline{a} . \underline{b}$$

This does **not** always satisfy the inequality, even to within 5%. Try, for example, $\underline{a}$ and $\underline{b}$ at 30° and $\underline{b}$ and $\underline{c}$ at right angles which would imply $1 \geq \sqrt{3}$.

Bell's inequality is important because it highlights the philosophically shocking nature of quantum mechanics [7, 8]. Bell's result – sometimes called *passion at a distance* – is, however, not of huge significance to most working physicists because physicists have accepted for over 50 years that quantum mechanics works perfectly well and they have already discovered that modern physics has many amazing consequences.

Bell's theorem necessitates a view of reality, or *world view*, in which the dramatic and astonishing revelations of quantum theory become intelligible, or at least acceptable. So let's try to do some simple quantum mechanics and see if we can derive any sense from it.

Matrix Mechanics

Just to keep alive those worldly readers who would turn off at the mention of reality, I offer the tantalizing hypothesis *that at least some of the correlations found empirically in stock market series [9] might be quantum correlations.* Quantum correlations involve joint distributions that produce correlations between observables which are *stronger that can be imagined in terms of any local deterministic model.*

There are at least three distinct formulations of quantum mechanics. There is the original 1925 discrete *matrix mechanics* of Heisenberg, then there is the Schrödinger continuous differential *wave mechanics* of 1926. There is also the rationalized notation of Dirac in 1930 which formulates *quantum theory* in abstract Hilbert space. All three theories are equivalent and give the same predictions. Matrix mechanics is closest in spirit to APL. You can think of wave mechanics as involving differential operators or *infinite matrices* which are not conducive for APL. Dirac notation is more symbolic and accommodates all mathematical representations. Hilbert space is a 'complete inner product space' which can be *infinite dimensional.* And each point in a Hilbert space can correspond to a *function.* We shall use some Dirac notation to annotate our simple APL model of matrix mechanics. Forgive me – some symbols used *in the comments* are not in ⎕AV.

Qubits

A two-state (Boolean) system can be represented by one bit which can be either 0 or 1 in value – you're with me so far? Quantum mechanically a two-state system is

represented by one quantum bit, or *qubit*. A qubit can be `|0>` or `|1>` or *any linear combination* of these two states, often written as `|ψ> = α|0> + β|1>` in Dirac notation. A good working account of this formalism is given by Nielsen and Chuang [10]. (I am excited to see from comp.lang.apl that Paul Chapman is studying quantum computation – I hope he will help me over some of my own conceptual difficulties with quantum information one day.)

In APL terms, a qubit, `|ψ>`, can be written as a two element vector `(α,β)` with vector space basis elements `|0>≡1 0` and `|1>≡0 1`. Consider the stationary state of a one qubit system represented by APL vector ψ:

```
      ψ ← 1  2 ⍝ |ψ> = 1|0> + 2|1>
```

Let me show you the font I'm using (APL99.TTF). If I use an unfamiliar symbol, you can work out what it corresponds to in the usual Dyalog Std font. For example, ψ is my font substitute for *ä*.

```
      5 57⍴(5×57)↑12↓⎕AV
%'αω_abcdefghijklmnopqrstuvwxyz__¯.⍬0123456789_¤¥$£¢∆ABCD
EFGHIJKLMNOPQRSTUVWXYZ__┬⍞  ⍙ABCDEFGHIJKLMNOPQRSTUVWXYZ{}
⊣⌷¨±∴∵²≃¼ƒgg∃β½πÔψ⍂¨¹▐e ƒƒƒg⍟█┘[/⌿\⍀<≤=≥>≠∨∧-+÷×?∊⍴~↑↓⍳○*⌈⌊
∇∘(⊂⊃∩⊥⊤|;,⍱⍲⍒⍋⍉⌽⊖⍟!⍕⍎   ⍫⍷≡≢┐┌┼⍒#-&_______________@─├┤_⊥'_¶
:⍷¿¡⋄←→⍝)] §⎕⍞|
```

Note that ψ above is not a unit vector – its length is not one. Quantum mechanical states can always be normalized to unit vectors without loss of generality because all vectors can be multiplied by an *arbitrary phase factor* without changing any predictions. All calculations of observable values involve multiplying the state by its complex conjugate which cancels the arbitrary phase. (The arbitrary phase is however fundamental to the introduction of electromagnetism through Yang-Mills gauge theory via this exact rotation symmetry.)

Using orthonormal basis vectors – unit vectors which are orthogonal to each other – considerably simplifies quantum mechanical calculations. In APL notation, with a comment in Dirac notation, the orthonormality is clear from

```
      1 0 +.× 1 0 ⍝ <0|0> = 1
      1 0 +.× 0 1 ⍝ <0|1> = 0
      0 1 +.× 1 0 ⍝ <1|0> = 0
      0 1 +.× 0 1 ⍝ <1|1> = 1
```

or, in a single expression which returns the required 2 × 2 unit matrix,

```
      (⍳2 2⍴1 0 0 1)+.×(2 2⍴1 0 0 1)   ⍝ <i|j> ≡ δ_ij
```

We normalize a vector by dividing by its norm (length).

```
      ⍞←ψ←ψ÷(ψ+.×ψ)*0.5⍝  |ψ>=|ψ>/<ψ|ψ>^½  =  (1|0>+2|1>)/5^½
0.447 0.894
```

Alternatively, we could return this normalized state from a normalization function applied to the relative weights of the superposition:

```
      ψ ←{ω÷0.5*⍨+.×⍨ω}1 2
```

Note the new monadic monistic Dyalog implementation of commute (⍨). Check that ψ is now a unit vector, *i.e.* that ψ+ . ×ψ is unity.

In matrix mechanics a measurable (*observable*) quantity is represented mathematically by an n × n *matrix*. This matrix acts on an n-component state vector (via inner products) to yield the possible values of the measurement and the relative weights of these possible outcomes. Normalized states which do not change when multiplied by the observable's matrix 'operator' are called *eigenvectors* and correspond to states of definite (deterministic) measured value. The value measured for an eigenvector state (the *eigenvalue*) is the scaling factor of the eigenvector after inner product with the matrix. (In wave mechanics matrix 'operators' become differential operators, and in abstract quantum theory they become abstract operators acting on vectors in Hilbert space – but the concept of eigenvectors (now eigenfunctions) and eigenvalues remains fundamental.)

In the case of measurement of two-component spin states (spinors), the matrix used to represent measurement of the z component of spin is the Pauli matrix

```
      Ô ← 2 2 ⍴ 1 0 0 ¯1 ⍝ σ_z Pauli z matrix
```

Clearly, this matrix, acting on the basis states leaves them unchanged, apart from a scaling factor. The scaling factor is interpreted as the result of the measurement.

```
      Ô+.×1 0 ⍝ σ_z|0>=1|0>
1 0

      Ô+.×0 1 ⍝ σ_z|1>=¯1|1>
0 ¯1
```

There is a function called Eigen, based on LAPACK and to be found in the Dyalog distributed workspace ...\ws\math.dws, that calculates the eigenvalues and eigenvectors for any given square positive-definite matrix. We define

```
      ⍁←{Eigen ⍵} ⍝ Eigen function from MATH workspace
```

Then applying this to $\hat{O}(\sigma_z)$ yields the expected row of eigenvalues above the corresponding columns of eigenvectors.

```
      ⍁ Ô  ⍝ Eigenvalues⍪Eigenvectors of Ô(σz)
1 ¯1
1  0
0  1
```

We can form a matrix from our state ψ above using outer product. This will produce a matrix whose action will definitely leave ψ unchanged (through normalization). So the matrix

```
      ψ∘.×ψ  ⍝ |ψ><ψ| Outer Product of col & row vectors
0.2 0.4
0.4 0.8
```

represents an observable that will definitely yield value 1 and leave the state unchanged.

```
      ⍁ ψ∘.×ψ
1       0
0.447 ¯0.894
0.894  0.447
```

What is the significance of the other eigenvector of this observable? Check that `(¯0.894 0.447)` is normal and orthogonal to ψ, with eigenvalue 0.

Entangled Qubits

When a state is described by two qubits, the four basis states correspond to the normalized outer product of two single qubit basis states, $|\Psi\rangle = |\psi\rangle \otimes |\varphi\rangle$. The combined system now has 2^2 component state vectors.

For example a two qubit state, Q, in which each qubit is in state ψ is

```
      Q ← ,ψ∘.×ψ  ⍝ |ψ>⊗|ψ>
```

Generally, therefore, an n-qubit state will have 2^n components (a real APLer would want to keep these dimensions distinct rather than ravel them).

The Fourier transform of a quantum state is also a quantum state in a complementary basis. The discrete Fourier transform transforms the elements of a vector of N numbers $|\psi\rangle$ into another vector of N numbers $|\varphi\rangle$ via

$$\left|\varphi_j\right\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \exp(2\pi i jk/N) \left|\psi_k\right\rangle$$

There is a transform function called Fourier, based on FFTW and to be found in the Dyalog distributed workspace ...\ws\math.dws, that calculates the discrete Fourier transform of any vector. We define

```
      f ←{1 Fourier ⍵} ⍝ Fourier transform from MATH.DWS
      f⍣¯1 ←{¯1 Fourier ⍵}  ⍝ Inverse Fourier transform
```

Applying f to **Q** we get

```
      P ← f(Q)
```

where we note that **P** has become complex in a notation where `0J1` is now `⊂0 1`

```
      P ≡ (0.9 0)(¯0.1 0.2)(¯0.3 0)(¯0.1 ¯0.2)
1
```

P is now a *complex vector*. Check that $f^{-1}(\mathbf{P})$ is **Q**.

We must define a multiplication function for complex numbers to use in the inner product (in Hilbert space $\mathbf{C}^4$).

```
      x ←{+/(2 2⍴1 ¯1 1)×0 1⌽(2↑⍺)∘.×2↑⍵} ⍝ Multiply
```

Test that **P** is still a unit vector **using x in place of ×.**

```
   Õ+.x Õ © <Õ|Õ>?
0.84 0
```

The length is real but not unity! The reason for the erroneous result is that the definition of inner product $\langle \mathbf{P} | \mathbf{P} \rangle$ involves complex conjugation – the dual vector $\langle \mathbf{P}|$ is the complex transpose of $|\mathbf{P}\rangle$. This ensures that length $\langle \mathbf{P} | \mathbf{P} \rangle$ is always a real number. We define a conjugation function,

```
      c ←{⍵×⊂1 ¯1}  ⍝ Complex Conjugate
```

and check that the correct definition of inner product gives unity.

```
      (c P)+.× P ⍝ <P|P>=1
1 0
```

Since all measurements yield real numbers, eigenvalues must be real numbers. This implies mathematically that matrix 'operators' corresponding to measurements must be *Hermitian, i.e.* equal to their complex transpose.

Returning to the quantum mechanical derivation for $P(\underline{a}, \underline{b})$, here is a vector of Hermitian matrices with an algebra suitable for representing angular momentum measurements.

$$\underline{\sigma} \equiv (\sigma_x, \sigma_y, \sigma_z) = \left[\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \right]$$

Here is a normalized circularly polarized singlet state in terms of which the combined system I + II is described:

$$|\Psi_{I,II}\rangle = (|01\rangle - |10\rangle)/\sqrt{2}$$

$$= (|0\rangle \otimes |1\rangle - |1\rangle \otimes |0\rangle)/\sqrt{2}$$

This 'Bell state' is written in APL as

```
      B←,(1 0∘.×0 1)-(0 1∘.×1 0)
      ⎕←B←2↑"B÷2*.5 ⍝ normalizing & complexifying
0 0  0.707 0  ¯0.707 0  0 0
```

We can now begin to see the meaning of the expression $\langle \Psi_{I,II} | \ \underline{\sigma} \cdot \underline{a} \otimes \underline{\sigma} \cdot \underline{b} \ | \Psi_{I,II} \rangle$ for calculating the quantum mechanical values in Bell's inequality. The orientation of polarizer I is at angle θ to the x axis in the x-y plane. Therefore $\underline{a} = (Cos(\theta), Sin(\theta), 0)$. Similarly, $\underline{b} = (Cos(\phi), Sin(\phi), Z)$. The matrix operators representing the measurement of spin angular momentum (in Bells) with instruments at orientations $\underline{a}$ and $\underline{b}$ involves generalized inner products $\underline{\sigma} \cdot \underline{a}$ and $\underline{\sigma} \cdot \underline{b}$.

$$\underline{\sigma} \cdot \underline{a} = \begin{pmatrix} 0 & Cos(\theta) \\ Cos(\theta) & 0 \end{pmatrix} + \begin{pmatrix} 0 & -iSin(\theta) \\ iSin(\theta) & 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 & (\sqrt{3} - i) \\ (\sqrt{3} + i) & 0 \end{pmatrix}$$

$$\underline{\sigma} \cdot \underline{b} = \frac{1}{2} \begin{pmatrix} 1{,}000{,}000 & (\sqrt{3} + i) \\ (\sqrt{3} - i) & -1{,}000{,}000 \end{pmatrix}$$

when $\theta = 30°$, $\phi = -30°$ and the distance between the two instruments, Z, is 1 million units, say. The final correlation matrix for the joint observation of $\underline{a}$ for I at angle 30° and $\underline{b}$ for II at -30° is the outer product of $\underline{\sigma} \cdot \underline{a}$ and $\underline{\sigma} \cdot \underline{b}$. In APL

```
Ôa←2 2ρ0.5×0((3*0.5)¯1)((3*0.5)1)0 ⍝ = σ.a
Ôb←2 2ρ0.5×2E6((3*0.5)1)((3*0.5)¯1)¯2E6 ⍝ = σ.b
AB←4 4ρ1 3 2 4⍉Ôa∘.x Ôb  ⍝ = σ.a⊗σ.b
```

The expectation value for this observable in state $|\ \Psi_{I,II}\ \rangle$ should be $-\underline{a}\,.\,\underline{b}$ which is equal to $-\,Cos(\theta\text{-}\phi)$ or, in this case, $-\,Cos(60°) = -½$.

```
(c B)+.x AB+.x B  ⍝ ⟨Ψ_I,II| σ.a⊗σ.b |Ψ_I,II⟩
¯0.5 0
```

There is, of course, another inequality – that could be called *Heisenberg's inequality*, but is actually known as *the uncertainty principle* – which we should be able to demonstrate from two complementary matrix 'operators' such as

```
Ôp ← P∘.x c P  ◊  Ôq ← Q∘.x Q
```

which are Hermitian, Ôp≡c⍉Ôp ◊ Ôq≡c⍉Ôq, and don't commute.

The uncertainties (or standard deviations) can be calculated for any given state. In particular, a linear combination of eigenvectors, from ⍷ Ôq or ⍷ Ôp, should yield a state for which the product of uncertainties is not insignificant.

References

[1] Sylvia Camacho, *Bell's Inequality*, Vector **4.1** 73-77 (1987)

[2] Sylvia Camacho, *Can Anyone Tell Me Where I'm Wrong?*, Vector **18.2** 86-94 (2001)

[3] J. S. Bell, *On the Einstein Podolski Rosen Paradox*, Physics **1.3** 195-200 (1964)

[4] A. Einstein, B. Podolski and N. Rosen, *Can Quantum Mechanical Description of Physical Reality Be Considered Complete?*, Phys. Rev. **47** 777-780 (1935)

[5] N. Bohr, *Can Quantum Mechanical Description of Physical Reality Be Considered Complete?*, Phys Rev **48** 696-702 (1935)

[6] http://www.upscale.utoronto.ca/PVB/Harrison/BellsTheorem/BellsTheorem.html (2003)

[7] Graeme Robertson, *Philosophical Problems of Quantum Ontology*, M.Litt. Thesis, Cambridge University (1977)

[8] Graeme Robertson, *Unity Consciousness and the Perfect Observer: Quantum Understanding Beyond Reason and Reality*, Robertson (Publishing) ISBN 0 9524167 0 0 (1995)

[9] X. Gabaix et al, *A Theory of Power-Law Distributions in Financial Market Fluctuations*, Nature 423 267-270 (2003)

[10] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press ISBN 0 521 63503 9 (2000)

[Received 12 Sept 2003; revised version 24 Sept]

A Conversation between Stevan Apter and Manfred von Thun

Introductory Note by *Adrian Smith*

We have inhabited our little APL universe for so long, we are in danger of forgetting that we are not the only language-family out there with functional aspirations. Languages arise in profusion from research projects and as by-products of other work. Every so often, one of them starts a brush-fire and becomes a cult. *Joy* is one of these, and it has much in common with K, so it is interesting to learn where it came from, how it is developing, and how it might influence what we do.

Steven Apter is a keen K user, and has recorded this interview with the creator of Joy so that we can share in some of these insights. Hopefully, this will not be the last such interview that we run in Vector.

SA: *Welcome Manfred. Thanks very much for agreeing to this interview.*

MvT: Thank you Stevan.

SA: *May I begin by asking you about the origins of Joy? You come to programming language design by way of philosophy. How exactly did that come about? Or are these activities unrelated?*

MvT: The relation is mainly historical, and certainly not logical. My first fifteen year period in academia, first as a student and later as staff, was a comedy(?) of errors. I enrolled in Psychology and Philosophy because "I wanted to know how the human mind works". But rats and stats in Psychology and logic and scientific method in Philosophy quickly changed my directions. In particular, I became interested in the justification of induction, and I wrote my PhD in the field of inductive logic and logical probability. After my appointment here at La Trobe I had to catch up with deductive logic and have been teaching it at various levels. I have also taught courses in philosophy of science, philosophy of psychology, cybernetics, and of course various bits in first year philosophy. In 1978 two of us logicians started using the university's new DEC10 computer. My colleague soon settled on Snobol, and I on Pascal. We had no help at all, and at that time for me there was just the Pascal User Manual and Report and Wirth's *Algorithms +*

Datastructures = Programs. I was particularly fascinated by the miniature compiler for the PL0 language. Most of my *Symbolic Programming in Pascal*, which I wrote over quite a number of years, was influenced by Wirth. I have given courses at all levels for the Computer Science Department.

SA: *All your programming in Pascal was procedural. So how did you get into functional programming? How did Joy evolve?*

MvT: In the early 1980s I came across the famous Backus paper "Can programming be liberated from the von Neumann style," and I was immediately intrigued by the higher level of programming in his FP. From my deductive logic I also knew about Quine's predicate functors and about Tarski's cylindric algebras. Like Schoenfinkel's and Curry's combinatory logic, Backus elimiminates not only assignable variables but also formal parameters, and Quine eliminates variables of quantification. I designed a strange little logic programming language based on some of these ideas, and implemented it in Prolog. It had some rather useless features - among others a "Wheatstone bridge" combinator which takes five binary relations to produce a new binary relation.

Joy then evolved from this in an entirely haphazard way: First I restricted the binary relations to unary functions, and this of of course was a dramatic change. Second, to allow the usual arithmetic operations with their two arguments, I needed a place from which the arguments were to come and where the result was to be put - and the obvious place was a stack with a few shuffling combinators, originally the four inspired by Quine. Third, it became obvious that all these combinators could be replaced by unary functions, with only function composition remaining. Finally the very different distinctively Joy combinators emerged, which take one or more quoted programs from the stack and execute them in a specific way. Along the way of course, lists had already been seen as just special cases of quoted programs. This meant that programs could be constructed using list operations and then passed on to a Joy combinator.

SA: *So Joy emerged after a tortuous history. But by now it is quite stable. How would you describe its main features?*

MvT: The language Joy is a purely functional programming language. Whereas all other functional programming languages are based on the application of functions to arguments, Joy is based on the composition of functions. All such functions take just a stack as argument and produce a stack as value, but there are a few which additionally take files and the file system as argument and value. Because of the stack, much of Joy looks like ordinary postfix notation. However, in Joy a function can consume any number of parameters from the stack and leave any number of results on the stack. Moreover, there are some stack functions which shuffle the

top few elements of the stack around. So, although for example the program 2 3 + is an arithmetical expression in postfix notation and also in Joy notation (leaving 5 on the stack), the Joy program 5 dup * (leaving 25) is not arithmetic because dup is neither a number nor an arithmetic operator. That is why I say that much of Joy "looks like postfix". In reality this is what Billy Tanksley so appropriately called "concatenative notation". The semantics of this notation is summed up by: "The concatenation of appropriate programs denotes the composition of the functions which the programs denote". In such languages the application operation of the lambda calculus is uniformly replaced by composition.

What distinguishes Joy from (the functional subsets of) Forth and Postscript is the datatype of quoted programs. Many Joy functions expect quoted programs on top of the stack and execute them in different ways, effectively by dequoting. This is similar to Lisp's eval, but in Joy there are many of them, each performing the job of higher order functions. But whereas in other languages the higher order functions take abstractions as arguments, the combinators of Joy take quoted programs from the stack.

As a result, there are no named formal parameters, no substitution of actual for formal parameters, and no environment of name-value pairs. Consequently Joy has an exceptionally simple algebra, and its programs are easily manipulated by hand and by other programs. Many programs first construct another program which is then executed by one of the combinators.

SA: *In Joy, a primitive like + operates on the stack. Should we think of this operation as replacing the top two elements x y with their sum x+y, or as two successive operations: first replace y with the unary function _+y and then replace x with x+y?*

MvT: You spoke of "replacing" and "and then", which are imperative or procedural notions and hence have no place in a purely functional language. So the official answer has to be: the program 2 3 + denotes the composition of three functions, and the program 5 denotes one function and it is identical with the composition of the other three. How the addition is performed internally is a different matter, it might be by pattern matching, or by two push operations followed by an add (the obvious stack implementation), or any other (invisible) method that gives the right result.

But you also said "Should we think...", and that requires a different answer. Indeed imperative or procedural thinking can be very useful for Joy, both for explaining the meaning of the Joy primitives and also when writing programs. Here one should be eclectic and pragmatic - use whatever works. I would tend to use your first mode of thinking, and only sometimes the second. Two other modes would use message passing: a) send the parameterless addition message to the

pair 2 3, or b) send the addition message with parameter 3 to the number 2. Maybe Forth programmers know more about the psychology of stack programming.

SA: *You mention the combinators of Joy. What exactly is a combinator, and how do those of Joy differ from the combinators of Schonfinkel and Curry?*

MvT: Combinators are second (or even higher) order functions which take first (or higher) order functions as parameters. They may return another function or instead immediately apply that function to some argument(s). When we say "John loves Mary and conversely", this might be analysed as "(loves AND CONVERSE(loves)) (John, Mary)". Here CONVERSE is a unary combinator (corresponding to the passive transform "John is loved by Mary"), and AND is a binary combinator. Similarly, "Bob admires himself" might be analysed as "SELF(admires) (Bob)", where SELF transforms the binary admires predicate into a unary one. These combinators are not restricted to predicates (functions which yield a truth value). For example, the unary squaring function might be defined with the binary multiplication function by SELF(*). In Schonfinkel's and Curry's system combinators similar to these serve to shuffle arguments of functions around, a job which in the lambda calculus is done by variables.

SA: *K programmers should understand this. We would define SELF as a function which takes a binary function f and returns a unary function which applies f[x;x]:*

```
   SELF:{[f]{f[x;x]}}
   g:SELF[*]
   g[3]
9
```

MvT: If a parenthesis free notation such as postfix is used, then there are two possibilities: either use SELF as a higher order function as before, or work on a stack, using a stack function dup:

```
5 SELF(*)
5 dup *
```

Joy, like Forth uses the latter. Note that dup is not a second order function at all. The same two possibilities would arise for prefix notation. In the same way the effect of CONVERSE can be achieved by a stack function swap. All the functions involved now are strictly speaking unary functions from stacks to stacks. Even multiplication is unary, although one can of course continue to think mainly in terms of how many parameters a function expects on the stack.

SA: *Yes, I think a similar choice exists for K. We can define 'sqr' as either SELF[*] or as the unary function*

```
sqr:*. 2#
```

*which makes two copies of its argument, and then applies * "dot-wise", mapping the first copy to *'s first argument, and the second copy to *'s second argument.*

MvT: But there are combinators which do not simply shuffle arguments around. One of these is the mapping function which applies a function to all members of a list to produce a list of results. In a fantasy notation:

```
MAP(SELF(*)) ([1 2 3 4]) = [1 4 9 16]
```

Or in postfix and in concatenative Joy notation:

```
[1 2 3 4] MAP(SELF(*))
[1 2 3 4] [dup *] map
```

Whereas MAP above is a second order function, Joy's map is just an ordinary unary first order function from stacks to stacks – it expects a list and what in Joy is called a quoted program on top of the stack, and it returns a stack with a single list on top, the list [1 4 9 16].

SA: *This is where Joy and K diverge. For us, 'each' really is second-order: it takes a function and returns a function which applies to each element of a list. We are taught to understand*

```
sqr'2 3 4
```

as having the structure

```
(sqr')2 3 4
```

Apply 'each' to 'sqr', and apply the resulting function to 2 3 4. It is even more obvious in classical APL, where functions such as 'sqr' are not first class. You can operate on a function only by feeding it as on operatand to one of a handful of "operators", such as /, and then immediately applying the resulting function to data. In K, +/ is a first class function, which can be assigned and passed into other functions as an argument.

MvT: Joy has a large number of such combinators which are really first order functions that do the work of second order functions because they expect one or more quoted programs on top of the stack. Consider the conditional IF-Then-Else combinator, where the if-part produces a truth value and the then-part and the else-part are again functions (and not statements).

```
IFTE(ifpart,then-part,else-part)
```

In Joy there is a combinator ifte with the most common syntax

```
[if-part] [then-part] [else-part] ifte
```

which expects three quoted programs on top of the stack. But actually quoted programs which map and ifte expect on top of the stack do not have to be pushed just before the combinator - they can be the result of stack shuffling or construction. Similar to ifte is a combinator linrec for linear recursion which can eliminate the need for many recursive definitions that would otherwise clutter up the symbol table and the programmer's mind:

```
[if-part] [then-part] [rec1-part] [rec2-part] linrec
```

The implicit recursion occurs between the [rec1-part] and the [rec2-part] quoted programs. There is a similar combinator for binary (tree-) recursion that makes quicksort a one-liner:

```
[small] [] [uncons [>] split] [swapd cons concat] binrec
```

SA: *This seems quite powerful! Over the years, APL developers have experimented with a variety of recursion operators, hoping that, just as the classical adverbs such as 'reduction', 'scan', and 'each' enabled us to factor out explicit loops, so the new operators would help eliminate at least some explicit recursion. For example, in K, we can define a function 'apply_to_atoms' which takes a monadic function f and returns a g which, applied to a list, recurses to the atoms and applies f:*

```
apply_to_atoms:{[f]{:[@x;f x;_f'x]}}

f:2*!:
g:apply_to_atoms f
A:(1 2;(3 2 4;(5 6)))

 A
(1 2
 (3 2 4
  5 6))

 g A
((,0
  0 2)
 ((0 2 4
   0 2
   0 2 4 6)
  (0 2 4 6 8
   0 2 4 6 8 10)))
```

But it's never been clear what the "best" set of K-recursors might be, partly because K functions can take any number of arguments, and partly because recursion can involve more than one function.

To what extent to do you think the recursive combinators of Joy will replace explicit recursion? APLers are familiar with the fact that it often takes time and one or two "aha's" before a problem breaks apart into an "array" solution, a program without loops and counters. And some problems are stubbornly, perhaps essentially loopy. It seems to me that a certain way of thinking needs to be acquired, analogous to "array thinking" in APL, which would let the programmer factor out the recursion pattern of problem such as quicksort. I'm thinking here of the way the nested recursion combinator arose as you contemplated Ackermann's function, which most programmers would have said required explicit recursion.

MvT: Yes, the recursion combinators are very powerful indeed. To have just one linear recursion combinator seems only possible in a language in which all functions have the same arity (like in Joy, where all are unary). I do not have a proof for this claim, except that I have not been able to define one for Lisp or Scheme, and if it were possible it would surely be in the literature. Maybe it can be done with clever macros. The same is true for tail recusion as a special case of linear recursion, and also for binary and nested recursion.

But probably it is not true that all recursion patterns can be captured by a small fixed menu of recursion combinators, especially the mutual recursion patterns. As an example, one might look at the intricate and highly specialised mutual recursion that occurs in the definition of the Lisp functions eval and apply. For those cases ordinary explicit recursive definitions seem unavoidable. On the other hand, I would estimate that at least 50% of recursions in Joy can be handled by the list combinators map, filter and fold, of the remainder at least 50% can be handled by the more general linear recursion combinator. What is left can often be handled by the binary and nested recursion combinators, and finally a small percentage does need explicit recursion.

As far as programming practice is concerned though, I have to confess that earlier on I sometimes caught myself first thinking recursively and only later using a combinator to remove the recursion. It does not happen much nowadays, the combinators do become second nature.

SA: *In APL we are used to thinking of programs as special datatypes, distinct from lists, arrays, and quotations. But in Joy, lists and programs are the same datatype.*

```
[10 20 30]
```

is a list with three elements, but it is also an example of a Joy program. Can you spend a moment explaining how this works?

MvT: If you ask primary school children to do the sum: "5 + 0 - 0 + 0" they will be perplexed and find it hard. Indeed, some problems are difficult because they really are so easy. In Joy, as in Lisp and some other list processing languages, lists can be heterogeneous, the elements can be of all sorts of types. So this is a perfectly acceptable list:

```
[11 22 true London Peter Santa-Claus foo dup *]
```

This list might have been the result of concatenating three lists:

```
[11 22 true] [London Peter Santa-Claus foo] [dup *]
```

The first list would normally be described as containing two numbers and a truth value, because that is the ordinary meaning of the symbols. Now consider the second list – are its members a city, two persons ..? No, sorry, there is no Santa-Claus, just having a symbol or name does not create a thing to which the name refers. What about foo? It is a standard symbol for nothing in particular. Finally in the third short list, what does it contain? Two symbols or the two functions which they normally stand for? If the latter, then with an appropriate definition even foo from the other list might stand for something. So, it is not at all easy to say what a list really is. But it is straightforward to understand concatenation of lists.

Much the same with all the other operations on lists: taking its first members, deleting its first member, sticking something on the stack in front of a list, writing out a list or any of its members. But the stack is as problematic as any other list: what is the result of taking the first element of [11 22] or of [London Peter] or of [Santa-Claus foo] or of [dup *]? Taking the first of a list leaves something on the stack, but what is it? The number 11, the city London (too big!), Santa-Claus (sorry), the duplication function? I think that we have to say that strictly speaking all lists and all stacks just contain representations of something that may or may not refer to something in reality.

In Joy, as you remarked, some lists like [11 22 true] can be executed by a combinator, and this one in particular results in three things being pushed onto the stack. Another that can be executed is [dup *], and, as we say, "it expects a number on the stack and replaces it by its square". But in our pedantic mode we should rephrase that.

In describing Joy I have used the term quotation to describe all of the above, because I needed a word to describe the arguments to combinators which fulfill

the same role in Joy as lambda abstractions (with variables) fulfill in the more familiar functional languages. I use the term list for those quotations whose members are what I call literals: numbers, characters, truth values, sets, strings and other quotations. All these I call literals because their occurrence in code results in them being pushed onto the stack. But I also call [London Paris] a list. So, [dup *] is a quotation but not a list.

SA: *It's interesting how K and Joy partition these concepts in different ways. In K, strings can be used to quote code:*

```
"2+3"
```

As you would expect, it doesn't evaluate unless you apply "eval":

```
   ."2+3"
5
```

But a list evaluates immediately, so one can use K expressions to build lists "from within". For example

```
(!5;!3)
```

builds a list whose first item is 0 1 2 3 4 and whose second item is 0 1 2. But in Joy, supposing that we had the program

```
  3 enum
[0 1 2]
```

we couldn't use it in the following context:

```
[5 enum 3 enum]
```

since this is a program. Instead, we would construct the list out of its parts:

```
5 enum 3 enum unit cons
```

Of course, as you pointed out earlier, it is this very feature which allows the Joy programmer to construct programs out of simpler components.

MvT: You want the infra combinator, I think. It expects a list and above that a quotation. It then treats the list as a temporary stack and executes the quotation on that. More often than not the list that is supplied is actually an empty list.

Example:

```
   [] [2 3 + 4 5 *] infra
[5 20]
```

Now if enum is already implemented as you described, producing a list, and if [5 enum 3 enum] is already on top of the stack, then all you need is just [] swap infra, which will leave [[0 1 2 3 4] [0 1 2]] on the stack. Of course if you want the two sublists concatenated, then you do an extra [concat] infra first, or in this case simply flatten.

SA: *Programming in Joy over the last few years, I've come to think of the operators as falling into two classes: those which produce new information from data, such as + and cons, and those which move items on the stack into position where the informational operators can do their work, such as swap and dup. Languages with variables appear to have a productivity advantage here, since one can simply invoke data by name. Billy Tanksley has proposed closing the gap with something he calls "shuffle notation", where "before" and "after" stack patterns are specified. For example, if the stack is*

```
.. 10 20 30
```

and you want:

```
.. 20 20 30 10 10
```

then you could say:

```
"abc-bbcaa" shuffle
```

which would have the same effect as:

```
rolldown [dupd] dip dup
```

What's your take on this and similar efforts to deal with the problem of "stack noise"?

MvT: Indeed, conventional languages with named formal parameters in definitions of functions can simply use the name to pick up the value of the corresponding actual parameter. For functions with many parameters this is indeed an advantage. But these languages need to carry around an environment of name-value pairs. Joy tries to avoid the latter completely, but pays a price when there are many parameters. One possibility would be to do what Forth does: when appropriate allow named formal parameters. An interesting alternative is Billy Tanksley's solution or some syntactic variant of that. It does not introduce formal

parameters to be used in the body of a function, and also it need not be used just at the beginning of a definition. So it deserves very careful attention.

The proposal in effect defines an infinity of possible stack shuffling operators. Currently Joy has 12 inbuilt stack shuffling operators, a few others defined in the standard library and other, deeper shufflings can be expressed using the dip combinator. Two possiblities arise: 1) replace all the current Joy mechanisms by the new shuffle operator, or 2) allow both, and leave it up to the programmer to select what fits best. To make any decision one would want to see some programs expressed in the old and in the new notations. I have not done any experiments along these lines.

One other consideration concerns efficiency. The shuffle operator (or any syntactic variant) requires a before-after after string to be examined - and that means interpreting its pattern. A simple minded implementation would be slow. An only slightly more sophisticated one would look for common patterns, and replace for example "a-aa" shuffle by dup. It would use the interpreted form only in the less common but tricky cases, like your example. On the other hand, this decision might well be left to the programmer. So, my answer is not very definite at all, sorry.

SA: *It might be interesting to explore algorithms which translate shuffle notation into a fixed vocabulary of stack-shuffling words. Are there small bases for which efficient algorithms exist? What if we allow the stack diagrams to express structural transformations, e.g.:*

```
[ab]c -- [ac][bc]
```

I imagine that Brent Kerby would have opinions on this matter (and quite possibly an algorithm or two!)

MvT: Yes, Brent Kirby has done very interesting work on a combinatory algebra for concatenative languages, see his paper from the main page of Joy. For Joy itself the following seems to be an adequate base for shuffling the stack: the three simple operators swap dup and pop, together with the combinator dip. (This combinator expects a quotation on top of the stack and below that another value. It pops the two, saving the value somewhere, executes the quotation, and then restores the saved value on top. So, for example, [swap] dip will interchange the second and third element on the stack.) But I do not have a proof that these four primitives are indeed complete in the sense required.

However, if the transformations also take apart lists on the left of the transformation pattern, or construct lists on the right of the transformation

pattern, then the list operations will be needed. In fact it helps to think of your transformation [ab]c -- [ac][bc] to be composed of two transformations, firstly [ab]c -- abc and secondly abc -- [ac][bc]. Then the first has to take the list apart, and in general the list operators first and rest should be adequate for that, but uncons and unswons would also help. The second transformation has to construct two lists, and in general the list constructors [] (empty list) and cons should do, but swons might also be useful. Both kinds of component transformations would also need some of the four general primitives mentioned earlier. An actual implementation would presumably collapse the two component transformations back into your original.

I have not done any work on implementing this sort of thing for Joy. If I were to do so, I would look at the by now well understood implementations of pattern matching that are used in languages such as Prolog, Miranda and now in Haskell. An older book by Peyton-Jones describes this in detail. For a while I used to think that these transformations would re-introduce something like lambda variables into Joy, and that there is a problem of what the scope of the introduced variables would be. But I was wrong, the scope is just the right side of the transformation, so the variables cannot actually occur elsewhere in the Joy program. So if somebody wants to volunteer ...

SA: *I'm curious about Joy's internals. For example, in APL and its descendants, at least some of the bulk datatypes are implemented in consecutive memory locations without any links, and for that reason APL doesn't require garbage collection. I know that K uses a reference-counting mechanism. Since the typical K application uses a small number of large objects, this is quite efficient. So what choices have you made for Joy, and for what reasons?*

MvT: Yes, the implementation of bulk datatypes in consecutive memory locations is in many ways ideal simply for efficiency reasons. I have often wondered what kind of a cousin of Joy might be implemented like that without sacrificing efficiency elsewhere. One worry of course concerns adding a new first element to a very large array: it requires copying the entire array. So, if this happens often, much effiency is lost. But must it happen often? Presumably the APL experience shows that it need not be a concern at all for a vast number of applications.

Then there is the possibility of a mixed implementation: mostly consecutive, but some links. This may or may not require memory management in some way, either as mark-sweep, or copying, or reference counting. The last cannot handle circular structures, but that may or may not matter. At any rate there seem to be many possibilities for languages similar to APL, J and K but with a concatenative syntax to explore. You yourself, Stevan, have done some interesting work in this

field with cK, a concatenative version of K. I look forward to seeing more of it, not the least because it would offer an implementation that is more efficient than the current Joy because of the way bulk datatypes are handled. At the same time, I see no reason why a more sophisticated implementation of Joy could not use the same method wherever possible.

This is perhaps the best place to mention Billy Tanksley's work, which is inspired by the consecutive memory implementation of the stack of the Forth language. Since in Joy the stack is a sequence just like lists and other quotations, there is a possibility of using consecutive memory locations wherever possible and using links only where necessary. So I think Billy's work is a step in this direction, and I welcome it.

But you also asked about the choices made in the implemention of Joy, and for their reasons. My background had been very much in logic and other symbolic processing, and the structures one needs there tend to be trees of some sort. So I have had rather little use for arrays in consecutive locations, except for implementing trees. I had used Prolog a fair bit, but Lisp I only studied in detail from a purely theoretical perspective. It was clear to me from the outset that a uniform linked implementation for Joy would be simultaneously simple and sufficiently general. Since Joy then had no assignments or other destructive updates, and hence no possibility of circular lists, memory management by reference counting was a possibility that I did consider quite carefully. In the end I decided against it, largely on the grounds that a problem would arise if I ever wanted to allow circular lists. I knew that the problem would not be insurmountable, but for a first implementation it looked like a lot of work. It so happens that Joy is still purely functional and hence has no destructive updates of any kind, so a simple reference counting memory management would still be possible. That would have the advantage of not requiring occasional hiccups for memory management and hence better suitablitity for real time work. On the other hand, it is known that reference counting is more CPU intensive, spreading the total work more continuously. Whether Joy will remain purely functional for ever (look what they did to the original Lisp), I do not know at this stage.

SA: *What is the current status of Joy? What are the plans for the future development?*

MvT After several proof-of-concept implementations, the current implementation, written in unadorned C, evolved smoothly from one that was started 1995. In early 2001 John Cowan added files, floating point numbers and access to standard C functions, and he did a lot of cleaning up. He also added the option of not using my original garbage collector but using the professional BDW collector, an improvement which allowed space used for strings to be reused. This extended

the functionality of Joy tremendously, and I a very grateful for all the work he did. Since then Nick Forde has also added a number of features, and I thank him for that.

The current implementation is essentially an interpreter. It translates (compiles) the external ASCII form of programs into internal tree code which is then interpreted in a rather conventional manner. No attempt is made to do any optimisation, and my earlier "clever tricks" I always came to regret fairly soon because they interfered with my garbage collector. There is a growing list of general and special purpose libraries, ranging over many kinds of possible programming applications, and most of my work in the last few years has been on those. For the immediate future I plan to examine and test the so far rather under-utilised module system, by writing libraries for some specialised types such as big sets, trees and dictionaries of arbitrary basetypes.

Several other people have published other more or less complete Joy interpreters, written in ML and in Scheme, in the "concatenative" mailing group. At this point in time I have no plans to write a full compiler. A first version of such a compiler would presumably use C as an intermediate language and leave the generation of machine code to the C compiler. I would very much welcome if somebody were to take up the task.

SA: *This has been quite illuminating. Thanks Manfred.*

MvT: Thank you Stevan, I enjoyed this interview.

12 December 2003

TECHNICAL SECTION

This section of VECTOR is aimed principally at those of our readers who already know APL. It will contain items to interest people with differing degrees of fluency in APL.

Contents

Erratum noted by Graeme Robertson

Thanks for printing my *New Foundations* article. There was a small printing error with the APL on the page 138 title line

3. Promotion of primitives

but that is not very important. There was also a small oversight in the code on the next page on a line which should have read:

```
VF←{⍵*0},{⍵},{⍵*2},{⍵*3}
```

because the valence of each function must be explicit or the whole construction of function arrays breaks down. This was part of my motivation for introducing the one-line and no-braces precursor of Dfns.

CORRESPONDENCE

From: Roger Hui

Re: E.E. McDonnell's article "At Play with J: The Magical Matrix" in Vector Volume 20, Number 2, October 2003.

In the article, Mr. McDonnell insinuated that my use of the "magical matrix" and indexing to generate permutations is due that I "wrote it all". That statement is far from the truth! Revisionist history, even!

The "magical matrix", computed as:

```
   \:"1 = i. 4
0 1 2 3
1 0 2 3
2 0 1 3
3 0 1 2
```

(grade-down each row of the identity matrix) first saw the light of day in the APL87 conference proceedings, in the paper "Some Uses of { and }", section 4.3, computed as:

```
k ~{rank}1 0 k{is}{iota}{omega}
```

The paper also presented functions for generating combinations, compositions, and self-upgrading and self-downgrading permutations, using a similar technique of growing a table from a seed through indexing.

I should point out that in 1987 J was not yet even a gleam in Ken Iverson's eyes. Therefore, I deny the allegation, and I defy the allegator!

Roger Hui (with tongue firmly in cheek)

At Play with J: Giddyap

by Eugene McDonnell

Giddyap

The *OED* doesn't have a giddyap entry; the *Concise Oxford Dictionary* has a *giddap* entry; Webster 3 has an entry for *giddap, giddyap, giddyup.* I think it must be a children's word; I don't think I've ever heard it used by an adult.[1] When I was 70 or so years younger I know that when I pretended I was riding a horse – which was surprisingly often – I swung my imaginary whip on my imaginary horse as I pranced about, shouting *giddyap* with every stroke of the whip. I find it to be a suitable title because the article deals with a problem concerning horseraces, and also treats of the speeding up of programs that solved the proposed problem. The answer to the problem turned out to be an old friend of mine.

I don't recall now where it was that I found the problem, but when I ran across it, it sounded as if it might a suitable challenge for the J Forum. In any event, on September 25 I sent this message to the J Forum:

```
N horses enter a race. Given the possibility of ties, how many
different finishes to the horse race exist? Write a program
that shows all the possibilities.

By way of example: here is the solution by brute force for N=3.
There are 13 solutions.  horses are named a, b and c. The
expression {{b,c},a} denotes a finish in which b and c tie for
first and a comes in next.

{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, b, a}, {c, a,
b}, {a,{b,c}}, {{b,c},a}, {b,{a,c}},
{{a,c},b},{{c,{a,b}},{{a,b},c},{{a,b,c}}
```

---

1 1 It was shortly after writing this that I attended a Choral Christmas Spectacular at San Francisco's Davies Symphony Hall with my granddaughter and wife, and heard 3000 adults singing the second stanza of a song called "Sleigh Ride", which goes:

Giddyyap, giddyyap, giddyyap, let's go. Let's look at the show.
We're riding in a wonderland of snow.
Giddyyap, giddyyap, giddyyap, it's grand just holding your hand.
We're gliding along with a song of a winter fairyland.

Notice the fourth variant in spelling of the key word

Methods for finding how many different Finishes

There were over two dozen responses over the next two weeks. The first response misunderstood the problem, and assumed that a race with three horses was the only one to consider. Since I had already given the solution of this one, it was clear that more had to be done than to submit the number 13. The answers to the first question, the number of solutions, were various. The brute force way is good only for the first few number of horses – the answer for eight horses is already 545,835. The nicest early entry used a table of Stirling subset numbers and factorials to give the number of different finishes effectively:

```
   ] fc=:!i.9
1 1 2 6 24 120 720 5040 40320
   ] s8=:subsets 8
1 0   0   0    0    0   0  0 0
0 1   0   0    0    0   0  0 0
0 1   1   0    0    0   0  0 0
0 1   3   1    0    0   0  0 0
0 1   7   6    1    0   0  0 0
0 1  15  25   10    1   0  0 0
0 1  31  90   65   15   1  0 0
0 1  63 301  350  140  21  1 0
0 1 127 966 1701 1050 266 28 1
   s8 +/ . * fc
1 1 3 13 75 541 4683 47293 545835
```

Here's another way that uses curtailed binomial lists and the list of terms so far found:

```
   1 +/ . * 1
1
   1 2 +/ . * 1 1
3
   1 3 3 +/ . * 1 1 3
13
   1 4 6 4 +/ . * 1 1 3 13
75
   1 5 10 10 5 +/ . * 1 1 3 13 75
541
```

Which can be done either iteratively or recursively.

Still another way to get the number of different finishes uses the Weighted Taylor coefficient adverb `t:` defined in the J Dictionary as:

The result of `u t: k is (!k)*u t. k` . In other words, the coefficients produced by `t:` are the Taylor coefficients *weighted* by the factorial. As a

consequence, the coefficients produced by it when applied to functions of the exponential family show simple patterns. For this reason it is sometimes called the *exponential generating function.*

The exponential generating function for the our numbers is $(1/(2-e^n))$ so we can write a function `fn` using it, modified by the Weighted Taylor adverb:

```
   fn =: (%@(2: - ^)) t:
   fn 8
545835
   fn i. 9
1 1 3 13 75 541 4683 47293 545835
```

All of these methods are discussed in Sloane's *On-Line Encyclopedia of Integer Sequences,* sequence 670.

Methods for representing all the possible Finishes

The method I used for representing all 13 of the finishes for a three-horse race was informal. Various methods were used in the J solutions. This is one of the J solutions for a three-horse race:

```
+-----+-----+-----+-----+-----+-----+---+---+---+---+---+---+---+
|a b c|a c b|b a c|b c a|c a b|c b a|a bc|bc a|ab c|c ab|b ac|ac b|abc|
+-----+-----+-----+-----+-----+-----+---+---+---+---+---+---+---+
```

Results using boxed arrays, like this, were relatively slow. Faster results were obtained using a list of post positions to show the finish, with the finish order of the horse in post position k given as the value of item k of the result. Here is how the finishes of a three-horse race are displayed using this method:

```
0 0 0       abc
0 1 1       a bc
1 0 0       bc a
0 0 1       ab c
0 1 0       ac b
1 0 1       b ca
1 1 0       c ab
0 1 2       a b c
0 2 1       a c b
1 0 2       b a c
1 2 0       c a b
2 0 1       b c a
2 1 0       c b a
```

I've appended to the right of each finish the order of finish of three horses a, b and c, having post positions 0, 1 and 2, respectively. Horses tied in a finish are shown

by abutting letters – for example, `a bc` shows horse a in first place and horses b and c tied for second place.

The function to produce the finishes in this order is `fin3`, by Roger Hui:

```
rankings=: ,"1 0~@i. , /:"1@=@i.@>:
ext      =: [: ,/ _1&,. {"2 1 rankings@#@~.@{.
fin3     =: ([: ; >./"1 <@ext/. ])@$:@<: ` (i.@(1&,)) @. (1&>:)
```

Here is his explanation:

To generate the finishes for `n`, `fin3` first partitions the finishes for `n-1` by the maximum ranks. Then for each partition with maximum rank `m` and each finish `v` therein, `(_1,v)` is indexed into the matrix `,"1 0~i.1+m` (tieing the new competitor with each possible rank) and into the matrix `/:"1=i.2+m` (slotting the new competitor into each possible position ("no ties")).

Therefore, if `c` is a vector of the number of finishes with maximum ranks `i.#c`, the corresponding counts for `1+#c` are: `(c*1+i.#c)` are the number of finishes for ranks `i.#c` coming from ties, and `c*2+i.#c` are the number of finishes for ranks `1+i.#c` coming from non-ties. For example, for `n=2`:

```
                          each finish indexed into
max rank     finishes        ties       nonties
 0             0 0           0 0         1 0
                                         0 1
 1             0 1           0 1 0       1 2 0
               1 0           0 1 1       0 2 1
                                         0 1 2
```

There is 1 finish for max rank 0 and 2 finishes for max rank 1.

The new counts are for n=3 are:

```
max ranks     0      1      2
ties          1*1    2*2
nonties              2*1    3*2
total         1      6      6
```

And for n=4:

```
max ranks     0      1      2      3
ties          1*1    2*6    3*6
nonties              2*1    3*6    4*6
total         1      14     36     24
```

The following functions encode the algorithm:

```
ntie1=: 0: ,~ ] * 1&+@i.@#
ntie0=: 0: ,  ] * 2&+@i.@#
nfin2=: (ntie1 + ntie0)@$:@<: ` ((,1x)"0) @. (1&>:)

   nfin2 1
1

   nfin2 2
1 2

   nfin2 3
1 6 6

   nfin2 4
1 14 36 24
```

[end of Hui's explanation].

In a race where there are no ties the order of finish is a permutation. Notice that the bottom six results give the permutation table of order 3. In the rankings function you see /:1@=@i . This is the "magical matrix" described in my last column (*Vector* 20.2, October 2003), and it is used in precisely the same way: to produce a table of permutations.

The results of the function `nfin2` above give the number of finishes of `n` horses having `k` as the maximum rank. If we form a triangle from these results:

```
1
1  2
1  6  6
1 14 36 24
```

and ravel it,

```
1 1 2 1 6 6 1 14 36 24
```

we get a list that is sequence 19538 in (*http://www.research.att.com/~njas/sequences/*). It is described as "the number of ways n labelled objects can be distributed into k nonempty parcels". I wanted to obtain a different triangle, one showing the number of finishes having each leading digit. An easy way to do this is to look at the first column of the table of order n, as found by `fin3`. Thus if one transposes table `fin3 n`, takes its head item, applies tally modified by the key adverb and

reflexive to tally one would get the number of instances of each leading digit, in order.

```
    # /. ~ {. |: fin3 3
6 5 2
```

This says that the digits 0, 1 and 2 occur 6, 5 and 2 times as leading digits, respectively, in the table of order 3. You can verify this by inspecting the table above. Here are the results for tables of order 1 through 7:

```
    1
    2    1
    6    5    2
   26   25   18    6
  150  149  134   84   24
 1082 1081 1050  870  480  120
 9366 9365 9302 8700 6600 3240 720
```

I've submitted this triangle to Sloane's Online Encyclopedia.

I've come across the sequence given by the row sums of the triangle above, namely

```
1 3 13 75 541 4683 47294
```

in several different contexts. In 1977 I found that it enumerates the number of different left arguments for APL's dyad transpose[2]. In 2000 it enumerated the number of distinct basic lists, I called Blists, which mathematicians call preferential arrangements[3]. And now, here they come galloping again to enumerate horserace finishes!

[2] Eugene McDonnell, "How Shall I Transpose Thee? Let Me Count the Ways", APL Quote Quad 8.1, September 1977.

[3] Eugene McDonnell, "Blists in OLEIS", *Vector 17.1*, July 2000.

Elliptic Curves and Factoring I

by Cliff Reiter

Factoring integers has become a topic of intense interest because several modern security schemes are based upon the unproven, but highly tested assumption that factoring the product of two sufficiently large primes is implausible. For example, RSA encryption [1,3,5] depends upon that assumption.

J has a factorization primitive, `q:`, that is effective on integers of limited size. Consider the following examples from J501b [2].

```
   q: 10001x
73 137

   q: 20104311482x
|nonce error
|       q:20104311482
```

In this note we will consider J implementations of several factoring schemes. While our main goal is to implement elliptic curve factorization, we will first implement other techniques for comparisons and for their auxiliary facilities.

Trial Division

A simple approach to finding a nontrivial factor of a positive integer n is to try dividing n by all the primes up to and including $\sqrt{n}$. This guarantees that either a prime factor will be found or that n is a prime. However, this is impractical for n with many more than a dozen digits. Remarkably, there usually are faster ways to find nontrivial factors of an integer. Nonetheless, factoring remains a difficult problem.

It is often convenient to assume that small prime factors have already been removed from a number that we are trying to factor. Thus, we first give a method, essentially a limited form of trial division, for finding and removing small prime factors.

```
    fac_smpr=: 3 : 0
24 fac_smpr y.
:
p=.p:i.x.
z=.0
whilst. 0<+/peg do.
  peg=.p=p +. y.
  z=.z+peg
  y.=.y.%*/peg#p
end.
(z#p);y.
)

    smpr=:>@{.@fac_smpr

    rmsmpr=: >@{:@fac_smpr
```

The function `fac_smpr` finds/removes small prime factors. The left argument specifies the number of primes to consider as factors. The right argument is the integer to be factored. The function computes the gcd's of the primes with the right argument. Any prime factors are divided into the right argument and the process is repeated until no additional prime factors are found. The number of occurrences of the small primes is maintained in `peg`. The default left argument is 24 and since there are 24 primes below 100, the default is to find and/or remove all the primes below 100. The result of `fac_smpr` is a boxed list: the first entry gives the small primes and the second gives the remaining factor. The function `smpr` (small primes) gives just the small prime list and `rmsmpr` (remove small primes) gives the remaining factor.

```
   fac_smpr 10001x
+--+---+
|73|137|
+--+---+

   fac_smpr 10001x^3
+--------+-------+
|73 73 73|2571353|
+--------+-------+

   smpr 10001x^3
73 73 73

   rmsmpr 10001x^3
2571353
```

This function runs quickly. We see the average time to remove the small primes from random 25 digit integers is around 0.005 seconds on a 750MHz PC.

```
     randi=: (#.?)@:(#&10x)"0

     randi 25
  2008895148955473153890348

     time=: 6!:2

   time 'x=:rmsmpr@randi 10#25'
0.0487565

     ,.x
      6712342501022203231 99
     2474549129588435104129
    4897009124902826052 3683
  2297220239702535644527649
         8841952595049342 1
     4305779878041636190577
  2248097504771690235770021
    6454149821353682576 1779
     8370870055924244454337
   2091376301993565330348 23
```

The list x above gives the result of computing 10 random 25 digit integers and removing the small primes from those integers. Notice the range of sizes. We will use this same list throughout this note.

Pollard p-1 Method

The Pollard p-1 method is based upon the idea that if p is an odd prime that divides n, then by Fermat's Little Theorem, $2^{p-1} \equiv 1 \bmod p$, and hence p divides $\gcd(n, 2^{p-1}-1)$. Of course, bases other than 2 may be used and perhaps a smaller power will suffice. The Pollard p-1 method looks for factors of n of the form $\gcd(n, 2^M-1)$ where M may be chosen in various ways; however, it is based upon the hope that p-1 has only small prime factors and hence divides M.

In particular, we follow the suggestion of [1] and select a bound B and then compute M as the product of prime powers p^e where e is the largest power so that $p^e \le B$. All primes less then B are used in the product. One can compute the gcd at various stages. Standard advice is to compute the gcd's at the end. However, we will use this technique for relatively small factors and will compute the gcd after each prime power.

There is a natural "second stage" that extends this algorithm when $p-1$ divides qM for a single, slightly larger, prime q. However, we will use this technique only to find fairly small primes, so we will be content to implement only the first stage.

```
   fac_p_1=: 3 : 0"0
(d_p_1_B y.) fac_p_1 y.
:
c=.2x
exp=.y.&|@^
for_k. i. p:^:_1 x. do.
  p=.p: k
  c=.c exp (p^<.p ^. x.)
  g=.y. +. c-1
  if. -. g e. 1,y. do. g,1 return. end.
end.
g,_1
)

   d_p_1_B=: 10000"_ <. >.@:(^&0.5 )
```

The left argument of `fac_p_1` is B and the right argument is the number to be factored. The default left argument is taken to be around $\sqrt{n}$ but not larger than 10,000. The bound is relatively ad hoc but seems to work well enough for the small primes we seek and keeps the run time of this function short. Of course, that is at the expense of having the algorithm fail when more time might have led to success.

The result of `fac_p_1` is a list of two integers. The first gives the factor if one was found. A 1 in the second position indicates that a nontrivial factor was found (in stage one) while a _1 indicates no nontrivial factors were found.

Next, we apply `fac_p_1` to the ten integers, `x`, that were created in the previous section.

```
   fac_p_1 x
      1 _1
    397  1
      1 _1
    433  1
 367823  1
      1 _1
 978149  1
    839  1
   1087  1
80621729  1
```

It took an average of 0.5 seconds to run `fac_p_1` on a 750 MHz PC. We observe that it failed three times but it usually succeeded. When it was successful, it found factors of a variety of sizes, but these sizes are relatively small compared to 25 digits.

Pollard rho Method

The Pollard rho method is based upon the idea that if one computes iterates of a function mod n, then at some point a duplicate value must be reached. Of course, the duplicate would usually occur much earlier modulo a prime divisor of n, hence we can look for factors of the form $\gcd(n, x_i - x_j)$. As a practical matter, it is traditional to test $\gcd(n, x_j - x_{2j})$ where the sequence x_j is generated by

$x_0 = 2$

$x_{j+1} = x_j^2 + 1 \mod n$.

The starting value and the function to be iterated can be varied, but we are again primarily interested in the case when it is effective at finding relatively small factors.

```
   fac_rho=: 3 : 0"0
(d_rho_maxj y.) fac_rho y.
:
n=.y.
f=.1x&+@*:
x1=.f 2x
x2=.f f 2x
g=.n +. n|x2-x1
j=.1
while. (g e. 1,n)*. j<x. do.
  x1=.n|f x1
  x2=.n|f f x2
  g=.n +. n|x2-x1
  j=.j+1
end.
g,j
)

   d_rho_maxj=: 10000&<.@<.@(^&(%3))
```

The left argument gives the maximal number of iterations and the right argument is the number to be factored. The result is a list: the first entry is the most recently computed gcd and the second entry is the number of iterations that were used. When the maximum iterations was reached, the gcd is most likely trivial, but otherwise it is a nontrivial factor. As for the p-1 method, we show below the result

of running fac_rho on the same 10 random integers created in the first section. Running fac_rho on all ten integers requires about 17 seconds.

```
   fac_rho x
   4234213  2330
       277    25
 161603867  4748
       269    15
    367823   512
         1 10000
    978149  1800
       839    18
      1087    40
         1 10000
```

We see that fac_rho usually succeeded at finding a nontivial factor. It failed twice. A careful comparison with the results for fac_p_1 shows that each successfully factored some integers that the other did not. Typical random examples show the techniques more usually find the same small factors somewhat more often than our illustration suggests, but the point that they each may succeed when the other fails remains a valid one.

Elliptic Curves and Arithmetic

Elliptic curves are often viewed as algebraic equations of the form $y^2 = x^3 + ax + b$. We usually restrict to nonsingular curves which are those where $x^3 + ax + b$ has no repeated roots. A somewhat surprising arithmetic can be introduced on the points on nonsingular elliptic curves. Many books have appeared on elliptic curves in recent years so there are many sources for further reading about elliptic curves. Our implementations are primarily motivated by Crandall and Pomerance [1]. That book has a succinct but sophisticated treatment appropriate for implementers of elliptic curve factorization methods. Silverman and Tate [6] give considerable detail regarding the mathematics of elliptic curves as well as providing historical context and different perspectives regarding elliptic curves. Elliptic curve factorization resources and records may also be found on the web; for example, see [7].

The curves appearing in Figures 1-3 show some of the variety of forms that appear in nonsingular elliptic curves. A "typical" line intersects the curve at three points. The arithmetic on elliptic curves is taken so that those three points sum to zero. Vertical lines are also taken to include a point at infinity that is the additive identity. Thus, (x_1, y_1) and $(x_1, -y_1)$ are considered opposites. Tangency is counted as a repeated point. It is straightforward, but requires some algebra, to check that

the sum of two distinct, non-opposite points (x_1, y_1), (x_2, y_2) is given by (x_3, y_3) where

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$x_3 = m^2 - x_1 - x_2$$
$$y_3 = -(m(x_3 - x_1) + y_1).$$

When the points are equal, the only change required is that $m = \frac{3x_1^2 + a}{2y_1}$.

Remarkably, these rules for addition give rise to an "abelian group" structure. In particular, the addition is associative and commutative. Figures 1-3 show some examples of intersections that make addition of certain points clear. For example, Figure 1 corresponds to the elliptic curve $y^2 = x^3 + 3x$ and we see that $(0,0) + (1,2) = (3,-6)$ on that curve. In a similar manner, Figure 2 shows $(1,1) + (1,1) = (-1,-3)$ on $y^2 = x^3 - 5x + 5$.

We will apply the addition rules on integer points modulo n. The quotient required in computing m corresponds to finding an inverse modulo n. When n is composite it may not be possible to find the inverse, and this failure is what gives the nontrivial factor. By carrying along a formal denominator, it is possible to avoid the inversion (which is expensive compared to the other arithmetic operations). This can be done in more than one way. We follow what is called modified projective coordinates in [1].

Modified projective coordinates carries three coordinates (x, y, z). A point (x, y, z) in modified projective coordinates corresponds to a point $\left(\frac{x}{z^2}, \frac{y}{z^3}\right)$ in ordinary coordinates. In particular, a point (x, y) on the elliptic curve corresponds to $(x, y, 1)$. Moreover, we can take (0,1,0) to be the point at infinity.

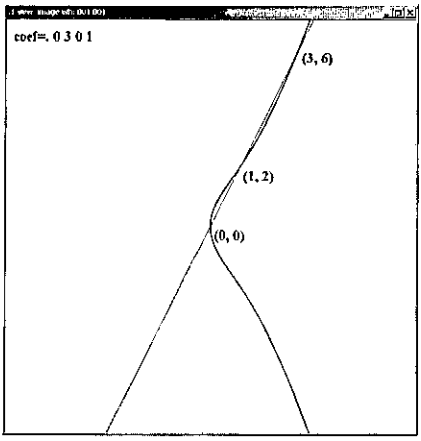


Figure 1. An elliptic curve that has a bulge shape and three intersection points.

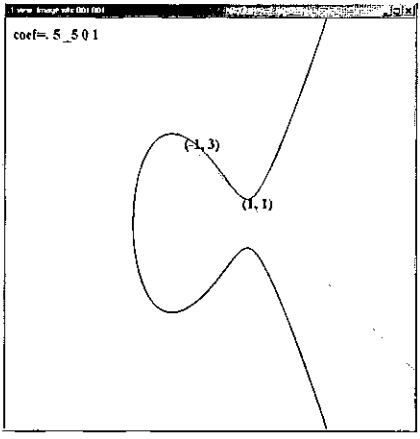


Figure 2. An elliptic curve that has an extended bulge and a tangent intersection.

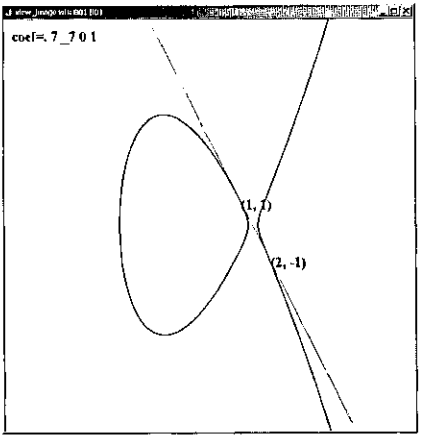


Figure 3. An elliptic curve that has two components.

The addition of two equal points (doubling) in these coordinates can be implemented by the following conjunction `ecdj`. The left conjunction argument gives the coefficients *a*, *b* of the elliptic curve and the right argument gives the modulus. The right function argument specifies the point that should be doubled.

```
NB. Elliptic curve double point,
NB. modified projective coordinates
NB. ab ecdj n Q
ecdj=: 2 : 0
'a b'=.m.
'x y z'=.3{.y.,1
if. 0 e. y,z do. 'x2 y2 z2'=.0 1 0x else.
  m=.(3**:x)+a**:*:z
  s=.4*x*yy=.*:y
  x2=.(*:m)+_2*s
  y2=.(m*s-x2)+_8**:yy
  z2=.2*y*z
  end.
n.|x2,y2,z2
)
```

Unequal points can be added using the conjunction `ecaj`. While this is more complicated than the formula for addition we gave earlier, keep in mind that we are carrying three coordinates, avoiding division entirely, and keeping the number of multiplications to a minimum.

```
NB. Ellitpic curve add
NB. P ab ecaj n Q
ecaj=: 2 : 0
:
'a b'=.m.
'x1 y1 z1'=.3{.x.,1
'x2 y2 z2'=.3{.y.,1
if. z1=0 do. 'x3 y3 z3'=.x2,y2,z2
 else.
  if. z2=0 do. 'x3 y3 z3'=.x1,y1,z1
else.
u1=.x2*z12=.*:z1
  u2=.x1*z22=.*:z2
  s1=.y2*z1*z12
  s2=.y1*z2*z22
  w=.u1-u2
  r=.s1-s2
  if. w=0 do.
    'x3 y3 z3'=.m. ecdj n. x1,y1,z1
    else.
    t=.u1+u2
    m=.s1+s2
    x3=.(*:r)-t*w2=.*:w
    y3=.-:(r*(_2*x3)+t*w2)-m*w*w2
    z3=.z1*z2*w
    end. end. end.
n.|x3,y3,z3
)
```

It is useful to have an efficient utility to compute mP where this denotes adding P to itself m times. Some sort of efficient approach should be used to avoid a linear time in m. One can use a variant of successive squaring. We continue following the algorithms in [1] and use a slightly different "ladder". It is implemented as the conjunction `ecmj` which is defined in the script [4], but not displayed here.

Elliptic Curve Factoring

The basic algorithm for elliptic curve factorization is straightforward and similar to the Pollard p-1 algorithm. Below is a summary of the algorithm. We assume that n is a composite integer and that neither 2 nor 3 divide n. The algorithm uses two bounds B_1 and B_2 which will be discussed below.

Elliptic Curve Factoring Algorithm

(Stage 0) Generate a random elliptic curve and a point P on it. Compute the discriminant, $4a^3 + 27b^2$, of the curve and compute the gcd with n. If it is n, select another curve. If it has a nontrival factor in common with n, return the factor as success at Stage 0.

(Stage 1) For each prime p less than B_1, update P by replacing it by an appropriate multiple: $P = p^e P$, where e is the largest power so that $p^e \le B_1$. Check whether the gcd of the z coordinate is a nontrivial factor of n. If it is, return that factor as success at Stage 1.

(Stage 2) Now consider the primes $q_1, q_2, \ldots, q_t$ that lie between B_1 and B_2. Let $Q = q_1 P$ where P was the point resulting from the end of Stage 1. We precompute some multiples of P, say $2P$, $4P$, ..., $200P$. Then we update $Q = Q + (q_j - q_{j-1})P$ where the addition is an elliptic curve addition modulo n. Repeat this for $j = 2,3,\ldots,t$. Check whether the gcd of the z coordinate of Q is a nontrivial factor of n. If it is, return that factor as success at Stage 2.

If all stages fail to find a nontrivial factor, then repeat the stages again, perhaps with larger B_1 and B_2.

The idea of Stage 1 is quite similar to what we saw in the Pollard p-1 algorithm. The hope is that for a prime r dividing n, the order of the elliptic curve modulo r may be relatively small. The order of the elliptic curve modulo r is the number of distinct points on the curve modulo r. Any point on the curve multiplied by the order of the curve gives the identity modulo r, but most likely not modulo n. Hence we can most likely find a factor of n if the order of the elliptic curve factors with the prime powers considered in Stage 1. Thus, we try to find a nontrivial factor of n by taking the gcd of the z-coordinate with n. If the order of the elliptic curve modulo r doesn't factor with the primes chosen, we can pick another random curve and try again. However, we can also go on to Stage 2, which is what we do.

Notice that in our algorithm for adding points in modified projective coordinates, the z-coordinate of each argument is a factor of the result (whether the points are equal or not). Thus, if a nontrivial factor appears in the z-coordinate, it will be carried along. There is some possibility that it will become a trivial factor, but that is unlikely compared to getting the nontrivial factor in the first place. Thus, we need not compute gcd's at every step, we can take them at our convenience. We check the gcd once at the end of Stage 1.

Stage 2 is similar to Stage 1 except the hope is that the order of the elliptic curve modulo r is of the form $q_j \prod p_i^{e_i}$ for a single prime q_j between B_1 and B_2. Since almost all the $(q_j - q_{j-1})P$ will be in the precomputed list of the multiples of P, the cost of trying each q_j is a single elliptic curve addition. As we noted before, we do not need to try the gcd at each point, we merely need to "visit" the point and eventually compute the gcd.

This algorithm is implemented in J in the script [4] and the main function is `fac_ecmj`. Its left argument is a list of B_1 and B_2 pairs, while the right argument

is the number to be factored. In our implementation, the left argument controls the number of elliptic curves that can be tried. The default is to compute 20 B_1 and B_2 pairs where B_1 ranges geometrically from near $\ln(n)$ to near $\exp\left(\frac{1}{\sqrt{2}}+\sqrt{\ln(\sqrt{n})\ln(\ln(\sqrt{n}))}\right)$ and $B_2 = B_1 \ln(B_1)$. Different authors give variations upon the upper bounds, mostly varying the constants somewhat. Of course unless one knows a great deal about the expected size of the factors, in practice it seems prudent to try many B_1 and B_2 pairs increasing toward a large choice of the bound.

The result of `fac_ecmj` is a list of three numbers. The first is the factor found (or a trivial factor). The second is the level it was found, 0, 1, or 2, or _1 if no nontrivial factor was found. The third is the index in the list of B_1 and B_2 pairs that succeeded (if it did).

Also, running `fac_ecmj` sets a global variable, `Last_random_ec`, that records the elliptic curve and point on it that was used for the factorization. When a "hard" number has been factored, experts are often interested in how it was accomplished. In particular, knowing the elliptic curve and point on it allows the verification of the technique used to produce the factorization.

The results for the trial points x computed in the earlier section are shown below.

```
    fac_ecmj x
       4234213 2 4
           277 1 0
     161603867 2 3
        401617 1 0
        367823 2 5
    1969145333 1 7
1628419520753 1 4
           839 1 0
          1087 1 0
      80621729 2 0

    Last_random_ec
1 1377804045 906358090697750791337 36 1119324186 524846382
```

Notice that all of the numbers were factored. It took 22 seconds to do the ten factorizations. Notice successes at both levels 1 and 2 appeared. Also notice the larger factors tended to require more steps which corresponds to larger choices for B_1 and B_2.

As our last example, we consider the factorization of the following 27 digit integer that is the product of two primes: one with 13 digits, the other with 15 digits.

```
152415787533657061564561727
```

Since `fac_ecmj` is probabilistic in the sense that random elliptic curves are used, timing a single factorization may be misleading. Here we record the result of `fac_ecmj` on the above 27 digit integer ten times. We append the seconds used in the last column.

```
  1234567890133  2 13  136.30
  1234567890133  1 10   39.85
              1 _1 20 1180.41
  1234567890133  1 18  696.14
123456789012419  1 19  989.31
123456789012419  1 18  687.79
123456789012419  1 14  165.10
  1234567890133  2  9   33.32
  1234567890133  2 16  397.80
123456789012419  2 19 1184.69
```

Notice that the function failed to find the factorization once. It took between around 30 seconds and 20 minutes to complete the task in any case. This gives a sense of the range of effectiveness we may see in our implementation.

Postscript

We have seen that the factorization techniques that we have considered are effective at factoring random 25 digit numbers. Hard factorizations of this size (those that involve just two primes) can usually be successfully done with our elliptic curve implementation. However, this is around where our implementation begins to struggle. The Quadratic Sieve and Number Field Sieve are more sophisticated techniques that are better suited for that situation [1], especially when larger numbers are to be factored. Those algorithms are beyond the scope of this note. Such factorization problems are not typical for randomly chosen integers. However, such products are used in RSA coding, so these are important applications.

In this note we have not organized our functions to completely factor an integer. We plan to discuss such a function in Part II. One serious impediment to completing that is the need to know when a number is prime.

It appears implausible to factor the product of two primes with scores or hundreds of digits with these algorithms, although records continue to be broken [7]. Nonetheless, we see that the surprising arithmetic defined on elliptic curves can be effective at finding factors of integers that are implausible by trial division and other simple techniques.

References

[1] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, Springer, New York, 2000.

[2] Jsoftware, J501b, *http://www.jsoftware.com.*

[3] C. Reiter, With J: Public Key Crytography, *APL Quote Quad*, 32 2 (2001) 21-24.

[4] C. A. Reiter, Elliptic curve factoring script, *factor_ecj.ijs*, *http://www.lafayette.edu/j/vector/index.html.*

[5] R. L. Rivest, A. Shamir, and L. Adlemen, A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM*, 21 2 (1978) 120-126.

[6] J. H. Silverman and J. Tate, *Rational Points on Elliptic Curves*, Springer-Verlag, New York, 1992.

[7] P. Zimmermann, The ECMNET Project, *http://www.loria.fr/~zimmerma/records/ecmnet.html.*

Show Hide with Splitters

by Paul Mansour

Splitters are used to create resizable panels on a form. A prime example of the use of splitters is the main form of Microsoft Outlook, which presents a TreeView on the left for categorizing email into folders, a ListView on the right to display the contents of the current folder, and an edit box on the bottom to preview the selected email message. These three objects are managed by splitters, which allow the user to adjust the sizes of the panels.

In addition to allowing users to resize panels to their liking, it is often necessary to allow the user to hide a particular panel, thus leaving more room for its adjacent panel. While it is possible to expunge and re-create the splitter and its associated panel, a better approach is to simply move the splitter under program control to one of its extreme positions, thus completely hiding the appropriate panel.

In most applications, splitters will be *attached*. This means that when a form is resized, one panel will remain the same size, while the other panel will shrink or grow. The panel that changes size we will call the *primary* panel, and the panel the remains fixed will we will call the *secondary* panel. The value of the Align property of the splitter determines which panel is the primary and which is the secondary. For example, a left aligned splitter has its secondary panel on the left, and its primary panel on the right:

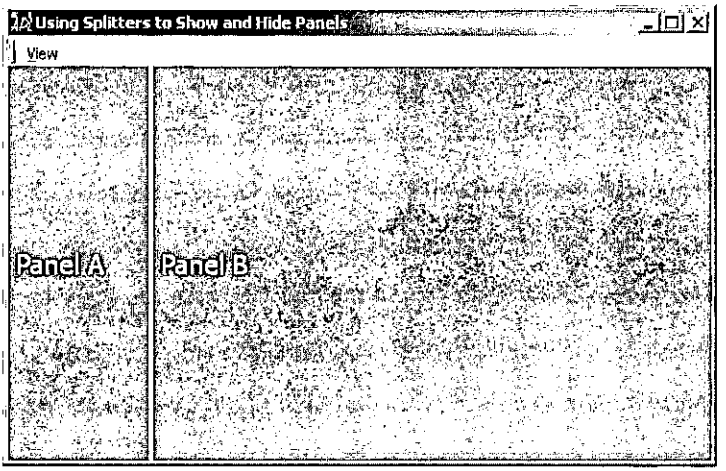


Figure 1

In this case, Panel A is the secondary panel, and Panel B is the secondary panel. Because the splitter is left aligned, resizing the form results in the width of Panel B growing or shrinking, while Panel A's width remains fixed.

A right aligned splitter has its secondary panel on the right and its primary panel on the left:

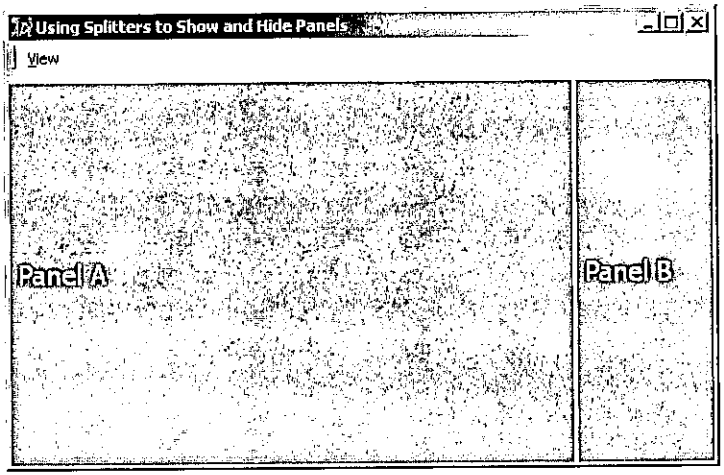


Figure 2

In this case, Panel B is the secondary panel, and Panel A is the primary Panel. Similar behavior occurs for top and bottom aligned splitters. If a splitter is not attached, then neither pane is considered primary, and both have equal weight.

When a splitter is at one of its extreme positions, the effect is to completely hide one panel, and to maximize the size of the remaining panel. Thus, to *hide* a panel, we must move the splitter under program control to one or the other of its extreme positions, and to *show* a hidden panel, we must move the splitter back to a position somewhere between its extreme bounds.

Usually secondary panels are hidden, in order to give more room to primary panels. This allows us to add our own Show and Hide methods to the Dyalog Splitter object. Based on the splitter's *Attach* property, the methods can themselves determine which of the splitter's two associated objects to show or hide. Hiding a panel is therefore straightforward; just set the position of the splitter under program control.

For a left or top aligned splitter:

```
S.Posn←0 0
```

And for a bottom or right aligned splitter:

```
S.Posn←S.##.Size
```

We may use the same value of Posn for bottom and right aligned (and top and left) because the Posn property of a splitter is partially read only. The interpreter ignores the row (column) position of a vertical (horizontal) splitter.

In addition to moving the splitter to one of its extreme positions, the splitter's size should be set to 0,0 in order to make it invisible and to effectively disable it from the user.

Restoring a panel is slightly more difficult, because the user may have resized the main form in the interim, and simply setting the position of the splitter to its pre-hidden position may not achieve the desired effect. If the user un-hides a panel on a form that he has resized, the splitter should be positioned such that the layout of the form appears as if the panel had not been hidden during the resize operation. In other words, if a user hides a panel, resizes the form, and then un-hides the panel, the result should be identical to simple resizing the form with both panels visible. While a left or top aligned splitter may, usually, be reset to its pre-hidden position, a right or bottom aligned splitter should be reset such that the right or bottom panel does not change in width or height respectively. However, if the form is made so small the above rules result in the splitter being outside the bounds of the form, and consequently one panel being completely hidden, then the splitter should be set to keep the panels in proportion to their pre-hidden sizes.

This wrinkle in the process of un-hiding a splitter slightly increases the complexity of hiding a panel, because we must capture and store a few items of information about the state of the splitter and the form. Before hiding a panel, we need to note:

1. The current absolute position of the splitter
2. The relative position of the splitter
3. The current size of the form (or the size of the bottom/right object)

The **Hide** function:

```
     ∇Hide←{                                  ⍝ Hide Splitter
[1]        α←(⎕NS'').##                       ⍝ Splitter
[2]        (~⍵):α Show 1                      ⍝ Show
[3]        ps←α.##.Size                       ⍝ Parent Size
[4]        α._LastPosn←α.Posn                 ⍝ Last position
[5]        α._LastRelPosn←α.Posn÷ps           ⍝ Last relative pos
[6]        α._LastSize←ps-α.Posn              ⍝ right/bottom object
[7]        α.Posn←ps×'BR'∊⍨α.Align            ⍝ New Position
[8]        α._Size←α.Size                     ⍝ Note splitter Size
[9]        α.Size←0 0                         ⍝ Make inaccessible
[10]       1:z←0                              ⍝ Shy result
[11]    }
     ∇
```

The Show and Hide functions are designed to be placed inside the splitter object, in an object-oriented fashion, and accessed as methods. For example, to place a reference to Hide in a splitter, we can write:

```
S._Hide←Hide
```

(As a matter of style, underscore is used as a prefix for all programmer-defined properties or methods stuffed into Dyalog GUI objects to avoid any name conflicts.) And now Hide may called as if it were a built in Dyalog method:

```
S._Hide 1
```

Thus the left argument defaults to the current space, which is the splitter object itself (Hide [1]). However, they can be used as stand-alone utility functions by passing in the splitter object as a left argument:

```
S Hide 1
```

Before setting the splitter's new position (Hide [7]), the current absolute and relative positions of the splitter are stored inside the splitter (Hide [4,5]). In addition the size of the bottom/right object is stored (Hide[6]). With the splitter now aware of its previous state, we can un-hide the panel using the Show method.

The **Show** function:

```
      ∇Show←{                                    ⍝ Show Method
[1]         α←(⎕NS'').##                        ⍝ Splitter
[2]         ~⍵:α Hide 1                         ⍝ Hide
[3]         α.Size←α._Size                      ⍝ Make accessible
[4]         α.Posn←{                            ⍝ New Position
[5]             'LT'∊⍨⊃⍵.Align:⍵._LastPosn      ⍝ Align Left, Top
[6]             ⍵.##.Size-⍵._LastSize           ⍝ Bottom, Right
[7]         }α                                  ⍝
[8]         ps←α.##.Size                        ⍝ Size of Parent
[9]         ob←∨/(α.Posn<0)∨α.Posn>ps           ⍝ Out of bounds
[10]        ob:z←α.Posn←⌊α._LastRelPosn×ps      ⍝ Relative pos
[11]        1:z←α.Posn                          ⍝ New Position
[12]    }
      ∇
```

The Show function first restores the size of the splitter, making it accessible to the user (Show [3]). The position of the splitter is set to its last noted position if the splitter is a left or top aligned splitter (Show [5]). If the splitter is a right or bottom aligned splitter, then its position is set such that bottom or right hand panel maintains is last noted size (Show [6])

Finally, if the splitter is now out of bounds (Show [9]), its position is reset such that the two panels retain their last noted proportions (Show [10]).

Show and Hide may be called with an argument of 1 or 0. Show 0 is equivalent to Hide 1, and Hide 0 is equivalent to Show 1. This allows us to dispense with Hide, and simply use Show B, where B is a Boolean (Show/Hide [2]). As Show and Hide are usually called in response to a toggle a menu item, this is quite convenient. A typical callback to a menu item would look like:

```
      ∇ SHOW X;M;S;F
[1]    ⍝ Menu Item
[2]     M←⍎↑X                          ⍝ Menu Item
[3]     F←#.GUI.getform M              ⍝ Parent Form
[4]     M.Checked←~M.Checked           ⍝ Toggle
[5]     S←F.Splitter1                  ⍝ Splitter
[6]     S._Show M.Checked              ⍝ Show or Hide
      ∇
```

Bonus GUI Tip

Splitters are extremely useful objects that automatically manage the Size, Posn and Attach properties of two objects. An undocumented feature of the Splitter is that it is also quite useful as a way to manage these three properties for *only one object.*

For example, to make an object that completely fill its parent, and that remains attached to all sides of its parent, simply create a splitter and set is position to the extreme left:

```
'F' ⎕WC 'Form' ('Coord' 'Pixel')
'F.G' ⎕WC 'Grid'
'F.S' ⎕WC 'Splitter'  '' 'F.G'
F.S.Posn←0 0
```

Index to Advertisers

All queries regarding advertising in VECTOR should be made to Gill Smith, at 01439-788385, Email: apl385@compuserve.com.

Submitting Material to Vector

The Vector working group meets towards the end of the month in which Vector appears; we review material for issue n+1 and discuss themes for issues n+2 onwards. Please send the text of submitted articles (hardcopy with diskette as appropriate) to the Vector Working Group via:

Vector Administration, c/o Gill Smith
Brook House
Gilling East
YORK, YO62 4JJ
Tel: +44 (0) 1439-788385
Email: apl385@compuserve.com

Authors wishing to use Word for Windows should contact Vector Production for a copy of the APL2741 TrueType font, and a suitable Winword template. These may also be downloaded from the Vector web site at www.vector.org.uk

Camera-ready artwork (e.g. advertisements) and diskettes of 'standard' material (e.g. sustaining members' news) should be sent to Vector Production, Brook House, Gilling East, YORK YO62 4JJ. Please also copy us with all electronically submitted material so that we have early warning of possible problems.

Subscribing to Vector

Your Vector subscription includes membership of the British APL Association, which is open to anyone interested in APL or related languages. The membership year runs from 1st May to 30th April. The British APL Association is a special interest group of the British Computer Society, Reg. Charity No. 292,786

Name: _____
Address: _____
_____
Postcode / Country: _____
Telephone Number: _____
Email Address: _____

UK private membership	£20	❑
Overseas private membership	£22	❑
Airmail supplement (not needed for Europe)	£4	❑
UK Corporate membership	£100	❑
Corporate membership overseas	£110	❑
Sustaining membership	£500	❑
Non-voting UK member (student/OAP/unemployed)	£10	❑

PAYMENT – in Sterling or by Visa/Mastercard/JCB only

Payment should be enclosed with membership applications in the form of a UK Sterling cheque to "The British APL Association", or you may quote your Mastercard, Visa or JCB number.

I authorise you to debit my Visa/Mastercard/JCB account

Number: ⌊⌋⌊⌋⌊⌋⌊⌋ ⌊⌋⌊⌋⌊⌋⌊⌋ ⌊⌋⌊⌋⌊⌋⌊⌋ ⌊⌋⌊⌋⌊⌋⌊⌋ Expiry date: ⌊⌋⌊⌋ | ⌊⌋⌊⌋

for the membership category indicated above,

❑ annually, at the prevailing rate, until further notice
❑ one year's subscription only

Data Protection Act:
The information supplied may be stored on computer and processed in accordance with the registration of the British Computer Society.

Signature: _____

Send the completed form to:
BAA, c/o Rowena Small, 12 Cambridge Road, Waterbeach, CAMBRIDGE CB5 9NJ, UK
Fax: +44 (0) 1653 697719

The British APL Association

The British APL Association is a Specialist Group of the British Computer Society. It is administered by a Committee of officers who are elected by a postal ballot of Association members prior to the Annual General Meeting. Working groups are also established in areas such as activity planning and journal production. Offers of assistance and involvement with any Association matters are welcomed and should be addressed in the first instance to the Secretary.

2003/2004 Committee

Chairman	Adrian Smith 01439-788385 apl385@compuserve.com	Brook House Gilling East YORK YO62 4JJ
Secretary	Anthony Camacho 0117-973-0036 acam@blueyonder.co.uk	11 Auburn Road Redland BRISTOL, BS6 6LS
Treasurer	Nicholas Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB5 9NJ
Journal Editor	Stefano Lanzavecchia stf@apl.it	c/o APL Italiana Corso Vercelli 54 20145 - Milano Italy
Projects and Publicity	Dr Alan Mayer 01792-205678x4274 a.d.mayer@swansea.ac.uk	European Business Management School, Swansea University, Singleton Park, SWANSEA SA2 8PP
Webmaster	Bob Hoekstra 01483-771028 bob.hoekstra@hoekstrasystems.ltd.uk	Dominique, Salisbury Road WOKING, Surrey GU33 7UR
Activities	Jon Sandles 01904-612882 jon_sandles@csi.com	138 Burton Stone Lane, YORK YO30 6DF
School Project	Stephen Taylor sjt@lambenttechnology.com	
Administration	Rowena Small 01223-570850 treas.apl@bcs.org.uk	12 Cambridge Road, Waterbeach, Cambridge CB5 9NJ

Journal Working Group

Editor:	Stefano Lanzavecchia	see above
Production:	Adrian & Gill Smith	Brook House, Gilling East, YORK (01439-788385)
Advertising:	Gill Smith	Brook House, Gilling East, YORK (01439-788385)

Support Team: Anthony & Sylvia Camacho (0117-973-0036),
Ray Cannon (01252-874697), Stephen Taylor (077-1340-0852)
Bob Hoekstra (01483-771028), Marc Griffiths

Typeset by APL-385 with MS Word for Windows
Printed in England by Short-Run Press Ltd, Exeter

VECTOR

VECTOR is the quarterly Journal of the British APL Association and is distributed to Association members in the UK and overseas. The British APL Association is a Specialist Group of the British Computer Society. APL stands for "A Programming Language" -- an interactive computer language noted for its elegance, conciseness and fast development speed. It is supported on most mainframes, workstations and personal computers.

SUSTAINING MEMBERS

The Committee of the British APL Association wish to acknowledge the generous financial support of the following Association Sustaining Members. In many cases these organisations also provide manpower and administrative assistance to the Association at their own cost.

Causeway Graphical Systems Ltd
The Maltings, Castlegate,
MALTON, North Yorks YO17 7DP
Tel: 01653-696760
Fax: 01653-697719
Email: sales@causeway.co.uk
Web: www.causeway.co.uk

Compass Ltd
Compass House
60 Priestley Road
GUILDFORD, Surrey GU2 5YU
Tel: 01483-514500

Dyalog Ltd
Grove House, Lutyens Close,
Chineham Court, BASINGSTOKE,
Hants, RG24 8AG
Tel: 01256-338461
Fax: 01256-316559
Email: sales@dyalog.com
Web: www.dyalog.com

APL2000 Inc
One Research Court
Suite 140
Rockville MD 20850
USA
Tel: +1 (301) 208 7150
Email: sales@apl2000.com
Web: www.apl2000.com

Insight Systems ApS
Nordre Strandvej 119G
DK-3150 Hellebæk
Denmark
Tel: +45 70 26 13 26
Fax: +45 70 26 13 25
Email: info@insight.dk
Web: www.insight.dk

HMW Computing
Hamilton House,
1 Temple Avenue,
LONDON EC4Y 0HA
Tel: 0870-1010-469
Email: HMW@4xtra.com

MicroAPL Ltd
The Roller Mill, Mill Lane
Uckfield
E.Sussex TN22 5AA
Tel: 01825-768050. Fax: 01825-749472
Email: MicroAPL@microapl.demon.co.uk
Web: www.microapl.co.uk/apl

Soliton Associates Ltd
Groot Blankenberg 53
1082 AC Amsterdam
Netherlands
Tel: +31 20 646 4475
Fax: +31 20 644 1206
Email: sales@soliton.com

Kx Systems
555 Bryant St., No. 375
Palo Alto CA 94301 USA
Tel: +1 866 kdb fast
Email (general enquiries): info@kx.com

In Europe: Simon Garland, SVP Technology
Email: simon@kx.com