VECTOR

100+ pages of the Best in APL

The Journal of the British APL Association

ISSN 0955-1433

Vol.7 No.4 April 1991

A Specialist Group of the British Computer Society

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 (ideally with a photograph of the author and a brief biographical note). Unless otherwise specified, each item will be considered for publication as a personal statement by the author.

Please supply as much material as possible in machine-readable form, ideally as a simple ASCII text file on an IBM PC compatible diskette (any format). APL code can be accepted as camera-ready copy, or in workspaces from APL*PLUS/PC, IBM APL2/PC or Dyalog APL.

Except where indicated, items in VECTOR may be freely reprinted with appropriate acknowledgement. Please inform the editor of your intention to re-use material from VECTOR.

Membership Rates 1991-92

Category	Fee	Vectors	Passes
UK Private	£12	1	1
Overseas Private	£20	1	1
(Supplement for Airmail)	£8		
UK Corporate Membership	£100	10	5
Overseas Corporate	£155	10	
Sustaining	£430	50	5
Non-voting Student	£6	1	1

The membership year 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 or Mastercard 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

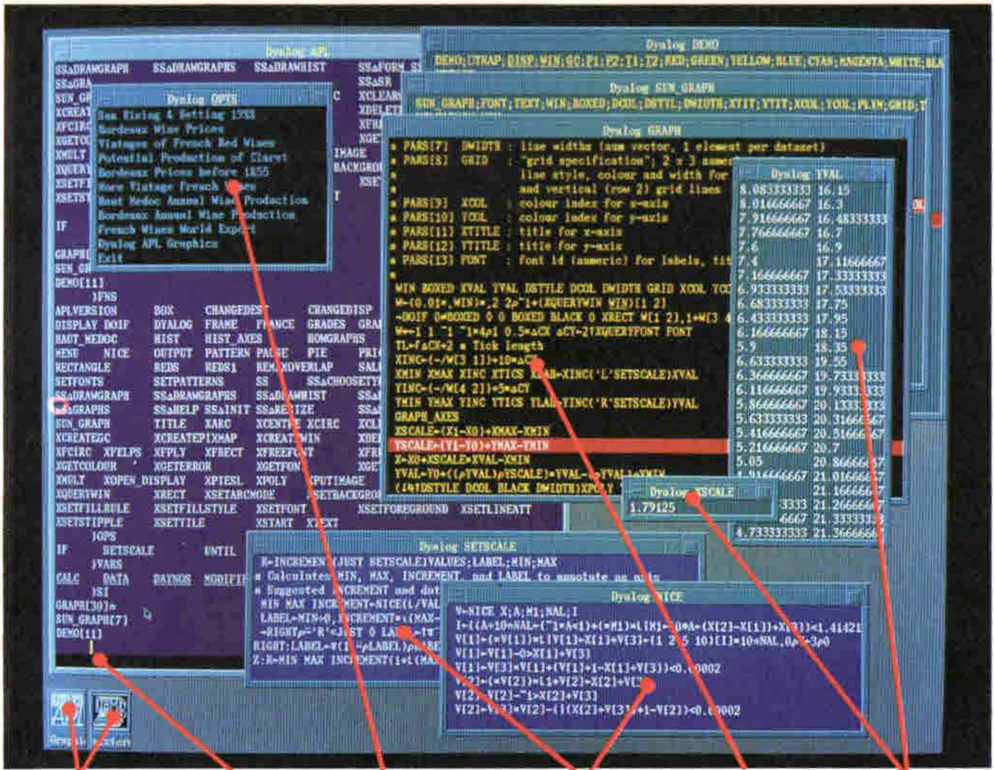
Advertisements in VECTOR should be submitted in typeset camera-ready A5 portrait format with a 20mm blank border. Illustrations should be black-and-white photographs or line drawings. Rates are £250 per full page, £125 for half-page or less (There is a £50 surcharge per advertisement if spot colour is required).

Deadlines for bookings and copy are given under the Quick-reference Diary. Advertisements should be booked with, and sent to, Alison Chatterton, whose address is given on the inside back cover.

CONTENTS

dyalog APL/X

GETS YOU THERE FASTER!



```
Dyalog APL
SSΔDRAWGRAPH   SSΔDRAWGRAPHS   SSΔDRAWHIST   SSΔFORM_S
SSΔGRA                                         SSΔSR
SUN_GR                                         XCLEAR
XCREAT                                         XDELETE
XFCIRC                                         XFR
XGETCO                                         XGET
XMULT                                          IMAGE
XQUERY                                         BACKGRO
XSETFI                                         XSE
XSETST
IF
GRAPH[
SEN_GR
DEMO[11]
      )FNS
APLVERSION    BOX        CHANGEDEST     CHANGEDISP
DISPLAY DOIF  DYALOG     FRAME  FRANCE  GRADES   GRA
HAUT_MEDOC    HIST       HIST_AXES      HOMGRAPHS
MENU    NICE  OUTPUT     PATTERN PAUSE  PIE      PRI
RECTANGLE     REDS       REDS1   REM XOVERLAP    SAL
SETFONTS      SETPATTERNS        SS    SSΔCHOOSETY
SSΔDRAWGRAPH  SSΔDRAWGRAPHS      SSΔDRAWHIST     SSΔ
SSΔGRAPHS     SSΔHELP SSΔINIT    SSΔRESIZE       SSΔ
SUN_GRAPH     TITLE   XARC       XCENTRE XCIRC   XCL
XCREATEGC     XCREATEPIXMAP      XCREATEWIN      XDE
XFCIRC  XFELPS XFPLY  XFRECT     XFREEFONT       XFR
XGETCOLOUR  ' XGETERROR          XGETFONT        XGE
XMULT   XOPEN_DISPLAY XPIESL     XPOLY   XPUTIMAGE
XQUERYWIN     XRECT   XSETARCMODE        XSETBACKGRO
XSETFILLRULE  XSETFILLSTYLE      XSETFONT        XSETFOREGROUND  XSETLINEATT
XSETSTIPPLE   XSETTILE           XSTART  XTEXT
      )OPS
IF      SETSCALE          UNTIL
      )VARS
CALC    DATA     DAYNOS   MODIFI
      )SI
GRAPH[30]*
SUN_GRAPH[7]
DEMO[11]
```

```
Dyalog OPTS
Sun Rising & Setting 1993
Bordeaux Wine Prices
Vintages of French Red Wines
Potential Production of Claret
Bordeaux Prices before 1855
More Vintage French Wines
Haut Medoc Annual Wine Production
Bordeaux Annual Wine Production
French Wines World Export
Dyalog APL Graphics
Exit
```

```
Dyalog DEMO
DEMO;⎕TRAP;DISP;WIN;GC;P1;P2;T1;T2;RED;GREEN;YELLOW;BLUE;CYAN;MAGENTA;WHITE;BLA
```

```
Dyalog SUN_GRAPH
SUN_GRAPH;FONT;TEXT;WIN;BOXED;DCOL;DSTYL;DWIDTH;XTIT;YTIT;XCOL;YCOL;PLYW;GRID;T
```

```
Dyalog GRAPH
⍝ PARS[7]  DWIDTH  : line widths (num vector, 1 element per dataset)
⍝ PARS[8]  GRID    : "grid specification"; 2 x 3 numer
⍝                    line style, colour and width for
⍝                    and vertical (row 2) grid lines
⍝ PARS[9]  XCOL    : colour index for x-axis
⍝ PARS[10] YCOL    : colour index for y-axis
⍝ PARS[11] XTITLE  : title for x-axis
⍝ PARS[12] YTITLE  : title for y-axis
⍝ PARS[13] FONT    : font id (numeric) for labels, tit
⍝
 WIN BOXED XVAL YVAL DSTYLE DCOL DWIDTH GRID XCOL YCO
 W←(0.01×,WIN)×,2 2⍴¯1↓(XQUERYWIN WIN)[1 2]
 →DOIF 0≠BOXED ◊ 0 BOXED BLACK 0 XRECT W[1 2],1+W[3 4
 W←W+1 1 ¯1 ¯1×4⍴1 0.5×ΔCX ΔCY←2↑XQUERYFONT FONT
 TL←⌈ΔCX÷2 ⍝ Tick length
 XINC←(-/W[3 1])÷10×ΔCX
 XMIN XMAX XINC XTICS XLAB←XINC('L'SETSCALE)XVAL
 YINC←(-/W[4 2])÷5×ΔCY
 YMIN YMAX YINC YTICS YLAB←YINC('R'SETSCALE)YVAL
 GRAPH_AXES
 XSCALE←(X1-X0)÷XMAX-XMIN
 YSCALE←(Y1-Y0)÷YMAX-YMIN
 X←X0+XSCALE×XVAL-XMIN
 YVAL←Y0+((⍴YVAL)⍴YSCALE)×YVAL-(⍴YVAL)⍴YMIN
 (↓⍉↑DSTYLE DCOL BLACK DWIDTH)XPOLY
```

```
Dyalog YVAL
8.083333333 16.15
8.016666667 16.3
7.916666667 16.48333333
7.766666667 16.7
7.6         16.9
7.4         17.11666667
7.166666667 17.33333333
6.933333333 17.53333333
6.683333333 17.75
6.433333333 17.95
6.166666667 18.15
5.9         18.35
6.633333333 19.55
6.366666667 19.73333333
6.116666667 19.93333333
5.866666667 20.13333333
5.633333333 20.31666667
5.416666667 20.51666667
5.216666667 20.7
5.05        20.86666667
4.916666667 21.01666667
            21.16666667
       3333 21.26666667
       6667 21.33333333
4.733333333 21.36666667
```

```
Dyalog YSCALE
1.79125
```

```
Dyalog SETSCALE
 R←INCREMENT(JUST SETSCALE)VALUES;LABEL;MIN;MAX
⍝ Calculates MIN, MAX, INCREMENT, and LABEL to annotate an axis
⍝ Suggested INCREMENT and dat
 MIN MAX INCREMENT←NICE(L/VAL
 LABEL←MIN+0,INCREMENT×⍳(MAX-
 →RIGHT⍴⍨'R'=JUST ◊ 0 LABEL
RIGHT:LABEL←⍕(1
Z:R←MIN MAX INCREMENT(1+⍳(MA
```

```
Dyalog NICE
V←NICE X;A;M1;NAL;I
I←((A+10⋆NAL←(¯1×A<1)+(×M1)×⌊|M1←10⍟A←(X[2]-X[1]+X[3])<1.41421
V[1]←(×V[1])×⌊|V[1]←X[1]÷V[3]←(1 2 5 10)[I]×10⋆NAL,0⍴V←3⍴0
V[1]←V[1]-0>X[1]÷V[3]
V[1]←V[3]×V[1]+(V[1]+1-X[1]÷V[3])<0.00002
V[2]←(×V[2])×⌊1+V[2]←X[2]÷V[3]
V[2]←V[2]-¯1>X[2]÷V[3]
V[2]←V[3]×V[2]-(|(X[2]÷V[3])+1-V[2])<0.00002
```

You can "iconify" and restore your windows as and when required. Here you see icons for an APL graphics window and a second Unix session.

You can execute expressions in the Session whilst Edit and Trace windows remain open.

You can edit character matrices and vectors of character vectors.

You can edit several functions and/or operators at the same time. Lines can easily be copied/moved between functions or to and from the Session.

The TRACE facility lets you step through your application line by line. As you do so, the APL stack is displayed and the current execution line is highlighted.

Watch your variables change as you run your code.

A 1-user licence for Dyalog APL/X costs just £1,500 and is available for a range of UNIX workstations including IBM RISC System/6000, Sun SPARCStations, and Hewlett-Packard HP9000-300/400 Series.

In addition to the unique Development Environment pictured above, Dyalog APL/X provides a powerful set of tools to help you produce good-looking, responsive and easy-to-use interfaces for end-user applications. Other important features include graphics support, external variables, a really fast component file system and a native file interface. You can easily access external functions written in C or Fortran, and an interface to Oracle is available on most systems.

For further details contact:

Dyadic Systems Limited
Riverside View, Basing Road, Old Basing,
Basingstoke, Hampshire RG24 0AL.
Tel: (0256) 811125 Fax: (0256) 811130

DYADIC

Editorial: a Babel of APL Dialects?

by Jonathan Barman

Are there too many versions of APL? Moving applications from one version to another seems the most painful, pointless and unrewarding task, only to be undertaken when absolutely necessary. How one wishes for absolute uniformity when doing such work.

However, it would be a very sad day if APL were so moribund that all implementations were the same. The new generation of APL interpreters are full of exciting features, which is evidence of an alive and progressive APL community. We need discussion and debate of the relative merits of the various features so that eventually a consensus view can be incorporated in the APL Standard.

The APL Standard seems to have had very little influence in removing differences between APL implementations. The idea of 'conforming extensions' means that most implementations can conform to the standard whilst retaining all the various extra features. In my view this just allows all the differences to persist, and the real crunch will come when the next Standard has to decide between competing features. At the moment the features battle is being decided in the market place.

The principal contenders in the market place for second generation APLs are IBM's APL2, STSC's APL*PLUS II, Dyalog APL, and MicroAPL who have entered the fray with their APL.68000 Level II interpreter which is reviewed in this issue. All these versions are slightly different, but not as fundamentally different as Sharp APL with its grounded system of nested arrays. Sharp APL is in a minority at the moment, so it is possible that the grounded system will fade away.

If it were possible to stick to just one flavour of APL then there would be no problem. Obviously, when one writes an application one can only use the interpreter available at the time, and the problems of migration to another version of APL can seem to be quite minor compared with the problem of getting the job done on time. However, migration is becoming more and more necessary. Mainframe applications are being transferred to Personal Computers. Applications on old PCs subject to the DOS memory limits are being revitalised by transferring to the 386 architecture where memory limits are removed. Soon everyone will want windows and networks which may require different APL implementations.

When writing APL it is good practice to ensure that the design does not rely on special features. This is much easier said than done. Quite minor 'features' can cause extraordinarily difficult migration problems, which cannot be predicted in advance. An example of a very minor change causing migration problems was the change in the Format primitive made by APL2. In VSAPL a leading space was guaranteed when using monadic format on a matrix, but in APL2 there is no leading space. The ramifications of such a tiny change were very surprising, with migrated code falling over in quite unexpected places.

Designing function calls which can be tailored to cater for major differences in features is quite a difficult task. Screen handling is the prime example of the problem. STSC's ǒWIN leads to screen driving functions being written in a quite different way from those that use Dyalog's ǒSM, which is different again from the AP124 processor in IBM's PC APL2. File handling is another area where a plethora of radically different features often makes an otherwise good design difficult to migrate. The article on WAGS in the Random Vector in the last issue (7.3) was a particularly interesting exercise in designing graphics functions which could be implemented in any APL implementation.

The wide range of different features that are currently available needs to be reduced. Of course there are problems. APL vendors are obliged to provide an upgrade path for their existing users so that their code continues to work with any new version. Special features are bound to be seen in the light of competitive advantage. However, convergence is possible, and actually happening now. MicroAPL have deliberately made their Level II as compatible as possible to IBM's APL2.

Everyone who uses APL should be aware of those features that are likely to be included in the next Standard and those that should be avoided. There is a need for publicity and debate so that nobody is surprised should a feature be downgraded at some time in the future. Those who work with more than one version of APL have a duty to the APL community to give their views on the relative merits of the various features. Let us see if we can make Vector the forum for discussion on this topic.

Quick Reference Diary 1991

Date	Venue	Event
26 April	IEE London	BAA Meeting
7 June	IEE London	AGM and BAA Meeting
4th - 8th August 1991	Stanford	APL91: The Next 25 Years
20 September	IEE London	BAA Meeting
25 October	IEE London	BAA Meeting
22 November	IEE London	BAA Meeting

Starting from February 1991, all British APL Association meetings will be held in the IEE, Savoy place. Nearest tube outlets: Temple or Embankment.

Meetings are normally held on the 3rd Friday of the month throughout the autumn and spring.

Overseas APL User Groups

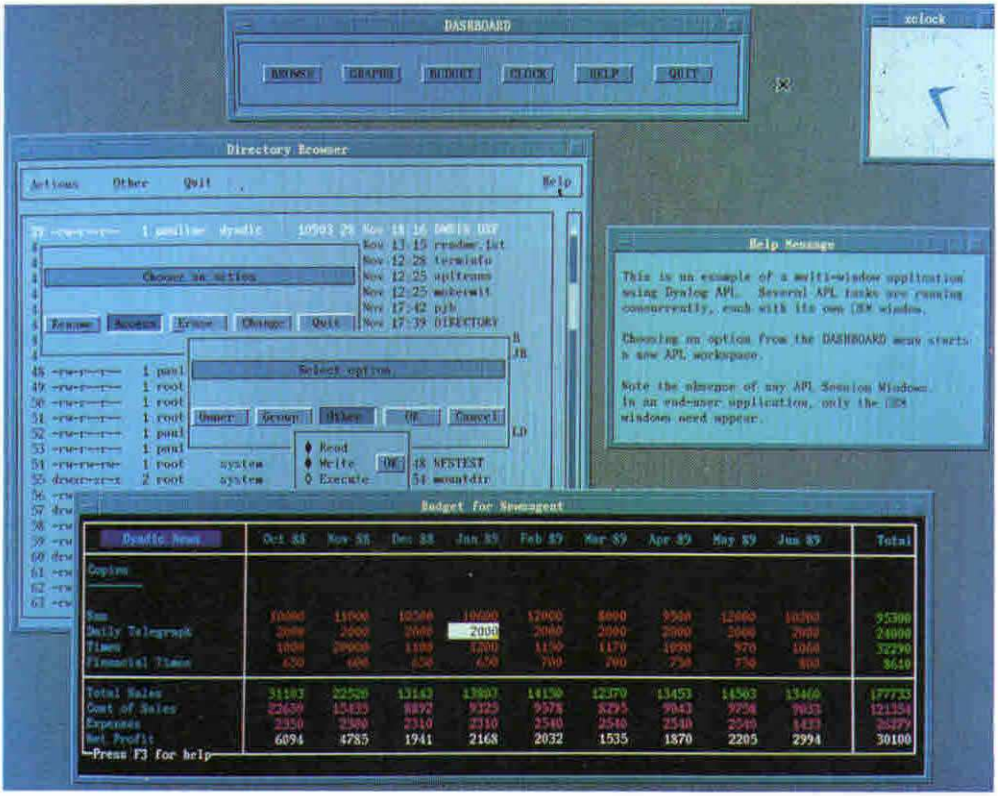
The Vector working group are anxious to maintain contact with as many overseas associations as possible. If you are running an APL group and would like to receive Vector in exchange for your own newsletter or journal, please contact Alison Chatterton at the address on the inside back cover. We will endeavour to compile and maintain an accurate list of contacts of overseas groups, and publish this regularly with our APL Product Guide.

Dates for Future Issues of VECTOR

	Vol.8 No.1	Vol.8 No.2	Vol.8 No.3
Copy date	1st June 91	5th Sept 91	1st Dec 91
Ad booking	8th June 91	12th Sept 91	1st Dec 91
Ad Copy	15th June 91	21st Sept 91	12th Dec 91
Distribution	July 91	October 91	January 92

dyalog APL/X

AS YOUR USERS SEE IT!



Help Message

This is an example of a multi-window application using Dyalog APL. Several APL tasks are running concurrently, each with its own ⎕SM window.

Choosing an option from the DASHBOARD menu starts a new APL workspace.

Note the absence of any APL Session Windows. In an end-user application, only the ⎕SM windows need appear.

Budget for Newsagent

Dyadic News	Oct 88	Nov 88	Dec 88	Jan 89	Feb 89	Mar 89	Apr 89	May 89	Jun 89	Total
Copies										
Sun	10000	11000	10500	10000	12000	8000	9500	12000	10200	95300
Daily Telegraph	2000	2000	2000	2000	2000	2000	2000	2000	2000	24000
Times	1000	20000	1100	1200	1150	1170	1090	970	1060	32290
Financial Times	650	600	650	650	700	700	750	750	800	3610
Total Sales	31183	22520	13142	13807	14150	12370	13453	14563	13460	177735
Cost of Sales	22659	15435	8892	9325	9578	8295	9043	9758	9033	121354
Expenses	2350	2300	2310	2310	2540	2540	2540	2540	1433	26277
Net Profit	6094	4785	1941	2168	2032	1535	1870	2205	2994	30100

Press F3 for help

This photograph illustrates the sort of modern windows applications you can develop using Dyalog APL/X. At the heart of the user-interface is a system variable ⎕SM which represents the layout, content and behaviour of a set of fields within an X window on the display. Users interact with ⎕SM using the keyboard and/or mouse through a system function ⎕SR. Together these tools provide an extremely powerful yet natural and easy-to-program user-interface for your APL applications.

Dyalog APL/X is just one of a fully compatible family of APL products for UNIX workstations, multi-user UNIX systems and DOS/386 PCs.

For further details contact:

Dyadic Systems Limited
Riverside View, Basing Road, Old Basing,
Basingstoke, Hampshire RG24 0AL
Tel: (0256) 811125 Fax: (0256) 811130

DYADIC

British APL Association News

Recruitment Officer's Report by Jill Moss

The role of Recruitment Office for the British APL Association is to endeavour to increase the membership by encouraging people to join and by persuading lapsed members to renew. The Recruitment Officer also tends to be the focal point for enquiries about the Association. I always try to send a personal letter in response to each individual enquiry, accompanied by one of the Association's leaflets, a membership form and information about Vector. Invariably, this has the desired result, with the enquirer deciding to take the plunge and join!

During the two years in which I have held the post of Recruitment Officer for the Association, various other initiatives have been tried. These have met with varying rates of success, and have included mail shots extolling the benefits of membership:

- to lapsed members, both in the UK and abroad
- to individuals on APL People's own mailing list
- to attendees of the ASL Conference in Swansea (drawing their attention to the support provided by the BAPLA)
- to delegates of APL 90

Mail shots have proved to be quite an effective means of pulling in new members and convincing former members to rejoin. We have also tried:

- Advertising in the Computer Bulletin (Journal of the British Computer Society)
- running a competition to encourage existing members to persuade other people to join (by offering them the chance to win a trip to APL 90 in Copenhagen)
- getting all the major APL vendors to promise to enclose one of the Association's publicity leaflets with every APL interpreter sold

The latter of these is quite effective (provided vendors remember to enclose the leaflets of course!). Unfortunately, the previous two initiatives failed to produce much in the way of results and therefore will not be repeated.

As far as future efforts go, we plan to target specific groups of people, such as teachers, and also to produce a range of leaflets to promote certain benefits of membership, such as the Software Library. Anyone who would like to help with writing or distributing leaflets, or who has any bright ideas for other ways of promoting membership, please let me know - all offers gratefully received!

News from Sustaining Members

Compiled by Alison Chatterton

APL People Limited

The New Year got off to a promising start with several people taking new jobs and more contract work being secured for the Company. However, signs that the economy has slowed down considerably are now becoming evident as more companies are reluctant to commit themselves to taking on extra staff. We now find ourselves in the unusual position of having more people seeking jobs than we have vacancies! So any companies out there who are thinking of taking on skilled 'APLers' this year, why not get in touch with us now, while we have a good selection of candidates for you!

APL Manufacturing's production control software continues to sell well, particularly to companies in the defence industry (so some good will have come out of the Gulf war after all!) The latest version of PEFAC, which is designed to run much faster on the hard disk, is currently on trial with the Ministry of Defence, who are looking for software to control the pricing of all their sub-contract work.

HMW Computing Limited

HMW Computing Limited continue to support, develop and market "4XTRA" our front end Foreign Exchange Dealing and Position Keeping System.

Each time I write news for Vector it seems that we have just released another upgrade of software. To keep the tradition going, we release another upgrade on 1st of March (tomorrow) which includes enhanced user monitoring software. This allows us simultaneously to monitor all machines running 4XTRA on a network from any point on the network. The monitoring software (4ARMS) picks up any stations APL errors instantly, and can then "take over" the users machine, investigate and fix it up, and return control to the user. With over 100 users, this gives us the facility to offer instant support, without physically having to visit the station on the network.

We have recently moved to larger offices within Hamilton House which has given us much more room for our consultancy teams and for visiting clients.

If you would like to see the latest 4XTRA system, we are exhibiting at a number of exhibitions and would be please to offer invitations or if you prefer a demonstration can be arrange at our offices or on your own premises. à

Dyadic Systems Ltd

Dyadic is now shipping Dyalog APL Version 6.1 for DOS/386, and certain UNIX, and X Windows implementations. The new version contains the following enhancements:

Browsing/Editing Variables

You can now open a display/edit window for a variable. The window is opened in the same way as for functions and operators, i.e. by using *)ED* or by positioning the cursor over the name and pressing the <EDIT> key. In Dyalog APL/X you can also "point-and-click" with the mouse. The appearance of the variable in the window is exactly as if it were displayed using standard output ignoring *⎕PW*.

Data in an Edit window can be scrolled vertically and horizontally so that you can easily browse through a large array. ANY variable can be displayed, although not all types of array can be edited. Additional colour parameters may be specified to distinguish between variables and functions and between "editable" and "non-editable" arrays.

If you assign new values to a variable for which there is an open Edit window, the window is immediately updated. This is particularly useful for debugging when using the *TRACE* facility, because you can watch the value of one or more variables change as you step through a program.

⎕ED

A new system function, *⎕ED*, has been introduced to permit the browsing and/or editing of variables and functions/ operators under program control.

Screen Manager Improvements

The Dyalog APL Screen Manager (*⎕SM*/*⎕SR*) has been enhanced. In all implementations, the handling of TIMEOUT events has been improved. In the UNIX and X Window versions, *⎕SR* can now be interrupted by another process (e.g. another APL task or an Auxiliary Processor) and the interrupt trapped. This allows several independent APL tasks to signal events to one another and broadens the scope for processing "real-time" data feeds.

MicroAPL Ltd

Sales of our new second-generation APL - APL.68000 Level II - are continuing to be very strong and we are now shipping significant volumes to the USA. We have made a couple of minor alterations to the interpreter since its release last year and we are now shipping release 1.17. Users who have not received upgrade notes should contact MicroAPL (or their local distributor) for details of the changes.

We are pleased to announce the appointment of Uniware as a distributor for APL.68000. Uniware are probably the leading French APL specialists and their appointment fills a major gap in our overseas representation. France has a large and enthusiastic APL community and we feel that a local supplier for APL.68000 will enable us to offer a better service to our current and prospective French clients.

In addition to Uniware distributing our APL software in France, MicroAPL has been appointed as a dealer for Uniware's range of software products for the PC. The Uniware software range includes an APL debugger, report formatter, menu generator and spreadsheet-like data entry utility. The software is available for APL*PLUS and APL*PLUS II. We feel that the APL Debugger is an especially powerful utility, enabling APL programmers to view the code being executed and the result generated at the same time, with a facility to coexist with APL software which itself makes window calls.

The entire range of products is very reasonably priced and offers a significant boost to programmer productivity for APL*PLUS users. Site licences for the software are also available.

As well as adding the Uniware range of software to our product list, MicroAPL has released an upgrade to our MicroPLOT software for the IBM PC - MicroPLOT/GSS. MicroPLOT provides a number of simple but powerful graphics commands to facilitate the production of two dimensional business graphics. MicroPLOT/PC, the implementation for APL*PLUS PC, has been available for some time now, incorporating drivers for the various types of PC display and also for HP-family graph plotters.

MicroPLOT/GSS is an implementation of MicroPLOT which uses the APL*PLUS PC interface to the GSS subsystem. APL programmers can use the GSS subsystem via the high level MicroPLOT functions and produce device independent graphs without having to invest a large amount of time and effort in understanding the ramifications of the many GSS interface calls.

MicroAPL is now shipping release 10 of APL*PLUS PC. The upgrade includes a number of useful new features including ⎕*NA* for access to non-APL routines and a new system function ⎕*MOUSE* (what else!) to allow access to the mouse under program control.

REUTER:FILE

Reuter:file are unable to attend the Vendors' Forum meeting on 26th. April. We will continue to support the BAA as a sustaining member and take an interest in its work.

APL91 Paper Abstracts at 13 March 1991

from Charles Schulz

A0 System - APL in Construction Planning. Alexei I. Miroshnikov, Central Research Economic Institute, USSR. This paper describes an automation system, named A0, for construction planning. A0 was developed for use by big building companies at various stages of preparing and executing construction works. It supports different tasks starting with calculation of project estimates, bills of materials, and schedules of works, and ending in providing the company executives with complete information for the supervision of the building process in their company. A0 also includes a DBMS or the all-Union data base on construction. The amount of data in the data base is about 45MB for the Leningrad region alone. At present, the system is used by the Leningrad Building Corporation. A0 is written in APL*PLUS/PC. The system will be distributed with the run-time version of APL*PLUS for Soviet customers after performance tests. A0 was developed in market competition with six similar projects based on different programming systems: FoxBase, Clipper, Turbo-Pascal, and MS C compiler. The advantages of the system under consideration were in many ways ensured by choosing APL. Some advantages and disadvantages of using APL for this kind of application are discussed as well.

An Oracle SQL - APL2/PC Auxiliary Processor. Stephen Deerhake, Westport Systems, Inc., USA. An APL2/PC auxiliary processor for interfacing ORACLE SQL with APL2/PC modelled after APL2/370's AP 127 auxiliary processor, is described. While providing direct compatibility with AP 127 service requests, the AP described here also offers several enhancements to AP 127 service requests. Although written for ORACLE SQL and 16-bit APL2/PC, the programming style employed provides a great deal of isolation between the DBMS and target interpreter, thus providing a high degree of portability to other DBMS implementations (e.g., Database Manager under OS/2) and target interpreters. Besides providing SQL services for a desktop APL environment, the AP demonstrates that it is possible to write large auxillary processors in a high-level language for APL2/PC which can be installed in extended memory and run in protected 80x86 CPU execution mode.

APL as an Embedded Language: The Ultimate Application? Jean Jacques Girardot, Ecole des Mines, France. This paper describes a new approach to the development of customized applications. It first discusses two problems with APL programming: writing efficient programs and building user interfaces. It then describes the proposed solution, consisting of writing the skeleton of the application in an efficient compiled language, using some predefined building blocks, and developing the other parts in APL. This approach is closer to integrated systems, such as spreadsheets or data-base managers, than to traditional APL applications executed under the control of an interpreter. It differs from these integrated systems in the fact that the development cost is kept low, so that new applications, highly customized for specific end users, may be built from scratch or, more exactly, from predefined building blocks.

APL Composition, Inversion and Evaluation of Programs Synthesized with Monadic Functions. Alvin J. Surkan, University of Nebraska, USA. APL is used to facilitate the automatic inversion and execution of programs which are synthesized by strict functional composition. Programs for performing high level functions and their inverses can be logically synthesized and then tested in arbitrary instances. APL is used to compose higher level functions by repeated left application of functions and operators designed expressly so that their syntax is limited to be monadic. A small system of auxiliary functions is sufficient to do the synthesis of inverse programs. This system operates on sequences of applicative functions which must be provided originally with their forward form coded in functional style APL. The system inverts higher level user-defined functions and generates correctly sequenced function calls which are used in testing the synthesized inverses. Using this system, the mechanistic reformulation of the forward functions and their exact, or even their approximate, inverses can be produced and then checked for consistency.

APLITDS: An APL Development System. Carlo Alberto Spinicci, APL Italiana srl, Italy. APL ITaliana Development System (APLITDS) is the internal application development system for APL Italiana, which aims to reduce development efforts, simplify maintenance, and standardize programming styles. It was originally produced by one programmer in his spare time. Details of the system and development methods are discussed.

An APL Rule-Based System Architecture for Image Interpretation Strategies. P. Bottoni, P. Musslo, M. Protti, Universita di Milano, Italy. APL is both a prefered language for image processing and description tasks, and a language for artificial intelligence applications, typically expert systems. This paper presents the architecture of an APL rule-based system to realize description synthesis strategies, i.e. sequences of actions evaluating the properties of structures detected in an image and to be related with objects in the real scene from which the image was drawn. Descriptions of images are stored in a data structure which is formally presented in terms of the APL2 syntax. The system processes rules whose format is: search for instances of an antecedent, evaluate a condition on the attributes of the found instance, perform an action if the condition is satisfied. Describing an image often requires that reflective actions be taken, i.e. actions in which the system examines the state of the computation and its internal state to select the next action to be executed. It is shown how APL provides all the features by which a reflective mechanism can be realized through the use of metarules of the same format and to be interpreted by the same program as rules.

APL Technology of Computer Simulation. A.Yu. Boozin, I.G. Pospelov, Academy of Sciences, USSR. Advantages and disadvantages of APL language usage for computer simulation are considered. The language has been actively used in the Computing Center during the last ten years for simulation of socio-economic systems. The original software package was developed with the aid of APL*PLUS/PC. This package makes the simulation researches easy and helps to get the results in graphical form.

Application of Nested Arrays to Databases for Engineering Design. Yehonathan Hazony, Boston University, USA. Conventional computer-aided design (CAD) systems are described. The addition of new mathematical tools to an existing design system is generally difficult since it has to be accomplished through a software interface to an external system. These difficulties are compounded by the structural complexity of the data representing an engineering design. The software interface should include a rich set of tools to extract appropriate data for external analysis. Furthermore, it is expected that results of analytical processes be inserted back into the design database for further use in the design process. An APL2-based engineering design system, which includes both the requisite data management capabilities and a rich set of mathematical tools suitable for engineering design and analysis, is described.

Assembler Utility Functions for APL2/PC. Tauno Ylinen Finland. Efficiency obsessions of APLists may date from years when interpretive overhead was a financial penalty on mainframes and a prohibitive handicap on micros. Now, when the general tendency is towards graphical interfaces and other complexities, which usually also lead to prohibitive costs, such still-managable systems as APL suddenly appear competitive with regard to speed, space, and cost. However, compiled functions still have benefit for bottlenecks. There is also a pleasure in writing lightning-fast and very small routines in assembler which is not very different from finding more and more concise idioms in APL. Several approaches to compiled sub-programs in APL are discussed: compiling APL itself, building interfaces between APL and compiled language programs, and building assembly language modules called from APL.

Automated SQL Documentation Using APL2. Rexford H. Swain, Independent Consultant, USA. An application programmer working with APL2 and SQL/DS will often want to investigate and/or document the definitions of SQL tables and views, particularly when working with tables created by others. Unlike conventional APL file systems, SQL "knows" quite a bit about the objects it is storing, but this information is scattered throughout several system catalogs. An APL2 tool which combines and neatly formats available information about a table is presented. Interpretation of this information may point out conditions which are causing SQL to perform less than optimally. Some issues which influence SQL performance are considered, and some general guidelines for improving performance are suggested.

The Boston University Manufacturing Expert System (BUMES): An APL- based CASE application. T. Shojaie, S. Sadri, L. Zeidner, Y. Hazony, Reuters Information Services and Boston University, USA. Manufacturing engineering involves the processes and equipment required to design, analyze, fabricate, and test products. The transition from an engineer's first sketch to a finished machined part is traditionally a long, expensive, difficult path. This path includes a variety of separate software solutions to individual subproblems. These solutions raise confusing issues of capability, compatibility, and design integrity. This paper presents the design and implementation of a fully integrated solution to this problem. This system enables a designer to sketch a part graphically, and immediately have it machined. Because this rapid-prototyping system was designed as an integrated solution, it allows the process rules to be available in the part design stage, so that unmanufacturable design features are identified immediately when the designer attempts to add them to the part. Design changes and subsequent machining are rapid because of relationships between the part's geometric elements, so that if some are changed, all can be adjusted automatically. The prototype Boston University Manufacturing Expert System (BUMES) was designed and implemented using the Expert System Generator (ESG) CASE tool in APL2.

Building an APL2 X-Window Interface for VM and AIX with a General APL2-to-C Interface. John R. Jensen, Kirk A. Beaty, IBM Corp., USA. This paper describes APL2/X, an interface between X-Windows and APL2, designed and built at the IBM Cambridge Scientific Center. It currently runs under VM/CMS and AIX. The AIX version of APL2 is an experimental prototype of APL2 in that environment; it was demonstrated at the APL90 conference held in Copenhagen. The APL2/X VM version of the interface uses the APL2 associated processor 11 to communicate with X. The APL2/X AIX version uses a new auxiliary processor to achieve the same functionality.

Calling APL2 from COBOL. Don Erickson, The Travelers Insurance Company, USA. This paper will discuss the reasons why a need came up for COBOL to call APL2 functions as executable modules, the methods used to complete the interface, and the benefits and concerns that arose during implementation and maintenance.

CATS - Computer Aided Testing of Software. Maurice Jordan, British Airways, England. This paper describes the implementation of an automated test system for APL functions. It extends assertive comments in APL to derive a notation for formal specification using pre and post conditions. These conditions are APL statements and so can be built into test functions. Data supplied to provide examples is subjected to mutations so that edge conditions can be tested for. It makes extensive use of modern APL ideas such as defined operators, phrasal forms, and function assignment.

Comparison of the Functional Power of FORTRAN 8x and APL2. Robert G. Willhoft, IBM Corp., USA. APL2 and FORTRAN, although very different, share the challenge of remaining "competitive" in the light of an onslaught of "modern" computer languages. To meet this challenge, both have attempted to enhance their position by adding significant new features to their language. APL2 is a specific example of an extension of APL. FORTRAN has also attempted to meet the challenges of modern programming by developing a new FORTRAN draft standard called FORTRAN 8x. The standard revises many areas of FORTRAN, but this paper will concentrate on those that affect the computational power of FORTRAN. Many of these changes were motivated by the increased use of vector and array "supercomputers". Therefore array features, the ability to act on entire arrays instead of individual elements, are an important part of this new standard. In doing this work, the FORTRAN community looked to APL as an example of a powerful array language. This paper will answer several questions regarding this new standard. First, from a computational or functional point of view, what are the major features of FORTRAN 8x? Next, how do these features compare with APL2? And finally, what can APL2 learn from the FORTRAN 8x work?

Compiling APL for Parallel and Vector Execution. Timothy A. Budd, Rajeev K. Pandey, Oregon State University, USA. The inherent parallelism of applicative languages such as APL and functional languages such as FP present a little-exploited somewhat unorthodox means of parallel programming. Here we summarize our investigation of a new approach to compiling such programs for execution on various types of parallel hardware. Our method centers around an intermediate form that is an extension of the lambda calculus. We present evidence that APL programs are easily translated into this intermediate form, and that this intermediate representation lends itself readily to code generation for a variety of parallel hardware.

A Dance of Rounds. J. Philip Benkard, IBM Corp., USA. Two different methods of getting sums of rounded numbers to add up to the rounded sum are discussed. The case of cascaded rounding, in which the individual numbers are replaced by sets of numbers to be rounded, is covered. Geometric properties of grade are reviewed.

Dependence of Learning Rate and Generalization on Number of Processing Elements in a Sparse Distributed Memory. Richard M. Evans, Alvin J. Surkan, Defense Training and Performance Data Center and University of Nebraska, USA. A simulated neural network was developed with APL on an 80386 microcomputer. The network was configured to associate task descriptions with ten categories of military occupational specialty. The number of processing elements in the problem was varied. Increasing the number of processors increased the speed of learning in the simulation. Generalization was not significantly different for various numbers of processing elements, except for one intermediate number at which generalization occurred about 15 percent higher. Analysis of the performance of the present network suggests that low level, natural language understanding is a form of text processing which promises to become an important application area for neural model-based computing.

Designing a Kanban Manufacturing System Using the Server Network Generator (SNG) CASE Tool. A. Bouchentouf-Idriss, L. Zeidner, Reuters Information Services and Boston University, USA. Developing concurrent real-world software systems requires a different sort of CASE tool than those designed for standard sequential von Neumann software development. Issues of concurrency, data flow, distribution, bottlenecks, load levelling, and distributed versions must be addressed. For cooperative distributed computing to be widely applicable, suitable CASE tools must be developed. This paper presents a manufacturing engineering application, a concurrent cooperative processing model of this application and the CASE tool that was used to design and implement it as a distributed software system. The application is a Japanese manufacturing control strategy called the "Kanban System". The application CASE tool used for implementation is the Server Network Generator (SNG).

DSS Structure and Algorithmic Transparency in APL. W.E. Cundiff, Griffith University, Australia. Earlier discussion focused on APL as an executable notation for understanding the inner workings of dialogue generation and data/model base management in decision support systems (DSS). The present work applies the notion of algorithmic transparency to the broader properties of system structure that connect the technology components of DSS. Using a concise set of idioms, embodying only the control construct of sequence, APL's direct definition form is employed in revealing patterns in the specification of a functional prototype, Group DSS.

The Dual Structure of Ordered Trees. Gerard A. Langlet, Commissariat l'Energie Atomique, France. Although they were described by K. Iverson in his 1962 book, ordered trees have never been implemented as APL objects. However, APL2, as well as ISO-APL, provide many facilities to handle them. This paper illustrates with simple examples the duality between the tree structure and what will be presented as an "object-oriented definition structure". APL functions are proposed to facilitate the conversion of the one form into the other one. The utility of trees is great in DB processing, in science as well as in business applications. Interfaces with the "outer world" - e.g. word processors or LISP and PROLOG can become easy to build with the help of this simple concept of duality.

Extending Structure, Type, and Expression in APL2. J. Philip Benkard, IBM Corp., USA. The two principal directions of APL are compared. Suggestions are made for the future enhancement of APL2.

Forecasting System of Employment Pension Scheme. Timo Korpela, Bo Lundqvist, Central Pension Security Institute, Finland. This paper describes the long-term forecasting system of the Finnish employment pension scheme. All employees and self-employed persons are included in the scheme. The forecasting system has three main models: demographic, pension expenditure, and financing. All models are programmed in APL2, with use of other products in the same environment, e.g. APE, SQL, ICU, AFP. The models are composed of modules, which make it possible to change the model at all levels. This structure also gives self-documentary programming code. Flexibility and documentary reasons have guided the programming so that compact and efficient code have not been primary objectives. The system gives all end users - also those who have no knowledge of APL - the ability to make tables and ICU graphics of the data.

Genetic Algorithms. Manuel Alfonseca, IBM Corp., Spain. Genetic algorithms, emulating biological evolution, are easy to program in APL. This paper shows a simple way in which they can be tested and analyzed.

Gerunds and Representations. Robert Bernecky, Snake Island Research, Inc., Canada. Gerunds, verbal forms that can be used as nouns, are recognized as having utility in the realm of programming languages. We show that gerunds can be viewed as arrays of atomic representations of verbs (functions) in a way which is consistent with the syntax and semantics of APL. We define derivations of verbs from gerunds for the J dialect of APL, show how these derivations perform sequencing, selection (in the sense of generalized forms of CASE and IF/THEN/ELSE), iteration (DO WHILE), recursion, and parallel (MIMD) computation. We conclude with alternative representations of verbs which are useful in other contexts.

How to Manage Large APL Projects - a User Interface Management System Approach. Richard R.N. Eller, TMT-Team Oy, Finland. Many new graphical user interface (GUI) needs can be handled by user interface toolkits or libraries. However, use of these will not solve all the complexities of a good user interface. Another approach is to choose, or design, a user interface management system (UIMS) that is responsible for all user interface actions and, only when necessary, will call actual application code to perform specific application tasks. What is APL best for? Calculations, experimentation, and data handling? So why should one include all the user interface complexities in the compact APL application code? This paper describes the architecture of a UIMS-based large APL application project. Since a UIMS approach differs from traditional hierarchical programming, many of these differences are described in detail. Particular emphasis has been given to the effect of this approach on project management and the various tasks of a software project. The paper ends by presenting some thoughts about future APL application development systems.

An Interactive Data Analysis System Developed Under APL. Peter I. Day, Unocal, USA. Unocal has used STSC APL*PLUS to build a software system for analyzing data from oil wells. Such data is routinely gathered from a variety of electrical, mechanical, and nuclear sensors lowered into the well- bore and, for our purposes, can be regarded as coincident depth-series vectors. The system was designed to allow the data analyst to perform two key tasks: rapid "visualizing" of the data; and selecting algorithms and parameters for interpreting the data. We have used APL to fulfil two different needs. At a fundamental level, there is one set of algorithms that actually performs the data analysis. At a higher level is a second set of functions that provides an interactive interface between the user, the data, and the interpretation algorithms. This interface is provided through menus that operate in both text and graphics mode, through text-entry panels, and through an extensive series of graphical routines. In text mode operation, we have made considerable use of ⎕WIN and related functions under keyboard control, whereas in graphics mode we have used ⎕G-style graphics under mouse control provided through ⎕INT 51. A demonstration of the system, concentrating on the higher-level interface, will be given.

L1...L3: Considered Harmful. F.H.D. van Batenburg, Leiden University, The Netherlands. It is said by non-APL-programmers that APL code is hard to read and that it is unstructured. Here we argue that APL programmers may refute this by pointing out some misunderstandings, but that a final analysis will show a deeper truth in these criticisms. We will show that APL gives ample opportunity for unstructured code. Two proposals are presented to address this problem. The first one rejects the established standard for labelling and suggests the adoption of a proper style of programming, enforced by a new standard of labelling. This standard will abolish unstructured code. Some negative aspects as well as the positive aspect of this proposal are discussed. The second proposal revives an old idea to introduce one single proper control structure in the language. This would make the current branch superfluous and enforce structured code.

Mastering J. Donald B. McIntyre, Scotland. It is exciting to recognize a major advance after three decades of APL experience with the introduction of the dialect called J. This paper introduces the reader to J by showing what problems the author had to overcome in learning it and what techniques he developed to aid him in writing programs in J.

An N-dimensional Data Structure in Support of Electronic Data Interchange (EDI) Translation. Georges Brigham, Edward Shaw, The APL Group, Inc., USA. A method is described by which data in a database system are named using sets. The sets exist in an n-dimensional data space in which each axis represents a homogeneous set and all axes (sets) are orthogonal. Data are named using an ordered combination of the names of the sets. Elements of data are identified by referring to the coordinates along the sets (axes of the database). An executable notation is used to describe relationships between the sets for purposes of discussion or execution on the computer. This methodology lends itself quite conveniently to APL, and currently forms the foundation for a robust commercial application performing electronic data interchange (EDI) translation.

Notes on C Programming for APL Programmers. Stephen Deerhake, Westport Systems, Inc., USA. As the domain of "callable" languages from APL increases, it is quite likely that APL programmers will find an increasing need for the ability to program in other languages as part of their support activities for APL programming. This is already becoming prevalent in mainframe APL2 (APL2/370), where it is not uncommon to find hybrid systems consisting of APL2, REXX and/or FORTRAN. Since most "external language" activity in APL-based systems centers on manipulating APL arrays, it is appropriate to examine the programming techniques in callable languages from the standpoint of facilitating APL array manipulation. Specifically, C language programming techniques for handling APL2/PC arrays are reviewed. The techniques discussed emphasize isolation of the application code from the underlying array structure, thus maximizing application code portability. While immediately applicable to the management of APL2/PC arrays, the programming techniques and tips offered are applicable to other APL array implementations as well.

Nuclear Power Plant Diagnostics in APL. Alexander O. Skomorokhov, Institute of Physics and Power Engineering, USSR. This paper discusses the application of APL to the development of diagnostic technology for the operation of nuclear power plants. The application is illustrated by an example of detecting and locating failed fuel elements through the use of Delayed Neutron Detectors (DNDs). This is accomplished by APL algorithms examining DND signals as the power distribution in various regions of the reactor core is altered.

On Performance and Space Usage Improvements for Parallelized Compiled APL Code. Dz-ching Ju, Wai-Mee Ching, Chuan-lin Wu, University of Texas at Austin and IBM Corp., USA. Loop combination has been a traditional optimization technique employed in APL compilers, but may introduce dependencies into the combined loop. We propose an analysis method by which the compiler can keep track of the change of the parallelism when combining high level primitives. The analysis is necessary when the compiler needs to decide a trade-off between more parallelism and a further combination. We also show how the space usage, as well as the performance, improves by using systems calls with the aid of garbage collection to implement a dynamic memory allocation. A modification of the memory management scheme can also increase available parallelism. Our experimental results indicate that the performance and the space usage improve appreciably with the above enhancements.

Programming for Events. D.S. Eastwood, MicroAPL Ltd., England. Modern windowing user interfaces offer both a challenge and an opportunity for the APL programmer. This paper discusses some of the factors that need to be taken into account when designing APL applications in a windowing environment. Some of the typical techniques required to produce a robust windowing application are discussed, and applications examples are quoted using APL.68000 on the Apple Mac.

Psycho-biographical Analysis with APL. Andrew V. Kondrashev, Alexander A. Kronik, Academy of Sciences, USSR. This paper deals with computer methods of psychological age measuring, psycho-biographical express-diagnostics of personality, and data analysis with APL.

Pure Functions in APL and J. Edward M. Cherlin, APL News, USA. Any expression in combinatory logic made up of combinators and variables can be abstracted into a pure combinator expression applied to a sequence of variables. Because there are great similarities between combinators and certain APL operators, a similar result obtains in many APL dialects. However, rewriting arbitrary APL expressions as pure functions requires new operators, not provided as primitives by any dialect. This paper defines functional completeness, gives a construction for achieving it, proves a conjecture of Kenneth Iverson that J is functionally complete, and shows how closely the major APL dialects approach those conditions.

The Server Network Generator (SNG): A CASE Tool for Distributed Cooperative Processing. L.E. Zeidner, Boston University, USA. The Server Network Generator (SNG) is a CASE tool that employs a problem solver's ability to represent an application as an ordinary block diagram, a graphical specification of its macroscopic structure. This functional decomposition provides a natural mechanism for subdividing the application into processing tasks that can be distributed across a computing network. Each "server" is a software process that assumes the role of one block in the diagram, performing one processing task, employing interprocess communication as indicated graphically. A distributed network of IBM 7437 VM/SP Technical Workstations are shown as a powerful platform for problem solving using the SNG. Distributed computing models based upon networks of microprocessors have long been proposed as an alternative to centralized mainframe computing. The 7437 provides a genuine VM/370 computing environment, and so can use mainframe systems as powerful nodes in the network, rather than attempting to replace them. An auxiliary processor is presented that was required to support interprocess communication via the shared-variable interface between virtual machines on different hosts.

Supply-Chain Management at Rowntree: Critical Success Factors for APL. Adrian Smith, Rowntree Mackintosh Ltd., England APL has been in use at Rowntree Mackintosh since late 1978; during this time it has developed from an initial application to 'little local systems' to play a vital role in the supply-chain management of the company. APL systems are now deeply embedded in all stages of company operations, from recipe modelling to the scheduling of raw materials, from five-year planning to detailed shift-by-shift production scheduling. This paper sets out to explain the critical success factors which encouraged the directors of Rowntree Mackintosh to entrust such a vital part of their business operations to an obscure Greek language which executes backwards.

Tacit Definition. Roger K.W. Hui, Kenneth E. Iverson, Eugene E. McDonnell, Iverson Software, Inc., Canada and USA. J permits a form of functional programming we call tacit definition, in which no variable or assignment appears. We show how many conventional programs can be transformed into tacit definitions. Many uses of these forms are given.

The User Command Processor. Jim Weigang, APL Consultant, USA. The User Command Processor is a new feature of several APL*PLUS systems which allows users to define commands, analogous to system commands, that can be executed from within any workspace. This enhancement is effected by means of two simple changes to the APL interpreter. Coupled with a suite of two-dozen predefined commands, the result is a file-based program storage and execution environment that integrates many important features not provided in standard APL systems. Using the command processor, applications of unlimited size can be developed, run, and maintained without many of the headaches that are characteristic of workspace-based systems. This paper describes the basic methods whereby the command processor operates, provides an overview of the predefined commands, shows how a new command can be defined, and illustrates how a large application can be built using the command processor.

Using Boolean Arrays to Build and Completely Analyze Function Networks. Kenneth Fordyce, Jan Jantzen, Gerald Sullivan, Gerald (Jay) Sullivan, Jr., IBM Corp., Technical University of Denmark and Rensalear Polytechnic Institute, USA and Denmark. A critical computational requirement for many of the decision technologies in the fields of MS/OR, AI/KBS, and DSS is the development and manipulation of a function network describing the relationship between "actors" involved in the application of the decision technology to a specific problem. This paper describes how we can fully build and manipulate a function network with boolean arrays including focusing networks, finding circular conditions, and grouping functions based on relative independence to identify parallel computational opportunities and substantially reduce the non-procedural aspect of the problem.

THE EDUCATION VECTOR

April 1991

Editor Alan Sykes

This Education Vector has been reprinted from VECTOR Vol.7 No.4. VECTOR is the Quarterly Journal of the British APL Association. For more information about the British APL Association, contact: Anthony Camacho, 2 Blenheim Rd, St Albans, Herts AL1 4NR Tel: 0727 860130.

Contents

Dr Alan Sykes,
c/o European Business Management School
Swansea University
Singleton Park, Swansea SA2 8PP
Wales, UK

Editorial

by Alan Sykes

'Hello and Welcome' as they say. The compilation of this Education Vector comes at quite a busy time. The APL Statistics Library Project (ASL!) is preparing for its launch, and not surprisingly I have been very much involved in such activities. I hope that in a future edition the ASL project and particularly its library of functions for basic statistics (up to A-level standard) will be reported.

Hence I am particularly grateful to Walter Spunde for his timely article entitled 'Shape, Ravel and Roll'.

It is now nearly three years since I took up the editorship of Education Vector. I have enjoyed my involvement with members of the British APL Association and particularly the opportunity to contribute to the columns of Education Vector. Readers of it have been very kind in expressing appreciation of the efforts that have been made to build up material that is intended to be helpful in introducing newcomers to the delights of using APL.

Nevertheless I am well aware that there is so much more that could be done. If only there were more than 24 hours in a day!

So I end this editorial with yet more words of encouragement to users of APL throughout the world to submit to Education Vector material that is appropriate to the aims and aspirations of it, together with any news and notes that readers may be interested in.

Shape, Ravel and Roll

by Walter G. Spunde

For over a decade keen mathematics instructors have been searching for ways to incorporate computing power into their teaching of maths. Language requirements have tended to deflect attention away from mathematical principles and the maths class has been in danger of becoming a lesson in the syntax of some computer language or package. An obvious answer was to use APL, but it was expensive, and difficult to justify to un-informed superiors.

I-APL opened new doors. Free, and nearly as convenient as a calculator, it should be the answer to a maths master's dream. One of the most appealing features of APL for the newcomer is the ability, after only a few minutes' exposure to the interpreter, to produce results, and, with a little additional effort, to make improvements and enhancements. There is an immediate reward for every effort made, and immediate feedback on mistakes. It is a superlative tool for education; but, since the power of using APL is as seductive as the drudgery of other languages is confining, it has even more potential for itself becoming the focus of attention. This note is intended to show how little APL notation is really necessary to make working with an I-APL disk productive in a maths class, with the focus clearly on mathematical concepts, at tertiary, or upper secondary level. At the same time too, it may sound a little warning to APL enthusiasts.

A Mathematical Perspective

Linearity is a central concept in much of mathematical theory, and the study of linear algebraic equations is one of the first non-trivial examples illustrating the properties of linear operators that students encounter.

The small square systems of equations usually presented, since they are manageable with hand calculators, provide only poor examples of these properties, as they are apparent only in the non-typical cases and are difficult to discern. Solving a square system of equations is, in any event, such a common operation that, like taking the square root of a number, (i.e. solving the quadratic equation $x^2 = n$) there should be (as there is in APL) a primitive to do the job.

By a "primitive" we should understand a mathematical operation that is so common and well understood, that the details of the algorithm producing the result need not be a concern (even when they are not, or have never been, known to the user.)

The typical problem leading to a system of equations is an arbitrary number of equations in any number of unknowns. Setting up a system of four equations in ten unknowns, from data in Figure 1, is conceptually no more difficult than setting up two equations in two unknowns. It may actually be easier, since the pattern in adding up certain fractions of various quantities is more apparent when there are several of them.

NUTRITIVE CONTENT OF FEEDS
(POUNDS OF ELEMENT PER 100 POUNDS OF FEED)

FEED	DIGESTIBLE NUTRIENTS	DIGESTIBLE PROTEIN	CALCIUM	PHOSPHORUS
CORN	78.60	6.50	0.02	0.27
OATS	70.10	9.40	0.09	0.34
MILO MAIZE	80.10	8.80	0.03	0.30
BRAN	67.20	13.70	0.14	1.29
FLOUR MIDDLINGS	78.90	16.10	0.09	0.71
LINSEED MEAL	77.00	30.40	0.41	0.86
COTTONSEED MEAL	70.60	32.80	0.20	1.22
SOYBEAN MEAL	78.50	37.10	0.26	0.59
GLUTEN FEED	76.30	21.30	0.48	0.82
HOMINY FEED	84.50	8.00	0.22	0.71
DESIRABLE INTAKES (POUNDS DAILY)	74.20	19.90	0.21	0.67

Figure 1

The computing power available from APL implies that the size of the equations we work with in class is limited only by what can be conveniently displayed on the screen, and that we need no longer be concerned with the errors that students might generate (and thus destroying a discussion of the patterns apparent in the working).

Fundamental concepts

The procedure for "solving" equations involves subtracting multiples of one equation from the others until an equivalent reduced system is found. This conceptual structure can be maintained if we can assign a name such as *EQN*1 to the string of coefficients in any equation together with the demands. Working with equations as in Figure 2 requires a knowledge of only the assignment arrow beside the elementary operations and the ease of doing it would in itself be sufficient justification for using I-APL in teaching elimination techniques. However, if we concentrate on the idea of *linear combination* - the sum of multiples - which underlies the very formation of the equations, we soon depart from this approach to the problem.

```
      EQN1 ← 78.6  70.1  80.1  67.2  78.9  77    70.6  78.5  76.3  84.5  74.2
      EQN2 ←  6.5   9.4   8.8  13.7  16.1  30.4  32.8  37.1  21.3   8    19.9
      EQN3 ←  0.02  0.09  0.03  0.14  0.09  0.41  0.2   0.26  0.48  0.22  0.21
      EQN4 ←  0.27  0.34  0.3   1.29  0.71  0.86  1.22  0.59  0.82  0.71  0.67

      ⎕ ← EQN1 ← EQN1 ÷ 78.6
1 0.8919 1.0191 0.855 1.0038 0.9796 0.8982 0.9987 0.9707 1.0751 0.944

      ⎕ ← EQN2 ← EQN2 - 6.5×EQN1
0 3.6029 2.176 8.1427 9.5752 24.0323 26.9616 30.6083 14.9902 1.0121 13.7639

      ⎕ ← EQN3 ← EQN3 - 0.02×EQN1
0 0.0722 9.6E¯3 0.1229 0.0699 0.3904 0.182 0.24 0.4606 0.1985 0.1911

      ⎕ ← EQN4 ← EQN4 - 0.27×EQN1
0 0.0992 0.0248 1.0592 0.439 0.5955 0.9775 0.3203 0.5579 0.4197 0.4151
```

Figure 2

A linear combination is the essential construct in discussing linearity. An operator is *linear* if the transformation of any linear combination of elements is the same linear combination of the transformations of each of the elements. In the notation of APL2:

```
FN Coefficients +.× Elements = Coefficients +.×  FN¨ Elements
```

Linear combinations occur widely in mathematics: polynomials are linear combinations of powers, Fourier series are linear combinations of trigonometrics, the familiar dot product is a linear combination of scalars, pre-multiplying a matrix by a vector forms a linear combination of the rows of the matrix, pre-multiplying a matrix by another matrix forms several linear combinations of the rows of the multiplied matrix, and, of course, solving equations by subtracting multiples of one equation from another is forming linear combinations of the equations. When we represent a set of equations by a matrix containing in its rows the vectors (*EQN*1,..) for the various equations, it is easy to see that the elimination procedure is equivalent to a matrix pre-multiplication.

This observation is one of the basic goals of an elementary linear algebra course, and once it is appreciated, we can deduce the possible existence of an *inverse matrix* for a square system, and more generally, of LU factorisations. In APL, though not by hand, it is also a very practical observation, since we can write a function (Figure 3) which produces the required multiplier for a particular reduction, and use it to actually do that reduction.

```
          ∇REDUCER[⎕]∇
 [0]  Z ← Piv REDUCER Mat ;D;E;N     ⍝ Piv is pivot location in matrix Mat
 [1]   N ← (⍴ Mat)[1]                ⍝ measures the number of rows in Mat
 [2]   Z ← (N,N) ⍴ 1,N⍴0             ⍝ creates an NxN identity matrix
 [3]   D ← Mat[ Piv[1] ; Piv[2] ]    ⍝ gives the value of the pivot element
 [4]   Z[;Piv[1]] ← - Mat[;Piv[2]]÷D ⍝ alters appropriate column of Z,except
 [5]   Z[Piv[1];Piv[1]] ← ÷D         ⍝ for the diagonal entry, which is 1÷D
 [6] ⍝ The result is the pre-multiplier for a Gauss-Jordan reduction of Mat.

     REDUCE : (Piv REDUCER Mat) +.× Mat  ⍝ Performs G-J reduction
```

Figure 3

This considerably simplifies the reduction process (Figure 4), and facilitates changes from one set of leading variables in a "solution" to another, thus opening the door to easy discussions of Linear Programming.

```
      EQNS
78.6  70.1  80.1  67.2  78.9   77    70.6   78.5  76.3  84.5  74.2
 6.5   9.4   8.8  13.7  16.1   30.4  32.8   37.1  21.3  8     19.9
 0.02  0.09  0.03  0.14  0.09   0.41  0.2    0.26  0.48  0.22  0.21
 0.27  0.34  0.3   1.29  0.71   0.86  1.22   0.59  0.82  0.71  0.67

      1 1 REDUCE EQNS
 1     0.89  1.02  0.85  1      0.98  0.9    1     0.97  1.08  0.94
 0     3.6   2.18  8.14  9.58  24.03 26.96  30.61 14.99  1.01 13.76
 0     0.07  0.01  0.12  0.07   0.39  0.18   0.24  0.46  0.2   0.19
 0     0.1   0.02  1.06  0.44   0.6   0.98   0.32  0.56  0.42  0.42

      ⎕ ← ECHELON ← 4 4 REDUCE 3 3 REDUCE 2 2 REDUCE 1 1 REDUCE EQNS
 1     0     0     0     ¯2.5   ¯6.2  ¯9.65 ¯12.13 ¯0.51  3.76 ¯3.42
 0     1     0     0     ¯0.04   5     0.05   2.1   7.05  3.08  2.1
 0     0     1     0      3.18   2.64  9.72  11.17 ¯4.69 ¯5.53  2.32
 0     0     0     1      0.34   0.03  0.69  ¯0.16 ¯0.02  0.24  0.14

      ⎕ ← MAT ← 2 8 REDUCE ECHELON
 1     5.76  0     0     ¯2.73  22.64 ¯9.35   0     40.1  21.52  8.68
 0     0.48  0     0     ¯0.02   2.38  0.02   1      3.35  1.47  1
 0    ¯5.31  1     0      3.39 ¯23.92  9.45   0    ¯42.1 ¯21.89 ¯8.82
 0     0.07  0     1      0.34   0.4   0.69   0      0.5   0.47  0.3
```

Figure 4

Interpreting the "solutions" shows that the leading variables are given as the sum of the basic solution and an arbitrary linear combination of the columns of the coefficient matrix associated with the parameters (Figure 5). The basic solution is the particular solution obtained when the parameters are set to zero. The general solution is seen to be the sum of a particular solution and an arbitrary linear combination of vectors each of which must satisfy the associated homogeneous equation. This is the structure characteristic of all linear equations, be they algebraic or differential.

```
        MAT
 1       5.76  0     0     ¯2.73  22.64 ¯9.35   0     40.1  21.52  8.68
 0       0.48  0     0     ¯0.02   2.38  0.02   1      3.35  1.47  1
 0      ¯5.31  1     0      3.39 ¯23.92  9.45   0    ¯42.1 ¯21.89 ¯8.82
 0       0.07  0     1      0.34   0.4   0.69   0      0.5   0.47  0.3

  i.e. writing the equations out in standard form:
X1 = 8.68 - 5.76 X2 + 2.73 X5 - 22.64 X6 + 9.35 X7 - 40.1  X9 - 21.52 X10
X8 = 1    - 0.48 X2 + 0.02 X5 -  2.38 X6 - 0.02 X7 -  3.35 X9 -  1.47 X10
  etc.
      UNKNOWNS   ← ⍳10
      LEADING    ← 1 3 4 8                       Identifying the leading
      PARAMETERS← UNKNOWNS ~ LEADING             and parameter variables
      RHS ← 11

      ⎕ ← BASIC ← (MAT[;LEADING]+.×LEADING),[1.5] MAT[;RHS]
 1              8.678975116
 8              0.9974369095                     Displaying a basic solution
 3             ¯8.816863837
 4              0.2970915451

      ⎕ ← NULL ← ⌹ MAT[;PARAMETERS]
  5.76221445       0.4752073928   ¯3.307316511      0.07443774415
     2779616      ¯0.0191098723    3.394720669      0.340591928     Basis
 22.63536639       2.378120786   ¯23.91983696       0.4041169768    vectors
 ¯9.349943815      0.02447145983   9.4502154        0.6937845338    for
 40.09878806       3.349135337   ¯42.09819498       0.5013921457    null
 21.5243152        1.465355154   ¯21.89393919       0.4667109407    space

      ⎕ ← PARS ← PARAMETERS,[1.5] PVALUES ← 7 6⍴10
 2  9
 5  3
 6  5          Attaching arbitrary values to the parameters
 7  6
 9  5
10  3
      ⎕ ← SOLUTION←((BASIC[;1]),[1.5]MAT[;RHS]-PVALUES+.×NULL)⍪PARS
   1           ¯357.1416214
   8            ¯36.40127485
   3            367.8355076
   4            ¯11.48500957               Presenting
   2              9                            a
   5              3                         complete
   6              5                         solution
   7              6
   9              5
  10              3
      ⎕ ← SOLUTION ← SOLUTION[⍋SOLUTION;]
   1           ¯357.1416214
   2              9                          Ordering
   3            367.8355076                    the
   4            ¯11.48500957                 variables
   5              3
   6              5                     (Note the new primitive
   7              6                       introduced for this
   8            ¯36.40127485                   purpose.)
   9              5
  10              3

      EQNS[;RHS] - EQNS[;UNKNOWNS] +.× SOLUTION[;2]
 0 0 0 0
```

Figure 5

The shift of emphasis to linear combinations of columns that occurs in the process of writing the general solution opens further avenues of theoretical investigation, but no further computational tools are required to discuss the ideas of linear dependence, spanning sets and bases for n-dimensional vector spaces, or of column, row and null spaces, and the rank of a matrix. The concept of *orthogonality* can be seen as an algebraic convenience in the discussion of bases, which can be given an insightful geometric interpretation later.

Essential APL

To get to this stage, which may be second year tertiary linear algebra, we have not needed (beyond the usual mathematical symbols and the *assignment arrow*) anything more than *shape, ravel,* (*roll* is convenient for generating arbitrary examples -- and it's fun), the *index* brackets, *transpose,* the *inner product* and some knowledge of how to define and edit functions. The latter should not be regarded as a purely technical matter, since a concentration on the nature of a *function* is not only an essential pre-requisite to calculus but indeed a central mathematical concept.

One of the very pleasing side benefits from introducing APL notation to students is the opportunity to revise and to re-examine elementary concepts in mathematics - concepts that are not always as well understood by students as they should be, but that cannot be broached at tertiary level without inviting the scorn of students who suspect they're being insulted. This applies to the concept of a logical proposition (as embodied in an equation or an inequality) and the function concept, for which APL notation is superbly suited.

The history of APL's origins makes it no surprise that the notation provides so many insights to mathematical concepts. It really is a joy to work with (if only I-APL had a full screen editor!), and one can only wish that every student had a laptop with APL in front of him in every class. For many students, it is the lack of easy access to a micro that prevents their use of the notation, and it is with continual use that the best results come.

In providing material for teachers, recalling the history of the spreadsheet may be instructive. A great deal can be done with just the assignment arrow and the four arithmetic operations. Every element of APL notation represents a significant operation and should not be introduced any more quickly than normal mathematical notation. It should also be remembered that one's own discoveries are the most satisfying. Students who are stimulated to solve problems using APL will ask the questions they need answered. There is no virtue in giving them one iota of information more than they need, and it is a wonderful feeling for a teacher when the students clamour for more.

Execution Time

by Alan Sykes

The aim of this short note is to discuss the 'execute' function ⍎ and to present some important ways it can be used with great effect.

What is ⍎?

Newcomers to programming sometimes have difficulty distinguishing between 'string variables' and 'numeric variables'. For example in APL, the definitions

```
      A←'1 2 3'
      B←1 2 3
```

are quite distinct. To see this note that ⍴B yields the answer 3 and ⍴A 5. This is because A has allocated to it five characters - '1', ' ', '2', ' ', '3'; in fact the APL expression A←'1',' ','2',' ','3' is equivalent. Further differences arise if we try to execute the commands

```
      1+B
2 3 4


      1+A
DOMAIN ERROR
1+A
 ^
```

The statement with A can be rectified by 'executing' A first.

```
      1+⍎A
2 3 4
```

showing that the string contents of A have been converted into their numeric equivalent.

The use of ⍎ however is more far reaching than this. Let's take a slightly different example.

```
      A←'B←1 2 3 4'
```

The string contents of *A* now contain an executable APL statement. So not surprisingly when *A* is executed, the statement in quotes is processed by the APL interpreter.

```
      B
1 2 3
      ⍎A
      B
1 2 3 4
```

At the very least this provides one with a mechanism for repeatedly executing a complicated APL statement without having to write an APL program - possibly an attractive proposition for teachers who would like their students to use programs as soon as possible but cannot afford the time to show them how to write programs in APL. (Of course direct definition is a much better approach in this situation.)

However there are a number of ways in which execute can be exploited with great effect, particularly in the area of making choices within a program. The following sections demonstrate some ideas on this topic.

Incidentally, if we wish to allocate a string variable to *B* in the above example, then we must use a repeated quote:

```
      A←'B←''1 2 3 4'''
      A
B←'1 2 3 4'
```

Choices

Suppose in a program, you have a variable *C* which can take the values `1,2`. Also suppose that if `C=1`, you wish to perform `TASK1`, whilst if `C=2`, you wish to perform `TASK2`. If `TASK1` and `TASK2` both require a right argument *X* say, then instead of branching on condition *C*, ⍎ may be used in the form:

```
      ⍎'TASK',(⍕C),' X'
```

This works because if *C*=1, then ⍕*C* (format *C*) becomes the character '1' and so the argument of ⍎ is the string '*TASK*1 *X*'. (Note the space to the left of '*X*' is crucial.)

A second possibility for choice in programming makes use of the fact that

```
      1/'THIS IS A STRING'
'THIS IS A STRING'
```

whereas

```
      0/'THIS IS A STRING'
```

What is happening here is that the Boolean reduction (0 1 1/'*CAT*' giving the string '*AT*') is being used with a scalar argument (1 or 0) which is then effectively replicated to the same size as the string - hence 1/.... gives the whole string, whilst 0/.... gives the null string.

Hence an alternative version of the previous choice of *TASK*1 and *TASK*2 can be written

```
      ⍎(C=1)/'TASK1 X'
      ⍎(C=2)/'TASK2 X'
```

This use of ⍎ is particularly useful in programming, conditionally to execute a statement or pass through to the next line, without the need to branch.

IF ... THEN ... ELSE

This is a familiar construct in many programming languages, but not in APL. But if you want it, because you feel it helps you to read your programs, then you can have it, at the expense of writing three small programs. The idea is to construct *IF*, *THEN*, *ELSE* so that we could write an APL statement such as

```
      IF 'C=1' THEN 'TASK1 X' ELSE 'TASK2 X'
```

First the program *ELSE* which is designed to combine the two executable statements into a two-row character matrix.

```
      ∇ R←LA ELSE RA;M
[1]   ⍝ RA is string vector or scalar APL command
[2]   ⍝ LA is string vector or scalar APL command
[3]   ⍝ R is a two row matrix containing both commands
[4]    M←(⍴LA←,LA)⌈⍴RA←,RA
[5]    R←(2,M)⍴(M↑LA),M↑RA
      ∇


      R←'X←1 2 3' THEN 'X←1 2 3 4'
      R
X←1 2 3
X←1 2 3 4
      ⍴R
2 9
```

Now *THEN* uses as its right argument the two-row matrix constructed by *ELSE*, and together with its left argument which when executed tells it which row to select, returns the command to be executed.

```
      ∇R←CONDITION THEN ACTIONMAT
[1]   R←ACTIONMAT[1+~⍎CONDITION;]
      ∇
```

And finally, a 'syntactic sweetener' *IF* which merely executes its right argument

```
      ∇IF STRING
[1]    ⍎STRING
      ∇
```

Hence piecing these together we can write, for example:

```
      C←1
      IF 'C=1' THEN 'X←1 2 3' ELSE 'X←1 2 3 4'
      X
1 2 3
      C←2
      IF 'C=1' THEN 'X←1 2 3' ELSE 'X←1 2 3 4'
      X
1 2 3 4
```

A MENU Program

Finally in this article, we use ⍎ to construct a *MENU* program which activates a set of different functions by the careful selection of an appropriate key.

First we define the keys to be used; for example assume that there are four tasks denoted by the keys A,B,C,D.

```
      KEYS←'ABCD'
```

Secondly, we wish to explain on the screen what the options are that are selected by these keys. So we might use the *BOX* program of a previous issue to construct a matrix called *EXPLAIN*.

```
      EXPLAIN←BOX 'Option 1/Option 2/Option 3/Option 4/'
      EXPLAIN
Option 1
Option 2
Option 3
Option 4
```

Thirdly we list in matrix form the action programs that require execution. (Each line of *ACTION* could be a complicated APL expression which requires execution when selected.)

```
      ACTION←BOX 'ACTION1/ACTION2/ACTION3/ACTION4/MENU/'
```

Then a *MENU* program might look something like the following.

```
      ∇MENU
[1]    'Choose one of the following using only the keys ',KEYS
[2]    KEYS,'-',' ',EXPLAIN
[3]    ⍎,ACTION[KEYS⍳1↑⍞;]
      ∇
```

Some readers may not be familiar with the use of the ⍳ symbol in this way (usually referred to as the 'dyadic iota' construct). It is basically a vector look-up table function. For example

```
      3 1 2 4⍳1
2
```

as the first occurrence of 1 in the left-hand side is in position 2. Similarly

```
      'BACD'⍳'A'
2
```

However, note what happens when there is no occurrence.

```
      'BACD'⍳'X'
5
```

What is returned is 1 more than the length of the look-up table.

Hence in our application, *KEYS*⍳1↑⍞ returns the value 1 2 3 4 according to the first key pressed equalling A,B,C,D and 5 if the first key pressed is not in *KEYS*. To cater for this case we have appended *MENU* to the end of the action list so that if an incorrect key is pressed, the menu is repeated.

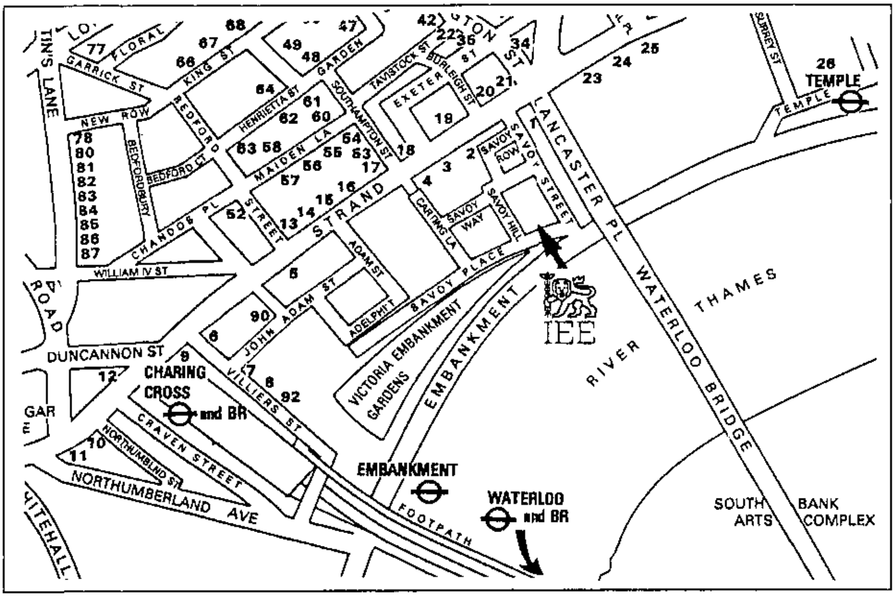
(Note also the reason for the comma after the execute symbol. This is because the argument of ⍎ must be a string scalar or vector. As the code is written, 1↑⍞ returns a vector of length 1 and hence the indexing of *ACTION* uses a vector, hence a one line matrix is created.)

As you can see, this basic *MENU* program is very concise, and a good demonstration of the usefulness of the execute function.

British APL Association Monthly Meetings

From February 1991, we have decided to change the venue of our monthly meetings in London. Meetings are now be held at the **Institute of Electrical Engineers**, Savoy Place, London (nearest tube outlets: Temple or Embankment). The committee believe that this move will greatly enhance the quality of our monthly meetings - but they are still **free**, and of course non-members are still most welcome. Come along and see if we've made the right decision.

Date	Venue	Event
26 April	IEE London	BAA Meeting: Vendor Forum
7 June	IEE London	**AGM** and BAA Meeting: Modern Productivity Aids for APL
20 September	IEE London	BAA Meeting
25 October	IEE London	BAA Meeting
22 November	IEE London	BAA Meeting



Renaissance Data Systems

Enlightenment Thru Information Processing

ALL APL BOOKS IN PRINT

Catalog excerpts as of September 1, 1990

ACCOUNTING STRUCTURED IN APL, Ijiri. 1984, 491p. $10.00
The basic principles of accounting with numerous APL expressions to model them.

COMPUTING IN STATISTICAL SCIENCE THRU APL, Anscombe. 1981, 426p. $42.00
Lots of statistics and APL. His workspaces are available from Yale University. A classic.

MATHEMATICAL EXPERIMENTS ON THE COMPUTER, Grenander. 1982, 525p. $64.00
Case studies using APL in a number of fields, including statistics, linear algebra, geometry, asymptotics, neural networks, invariant curves. Detailed description and analysis of APL functions in all topics discussed.

PROBABILITY IN APL, Alvord. 1984, 142p. $17.50
A delightful, yet serious development of 31 functions for fun and learning with roll, deal, binomial distributions, combinations, permutations, geometric distributions, and the World Series. Friendly examples. No previous APL required.
DISKETTE FOR ABOVE containing functions covered in book. $22.00

APL WITH A MATHEMATICAL ACCENT, Jones, Reiter. 1990, 200p. $41.95
Specially written for use in advanced high school or college math courses. By a student of Alvord's made good!

APL - THE LANGUAGE AND ITS ACTUARIAL APPLICATIONS, de Kerf, Goovaerts, Stiers. 1987, 223p. $89.00
Introduction to APL. Loss reserves, credibility, probability, numerical analysis, forecasting, with APL functions.

COMPUTATION FOR THE ANALYSIS OF DESIGNED EXPERIMENTS, Heiberger, 1989, 683p $59.95
Analysis of the construction of ANOVA programs using least squares techniques, including the parsing algorithms by which a language specification is interpreted. Emphasizes the geometry as well as the algebra of the methodology. All programs in the book are included on 5 1/4" diskettes in APL, BASIC, C, and FORTRAN.

APL2 AT A GLANCE, Brown, Pakin, Polivka. 1988, 444 p. $35.00
Solid, unintimidating, introduction to APL2. Clear illustrations, modern exercises in each chapter. Gets you started.

LEARNING APL: AN ARRAY PROCESSING LANGUAGE, Mason. 1985, 259p. $21.50
Emphasis on arrays, naturally. A readable, detailed introduction with thorough examples.

APL - ADVANCED TECHNIQUES AND UTILITIES, Bergquist. 1986, 450p. $44.95
Good discussion of alternative approaches to a wide range of programming tasks - efficiency, searching, dates, workspace documentation, file design, boolean techniques. Assumes knowledge of the basics. Many idioms
DISKETTE FOR ABOVE. $15.00
Contains all functions described in text. Formatted for APL*PLUS/PC, uploadable to Sharp APL, VS/APL, APL2.

APL AS A TOOL OF THOUGHT, PROFESSIONAL DEVELOPMENT SEMINARS, NY/SIGAPL; The first 5 years.
A wide range of topics in education and business. Logic, insurance, statistics, A. I., accounting, fractals, teachers toolbox, computer science, biology, graphics, engineering, data bases, and much more. 1983 - 1987, approximately 600p. $55.00

APL89 - STATISTICS TUTORIAL, Alvord, Traberman, et al, 72p. $10.00
A unique collection of papers on statistics and its exposition in APL.

Name: ____________________ Date: __________

Street: ____________________

____

City: __________ State: ______ Postal Code: ______

Payment in check or money order payable in $U. S. must accompany order. Send to:
Renaissance Data Systems
P. O. Box 20023
Park West Finance Station
New York, New York 10025-1510
(212)864-3078

Country __________ Telephone Day: __________ Evening: __________

We regret that we cannot accept purchase orders or credit cards.

Title ____________________ **Price** ______ **Quantity** ____ **Total** ______

Title ____________________ **Price** ______ **Quantity** ____ **Total** ______

Title ____________________ **Price** ______ **Quantity** ____ **Total** ______

Subtotal: ______

Postage and handling - U.S. and Canada ($2.50 minimum; $2.50 each item over $25.00): ______
- International (surface mail add 15%, airmail add 40%): ______

Total: ______

Thank You For Your Order!

Please send complete CATALOG of APL books and software ☐

The Random Vector

The Newsletter of the APL Statistics Library
Editor David Eastwood

April 1991

Contents

Editorial

In this second issue of 'Random Vector' Tony O'Hagan continues to explore the genesis of the ASL project, seeking to prove that if no one present at a meeting can remember what was said, he who writes something down creates the history! During the period in question (late 1988 to date) I have been a member of both the BAA Committee and the ASL Management Committee and so I can add some comments regarding the feelings of the BAA Committee when we were first officially approached for funding. I think it is fair to say that there was an immediate and enthusiastic acceptance of the ASL project within the BAA - we felt that ASL combined a number of themes dear to our hearts:

- ASL represents a means of demonstrating the appeal of APL in a key area of current usage.
- Within that area, ASL offers a route to enhance APL's level of acceptance by supplying the basic, tested, algorithms that new users have a right to expect.
- The proposal seemed unlikely to receive any commercial funding.
- The project proposals seemed to be realistic and not over ambitious

The initial queries raised by the BAA were largely concerned with points of detail within the various proposals put forward by the ASL team.

In the first issue of Random Vector we gave a brief overview of the contents of the 'Bottom Shelf' ASL functions. By the time this issue is delivered the official launch of ASL will have taken place at the March 1991 meeting of the BAA. In the run-up to this launch, a meeting was held on February 12 1991 at University College Swansea. Maurice Jordan and Jake Ansell braved the uncertainties of British Rail and its attitude to snow. Norman Thomson was unable to be present but communicated by telephone. Also present were Alan Hawkes, Alan Mayer and Alan Sykes from University College.

A profitable day was spent reviewing the mountain of paper produced by Maurice's 'APL test bed' as applied to the functions on the bottom shelf. This largely automatic approach to APL code testing was discussed by Maurice at the second ASL conference at Swansea in September 1990. Also discussed were the comments that had been received from testers who had attended that September meeting. Alan Mayer presented his work on producing accurate tail probabilities and quantiles for Normal, t, F and Chi-Square, which have subsequently been tested by Maurice.

Norman Thomson has been maintaining the specification document for the Bottom Shelf and has been incorporating the results of discussions such as this in the document. I shall publish as much of Norman's document as space permits in a future issue of Random Vector.

This issue, however, takes a look at the Regression Shelf in some detail. We reproduce some notes written by Alan Sykes which were issued with the first test versions of the Regression Shelf. Although, as Alan says in his article, these notes are now being revised, they do offer an insight into the scope of the Regression Shelf.

The last article included is not so directly related to the ASL theme, but David Appleton does offer an interesting view into the way in which the APL representation of a particular mathematical formula can be progressively refined. As with any article which dares to introduce the theme of coding techniques and style, I expect a lively reaction from readers, so, having lit the blue touch paper, I shall retire and await the postbag with interest.

The 3rd ASL Conference

I would like to conclude this introduction by making a preliminary announcement about the third ASL conference. This will be held at:

The University of Wales Conference Centre at Gregynog
from Monday 30 September 1991
till Wednesday 2 October 1991

As with previous ASL conferences, this is an invitational conference and the only costs for delegates are their travel costs. We are hoping to review the status of ASL and to chart out the next phase of development for the project. In particular we are hoping to attract more volunteers to work on the project!! The ASL team will be delighted to hear from anyone who would like to attend - we are trying to continue the mix of Statisticians from Higher Education and the state or private sector as well as theoretically professional APL programmers. If you would like to find out more about the next conference, why not contact one of the ASL Management Committee who are:

Chairman: Tony O'Hagan, Department of Mathematics,
University of Nottingham, University Park,
Nottingham NG7 2RD. Telephone: 0602-484848 x2800

The Genesis of ASL (2): October 1988

by Tony O'Hagan

First Approaches to the BAA

The very earliest beginnings of the ASL idea were chronicled in the first article in this series. The next key event was a meeting on 5th October 1988 at IBM South Bank, London. Present were Jake Ansell, David Eastwood, Norman Thomson and myself. This meeting drew up the following list of goals as part of "establishing an APL service to statistics users".

1. Define standard functions for basic statistical operations. Identify good implementations, possibly by soliciting code from the statistics/APL community.

2. Identify standards for delivering functions/workspaces to different APL systems.

3. Draft standards for more complex functions, their syntax and means of verification.

4. Create a library for statistical functions, comprising the standard basic functions and others submitted by users.

5. Prepare housekeeping procedures/functions to administer this library. These should cover receipt, verification and storage of new functions/workspaces, plus selection, packaging and delivery of requested functions/workspaces.

6. Document functions in book form, possibly loose-leaf.

Several ways of achieving these goals were discussed, but the favourite was for a student to do it as a one-year MSc project. I agreed to look into the possibility. Finally, the meeting considered ways of funding the project. The idea of an MSc student was attractive because it would be cheap, but still substantial money would be involved. It was here that involvement of the British APL Association was first suggested. Another possibility was funding by IBM.

My investigations concerning the MSc idea were not very successful, for similar reasons as caused us to drop the thought of a PhD project in our earlier discussions. But David Eastwood made contact with the BAA on our behalf at a BAA committee meeting in November. They expressed an interest in principle in providing support for our project. In particular, they suggested that the ball could be set rolling by having a conference on 'APL and Statistics', to establish demand and collect ideas. This very positive response to our tentative enquiries was extremely encouraging. For the first time, it looked like the 'statistics library' might be in business!

I have no record of what happened in the next couple of months. I believe that we were asked to clarify our thinking, but no written proposal seems to have been made then, nor any meeting held. The BAA committee met again in January 1989. Alan Sykes, who was the BAA Education Officer at the time, wrote me a letter dated 18th January reporting the committee's conclusions. A substantial part of Alan's Education budget for the year was to be given to the proposed conference. He had already made a booking for it at the University of Wales Conference Centre at Gregynog for that September. Furthermore, a firm proposal for funding of the project itself would be welcomed.

Another meeting took place on 24th February, again at IBM South Bank. This was publicised by Jake Ansell through the APL Statistics Users' Group, and attracted several interested people from industry. In all, the attendance was Dick Bowman, Mike Day, David Eastwood, Peter Lane, A MacGillivray, Robin Morphet, Alan Sykes, Norman Thomson and myself. A great deal was discussed. Dick Bowman suggested that Sig APL might also contribute to the ASL project. Dick later followed this up without success. It seems that the American group are very wary of supporting an initiative which might later compete ('unfairly' is implied) with commercial products.

The main achievement of this meeting was to establish two interim committees. A Conference Committee, chaired by Alan Sykes, would meet to plan the conference in September. A Proposal Committee, with me in the chair, was charged with drawing up detailed proposals for funding of the project proper, for submission to Sig APL and the BAA.

Another interesting suggestion made at this meeting was that in due course the Royal Statistical Society might be prevailed upon to put some kind of seal of approval on the functions produced. Although this has not been pursued any further, we still hope to do so when sufficiently battle-hardened software is ready. The meeting ended with a discussion of a name for the library, and for the project. S-APL was tentatively agreed.

My records are hazy again here. It is clear that I prepared a first draft proposal and sent it to members of the Proposal Committee only four days later, on 28th February. It seems likely that when the meeting on the 24th ended it metamorphosed into a Proposal Committee meeting, which went on to discuss detailed proposals at some length. The text of that original draft is no longer with me, either. It went through a series of modifications, including further feedback from the BAA in April, and emerged on 5th June 1989 as a substantial five-page document. The name had changed to ASL = APL Statistics Library (because S-APL was already used by I P Sharp), and that name has stuck. Since this document defines ASL in basically the form which BAA agreed formally to fund, it is worth setting down here the essential features.

1. The objectives were set out as "to establish:

   (a) a set of standards which will give coherence to the assembly of functions, make them easy to use singly and in combination, facilitate porting to a range of interpreters and hardware, and help with verifying their correctness;

   (b) a 'bottom shelf' of functions comprising the most fundamental statistical routines (descriptive statistics, calculations on standard distributions, etc.), plus utility functions (data handling and I/O, graphics, numerical procedures, etc.);

   (c) various other 'shelves' of functions for specific statistical applications, such as regression (linear models), categorical data analysis, multivariate statistics, reliability, analysis of variance, sample survey analysis, time series and forecasting, all taking functions from the bottom shelf, and possibly other shelves (e.g. a shelf for generalised linear models may take functions from the regression shelf)."

2. The benefits were described as follows. "One of the best-selling categories of software is statistics packages... We want ASL to be a flagship for APL in this important market." But it was stressed that ASL would not be just another statistics package. "... it will not be a closed package, allowing the user to do only what the programmers have thought to include. The full power of APL will always be available, ASL merely providing a sophisticated set of functions, which the user will then combine with each other, the APL primitives and his/her own functions. For the lay user, the documentation will provide idioms for operations such as selection of variables and cases, data transformations and passing data between functions. To such a user ASL will appear as a complete package, as general and user-friendly as conventional packages. To the APL user it will be much more."

3. It was also boldly claimed that "APL is the best language for doing statistics. Data analysis is essentially exploratory. It is not a simple matter of applying a single standard analysis to the data, but of trying different kinds of analysis, looking at the data in different ways, and drawing together a range of conclusions. The APL environment is ideal for this activity. The APL language also uniquely supports the processing of arrays of data neatly and simply."

4. Management of ASL was to be by a Management Committee, to be elected at the September conference. In addition to a Chairman, Manager (responsible for day-to-day running of the ASL effort), Treasurer and Secretary, it was hoped to have BAA involvement through its Projects, Technical and Education Officers.

5. A programme of work was described, culminating with a general release of the first 'shelves' in September 1990. These were to comprise a bottom shelf as in 1(b) above and a regression shelf. Although both first and second generation APLs were to be supported, 'APL2' implementation was not the first priority.

6. Funding was requested for a programmer, working for about 2 days per week for a year, for a PC for the programmer, and a small amount for expenses and consumables. The BAA was asked to commit money on a phased basis, payment at the second phase being subject to satisfactory progress in the first phase (to March 1990).

Since then, there have been minor changes in the formal objectives, the management and the timescale, which will be described in later episodes of this series ... (watch this space!), but the ideals of (2) and (3) have kept ASL going ever since.

The ASL Regression Shelf

by Alan Sykes

Introduction

The ASL project outlined initial 'shelves' to be produced in the first year. One of these was a Regression Shelf. An exploratory workspace was produced and discussed by the ASL committee. That workspace built on the work of Jake Ansell and Alan Sykes over recent years in using APL routines that behaved similarly to those implemented in the Royal Statistical Society's Software Package 'GLIM'. It was decided that the Regression Shelf should contain a set of functions that performed GLIM-type calculations in a way that allowed them to be incorporated into user functions.

Introduction to Regression

Most users of Statistics, or for that matter users of APL, understand the basic principle of 'Simple Linear Regression' where a response variable's average value depends on a predictor variable in a linear way. The equation of that line is estimated from bi-variate data by the method of least-squares, which in APL means the use of the dyadic version of 'domino'.

In this introduction, some basic terminology is explained and the simple linear regression model is extended into the concept of a statistical linear model.

There are three major attributes of a statistical linear model. First we have the RESPONSE variable, often referred to as the 'y-variable'. This is usually random, with distribution assumed to be NORMAL or GAUSSIAN. Secondly, we have one or more PREDICTOR variables, usually regarded as non-random, which predict the average value of the response variable in a linear way. If E(Y) denotes the 'expected' value or mean value of the response variable Y, and U,V,..W are predictor variables then we assume that

```
    E(Y) = a+bU+cV+....dW ,
```

where a,b,...d are unknown constants.

So much for the mean of Y; what about its variance? The simplest assumption to make is that the variance is constant, given by k say. Hence y has a Normal Distribution with mean a+bU+cU+...+dW and variance k. Regression Analysis concerns itself with the statistical procedures appropriate to estimating the unknown linear regression parameters a,b, ...d, together with the nuisance/scale parameter k. As such it is merely one example of the basic paradigm

```
Observation = Signal + Noise
```

Given a collection of values of the response variable, and the associated values of the predictor variables, the machinery of statistical linear models concerns itself with:

(a) Choosing appropriate models;

(b) Validating a chosen model;

(c) Estimating the parameters;

(d) Using the model for prediction.

Hence, functions are required to enable the user to:

(a) Specify a model;

(b) Estimate the parameters of the model;

(c) Test hypotheses concerning that model;

(d) Perform diagnostic tests and graphical plots to investigate the validity and appropriateness of the model.

Generalized Linear Models

Over the last twenty years the success of statistical linear models has been strengthened by relaxing two of their basic assumptions. First the assumption that the response variable has a Normal distribution is relaxed. Alternatives (from the 'Exponential family') such as Poisson, Binomial, and Gamma distributions are allowed extending the scope considerably. For example if Y is a count of radioactive particles emitted from a source in a time period of size x, then, rather than assuming that y had a Normal distribution with mean a+bx, it is likely to be more appropriate (since the y values are counts) to assume that Y has a Poisson distribution with mean a+bx - hence the idea of 'Poisson Regression'.

The second assumption that is relaxed is the idea that the 'linear predictor', i.e. the function a+bu+cv+..dw, need not specify the expected response E(Y). Rather, the linear predictor specifies a function of the expected response. For example in Poisson regression we may assume that a+bU+cV+..dW = log(E(Y)). This concept of a 'Link Function' extends the scope even further. Generalized Linear Modelling is now an accepted part of an applied statistician's armoury of tools. It is not surprising that the success of the package GLIM has resulted in the inclusion of such tools in many other statistical packages, such as SAS, GENSTAT and now ASL!

The Facilities Within the Regression Shelf

The first, draft, version of the Regression Shelf was distributed to interested parties during and after the second ASL conference in October 1990. Feedback gained as a result of that exercise has resulted in some modifications to the shelf which I shall detail below.

In the remainder of this article, I shall introduce the facilities offered in the Regression Shelf and in a future article I shall discuss the operation of the functions supplied within the shelf in more detail.

A Database of Examples

In order that the Regression Shelf may be tried and tested, a number of databases have been provided in a separate workspace. Each database is stored under a variable name, which specifies a character matrix with a simple but specific format:

1 Introductory Lines, specifying appropriate references

2 Blank Line

3 Variable Names; one per line with at least one space between the name and the (tabulated) start of the explanation of the variable names

4 Blank line

5 Lines of Data

As an example consider the data set *HOUSEPRICES*

```
      HOUSEPRICES
HOUSE PRICE DATA, SOURCE: Genstat 5 An Introduction, by Lane,
Galwey and Alvey, Page 55. 20 similar houses for sale in
Harpenden in 1977
```

```
PRICE  House price in pounds sterling
SPACE  Total floor space of the house in square-metres
GARDEN Area of the garden in square-metres
AGE    Age of the house in years

11500 131 140 88
15500 154 245 70
12950 137 150 66
14000 121 180 43
16500 135 260 17
17600 172 400 23
12450 112  90 52
15500 124 120 10
14900 141 180 43
16250 149 350 36
18400 170 320  7
11950  93 350 19
10400 111 280 62
17250 162 380 12
13450 148 190 23
10950 128 160 75
13950 152  95 92
11500 101 450 42
17500 145 275  0
```

Functions are supplied to allow you access to the explanation, the names and the data. A function is also provided to extract the data into numerical form attached to the names provided.

```
      EXTRACT HOUSEPRICES
PRICE +SPACE +GARDEN +AGE
```

(N.B. the explicit result of using *EXTRACT* is a string variable containing the variable names, separated by plus signs - this is useful for feeding into the regression routines. But in addition of course, the variables *PRICE, SPACE, GARDEN* and *AGE* are now in the workspace.)

An Introductory Example

A simple example of the basic regression directives using the database on *HOUSEPRICES* as described above gives a flavour of the Regression Shelf.

```
      EXTRACT HOUSEPRICES
PRICE +SPACE +GARDEN +AGE
```

The aim of the use of regression here is to attempt to explain the variation in house prices in terms of the information about the houses contained in the variables *SPACE, GARDEN* and *AGE.* Intuitively, a large house with a large garden will command a higher price than a smaller house. As for age, perhaps its influence on house price is debatable.

Since the object is to predict house prices, the variable *PRICE,* or some transformation of it must be declared to be the 'y- variable'. This is done using the directive *YVAR.*

```
      YVAR 'PRICE/1000'
  Y variable is PRICE/1000
  units set to 19
```

To predict this variable, we have the choice of using all or some of the variables *SPACE, GARDEN, AGE,* together with any variables that can be constructed using them, such as for example *SPACE×SPACE, SPACE×GARDEN, ÷AGE,* etcetera. In addition, we usually require a constant term, which is denoted by *'GM'*. Let's try using just the variable *SPACE,* and of course a constant. Predicting *PRICE* using *'GM+SPACE'* is then equivalent to finding the regression line of *PRICE* on *SPACE.*

The linear predictor is specified by using the directive LP.

```
  LP 'GM+SPACE'
```

To fit, follow this with the directive *FIT.*

```
      FIT

 Response Variable is PRICE/1000
 Fitted Model is GM+SPACE

 source                  ss      df             ms
 ------                  --      --             --
 Due to model        66.223       1         66.223
 Residual            46.083      17          2.711
 -----------------------------------------------
 Total Corrected    112.306      18          6.239

 Percentage Variation Accounted for =    58.97
 F-statistic =                           24.43
 p-value=                                  .00
```

This table presents a summary of the ability of the chosen linear predictor to predict *PRICE*÷1000. It takes the form of an 'ANOVA' table which breaks down the variation in the 'y-variable' (in this case 112.306), into that explained by the model (66.223) and the residual variation of the data about the fitted regression line (46.083). Hence we can see that using *SPACE* as a predictor variable explains 100×66.223÷112.306 or 58.97% of the variation.

This summary does not however give the estimated coefficients of the predictor variables (*GM* and *SPACE*). The directive *DISPLAY* does.

```
      DISPLAY

  Var            Est          Std Err  t-stat  p-val
  ---            ---          -------  ------  -----
  GM           2.576            2.410     1.1   .300
  SPACE         .086             .017     4.9   .000
```

Now it should be clear from this simple one-variable model how to fit more complicated models. Let's use all the supplied variables, for example.

```
      LP 'GM+SPACE+GARDEN+AGE'
      FIT

  Response Variable is PRICE/1000
  Fitted Model is GM+SPACE+GARDEN+AGE

  source                   ss        df             ms
  ------                   --        --             --
  Due to model         96.986         3         32.329
  Residual             15.321        15          1.021
  ------------------------------------------------------
  Total Corrected     112.306        18          6.239

  Percentage Variation Accounted for *  86.36
  F-statistic *                         31.65
  p-value*                                .00


      DISPLAY
  Var              Est            Std Err  t-stat  p-val
  ---              ---            -------  ------  -----
  GM             6.079              1.769     3.4   .004
  SPACE           .077               .011     7.0   .000
  GARDEN        -.001                .003     -.3   .751
  AGE           -.048                .010    -4.9   .000
```

From these results, we can see that with the additional variables, the total prediction capability is 86.36%. Note from the estimates, that the coefficient of *AGE* is negative (older houses tend to be sold for less than younger houses, other things being equal), and that the coefficient of *GARDEN* is less than its standard error, indicating that its predictive capabilities in the presence of the other variables is very small.

Fitting a model such as the above is fraught with problems. There are so many assumptions involved which should be checked. There is a vast array of suggested techniques for doing this and the regression shelf includes a number of the most popular regression diagnostics as well as a repertoire of functions to run and test more complex models. In the next article on the Regression Shelf I shall illustrate the full scope of the code supplied in the Regression Shelf.

Since January 1991, I have been working on a complete revision of the Regression Shelf. The basic weighted least-squares routine has been changed from one based on Householder's method to a version of Beaton's method.

The criticism raised by early testers that the Shelf was 'too packaged' has been considered seriously. However it does seem that it is difficult to provide useful functions including output without a good deal of packaging. So, in the revision, we have tried to reach a compromise - the package is there, but it is more modular, with core functions, such as the weighted-least-squares routine available to the user directly. With the aid of good documentation, a user has a choice of using the Regression package or choosing core routines for his own use.

The numerical computations have now been checked by reference to published examples in the GLIM manual and in Aitken's book on Generalised Linear Modelling. Additional functions have been added including Summary Statistics, correlation matrices (both for estimates and regression variables), and Box-Cox loglikelihood for power transformation.

Sterling's Function: a Case Study

by D R Appleton

Theme

Several different APL functions, for the evaluation of Stirling numbers of the second kind, are presented. Their advantages and disadvantages are discussed.

Exposition

A colleague doing some theoretical statistical work on the analysis of an experiment which had resulted in counts from the truncated Poisson distribution being observed, recently asked me if I had an APL function to evaluate the Stirling numbers of the second kind. I did not have such a function, but he told me they were defined for positive integers n and t by

a formula obtainable from Abramowitz and Stegun (1964). With APL it is easy to turn this expression into executable form, for example:

```
    < STIR1 >
```

However this function can be improved: it ignores the fact that the first term is identically zero, it does not make it clear that the result is always positive, and APL has a more concise way of summing alternating series. A better function is therefore given by

```
    < STIR2 >
```

It may or may not be numerically preferable to factorise out the n! and use

```
    < STIR3 >
```

Of course APL's power comes from its ability to handle several different values simultaneously. We could easily have a vector for *T*.

```
    < STIR4 >
```

However, it is likely that the values of T we would require, at least to form a table of the Stirling numbers (which might be useful in the long run to save calculation) would be all the positive integers up to T. This leads to

```
< STIR5 >
```

and we could have, had we wished, developed *STIR*2 to this form instead of *STIR*3.

If we wish to produce and save a table of the numbers we may build *STIR*5 into a loop:

```
<


STIR6


 >
```

It is now time to look at the table of numbers this function produces; this is shown in Table 1 for N=6 and T=6. It becomes very clear that a function which wastes its time calculating values for $N>T$ is unsatisfactory. This, of course, should have become evident if a little more algebra had been done before turning to programming, or if *STIR*2, say, had been properly tested, but the functions are so simple it hardly seemed necessary to test them all individually!

Table 1

		t					
		1	2	3	4	5	6
	1	1	1	1	1	1	1
	2	0	1	3	7	15	31
n	3	0	0	1	6	25	90
	4	0	0	0	1	10	61
	5	0	0	0	0	1	15
	6	0	0	0	0	0	1

Development

It is important to define more coherently the problem we are tackling. We now want to produce a table of Stirling numbers in which the element at row n and column p contains σ^{n}_{n+p}, and to solve the practical problem underlying the computing problem we would like n to take values up to at least 100 and p to reach at least 2n for as many values of n as possible. This means rewriting *STIR*5 as

```
< STIR7 >
```

Running this in a loop produces Table 2 for N=8 and P=6. The magnitude of the values appearing in the table gives rise to concern for the function: when will it overflow? This is certainly a matter which will have to be investigated, but there are other interesting aspects to the table.

Table 2

					p		
		1	2	3	4	5	6
	1	1	1	1	1	1	1
	2	3	7	15	31	63	127
	3	6	25	90	301	966	3025
n	4	10	65	350	1701	7770	34105
	5	15	140	1050	6951	42525	246738
	6	21	266	2646	22827	179487	1323652
	7	28	462	5880	63987	627396	5715424
	8	36	750	11880	159027	1899612	20912320

From row 2 it is clear that $\sigma^2_{t+2} = 2\ \sigma^2_{t+1} + 1$, and from row 3 it is not too difficult to deduce that $\sigma^3_{t+3} = 3\ \sigma^3_{t+2} + \sigma^2_{t+2}$, and hence to discover the recurrence relation

$$\sigma^n_t = n\ \sigma^n_t + \sigma^{n-1}_t$$

This enables us to write a function without calculating the factorials or binomial coefficients in our original definition.

```
<

 STIR8


 >
```

But what is this? An APL function with a double loop? Surely that is unnecessary. Instead of working a row at a time we must evaluate the table by columns.

```
<

 STIR9

 >
```

Now we have a much faster function (taking about a quarter of the time *STIR*8 does for *N*=80 and *P*=80), which is approaching its final stage, but let us look more closely at the table and the function we have written. Column 1 is just `+\⍳N` and column 2 is therefore `+\(⍳N)×+\⍳N`. Indeed column *P* is `⍎1↓(7×P)⍴'×+\(⍳N)'` an expression so concise it is hardly worth writing a function for. Notice that `⍎1↓(9×P)⍴'×+\(⍳N)'` prints out the entire table given by *STIR*9, although it transposes it and prints each line to a different format. This is avoidable, for small values of *N* and *P*, by `⍎2↓((12×P)⍴'×⍎⎕←10 0⍕+\K'),'←⍳N'` but we wish to store the table, not print it, a task which is done by our final version

```
< STIR >
```

which is slightly faster even than *STIR*9.

Recapitulation

We have seen how it is worthwhile to look at the problem from a theoretical point of view so that we may obtain different algorithms for calculation. Even with an apparently obvious formula, simply translated into APL, it may be more appropriate to use a recurrence relation. This is not only for reasons of time, which are not of vital importance if the values obtained are to be stored in any case, but for reasons of numerical accuracy. In passing we suggested that *STIR*2, which involves the calculation of a factorial and a set of binomial coefficients, might have different numerical properties from *STIR*3 which involves two sets of factorials.

To investigate this suppose *N*=50 and *T*=60, the sort of numbers which might indeed arise as the result of a small experiment. *STIR*1 and *STIR*2 give σ^{50}_{60} = 2.02×10^{25} while *STIR*3 and *STIR*4 give 5.06×10^{25}. The sensitivity of each function to the order of summing the series can be looked at by changing `-/` in *STIR*2 and *STIR*3 to `-/⌽`; the results change to 1.71×10^{26} and 7.09×10^{25} respectively. With *STIR*4 we can even alter the calculated value of *S* depending on which values other than 60 we include in vector *T*. *STIR* (and *STIR*9 which uses the same arithmetic) gives 9.55×10^{24} for σ^{50}_{60}, which I believe to be correct. Not only is the algorithm contained in `⍎1↓(7×P)⍴'×+\(⍳N)'` remarkably succinct, it is numerically preferable to the others. *STIR* works whenever `350>N+2*P` and occasionally outside that range. When the function fails it is sometimes because the workspace becomes full, and sometimes because a domain error occurs. The figure shows an isometric plot of the common logarithm of the Sterling numbers of the second kind, for values of n and t-n up to 80. The values given above, except for that of 9.55×10^{24} , are system-dependent; the ones quoted were obtained from IBM APL on a Personal Computer, the same phenomenon occurs with different values using TRYAPL2. Other systems give a domain error instead of inaccurate results.

Coda

The style of programming illustrated in *STIR* can be utilised in many instances of recursion, though the Stirling numbers of the second kind may be the most elegant. Readers might like to deduce what the following expressions evaluate.

```
< TXT8 >
< TXT9 >
< TXT10 >
```

Reference

[1] Abramowitz, M. & Stegun, IA (eds). *Handbook of mathematical functions with formulas, graphs, and mathematical tables.* Washington: US Govt. Print. Off. 1964.

APL Product Guide

Compiled by Alison Chatterton

VECTOR's exclusive APL 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 do depend on the alacrity of suppliers to keep us informed about their products so that we can update the Guide for each issue of VECTOR. Any suppliers who are not included in the Guide should contact me to get their free entry - see address below.

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. Where no UK distributor has yet been appointed, the vendor should indicate whether this is imminent or whether approaches for representation by existing companies are welcomed.

For convenience to readers, the product list has been divided into the following groups:

* Complete APL Systems (Hardware & Software)
* APL Timesharing Services
* APL Interpreters
* APL Visual Display Units
* APL character set printers
* APL-based packages
* APL Consultancy
* APL Training Courses
* Other services
* Vendor addresses

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.

Note: 'poa' indicates 'price on application'.

All contributions to the APL Product Guide should be sent to Alison Chatterton, at the address on the inside back cover.

COMPLETE APL SYSTEMS

COMPANY	PRODUCT	PRICES(£)	DETAILS
Active Workspace Ltd			
	AWL486	4,450	486 based 25MHz PC, 140MB Disk, 4MB RAM, VGA Colour. (Inc. 1 year on site maint.)
	AWL 386	3,095+	386 based 25 & 33MHz PC, 140MB Disk, 4MB RAM, VGA Colour. (Inc. 1 year on site maint.)
APL People	IBM PCs & compatibles	poa	Includes PC, mono/colour monitor, APL interpreter, operating system software, plus optional printers, graphics boards, additional memory etc.
Dyadic	IBM RS/6000 MD320	11,736	APL POWERstation (Greyscale) 27.5 MIPS, 7.4 Mflops RISC Processor 8Mb RAM, 120Mb Disk 19" 1280x1024 Greyscale Graph Display AIX, OSF Motif, Dyalog APL (1-user)
	IBM RS/6000 MD320	13,817	APL POWERstation (Colour) 27.5 MIPS, 7.4 Mflops RISC Processor 8Mb RAM, 120Mb Disk 16" 1280x1024 Colour Graphics Display AIX, OSF Motif, Dyalog APL (1-user)
	IBM RS/6000 MD320	22,658	Advanced APL POWERstation 27.5 MIPS, 7.4 Mflops RISC Processor 16Mb RAM, 320Mb Disk, 150Mb Tape 16" 1280x1024 Colour Graphics Display AIX, OSF Motif, Dyalog APL (1-user)
	IBM RS/6000 MD520	37,114	APL POWERsystem (8-users) 27.5 MIPS, 7.4 Mflops RISC Processor 16Mb RAM, 320Mb Disk, 150Mb Tape CD-ROM Drive, 16 Ports AIX, Dyalog APL (2-8 user licence)
	IBM RS/6000 MD530	72,054	APL POWERsystem (16-users) 34.5 MIPS, 10.9 Mflops RISC Processor 32Mb RAM, 1.34Gb Disk, 2.3Gb Tape CD-ROM Drive, 16 Ports AIX, Dyalog APL (8+ user licence)
	IBM RS/6000 MD540	122,842	APL POWERsystem (32-users) 41 MIPS, 13 Mflops RISC Processor 64Mb RAM, 1.7Gb Disk, 2.3Gb Tape CD-ROM Drive, 32 Ports AIX, Dyalog APL (8+ user licence)
MicroAPL	Aurora	20,000+	Multi-user APL computer using 68020 CPU. Std. configuration 2Mb RAM, 16 RS232 ports, 68 Mb hard disc, 720K diskette
	Spectrum	7,000+	Expandable multi-user APL computer using Motorola 68000. Std. conguration 1 Mb RAM, 12/36 Mb disc, 12 ports.
M.T.I.C.	386sx AT	2600+	Entry level 32-bit compact AT with DYALOG APL. 2MB ram, VGA, 40MB hard disk, 1.44MB floppy disk, 102-key keyboard, monitor, mouse,MS-DOS 4.01, WORKS and WINDOWS-386. Many expansion options available.

APL TIMESHARING SERVICES

COMPANY	PRODUCT	PRICES(£)	DETAILS
REUTER:FILE	SHARP APL	poa	International Network application systems and public databases.
Uniware	APL*PLUS	call	STSC's mainframe service

APL INTERPRETERS

COMPANY	PRODUCT	PRICES(£)	DETAILS
Active Workspace Ltd	DYALOG APL DOS 386		Dyadics PC 386 APL interpreter
APL Software	APL*Plus/PC Release 9	535	STSC's APL for IBM PC, PC/AT and PS2. Upgrades from earlier Releases also available.
	Run-time	poa	Closed version of APL*Plus/PC which prevents user exposure to APL.

	APL*Plus II	1600	Incorporates mainframe features & performance in a version of APL for the PC
	Run-time	poa	
	Dyalog APL	1000-10,000	2nd generation APL for Unix systems
	APL2/PC	376	IBM's APL 2 for the PC.
Cocking/Drury	APL*PLUS PC Rel 10	477	STSC's full featured APL for IBM's and compatibles - Version 10 includes the quad-NA facility to interface to non-APL software, support for MS Windows and mouse devices. The User-command processor has been built in to the interpreter. Upgrades to version 10 are available from Version 9 and earlier releases.
	APL*PLUS PC Run-Time	poa	Closed version of the interpreter for developers, prevents user exposure to APL.
	APL*PLUS II System	1600	High powered APL interpreter for the 80386 chip. Price includes one years maintenance and free upgrades - volume discounts VERSION 3 NOW AVAILABLE.
	APL*PLUS II Developer System	poa	All the features of the APL*PLUS II System plus unlimited free run-times to enable developers to distribute their applications.
	APL*PLUS UNX	poa	STSC's 2nd generation APL for all major Unix computers and workstations. Version 4 for existing platforms and for the IBM RS/6000 available from January.
	APL*PLUS VMS	poa	2nd generation APL for DEC VAX computers running under VMS.
	APL*PLUS Mainframe	poa	Enhances VS APL with many high performance, high productivity features. For VM/CMS and MVS/TSO offers simple upgrade from VS APL.
Dyadic	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 δSM Screen Manager. Requires 386/486 based PC or PS/2, at least 2Mb RAM, EGA or VGA, DOS 3.3 or later.
	Dyalog APL for Unix Systems	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.
I-APL Ltd	I-APL/PC or RML Nimbus	4.50	ISO conforming interpreter. Supplied only with manual. (see 'Other Products' for accompanying books)
	I-APL/BBC	4.50	As above
	I-APL/Archimedes	4.50	As above
IBM UK	IBM PC APL2	348	APL2 for the IBM PC. Program 5799-PGG/1, PRPQ number RJ-0411. From all IBM dealers.
MicroAPL	APL.68000 Level I	2000	First generation APL with numerous enhancements. Multi-user version (Unix, Mirage, MCS).
	APL.68000 Level II	2500	Second generation APL. Nested arrays, user defined operators, selective specification etc. Multi-user version (Unix, Mirage, MCS)
	APL.68000 Level I		
	Mac, ST, Amiga, QL	87	First generation APL. Single user, full windowing interface, software floating point support.
	Mac, Amiga	260	First generation APL. Single user, full windowing interface, hardware floating point.
	APL.68000 Level II		
	ST	170	Second generation APL. Full windowing interface, software floating point support.
	Amiga	260	Second generation APL. Full windowing interface.Hardware and software floating point support.

	Mac	520	Second generation APL. Full windowing interface.Hardware and software floating point support.
	APL*PLUS Rel 10	450	
	APL*PLUS II V 3.0	1395	
REUTER:FILE	SHARP APL	poa	for IBM mainframes
Uniware	APL*PLUS/PC	495	STSC's full feature APL for IBM PC/XT/AT, Compaq, Olivetti.
	Run-Time	call	Closed version of APL*PLUS/PC which prevents user exposure to APL.
	APL&PLUS/UNX	call	STSC's full feature APL for UNIX based computers
	APL*PLUS II	call	STSC's full feature APL for 386 machines.

APL VISUAL DISPLAY UNITS

COMPANY	PRODUCT	PRICES(£)	DETAILS
APL People	IBM & compatibles	poa	IBM and 'budget' APL VDUs - monochrome/colour/graphics.
Dyadic	IBM 3151	599	Monochrome APL/ASCII vdu with APL keyboard. Supports downloaded Dyalog APL font.
	IBM 6154	1,228	Colour APL/ASCII vdu with APL keyboard. Supports downloaded Dyalog APL font.
General Software	Mellordata	poa	
Shandell	HDS3200/10 APL	965	15" screen, 8 page memory, windows, 80/132 columns, full overstrike. Multi-host multi-session support. ANSI X3.64, DEC VT100, VT220, Tektronix 4010/4014 1024 x 390 resolution.
	HDS3200/25 APL	1065	As above plus switchable 25 or 50 line screen. 75 Hz refresh. Resolution 1024 x 780 in graphics mode.
	HDS3200/35 APL	1265	As HDS3200/25 plus local pan & zoom.
	HDS3200/5C APL	1395	14" colour monitor. 75 Hz refresh. 8 or 16 colours from palette o 256. Display memory 96 lines of 80 or 132 columns. APL processing & keyboard with full overstrike. Windows, multi-host, multi-session support. ANSI X3.64, DEC VT220, VT100, VT52 emulation.

APL PRINTERS

COMPANY	PRODUCT	PRICES(£)	DETAILS
APL People	Epson series	200	Inexpensive dot-matrix and NLQ printers
	Quen-data & Qume etc	500	Daisy-wheel printers
Dyadic	Various	poa	Range of APL printers available.
MicroAPL	Datasouth DS180+	1,195	See Datatrade entry
	Philips GP480	2,137	Matrix printer with letter & draft quality and APL (480 cps).
	Qume Letterpro20	549	APL/ASCII Daisy-wheel printer

APL PACKAGES

COMPANY	PRODUCT	PRICES(£)	DETAILS
Active Workspace Ltd	Syndicate Manager	poa	Lloyd's managing agent's syndicate / company accounting system. Stamp & Personal accounts (inc. Run offs)
APL-385	APL-385	50(PC),125(mf)	Including ...
	FSM-385		Screen development
	DRAW-385		Screen design

	DB-385		Relational W.S.
	GEN-385		Miscellaneous Utilities
APL Software Ltd (mainframe)	RDS	poa	Relation Data Base System
	IPLS	poa	Project Management System
	REGGPAK	poa	Regression Analysis Package
(microcomputer)	POWERTOOLS	295	Assembler written replacement function for commonly used CPU-consuming APL functions, includes a Forms Processor.
	REGGPAK	poa	Regression Analysis Package
	RDS	990	Relational Database System
APL IMPETUS	Impetus	poa	Hierarchical Planning System
Cocking/Drury (for VSAPL)	E'MENTS & SHAREFILE	poa	Component files, quad- functions & nested arrays for VSAPL under VM/CMS & MVS/TSO
	COMPILER	poa	The First APL compiler!
	FILEPRINT	poa	Print APL component files
	FILECONVERT	poa	Converts non-APL files to APL
	FILEMANAGER	poa	Extends APL primitives to database management
	TOOLS + UTILITIES	poa	APL Software development tools
	DATAPORT	poa	Information Centre spreadsheet incorporating data exchange between APL, FOCUS, IFPS, SAS, APL/DI, ADRSII, Lotus123, Visicalc, Multiplan & DIF
(for APL2)	SHAREFILE/AP	poa	STSC's shared access component file system for APL2. Comparable to all APL*PLUS file systems: multi-user storage of APL2 arrays with efficient disk usage
	FMT	poa	Full featured FMT for APL2
	WSDOC	poa	Workspace documentation utilities
	FILEMANAGER	poa	Extends APL primitives to database management
(for PC's)	APL*PLUS PC Tools	275	Utilities including: RAM disk, full screen data entry, menu input, report generation, exception handling and games.
	IRMA Module	90	327x IRMA support.
	FIN & STAT. LIBRARY	250	Financial & Statistical routines
	SPREADSHEET MGR	150	APL-based spreadsheet for APL*PLUS/PC. Cell arithmetic; transfers to ASCII & Lotus
H.M.W.	4XTRA	poa	Front-end Foreign Exchange dealing / pos keeping
	Arbitrage	poa	Arbitrage modelling
	Basket	poa	Basket currency modelling
	Menu-Bar	poa	pull-down menu for APL*PLUS/PC
HRH Systems	APL Utilities	poa	PC Utilities including: APLMAC (windows); Unlock (unlocks functions in .AWS); DTEX (text and spreadsheet imp/exp). Mostly available in English or French.
INFOSTROY	Russificator	poa	Drivers and documentation for use with APL*PLUS/PC system and other STSC software with Cyrillic alphabet (PC).
Interprocess	IEDIT	1900-3200	Full screen APL2 editor with immediate APL execution, and a full-screen debugger
(mainframe)	AFM	8200-9800	High performance component and keyed file system (VS APL and APL2)
	Format	1650	A QuadFMT data formatter for VS APL and APL2
	FSM124	1650	AP124 programming for APL applications without GDDM (APL2)

	PowerCode	1300	External functions for APL2
	CALL/AP	3000	for calling non-APL programs (VS APL and APL2)
	UCF	1800	Inter-user data transfer for VM users via IUVC
(PC)	AFM	115	Single user component and keyed files for APL2/PC
Mercia	STATGRAPHICS 4	684	Integrated statistics/graphics system for the PC. Now with macros. Bulk and educational discounts available.
	Upgrade 3 to 4	215	
	Upgrade pre 3 to 4	375	
	MICROSPAN	250	Comprehensive APL tutor
	LOGOL	poa	Logistics management system for PC and 386. Sales Forecasting, Inventory Control, Master Scheduling, Distribution Requirements, Planning etc
	TWIGS		A modular library of tools to teach and explore state-of-the-art materials management concepts.
	T	299	Time series forecasting
	W	99	Warehouse replenishment
	I	199	Inventory Management
	G	99	Grouping requirements into EOQ's
	S	99	Scheduling production/purchasing
		599	All 5 modules above
		4999	All 5 modules site licence
MicroAPL	MicroTASK	250	Product development aids
	MicroFILE	250	File utilities and database
	MicroPLOT	250	Graphics for HP plotters etc
	MicroLINK	250	General device communications
	MicroEDIT	250	Full screen APL editor
	MicroFORM	250	Full screen forms design
	MicroSPAN	250	Comprehensive APL tutor
	MicroGRID	poa	Ethernet & other networking
	APLCALC	400	APL spreadsheet system
	MicroPLOT/PC	250	For APL*PLUS/PC product
	MicroSPAN/PC	250	APL self instruction for APL*PLUS/PC
	STATGRAPHICS Rel 4	590	
REUTER:FILE			
	GLOBAL LIMITS	poa	Exposure management for banks
	IPSA/CONNECT	poa	Mainframe to micro link
	MAILBOX	poa	Electronic Mail
	MAILBOX/PC V.2	75	Full screen front end to IPSA mailbox
	upgrade to V.2	15	
	NEWSFLASH	poa	Real time message exchange
	VIEWPOINT	poa	4GL - Info centre product
Uniware (mainframe)			
	STSC's ENHANCEMENTS	poa	Quad-functions & nested arrays for IBM VSAPL under VM/CMS and MVS/TSO
	STSC's SHAREFILE	poa	component files for IBM VSAPL under VM/CMS and MVS/TSO amnd for IBM APL2

	PROGRAMMER TOOLS & UTILITIES	POA	
	FILEPRINT	poa	
	FILESORT	poa	
	FILECONVERT	poa	
	FILEMANAGER(EMMA)	poa	STSC's database package
	EXECUCALC	poa	Mainframe spreadsheet compatible with VISICALC and part of LOTUS 1-2-3 under VSAPL(VM or TSO)
(microcomputer)	STATGRAPHICS	poa	Statistics and graphics for PCs
	STATGRAPHICS UNISTAT	poa	An add-on module to STATGRAPHICS: Data analysis software.
	APL*PLUS TOOLS		
	-VOL1	poa	Incl. 327 Æ IRMA support, RAM disk, full screen data entry, menu input, report generation, games
	-VOL2	poa	Incl. File documentor, screen editor, exception handling.
	SPREADSHEET MNGR		poa APL spreadsheet with built-in ASCII, LOTUS and SYMPHONY interfaces.
	APL*PLUS/PC FIN & STAT.LIBRARY	poa	Collection of financial and statistical utilities.
	POCKET APL	poa	Smaller version of APL*PLUS/PC.
	UNIASM	poa	Collection of assembler routines for APL*PLUS/PC users.
	UNITAB	poa	APL*PLUS/PC spreadsheet-like data entry and validation system.
	The APL DEBUGGER	poa	First released APL*PLUS/PC debugger.
	APL2C	poa	Interface between APL*PLUS/PC and DATALIGHT C language
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.

APL CONSULTANCY

(prices quoted are per day unless otherwise marked)

COMPANY	PRODUCT	PRICES(£)	DETAILS
Active Workspace Ltd	Consultancy	poa	PC Based APL system design, programming and implementation.
Adfee	Consultancy	poa	Development, maintenance, conversion, migration, documentation,.of APL products in all APL environments
APL People	Consultancy	poa	Consultants available at all levels, with experience in: VS APL, APL*PLUS, APL2, Sharp APL, Dyalog APL, APL6800, C/Unix, TSO/MVS, VM/CMS, graphics, Operational Research etc. Expertise in APL system design, project management, prototyping, financial applications, decision support systems, MIS, links to non-APL systems, documentation, etc.
Buckland Management Systems	Consultancy	poa	Business and Technical systems in commerce and industry - designing, programming and implementing applications.
Camacho	Consultancy	poa	Specialising in programming & manual writing.
Chapman	Consultancy	150-300	24-hour programmer: APL, C, assembler, graphics; PC, mini, mainframe, network.

Cocking/Drury	Consultancy	175-275	Junior consultant
		275-350	Consultant
		300-450	Senior consultant
		400-600	Principal Consultant
		450-750	Managing consultant
Peter Cyriax	Consultancy	100-150	Junior Consultant
		120-200	Consultant
		160-300	Senior Consultant
Delphi	Consultancy	poa	APL system development on mainframes and micros.
Dyadic	Consultancy	poa	APL and Unix system design, consultancy, programming and training.
E & S	Consultancy	poa	System prototyping: all types of information system, engineering software, graphics and decision support systems APL*PLUS/PC, APL2, Dyalog APL
General Software	Consultancy	from 120	
H.M.W.	Consultancy	poa	System design consultancy, programming. HMW specialize in banking and prototyping work.
Ian A. Clark	Consultancy	poa	Computer-based Information Systems implementation where acceptance is critical. APL on PC and Macintosh. Human Factors of HCI; novice ease of use; online assistance; training courses; distance-learning materials.
INFOSTROY	Consultancy	poa	Localization of APL software for the Soviet Union software market
Intelligent Programs Ltd	Consultancy	175-350	Systems development, enhancements, support.
	Documentation	150-250	Preparation of new manuals, rewriting of existing materials.
	Training	150-250	Training for APL experts through to non-technical system users.
Mercia	Consultancy	poa	APL*PLUS & VSAPL consultancy.
MicroAPL	Consultancy	poa	Technical & applications consultancy.
M.T.I.C.	Consultancy	240-500	Business analysis and APL consultancy
Parallax Systems Inc	Consultancy	$750	Introductory APL, APL for End-user & Advanced Topics in APL
QB On-Line	Consultancy	250	Specialising in Banking, Financial & Planning Systems.
REUTER:FILE	Consultancy	poa	Consultancy & support service world-wide.
Rochester Group	Consultancy	poa	Specialise in MIS using Sharp APL
Rex Swain	Consultancy	poa	Independent consultant, 15 years experience. Custom software development & training, PC and/or mainframe.
Wickliffe Computer Ltd	Consultancy	poa	System design, consultancy, programming and documentation. Especially project management and decision support systems

OTHER PRODUCTS

COMPANY	PRODUCT	PRICES(£)	DETAILS
Adfee	Employment	poa	Contractors and permanent employees
APL People	Employment Agency	poa	Permanent employees placed at all levels. Contractors supplied for short/long-term contracts, supervised or unsupervised. Executive Search service available.
HMW	Employment	poa	Contractors and permanent employees placed.
I-APL Ltd	An APL Tutorial	2.50	45pp by Alvord & Thomson

	An Encyclopaedia of APL (2nd Edn)	5.50	228pp by Helzer
	APL in Social Studies	2.50	36pp by Traberman
	I-APL Instruction Manual (2nd Edn)	2.50	55pp by Camacho & Ziemann
	APL Programs for the Mathematics Classroom (Springer-Verlag)	14.50	185pp by Thomson
	** Please add one pound packing charge per order **		
REUTER:FILE	Productivity Tools	poa	Utilities for systems, operations, administration & analysts; auxiliary processors, comms software, international network.
	Databases	poa	Financial, aviation, energy and socioeconomic.
ISI	Tangible Math	$19	An APL Approach to Math - Shareware, includes base Sharp APL
	J	$24	Dictionary APL simplified and enhanced - Shareware (Mac,PC)
	SharpAPL/PC	$74	Registered Shareware and Reference Manual
	ISI APL	$99	Improved APL PC - enhancements, performance, large workspaces
Renaissance Data Systems	Booksellers		The widest range of APL books available anywhere. See Vector advertisements.

OVERSEAS ASSOCIATIONS

GROUP	LOCATION	JOURNAL	OTHER SERVICES	Ann.Sub.	Visa/Mcd
APL Bay Area	USA N. California	APLBUG	Monthly Meetings (2nd Monday)	$15	N/N
Dutch APL U.G.	Holland	-	Mini-congress, APL ShareWare Initiative		
APL Club Austria	Austria	-	Quarterly Meetings	200AS/person, 1000AS per company	N/N

VENDOR ADDRESSES

COMPANY	CONTACT	ADDRESS & TELEPHONE No.
Active Workspace Ltd	Ross D Ranson	Moulsham Mill Centre, Parkway, Chelmsford, Essex, CM2 7PX. Tel: (0245) 252414 ext.240
Adfee	Bernard Smoor	Dorpsstraat 50, 4128 BZ LEXMOND, Netherlands. Tel: 31.3474.2337, fax: 31.3474.2342
APL 385	Adrian Smith	Brook House, Gilling East, York. Tel: 04393-385
APLBUG	Jorge Mezei	117 East Creek Dr., Menlo Park, CA 94025, USA
APL Club Austria	Erich Gall	IBM Osterreich, Obere Donaustrasse 95, A-1020 Wien, Austria
APL Impetus Ltd	Cedric Heddle	Rusper, Sandy Lane, Ivy Hatch, SEVENOAKS, Kent TN15 0PD Tel: 0732-885126
APL People	Jill Moss	The Old Malthouse, Clarence St, BATH, BA1 5NS. Tel: 0225-462602
APL Software	David Alis Jill Moss	The Old Malthouse, Clarence ST, BATH, BA1 5NS.Tel: 0225-462602
Buckland Management Systems	John Buckland	Westwood, 19 Grange Road, Camberley, Surrey, GU15 2DH Tel: 0276 684327
Anthony Camacho		2 Blenheim Road, St. Albans, Herts AL1 4NR. Tel: St. Albans (0727) 860130
Paul Chapman		18, Trevelyan Road, London, SW17 9LN Tel: 081-767 4254
Cocking & Drury Ltd.	Romilly Cocking	180 Tottenham Court Road, LONDON, W1P 9LE Tel: 071436 9481 Fax: 071-436 0524

Datatrade Ltd.	Tony Checkseld	38 Billing Road, Northampton, NN1 5DQ. Tel: 0604-22289
Delphi Consultation	David Crossley	Church Green House, Stanford-in-the-Vale, Oxon SN7 8LQ. Tel: 03677-384
Dutch APL U.G.	Bernard Smoor (Sec)	Postbus 1341, 3430BH Nieuwegein. Tel: 03474-2337
Dyadic Systems Ltd.	Peter Donnelly	Riverside View, Basing Road, Old Basing, Basingstoke, Hants RG24 0AL. Tel: 0256 811125 Fax: 0256 811130
E & S Associates	Frank Evans	19 Homesdale Road, Orpington, Kent BR5 1JS. Tel: 0689-24741
General Software Ltd	M.E.Martin	22 Russell Road, Northholt, Middx, UB5 4QS. Tel:081-864-9537
H.M.W. Computing Ltd.	Stan Wilkinson	Hamilton House, 1 Temple Avenue, Victoria Embankment, LONDON EC4Y 0HA Tel: 071-353 4212 Telex: 926604 HAMHSEG Fax: 071-353 3325
HRH Systems	Dick Holt	Box 4496, Silver Spring, Maryland 20904
Ian A. Clark		9 Hill End, Frosterley, Bp. Auckland, Co. Durham DL13 2SX Tel: 038852-7190
I-APL Ltd	Anthony Camacho	2 Blenheim Road, St. Albans, AL1 4NR. Phone 0727-860130 for queries, order forms, bulk orders
J C Business Services	June Cutts	56 The Crescent, Milton, Weston-super-Mare, Avon, BS22 8DU N.B. PAID ORDERS ONLY
IBM UK Ltd	Nat.Enq. Centre	414 Chiswick High Rd, London W4 5TF Tel: 081-747 0747
INFOSTROY	Alexei Miroshnikov	3 S. Tulenin Lane, Leningrad 191186 USSR. Tel:812-238-6392 Fax:812-319-9709
Interprocess Systems	Stella Chamberlain	9040 Roswell Road, Suite 690, Atlanta, Georgia 30350-1131 Tel: (404) 992-8400
Intelligent Programs Ltd	Mike Bucknall	Unit 7, Hermitage Court, 6-10 Sampson Street, London E1 9NA Tel:071-481-4813
ISI	Eric Iverson Orders	33 Major Street, Toronto, Ontario, Canada M5S 2K9 Tel:(416) 925-6096 3612 Cameron Mills Road, Alexandria, Virginia, USA 22305-1103 Tel:(703) 548-1799
Mercia Software Ltd.	Gareth Brentnall	Aston Science Park, Love Lane, Birmingham B7 4BJ. Tel: 021-359 5096
MicroAPL Ltd.	David Eastwood	South Bank Technopark, 90 London Road, LONDON SE1 6LN Tel: 071-922 8866
M.T.I.C.	Ray Cannon	7 Pine Wood, Sunbury-on-Thames, Middx. TW16 6SH Tel: Sunbury(09327) 80848
Peter Cyriax Systems	Peter Cyriax	213 Goldhurst Terrace, London NW6 3ER Tel: 071-624 7013 (Answerphone) 0860-377963 (Mobile)
Parallax Systems Inc.	Kevin Weaver	Avery Road, Box 319, Garrison, NY 10524, U.S.A. Tel: 914-424-4265
QB On-Line Systems	Philip Bulmer	5 Surrey House,Portsmouth Rd Camberley, Surrey, GU15 1LB. Tel: 0276-20789
Renaissance Data Systems	Ed Shaw	P.O. Box 20023, Park West Finance Station, New York, NY 10025-1510, U.S.A.. Tel: (212)864-3078
REUTER:FILE	Paul Jackson	7th Floor B Block, Coventry Point, Market Way, Coventry CV1 1EA Tel: 0203 256582
The Rochester Group	Robert Pullman	50 S.Union St., Rochester NY 14607, U.S.A. Tel:716-454-4360 or 716-454-4641
Shandell Systems Ltd.	Maurice Shanahan	Chiltern House, High Street, Chalfont St. Giles, Bucks., HP8 4QH. Tel: 02407-2027. Fax:02407-3118
Sugar Mill Software Corp.	Lawrence H. Nitz	1180 Kika Place, Kailua, Hawaii 96734 Tel: (808) 261-7536
Rex Swain		8 South Street, Washington, CT 06793, U.S.A. Tel:203-868-0131 or 212-242-5816
Uniware	Eric Lescasse	15 Rue Erlanger, 75016 Paris, France. Tel:(1) 45-27-20-61. Fax:(1)45-27-20.61. Telex: 648348F UNIWARE
Wickliffe Computer Ltd	Nick Telfer	76 Victoria Rd., Whitehaven, Cumbria, CA28 6JD. Tel:0946-692588
Warwick Univ.	Prof.Jeff Harrison	Dept of Statistics, University of Warwick, Coventry, CV4 7AL Tel:0203-523369

ZARK: an APL Tutor for APL*PLUS/PC

reviewed by Emily Timson

ZARK is a software package which aims to teach APL to novices whilst providing a valuable reference aid to experienced APLers. My parents both work with computers and seem to be infatuated with the things: I have been fairly successful in avoiding anything to do with them. Then my husband bought one and also tried to get me interested and that's how I was persuaded to try ZARK and relate my experiences for Vector, a magazine I have never read. When I began I knew almost nothing about APL.

The ZARK APL Tutor Kit

The Software was on 4 floppy disks and a couple of pages of information about getting ZARK up and running on your PC all in a small disk container. The instructions were brief and straight forward for my husband. They worked first time. Within a couple of minutes he had started ZARK for me and I was sitting comfortably and ready to begin.

Zark is written in American, which sometimes makes it hard to follow, and is frequently amusing. I laughed out loud on many occasions. Here is a sample of the style from the starting instructions:

The Zark APL Tutor won't do you any good if your computer won't talk APL. These instructions may help to get you going. Follow them carefully.

1. *Find someone who's installed APL on his or her computer. Get down on your knees and ask for help.*
2. *Failing that, read the installation instructions that come with the APL product. Do what they say.*
3. *Failing that, follow the steps below. We've distilled them from the manuals .... If you have trouble, return to step 1 or 2 above.*
4. *Failing that, learn COBOL.*

The Introduction

I thought it would be wise to give this section a visit. (You see the Zark style is catching). Getting here was simple, I just pressed the SPACE BAR. Instantly, full colour easy to read text screens (we have VGA, but they work on Hercules too because we used to have that). Six pages followed in which I was told about the structure of the lessons and more importantly which THREE keys I had to press. SPACE BAR to proceed with the lesson, F1 for HELP and best of all ESC to get me out of all difficulties and back to the main menu.

Feeling in control and not yet confronted with anything confusing or difficult I moved straight on to the first lesson (SPACE BAR).

The LESSONS

ZARK seems to be meticulous in the way it has covered aspects of the language with a lesson for each one. If you know what subject you want to be taught you can quickly select a topic from the index screen and move straight into the required lesson. However you get there you are faced with a three part lesson:

Part 1. Tutorial

This section is like an extract from a text book. This is where ZARK teaches you about whatever topic you have selected. The text is very easy to follow and quite humorous. The lessons never seem to get too deep and yet at the end of each tutorial I felt I had covered everything well. Knotty points are sometimes covered by a discussion between three 'APL creators' which are represented by faces which pop up on the screen under the words expressing their point of view. Sometimes they argue with each other, but they always seem to agree on the best way of solving each problem.

Part 2. Reading

This is the shortest part of the lesson. ZARK illustrates the usage of the function being taught and invites you to predict the result of the expressions shown. This truly tests your understanding. Every usage is illustrated:

Predict the result of the following APL expressions:

```
      3 + 7  (Press Enter)
10
      3 - 7  (Press Enter)
¯4
```

Note the (¯) sign, it signifies that the result is negative. The (-) sign is a function which subtracts the right argument from the left argument.

```
      7 + (Press Enter)
SYNTAX ERROR
      7 +
        ^
```

... addition needs both left and right arguments.

It is easy to work through each example (Press Enter) and if you experience any trouble you can easily go back to the tutorial section to recap (Press ESC). You can also interrupt the lesson at any time by pressing SCROLL LOCK to enter immediate execution mode to experiment as you choose. Pressing SCROLL LOCK again returns you to the lesson at the place where you left it.

Part 3. Writing

This is the best part of the lesson. In this section you find yourself working in immediate execution mode. ZARK presents you with a simple problem such as:

> *"What length of fence is required to enclose a garden of length 7 metres and width 5.5 metres. Assign the answer to the variable LENGTH."*

You are required to type the APL expression to satisfy the question and then press ENTER. You can then check it by entering other APL expressions or submit it to ZARK by pressing SCROLL LOCK. ZARK replies with "Correct" if you are correct and gives you some alternative (equally valid) expressions. If you are wrong then ZARK offers you the chance to try again (twice). I enjoyed this hands-on practice. The problems posed by the tutor were worded to test your knowledge of APL, not your knowledge of solving puzzles. The questions are all very interesting and kept my attention all the way through the lessons.

```
      LENGTH ← 7 + 7 + 5.5 + 5.5
      LENGTH
25
[SCROLL LOCK]
```

Correct, the answer is 25. Other possible expressions are:

```
2×(7 + 5.5)   (2×5.5)+2×7   (7+5.5)×2
```

Topics Covered

ZARK contains 26 chapters labelled A to Z. As a tutor ZARK should be followed in sequence as the 'writing' section assumes that you have covered all preceding topics. The APL functions and the unique APL concepts such as Vectors, Matrices and Rank have separate sections of tutorial. The index seems thorough. The quantity of information in the index is immense. By searching I found a few topics which brought up a message that the topic was not covered in this course; encode and decode produced this message.

ZARK allows you to search through the index by function. This means you can scan the index for '+' and ZARK will take you directly to the appropriate point in the series of lessons and commence at the 'reading' stage. This is a very effective way of providing speedy and efficient reference after you have been through the course.

If, on the other hand, you know what you want to do but you do not know which APL function is appropriate then you can search the topics by subject index. This means you can search the index for 'Magnitude' or 'Modulus' and ZARK will swiftly transport you into the lesson which deals with the relevant APL function. If you know another programming language and you want to learn APL then this index would help you make the translation.

General Comments

ZARK impressed me because I did not expect it to be so much fun or such a variety of tricks to be used in the presentations. My husband says the instructions to install ZARK are terrific. The ZARK environment is clear and simple to negotiate.

The depth and variety of information offered by ZARK is staggering. I did not know there was so much in APL. I think that if I were going to write programs in APL this would be a very good way to learn. The truth is that I found APL rather difficult and haven't yet finished all the sections of the Tutor, but I think that anyone could pick up this package and be able to make good effective use of the language after working all through it.

What I found was that too often I was waylaid by the Tutor into making notes about the APL I was learning and forgetting entirely to make notes about the package I was supposed to be reviewing.

Little details such as telling you where a symbol is on the keyboard are never overlooked. It may seem trivial but how would you deduce or guess where the symbol for transpose is.

The tutor is positively enthusiastic about the versatility, flexibility and power of this language. This enthusiasm comes across in the lessons. A little has rubbed off on me!

Conclusions

I think this is a very good tutor for APL. Of course it is only for use with STSC's APL*PLUS/PC. I was lent version 8.0 for the review.

I am impressed with almost everything about Zark. I recommend it to anyone who wishes to learn APL or would like a handy encyclopedia of APL for quick reference.

APL PEOPLE

Serving the International APL Community

Consultancy	advice and assistance with all aspects of APL systems development
Recruitment	unique placement service for companies and individuals
Software	Competitive prices on all APL software tools and interpreters

Contact: Jill Moss, APL People Ltd.,
The Old Malthouse, Clarence Street,
Bath BA1 5NS

Tel: 0225 462602 during office hours
0225 333618 evenings and weekends

APL.68000 Level II

reviewed by Iain Hayward

Introduction

APL is almost unheard of on home computers, even though a full specification interpreter has been available for years. When MicroAPL ported APL.68000 onto popular machines like the Macintosh, Amiga and Atari, it was pioneering the use of WIMP interfaces in APL, and it made APL available to almost everyone.

MicroAPL recently announced the release of APL.68000 Level II, and renamed its original product as Level I. As an existing user of Level I for two years now on my Amiga, my only complaints have been the rather limited printer support, which I rectified using an auxiliary processor (Vector Vol.6 No.1, page 127), and the lack of nested arrays. Now it seems that MicroAPL has more than compensated for the latter problem by providing an upgrade which it claims is a superset of APL2. Although this review is based on the Amiga version, it concentrates on the machine-independent enhancements to the interpreter.

First Impressions

Any doubts about it being worth the price begin to fade as soon as you remove the wrapping; the manuals and disk come in a respectable looking box file as befits a serious software product. An initial glance at the contents gives further reassurance that MicroAPL has not tried to skimp on production costs. Even the disk containing the software is of an unusually high quality.

Two sets of APL key stickers are provided, one for the standard APL keyboard layout and one for the alternative Unified layout. The stickers should be attached to the front of each key rather than on top; an arrangement that works very well in practice but is surprisingly awkward to install (this shouldn't affect real APL'ers, of course, who already know where all the symbols are!).

For those who don't know quite where to start but are impatient to see their new purchase do something, MicroAPL has provided an automatic demonstration workspace which is quite impressive and inspiring.

Documentation

The package includes two conveniently sized A5 format manuals which stack well alongside copies of Vector and MicroAPL News. The first, thinner manual is concerned with system-specific information such as the user interface and access to features of the native operating system: graphics, windows, multi-tasking, etc. It is rather brief however, and doesn't really do justice to the software it describes. It also lacks an index.

The second manual however, the APL.68000 Level II Language Reference Manual, is superb. It is about two centimetres thick and ring bound so that it opens flat without one having to stand a coffee cup and a paper weight on it. It was obviously written by someone who knows and loves APL, and if you've got this manual then you may not need to buy a copy of Gilman and Rose. More than a third of the manual is devoted to teaching APL, and MicroAPL's experience in giving courses is clearly in evidence. There are convenient entry points for the complete beginner, for programmers who don't know APL, and for programmers who know APL but are unfamiliar with this implementation. There is also advice for users who are upgrading from Level I. The reference section of the manual should serve as an example to other suppliers. It is very readable, and gives carefully selected examples of how a function or operator might be used, rather than just a dry, technical definition. It is worth mentioning that the manual was completely re-written for Level II, instead of just having a new section added.

New Features

All of the enhancements to be found in Level II seem to have been added to provide compatibility with the APL2 standard. They include nested arrays, mixed arrays, multiple and selective specification, vector notation, four new primitives, defined operators and extensions to existing ones, and APL2-style error handling. Readers who are familiar with APL2 may wish to skip the next few sections and continue reading from the section on compatibility.

Nested Arrays

All of the functions that I had expected to see were there, together with a few that were new to me. All appear to conform to the APL2 standard.

I found the Partition function particularly useful. It divides its right argument into an array of nested vectors according to the specification given in its left argument, which is a scalar or vector of zeros or positive integers. A new element is created in the result whenever the corresponding element in the left argument is greater than its predecessor, whilst a zero causes the corresponding element to be discarded. This can be used to good effect in text processing, where you might want to separate words into a nested vector. First identify all the non-space characters as a boolean, then use this as the left argument to Partition. As a bonus, all single as well as multiple spaces are removed in the process.

Enclose increases the depth of its argument by one, producing a scalar. It has no effect on a simple scalar. It can be used with an axis specification to split and rearrange an array. Disclose used on a scalar reverses the effect of Enclose. If the argument is a nested vector then the result will be a matrix, with rows padded as necessary with their elements' prototype. Disclose can be used with an axis specification on higher dimensional arrays to rearrange the data. The shape of the result of Disclose is derived from a combination of the shape of its argument and the shape of the items within the argument.

The Pick function lets you pick out an item from a specified position and depth within a nested array. First returns the first element of an array, or its prototype if it is empty, and is a little more flexible than 1 Take. Enlist has the effect of ravelling and catenating every item in its argument (removing nesting in the process) to produce a simple vector out of anything. Depth returns the depth (amount of nesting) of the deepest part of an array. Although a simple scalar has a depth of zero, for some reason a simple array is considered to have a depth of 1. Thereafter the depth increases by one for each level of nesting.

The powerful Each operator has been provided of course, which lets you perform an operation on each element of an array without the need for a loop. The system functions ⎕*CR*, ⎕*FX*, ⎕*SS* and ⎕*DR* have all been extended to support nested arrays, and a utility function *DISPLAY* has been provided which displays the structure and contents of a nested array.

Mixed Arrays

It is now possible to mix character and numeric data in a single array. I don't think I would ever use this feature with simple arrays, but it is very useful with nested arrays where you might want to keep related data together, some elements containing numeric data and others character data.

Multiple and Selective Specification

Multiple specification or assignment is permitted, allowing you to assign a vector to a list of variable names in brackets, each element of the vector being assigned to a separate variable. It doesn't seem to work on system variables though, and I find that the diamond separator makes this feature largely unnecessary, but it's there for those who want it. Much more useful is selective specification, which makes it possible to assign to items deep within a nested array. The idea seems to be that if you can select it, then you can assign to it. There are some restrictions on the complexity of the expression used for selection, but I don't think APL.68000 is alone in this.

Vector Notation

Just as it has always been possible to create a numeric vector by entering a list of its elements, now it is possible to construct much more complex data structures in the same way. Each element can now be an array in itself, with parentheses and quotes being used where necessary to delimit the elements. This facility is particularly useful for constructing nested and mixed arrays.

Other New Primitive Functions

The Index function (squad character) is also implemented, providing a more powerful and neater way to do indexing than using brackets and semicolons. It can be used with an axis specification. Match will test if its arguments are absolutely identical in every way. This function is a welcome addition because it is quite tedious to have to write such a test in APL. Find is a generalized search function. It behaves like the string search system function but its arguments can be of any rank and type. I like the Without function, and feel it should have been included in APL from the very start. It returns its right argument with all the items occurring in its left argument removed. It only works on vectors though, but they can be nested.

Extensions to Operators and Defined Operators

Some of the most exciting enhancements to be found in the new product are still to come; operators can now be used with any primitive functions and with user-defined functions, so expressions such as *,/DATA* and *MYFN/DATA* are now valid. Furthermore, it is now possible to write your own operators, just like writing functions. The possibilities are enormous.

Error Handling

In a nutshell, APL.68000's error handling functions are still there, and IBM's error handling functions have been added.

The new system functions are:

⎕*EA*	Execute Alternate
⎕*ES*	Error Simulate
⎕*EC*	Execute Controlled
⎕*ET*	Error Type
⎕*EM*	Error Message

This apparent duplication of functionality actually turns out to be a good thing. Apart from the obvious benefits of compatibility, the two sets of error handling functions are in fact complementary, since the existing functions provide error handling at the function level, while the new functions provide it at the statement level.

Other Enhancements

The Overbar and Underbar characters can be used in object names, although not for the first character. Comments are allowed in function header lines, and spaces before comments are preserved (something I had wanted for a long time!). Ravel can now be used with an axis specification to create a new axis of length 1, or to combine a range of axes into one. Take and Drop can also be used with an axis specification to achieve Take and Drop along specified axes only.

Compatibility

Level II is upwards-compatible with Level I with just three slight exceptions. The first and most important one concerns the indexing of numeric constants. The following expression:

```
      1 2 3 4 5[3]
```

would give a *RANK ERROR* under Level II because the index binds to the scalar immediately to its left. The second exception concerns object class codes as defined for the system functions ⎕*NL* and ⎕*NC*. Code 4 in Level I meant invalid name; now invalid name is -1 and 4 means user-defined operator. The third exception is simply that numeric arrays are now displayed with each column formatted separately. These differences won't cause any of your existing applications suddenly to start behaving differently, though. A system variable ⎕*CS* (compatibility setting) has been provided to cause the interpreter to behave like Level I for all or any combination of the above three cases, and it is automatically set to give full compatibility when you load a Level I workspace.

I was particularly pleased to find that my assembler-coded auxiliary processor routines continued to function correctly. The original auxiliary processor interface (AP1) has remained the same, and just gives a *DOMAIN ERROR* when passed a nested or mixed array. A new AP interface (AP2) has been provided which will accept the new data structures.

MicroAPL has expressly stated that Level II is designed for close conformance with IBM's APL2 standard, and while there are some divergences, it is MicroAPL's intention to remove most of these in future releases. Even where Level II is different from APL2/370, it is usually compatible with APL2/PC, and in some cases it is more compatible. Among the features still to come are complex numbers, N-wise reduction, format-by-example, axis on scalar functions and matrix arguments to the sort primitives.

Portability

It has always been the case that a workspace saved on one machine can be loaded on any other machine running APL.68000, since the workspace format is identical. MicroAPL also claims that any workspace saved under Level I can be loaded under Level II, provided that the state indicator was clear when it was saved. You are warned however, that since the new interpreter is about 50K bigger, you may have less workspace available on single-user systems. So far, I have had no problems.

Due to the high degree of compatibility with APL2, it is now possible to move applications to and from APL2/370 and APL2/PC. This is achieved quite easily by using the system commands *)OUT* and *)IN* which create and read transfer files. How you move the files between machines is up to you, but you must ensure that no character translation takes place. There is also a system function called *⎕TF* which returns the transfer form of an object, or alternatively, decodes an object from its transfer form. The transfer form is a text representation that is suitable for transmission between dissimilar machines or implementations of APL.

System Limits

Most of the limits seem absurdly generous, for example the maximum depth of an array is 100, and the largest number allowed is about 1.8E308. The only limits which might conceivably be a problem are the maximum rank of an array (8), the maximum length of a name (30 characters) and the maximum symbol table size (6,021 symbols).

Performance

I am not keen on benchmarks, preferring to judge for myself whether the performance of a system is acceptable, but I did once run some standard benchmarks on APL.68000 Level I on my Amiga and found it to be slightly faster than an IBM PC/AT running APL*PLUS/PC. I found the overall performance of Level I to be a bit slower than C code, but it saved me so much development time that I would never go back to programming in C. MicroAPL has certainly optimized the displaying of text; results are flashed up on the screen like lightning. So far, I have not noticed any degradation of performance when running my existing applications under Level II. I can see one reason why this is so: the internal structure of simple variables has not changed, only the new data structures are more complicated, and they have been given a new type code. On the occasions where I have been able to replace existing code with a nested array solution I have in fact achieved a considerable improvement in performance, not to mention simplified code.

Conclusion

I have to conclude that Level II does everything that is claimed of it and, in the case of personal computer implementations, much more. The standard libraries provided with APL.68000 make it unbelievably easy to use the features of the native operating system; a single function call replaces many lines of quite difficult code and hours spent studying the reference manuals. There is even a terminal emulator included so that you can log on to remote APL systems. Now MicroAPL has enhanced its product to the standard of APL2 on a mainframe and made it portable as well. All this, together with a WIMP user interface that has to be experienced to be appreciated, would seem to make APL.68000 the obvious choice for anyone who wishes to do serious programming in APL.

APL*PLUS/PC PostScript Support

notes by Adrian Smith

On page 70 of Vector 7.3, Jonathan Barman regretted his lack of a PostScript printer, and hence his inability to test out the PostScript APL font shipped with APL*PLUS/PC release 10. Accordingly, I borrowed the review copy, and had a quick look at the font:

```
SJGKHZ×WY.ETOQDMF[UN←I
57]9B+PRVCA0/XL,~○?⌊|-
123846∧\⊃⎕;%(↓⊤→⍳⋆⍴∪∩⍺
¨¯<≠≤≥=>)∨⊥⊂÷⍵↑:∊⌈∘∇'∆
⌶⍫⍒⍋⌽⍉⊖⍟⍱⍲!⍪⍀⌿⍕⍎⍝⍞⍬⍙⍷≡
abcdefghijkl{}_⌹áíóúñÑ
mnopqrstuvwxyz¦£$%&#"@
⊣⊢ê ¢ ◊     ≢«»öÖß
```

Aesthetically, I'm afraid I don't particularly like it; the whole font looks a bit big and heavy, and there is no variation in the character height (e.g. I think that symbols like ⌈ ⌊ should show above and below the caps). It also bothers me that there is no attempt to be consistent about the overstruck symbols ... ∇ jumps all over the place depending on what has been combined with it! None of the PC line-drawing characters are included, which I would find an annoying omission.

Speed of printing is a little slow (the font is encrypted which must slow it down), a typical function listing took 156 sec to print on GoScript, as against 32 sec using my APL-2741 font.

The good news is that if you want to use the Port10 driver, you can easily substitute any other PostScript font which uses the same encoding, and the encoding is in plain text in the font definition (it looks like standard APL*PLUS/PC ⎕*AV*). For my part, I shall continue to work with my own font, trivially recoded to work with the APL*PLUS character encoding.

RECENT MEETINGS

This section of VECTOR is intended to document the seminars given at recent meetings of the association; it is of particular value to members who live away from London. It also covers other selected events which may be of general interest to the APL community.

If you would like to speak at one of the regular British APL Association seminars, please ring the Activities Officer (address on inside back cover) who will respond enthusiastically to your offer.

The Trials and Tribulations of using GSS Graphics with APL*PLUS/PC

talk by George MacLeod November 1990, notes by Jonathan Barman

George MacLeod gave us a heartfelt commentary on the problems in getting decent graphs out of GSS graphics with APL*PLUS/PC.

First George explained why high quality charts are needed. APL Impetus Ltd develop and market a financial planning package Impetus which is a descendant of the Boeing's TABAPL. Impetus is often used high up in companies and as a result the quality of the output is important, and high quality output must include high quality graphics.

The initial prototypes for Impetus used the ⎕*G* graphics available with the APL*PLUS/PC interpreter. The functions mirror those available in ROM BASIC, and the facilities provided are quite simple, for example only one font is provided which is increased in size by pixel replication. Sizes increase by squaring the number of pixels, so big characters look very crude. ⎕*G* graphics can be used with a relatively limited range of graphics boards and printers, but there are now a huge array of graphics devices. It has become impossible for STSC to support all the devices on the market.

GSS*CGI was introduced by STSC to solve these problems. GSS is a American company, Graphics Software Systems Inc, who concentrate on providing graphics drivers for every device on the market. CGI stands for Common Graphics Interface which is set of standardised graphics calls for programming languages such as C and FORTRAN. STSC have provided a set of functions in a workspace which reproduce each of the calls provided by GSS*CGI. Using these functions forces the use of extensive looping, as the GSS functions are really designed for compiled languages which can only manipulate scalars.

The first essential before using GSS*CGI is to buy the GSS Programmers Guide costing £50. The APL*PLUS/PC manual gives a simple list of functions and their titles, but not the details of the arguments and values required. For example, the function *V_OPEN_WKST* takes an eleven element numeric right argument which specifies default values for the device named in the left argument. The exact numbers to be used are only given in the GSS Guide.

George showed a sample chart using GSS*CGI as supplied. The fonts are only marginally better than the ⎕*G* fonts, and the quality is quite low. To get better fonts you need the GSS Completer Kit at £70, which contains lots of device drivers, FONTDRIV.SYS, but no fonts! Bitstream supply fonts at £140. What you actually get is sets of bezier curves defining the characters, together with a font maker. To actually use the Bitstream fonts you have to buy the GSS Font Maker which is part of the GSS Developers Toolkit costing £400. An apparently simple addition turns out to be quite expensive!

Now you have everything the problems really start. Whilst screen display works well the major problem is printing in high resolution. George showed sample charts which took many hours each to produce, and some of them were plainly wrong with headings missing! The problems are almost certainly to do with the way GSS manages memory, and GSS own up to having a problem with bitmaps. Whatever memory is given to GSS it always seems to need more. The memory has to be below the 640K boundary, and any memory allocated to GSS cannot be used by APL.

At this point George & Co. gave up and tried to use the Metafile Interpreter supplied by GSS at £225. The Metafile Interpreter consists of five diskettes full of subroutines, and you have to write a program in C or FORTRAN to produce what you want. So they had to buy a MicroSoft C compiler at £250.

Basically this is as far as George has got with high resolution graphics using APL*PLUS/PC and GSS*CGI. Other alternatives are being looked at such as APL*PLUS II and DYALOG APL. Both these versions of APL can run above 1 Meg and can let GSS run below 640K. APL*PLUS II and APL*PLUS/PC have essentially the same interface with about 120 APL functions matching each of the GSS commands. Dyadic have done rather better in that you can pass many GSS commands to the Auxiliary Processor in one nested array. Also, Dyadic have made some functions ambivalent so that settings can be passed as an optional left argument.

George summarised the situation as follows:

> The GSS Graphics system is unsatisfactory when used with APL*PLUS PC.
>
> Very limited experience suggests GSS will work satisfactorily with APL*PLUS II and DYALOG APL.
>
> Other PC products that offer graphics use GSS and they seem to work well. Perhaps most of the problems are due to poor cohabitation of APL*PLUS/PC and GSS.
>
> GSS is probably the best PC Graphics System available.

The Ideal Screen Editor

a talk given to the BAA on 23 November 1990 by Anthony Camacho

Five overhead projector foils were shown as an introduction to the screens which made defining a screen easy. Foil 1 was an introduction.

WHY A NEW SCREEN SYSTEM? [foil 2]

Most applications need one
Vendors' systems incompatible
Takes too long to write
Wish to make APL widely used
Wish to restore APL's advantages
Wish for screen to respond fast
Cannot spare much workspace

The motives for writing a screen system were to provide something independent of vendors that required as little beyond ISO APL as possible, written in APL so that anyone could amend it and straightforward enough to be used without a great deal of effort. Once, APL gave its users the best interface for interaction between the keyboard operator and the system; it would be nice to restore the advantage. Also a system would have to be fast enough and use little enough workspace so that it would be practical even in I-APL.

ORIGINS AND THE IDEA [foil 3]

Paul Chapman's port from Viz::APL
Use only window get/put cursor
All screen input/output is character
Pre-calculate variables for speed
Hold items on file to reduce space

To use:

```
⍝ R←SNAME ∆FS CHARVEC
⍝   SNAME ←→screen name on file
⍝ CHARVEC ←→chars for fields
⍝       R ←→ chars from fields
```

The idea was that a perfectly adequate screen system can be written using only four special functions, which are available in most versions of APL on micro-computers. The originator of the idea was Paul Chapman, who had to port a system from Viz::APL to APL*PLUS/PC in a hurry. He never completed or documented what he wrote, but it was in daily use for several years and the response was adequate even on an 8088 at 4.77MHz. Most data was held in vectors and as much as possible was pre-calculated. I learned about it by having to enhance the system that used it and the Ideal Screen Editor (ISE) was born when I considered how Paul's screen system could be adapted to meet the objectives mentioned.

The main limitation of ISE is that all input and output of the screen display functions is in characters. The file referred to on the foil is tied to *ΔSTIE*. A secondary limitation is that the main function *ΔFS* has to call subsidiary functions for any field which has more than one line.

FEATURES OF THE DESIGN [foil 4]

Three functions for simplicity

```
ΔFS Full screen (many fields)
ΔMA Matrix amend (one fld only)
ΔBB Bounce bar (one field only)
```

ΔMA & *ΔBB* handle multi-line fields

For each *ΔFS* the file has 9 components

The subsidiary functions are *ΔMA* which displays any matrix for amendment in a window (which may be a pop-up window) and *ΔBB* which handles the bounce-bar method of choosing an option; the choices may be scrolled past the window and selected with the highlight bar or, if the lines are numbered, by entering the option number.

The pre-calculated variables are held on a component file (or a pseudo-component file) in nine components, containing:

1. The design so that it can be re-entered with variations
2. The window for the screen
3. The initial contents of the screen
4. The indices of the characters that may be varied
5. The attributes of the initial contents

6. A table giving details of each field
7. A small matrix of executable lines (mainly standard validations)
8. A vector of the codes from keys that cause special actions
9. A corresponding character vector of labels (the action for the key)

The field table contains the window for each field, the navigation from field to field, three attributes of the field:

on display,
when highlighted with the cursor in it
and when validation has failed and it must be re-entered.

It also contains a pointer to the validation required, a pointer to any special action required on moving the cursor into the field, and, for multi-line fields, details of what is required.

To get the system into use, once it was established that it was feasible, the author had to do three things:

1. Produce an example to show how to use the screen functions
2. Demonstrate that the system was robust with a demanding test
3. Provide the potential users with an easy way of entering screens

These three needs could all be met if the screen system could prove its worth by using it to write the screen entry system. The screen entry system is the most important part of the design. If it is really easy to use then people may use it even if other features are less than perfect. On the other hand if the entry method is difficult to use then it probably will get ignored.

The author's purpose this afternoon therefore was to describe the entry procedures and to ask the BAA members present whether they could think of a simpler way of doing any of the tasks that are an essential part of entering a screen design.

WHAT IS THE EASIEST WAY? [foil 5]

Enter by amending a similar screen
Defaults (sensible) for everything
Never a need to count (characters)
Specify position by cursor
Choose from lists where possible
Show effects as soon as possible
Allow easy corrections

The simplest way to enter a new screen design would be to amend one which was similar. It is easier to alter a heading and an instruction or two than to key in a complete design. The simplest way to enter colours, validations, field sequences and the action to be taken on each kind of key depression is to have them all default to standard settings. The standards to which everything defaults should themselves be settable so that all the screens in an application can be given a distinctive house style. At every stage the delay between specifying something and showing the result should be as short as possible and the sequence to respecify it if needed should also be short and quick.

A series of draft screen designs were then shown and the proposed method of use for each described. In every case the aim was to find the entry method which would give the least trouble to the keyboard operator entering the design.

Fields are specified by putting special characters at the ends of the field (outside it so that default contents could be entered within the field). Ruled lines are specified by their ends, boxes by their corners and multi-line fields by marking the first and last lines after the fashion of single fields but with special characters. The screen may be framed with single or double lines by entering '+' or '#' in the top left corner.

Pop-up fields are specified by moving the cursor to the top left, marking the spot, and then to bottom right and fixing that. The same procedure is used to specify colour for the areas of the screen outside the fields. Fields themselves have three colours; all may be left to default values, the defaults may be changed and any field may have special colours set by moving the cursor into the field and specifying the colour.

Validations are specified by entering a character corresponding to one of the six standard validations or to a special validation in the field. If a special character is used then the validation required is entered as a line of APL (best to give a function name) which returns the validity as a boolean.

The fields may be navigated in two sequences and there is a third sequence for inserting the variable data into fields. The sequences have defaults. The default for any sequence may be amended by entering a decimal number in a field to give its new position in the sequence (like inserting a line under the del editor). On moving the cursor from the field the new sequence is calculated and displayed.

The allocation of actions to keys is done by displaying two bounce-bars. In one the key and a two letter code for its current action is shown; in the other a list of possible actions is shown. To allocate a different action to a key, first highlight it, then highlight or enter the new action, then press an F-key.

During the discussion afterwards only one improvement to ease of entry was proposed, by Peter Branson. Peter suggested that the method used for specifying a pop-up field should be used to specify multi-line fields. The author was glad to accept it and hoped that, as nobody could think of any further improvements, the design was now as good as it could be.

APL.68000 Level II

MICRO APL

APL.68000 Level II is a full specification APL2 for the 68000 family of processors. It incorporates all the user-oriented enhancements of APL.68000 Level I as well as MicroAPL's acclaimed windowing interface. APL.68000 Level II is available for computers such as the NCR Tower, Sun, HP, Bull, Nixdorf, Wicat, Mac, Amiga and ST. Features include:

- Nested and mixed arrays
- User-defined operators
- Selective specification
- Upwards compatibility with APL.68000 Level I

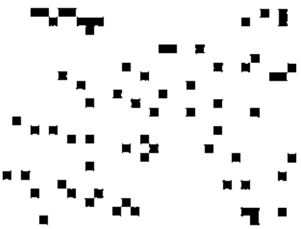
For more details contact:
MicroAPL Ltd.,
South Bank Technopark,
90 London Road, London, SE1 6LN.
071-922-8866

APL Graphics - First Principles

by Graham Parkhouse (Dept. of Mechanical Engineering, University of Surrey)

```
      ⍝ This is a demonstration of APL and computer graphics, written
interactively with my computer. Text following a six space
indentation is mine: all other text is my computer's response to my
instructions. Some of my text is prefaced by the lamp symbol, ⍝. This
is to indicate that the text following it is commentary and is to be
ignored by my computer.
      ⍝ The function PRINT displays a boolean matrix as a small-scale
bitmap. All the functions used are listed at the end of the article.
      BACKGROUND←10=?120 160⍴10
      PRINT BACKGROUND
```

```
      ⍝ BACKGROUND is a 120 by 160 boolean matrix with a sparsity of
10: of the 19200 numbers, 1920 are likely to be 1s, the rest zeros.
      +/,BACKGROUND                ⍝ Count the 1s
1888
      ⍝ Only 32 short: normal for a random process! Had I printed
BACKGROUND as 0s and 1s I would have filled ten pages. This
illustrates the efficiency of displaying information as a bitmap.
Next I am going to explore the bottom right hand corner in more
detail.
      BRHCORNER←¯24 ¯32↑BACKGROUND
      PRINT 10 MAGNIFY BRHCORNER
```

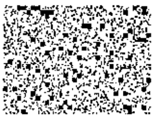


```
      ⍝ These big dots, each of 100 black pixels, are still much
smaller than the space occupied by the character 1. The wedge of
```

white space pointing up from the bottom of this pattern is visible i the bottom right hand corner of BACKGROUND.

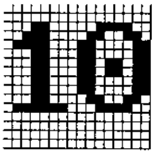
⍝ *If I think of BRHCORNER as my picture, then I can use BACKGROUND as its background by combining both matrices suitably magnified using* ∨.

```
      PRINT BACKGROUND∨5 MAGNIFY BRHCORNER
```

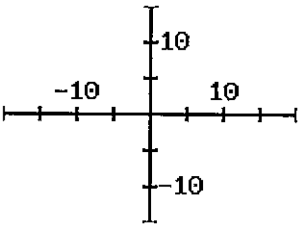


⍝ *I am going to choose PICTURE as the array I shall draw on, and with the help of the function BLOCK I can write on it as well. B assigning values to appropriate blocks of 'pixels' I can draw a set of axes onto PICTURE with the following instructions:*

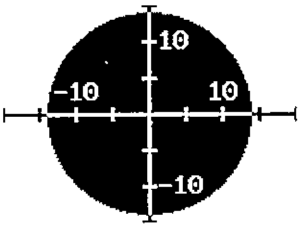
```
      PICTURE←120 160⍴0                       ⍝ Take a clean sheet
      PICTURE[60 61;]←1                       ⍝ Draw the x-axis
      PICTURE[;80 81]←1                       ⍝ Draw the y-axis
      PICTURE[56+⍳8;,1 20∘.+20×0,⍳7]←1        ⍝ Draw ticks on the x-axis
      PICTURE[,1 20∘.+20×0,⍳5;76+⍳8]←1        ⍝ Draw ticks on the y-axis
      ⍴TEN←BLOCK '10'                         ⍝ Block of pixels showing 1
16 16
      ⍴MINUSTEN←BLOCK '-10'                   ⍝ Block of pixels showing -
16 24
      BIGTEN←10 MAGNIFY TEN                   ⍝ Magnify TEN
      BIGTEN[10×⍳16;1]←1 ⋄ BIGTEN[;10×⍳16]←1
      PRINT BIGTEN                            ⍝ 16 × 16 pixel array
```



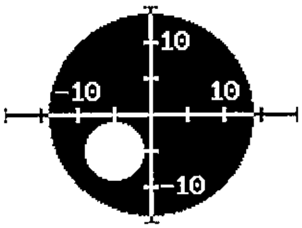
```
      PICTURE[40+⍳16;28+⍳24]←MINUSTEN         ⍝ Numbering x-axis
      PICTURE[40+⍳16;112+⍳16]←TEN
      PICTURE[92+⍳16;85+⍳24]←MINUSTEN         ⍝ Numbering y-axis
      PICTURE[12+⍳16;85+⍳16]←TEN
      PRINT 2 MAGNIFY PICTURE                 ⍝ Labeled axes
```



```
      AXES←PICTURE                              ⍝ Copy PICTURE for future use
      PICTURE←120 160⍴0                         ⍝ Take a clean sheet
      ⍝ Indexing pixels is low level APL graphics: very useful
sometimes, but rather fiddly. I now wish to draw some lines and
circles onto my PICTURE, which I wish to describe with reference to
my new axis system. I have written some functions that will enable me
to do this quite simply. They need information about my axis system,
and I have chosen the variable WINDOW to hold it. WINDOW is a 2×2
matrix holding:
      3 13⍴' XMIN | XMAX ------+------ YMIN | YMAX '
 XMIN | XMAX
------+------
 YMIN | YMAX
      ⍝ The axes I have drawn correspond to:
      WINDOW←2 2⍴¯20 20 ¯15 15
      ⍝ To register this information I call the function SETTINGS
which echoes the values in WINDOW and the size of PICTURE.
      SETTINGS
 WINDOW =  ¯20 20      ⍴PICTURE =  120 160
           ¯15 15
      ⍝ I am going to draw a solid circle centred on the origin (the
point 0 0) which has diameter 28 and is black (intensity 1).
      ADDCIRCLE 0 0 28 1
      PRINT 2 MAGNIFY AXES≠PICTURE
```

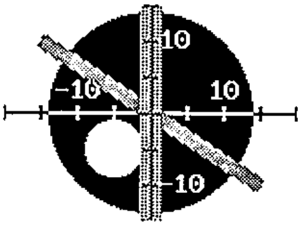


```
      ADDCIRCLE ¯5 ¯5 8 0                       ⍝ White circle, intensity 0
      PRINT 2 MAGNIFY AXES≠PICTURE
```

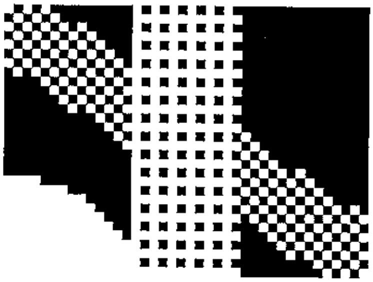


```
      ⍝ ADDLINE is a companion to ADDCIRCLE, having six components in
its argument: x1 y1 x2 y2 thickness and intensity.
      ADDLINE ¯15 10 15 ¯10 2 0.5
      ADDLINE 0 20 0 ¯20 3 0.25
```

```
        PRINT 2 MAGNIFY AXES≠PICTURE  ⍝ 2 circles overwritten by 2 lines
```

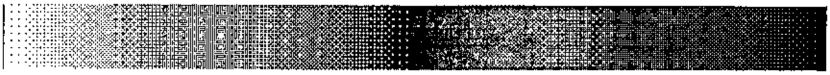


```
        ⍝ The last line was clipped by the window. Both ADDCIRCLE and
ADDLINE clip automatically.
        ⍝ I am now going to look at the middle of PICTURE magnified so
I can see the pixels clearly.
        PRINT 10 MAGNIFY PICTURE[45+⍳30;60+⍳40]
```

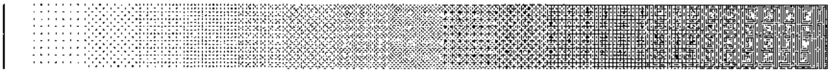


```
        ⍝ The intensities of the two lines are 0.5 and 0.25,
corresponding to a half and a quarter of the pixels being black. I
have 65 intensities available to me. The pattern for each is
described by the way the numbers 1 to 64 are distributed in the 8×8
array MASK:
        MASK
 1 33  9 41  3 35 11 43
49 17 57 25 51 19 59 27
13 45  5 37 15 47  7 39
61 29 53 21 63 31 55 23
 4 36 12 44  2 34 10 42
52 20 60 28 50 18 58 26
16 48  8 40 14 46  6 38
64 32 56 24 62 30 54 22
        ⍝ I am going to see what I can achieve by drawing a strip
across the page increasing uniformly in intensity from 0 to 1. First
I shall create a string of 901 intensities.
        INTENSITIES←(0,⍳900)÷900
        (5↑INTENSITIES),¯5↑INTENSITIES
0 0.001111 0.002222 0.003333 0.004444 0.9956 0.9967 0.9978 0.9989 1
        ⍝ Convert them to integers ranging from 0 to 64.
        INTENSITIES←⌊0.5+64×INTENSITIES
```

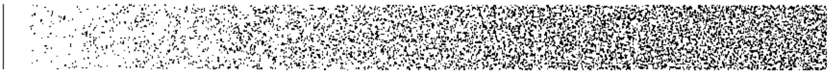
```
      (10↑INTENSITIES),¯10↑INTENSITIES
0 0 0 0 0 0 0 0 1 1 63 63 64 64 64 64 64 64 64 64
      ⍝ My strip is going to be made up of 72×901 pixels. BIGMASK is
going to be this size as well, got by repeating MASK both across and
down.
      BIGMASK←72 901⍴⍉901 8⍴⍉MASK
      PRINT 1,BIGMASK≤72 901⍴INTENSITIES
```



```
      ⍝ Notice the loss of contrast at the dark end. Two problems are
being demonstrated: isolated white pixels on a black ground are less
extensive than isolated black pixels on a white ground, a weakness of
the printing process, but even if this were not so there remains the
problem, or rather the fact, that eyes judge contrast to a different
scale. The printer problem disappears with magnification. The penalty
of magnification is sparsity of information: the same information
takes up more space. Alternatively, intensities can be kept below 0.5
or 0.25 so that black pixels never run into each other.
      INTENSITIES←0.25×(0,⍳900)÷900
      (5↑INTENSITIES),¯5↑INTENSITIES
0 0.0002778 0.0005556 0.0008333 0.001111 0.2489 0.2492 0.2494 0.2497 0.25
      INTENSITIES←⌊0.5+64×INTENSITIES
      (10↑INTENSITIES),¯10↑INTENSITIES
0 0 0 0 0 0 0 0 0 0 16 16 16 16 16 16 16 16 16 16
      PRINT 1,BIGMASK≤72 901⍴INTENSITIES
```

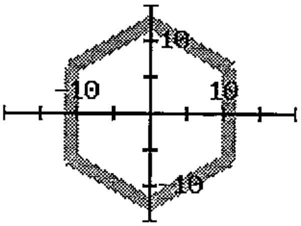


```
      ⍝ The pattern within MASK is rather beautiful and leads to a
family of pixel patterns that blend into each other very
harmoniously. A random array of integers between 1 and 64 in BIGMASK
provides a less orderly but equally versatile pattern.
      BIGMASK←?72 901⍴64
      PRINT 1,BIGMASK≤72 901⍴INTENSITIES
```



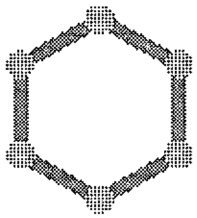
```
      ⍝ Now I shall move to a higher level, a level above ADDCIRCLE
and ADDLINE, to consider graphical objects described not as
arrangements of pixels but by their geometrical and material
structures. I have chosen to describe each object by 3 items: an
identifier, a co-ordinate array and a component array. HEXAGON is an
example of a graphical object.
```

```
      HEXAGON
  HEXAGON    10.83 10.83    0    ¯10.83 ¯10.83  0     1    1    1    1    1
              6.25 ¯6.25 ¯12.5  ¯6.25    6.25 12.5  0    0    0    0    0
              0     0       0     0        0     0    2    2    2    2    2
              1     1       1     1        1     1    0.5  0.5  0.5  0.5  0.
              1     1       1     1        1     1    1    2    3    4    5
                                                      2    3    4    5    6
      ⍝ I shall unpack HEXAGON and explain what it contains.
      ID COOR COMP←HEXAGON
      ⍴¨ID COOR COMP
 5 6  6 6
      ⍝ ID is a scalar enclosing the string 'HEXAGON'. COOR and COMP
are numeric arrays of size 5×6 and 6×6. I shall label their rows and
columns:
      ('' 'x-coor' 'y-coor' 'z-coor' 'h-coor (should be 1)' ''),(⍳6)
                         1     2       3     4       5     6
 x-coor                10.83 10.83    0    ¯10.83 ¯10.83  0
 y-coor                 6.25 ¯6.25  512.5  ¯6.25    6.25 12.5
 z-coor                 0     0       0     0        0     0
 h-coor (should be 1)   1     1       1     1        1     1
                        1     1       1     1        1     1
      ('type' '' 'thickness/dia.' 'intensity' 'end 1' 'end 2'),COMP
 type             1   1   1   1   1   1
                  0   0   0   0   0   0
 thickness/dia.   2   2   2   2   2   2
 intensity        0.5 0.5 0.5 0.5 0.5 0.5
 end 1            1   2   3   4   5   6
 end 2            2   3   4   5   6   1
      ⍝ Two rows are redundant and the h-coor row must always be
filled out with 1s. Type is 0 for circles and 1 for lines. The
numbers in the rows marked ends 1 and 2 refer to the columns of COOR
HEXAGON appears to be a hexagon having each of its sides 2.0 thick
and dark grey (intensity = 0.5). The hexagon should nearly fill the
current window, so I do not need to change WINDOW. I can draw HEXAGO
using my function DRAW.
      PICTURE←120 160⍴0
      DRAW HEXAGON
      PRINT 2 MAGNIFY AXES∨PICTURE
```

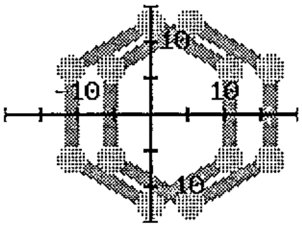


```
      ⍝ I have a set of functions that take graphical objects as
their arguments and return transformed graphical objects as their
results. An important one is AND, which collects the components in
each of the objects of its arguments and presents them as a single
```

```
object. First I shall create another object, NODES, which will be 4.0
dia. circles centred on each of the vertices of the hexagon.
      ID←⊂⊂'NODES'
      COMP[1;]←0                      ⍝ Set type to circles
      COMP[3;]←4                      ⍝ Set diameters
      COMP[4;]←0.25                   ⍝ Set intensities
      NODES←ID COOR COMP              ⍝ COOR is not changed
      OBJECT←HEXAGON AND NODES
      OBJECT[1]                       ⍝ Identifier
  HEXAGON  AND    NODES
      OBJECT[2]                       ⍝ Co-ordinates
 10.83 10.83   0    ¯10.83 ¯10.83  0    10.83 10.83   0    ¯10.83 ¯10.83  0
  6.25 ¯6.25 ¯12.5  ¯6.25   6.25 12.5  6.25 ¯6.25 ¯12.5  ¯6.25   6.25 12.5
  0     0      0     0      0     0     0     0      0     0      0     0
  1     1      1     1      1     1     1     1      1     1      1     1
  1     1      1     1      1     1     1     1      1     1      1     1
      OBJECT[3]                       ⍝ Components
1   1   1   1   1   1   0    0     0     0     0     0
0   0   0   0   0   0   0    0     0     0     0     0
2   2   2   2   2   2   4    4     4     4     4     4
0.5 0.5 0.5 0.5 0.5 0.5 0.25 0.25  0.25  0.25  0.25  0.25
1   2   3   4   5   6   7    8     9    10    11    12
2   3   4   5   6   1   8    9    10    11    12     7
      PICTURE←120 160⍴0
      DRAW OBJECT
      PRINT 2 MAGNIFY PICTURE
```



```
      ⍝ TRANSLATE and ROTATE operate on graphical objects.
      RINGS←OBJECT AND 5.41 0 ¯4 TRANSLATE OBJECT
      PICTURE←120 160⍴0
      DRAW RINGS
      PRINT 2 MAGNIFY AXES∨PICTURE
```

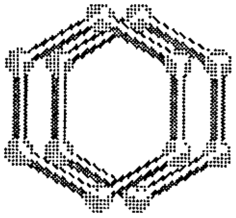


⍝ *Note that it is the object that has been translated, not the axis system nor the window. The z translation of ¯4 has affected the z co-ordinates of RINGS but has no effect on what DRAW does. DRAW uses only the x and y co-ordinates: it draws an orthogonal projection of the object onto the x-y plane.*

⍝ *Perspective projection can be performed using the function PROJECT which projects the co-ordinates of its right hand argument through the point STN onto the x-y plane at the origin. PROJECT returns a graphical object possessing not just the correct x,y co-ordinates but also the z co-ordinate of the middle of each component in the redundant row 2 of its component array. DRAW uses this row to order the elements before drawing them so that, provided STN has positive z, elements are drawn from the back of the scene towards the front. This permits hidden line removal.*

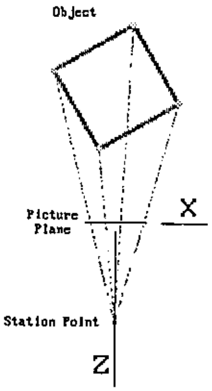
⍝ *Before advancing into 3-D, I want to add realism to my lines and circles to make them look like cylindrical rods and spheres. Function HIGHLIT will do this by providing a white reflection roughly in the middle and slightly in front of each component.*

```
DRAW PROJECT RINGS AND HIGHLIT RINGS
PRINT 2 MAGNIFY PICTURE
```

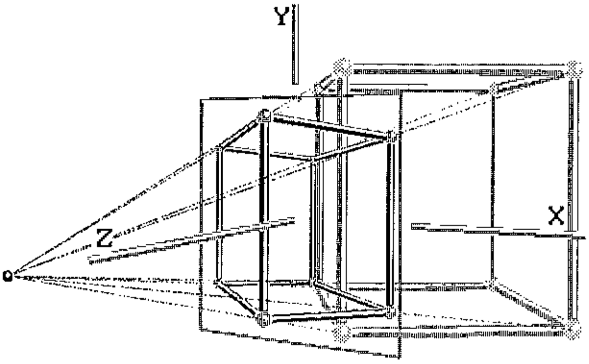


⍝ *I also have a function HALO that puts a thicker white component just behind each component to blot out some background to help components stand out.*

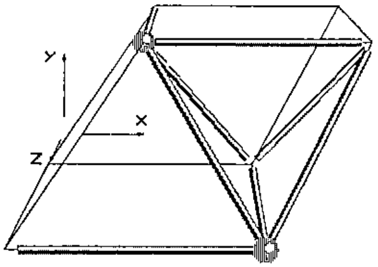
```
PRINT PLAN
```



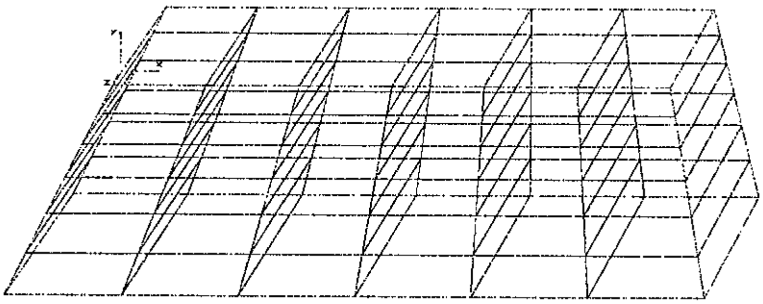
```
      ⍝ PLAN is a projection onto the z-x plane and shows a typical
arrangement for a perspective drawing of an object, in this case a
box. The picture plane is always the x-y plane through the origin:
the system cannot entertain any other arrangement. The station point
STN can be located anywhere, but in this instance is on the z-axis.
The corners of the box are projected onto the picture plane by the
four faint lines.
      PRINT PERSPECTIVE
```



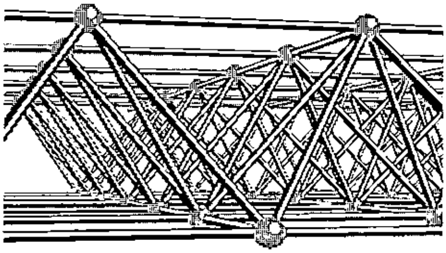
```
      ⍝ PERSPECTIVE is a perspective of the perspective projection
layed out in PLAN, viewed from the isolated point in the bottom right
hand corner of PLAN. There are now eight faint construction lines,
and the window is shown surrounding the drawing.
      ⍝ Much of what I illustrate either includes mathematical
curiosity or demands engineering precision. My APL graphics functions
are my sophisticated tools, while my simple tools are raw APL, which
can help me both at a lower and at a higher level. PERSPECTIVE was
drawn with the help of a little lower level manipulation, and now I
shall draw a latticed deck using the power of raw APL for high level
data definition.
      WINDOW←2 2⍴¯5 20 ¯7.5 7.5
      PICTURE←300 500⍴0
      STN←7.5 15 50
      DRAW PROJECT XYZ AND BOX AND CELL AND HIGHLIT CELL
      PRINT PICTURE
```



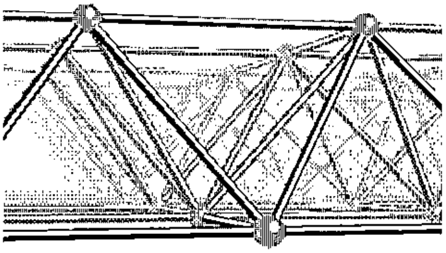
```
      ⍝ CELL, which contains 8 lattice members and two nodes, is a
convenient unit cell for a latticed deck. BOX marks the boundary of
the unit cell and XYZ shows the axes. The components of CELL can be
distinguished from the rest by the highlighting.
      ⍝ Before I replicate CELL I shall practise the replication on
BOX. BOX has a top and bottom which is 10 units square. To replicate
it I need to TRANSLATE it by multiples of 10 units in the x and z
directions. In Dyalog APL ⍳ is well suited to helping me do this:
      ⍳2 3 4
 1 1 1  1 1 2  1 1 3  1 1 4
 1 2 1  1 2 2  1 2 3  1 2 4
 1 3 1  1 3 2  1 3 3  1 3 4

 2 1 1  2 1 2  2 1 3  2 1 4
 2 2 1  2 2 2  2 2 3  2 2 4
 2 3 1  2 3 2  2 3 3  2 3 4
      ⍴⍳2 3 4
2 3 4
      ⍝ So ,(10×¯1+⍳6 1 6) will give me 36 translations appropriate
to a 6×6 deck. Applying these to BOX:
      STN←STN×6                     ⍝ Stand back 6 times further
      WINDOW←2 2⍴¯15 80 ¯30 10
      PICTURE←400 950⍴0
      DRAW PROJECT XYZ AND ⊃AND/,(10×¯1+⍳6 1 6)TRANSLATE¨⊂BOX
```



```
      ⍝ Repeating this replication on CELL and calling the result OB:
      OB←⊃AND/,(10×¯1+⍳6 1 6)TRANSLATE¨⊂CELL
      ⍝ Translate the deck towards the origin, rotate it 0.25 radians
about the y axis, and translate it back again:
      OB←35 0 35 TRANSLATE 0 0.25 0 ROTATE ¯30 0 ¯35 TRANSLATE OB
      STN←32 2 100
      WINDOW←2 2⍴10 50 ¯7 18
      PICTURE←300 480⍴0
      DRAW PROJECT (HIGHLIT OB) AND OB
      PRINT PICTURE
```



```
      ⍝ This is a close-up of the latticed deck. Much of the
structure has been clipped by WINDOW.
      ⍝ The last facility I want to demonstrate is HAZE. HAZE is a
two row matrix containing z-values in its top row and intensity
factors in its bottom row. Currently HAZE has just one column:
      HAZE
¯100
   1
      ⍝ All intensities of components having z-values greater than
¯100 had their intensities multiplied by 1. If I now define HAZE as:
      HAZE←2 20⍴(¯20+4×⍳20),0.8*⌽0,⍳19
      2 ¯7↑HAZE
36      40      44      48      52     56    60
 0.2621  0.3277  0.4096  0.512  0.64  0.8  1
      PICTURE←300 480⍴0
      DRAW PROJECT (HIGHLIT OB) AND OB
      PRINT PICTURE
```



```
      ⍝ Now components further back, with smaller z-values, have had
their intensities reduced more, giving an attractive haziness to the
picture.
      ⍝ All the funtions that have been used in this demonstration
are listed below. They have been run under Dyalog APL Version 6.0 on
a 33 MHz 386 PC with 8 Mb of memory. The original was printed on A4
paper by a Kyocera P-2000 PostScript laser printer, with graphics
mode set to 150 dots per inch.
      )OFF
```

```
 ADDCIRCLE CIRCLE;CENTRE;RADIUS;INTENSITY;PANE;BLACK;SIGN;⎕CT
⍝ ADDS A CIRCLE TO THE PICTURE
 CENTRE←CIRCLE[1 2] ⋄ RADIUS INTENSITY←0.5 1×CIRCLE[3 4]
⍝CALCULATE THE PANE FRAMING THE CIRCLE
 PANE←(CENTRE-RADIUS),[1.5]CENTRE+RADIUS
⍝INTERSECT IT WITH APERTURE
 ⎕CT←1E¯12 ⋄ PANE←((APERTURE⌈PANE),APERTURE⌊PANE)[;1 4]
 →(¯1 ¯1≢×-/PANE)/0                            ⍝NO INTERSECTION
 PANE←SIGN×⌊(SIGN←2 2⍴1 ¯1 ¯1 1)×PFX+.×1⍪PANE  ⍝PIXEL LIMITS
 →(∨/0>1 ¯1×-/PANE)/0                          ⍝NO INTERSECTION
⍝ LIST OF PIXEL INDICES
 PANE←,(PANE[1;2]+0,⍳-/PANE[1;])∘.,PANE[2;1]+0,⍳--/PANE[2;]
⍝ LIST OF PIXELS WHOSE CENTRES ARE INSIDE THE CIRCLE
 PANE←(RADIUS≥(+⌿((XFP+.×1⍪⍉↑PANE)-⍉((⍴PANE),2)⍴CENTRE)*2)*0.5)/PANE
 →(0=⍴PANE)/0                                  ⍝ IF NONE, EXIT
⍝ IS INTENSITY 0 OR 1?
 →(~∨/0 1=INTENSITY)/GREY ⋄ PICTURE[PANE]←INTENSITY ⋄ →0
GREY: ⍝ WHICH PIXELS ARE BLACK?
 BLACK←MASK[1+(⊂⍴MASK)|PANE]≤⌊0.5+INTENSITY××/⍴MASK
 PICTURE[BLACK/PANE]←1 ⋄ PICTURE[(~BLACK)/PANE]←0

 ADDLINE LINE;⎕TRAP;A;AB;AP;AT;APTR;B;BLK;C;D;D1;EDGES;I;IND;INTEN...
 ⎕TRAP←⊂3 'E' '→TRAP'                                  ...;L;M;N;PFT;S
⍝ ADDS A LINE TO THE PICTURE
 AB←⍉2 2⍴4↑LINE ⋄ D INTEN←¯2↑LINE ⋄ D÷←XFP[1;3] ⍝ D IN PIXEL UNITS
⍝ TRANSFORM TO P,Q HOMOGENEOUS COORDINATES
 AB←(PFX+.×1⍪AB)⍪1 ⋄ APTR←(PFX+.×1⍪APERTURE)⍪1
⍝ CALCULATE TRANSFORMATION MATRICES PFT AND TFP
 AP←AB,(AB[1 2;1]+1 ¯1×⌽-/AB[1 2;]),1         ⍝P,Q COORDS OF A,B,C
 AT←⍉3 3⍴0 0 1 ⋄ AT[(1 2)(2 3)]←L←(+/(-/AB[1 2;])*2)*0.5  ⍝T,N COORDS
 TFP←⌹PFT←AP+.×⌹AT
⍝ CALCULATE T COORDS OF I/SECN OF EACH LINE EDGE WITH EACH SIDE OF APTR
 APTR←APTR,⌽APTR ⋄ APTR[2;]←APTR[2;1 1 2 2] ⋄ APTR←↓⍉TFP+.×APTR
 EDGES←↓⍉3 4⍴0 0,L,L,(¯1 1 1 ¯1×D÷2),1 1 1 1
 T←(EDGES CROSS¨1⌽EDGES)[2 4]∘.CROSS APTR CROSS¨1⌽APTR
 T←(↑T)[;;1 3] ⋄ T[;;2]←OZ T[;;2]    ⍝OFFSET ZEROS IN 2ND COL.
 T←4 2⍴÷/T ⋄ T←T⍪CS T ⋄ AB←(⌈/0,⌊⌿T),⌊/L,⌈⌿T
 →(0≤-/AB)/0                         ⍝ NO INTERSECTION
⍝ SET UP SATURATION BOMBING GRID
 SC←|2⍴PFT ⋄ A←÷+/SC ⋄ D1←(÷A)-A××/SC ⋄ N←1⌈1+⌈(D-D1)÷A
 B←(D-D1)÷1⌈N-1 ⋄ L←|-/AB ⋄ M←1+⌈L÷A ⋄ A←L÷M-1
 IND←(M,N,3)⍴1 ⋄ IND[;;1]←⍉(N,M)⍴A×0,⍳M-1 ⋄ IND[;;2]←(M,N)⍴B×0,⍳N-1
 IND[;;1]+←1↑AB ⋄ IND[;;2]-←0.5×B×N-1 ⋄ IND←⍉((M×N),3)⍴IND
 →(N>1)/GOON ⋄ IND←((D÷D1)>0.001×M⍴RND←M⌽RND)/IND
GOON:I←↓[1]¯1 0↓IND←⌊0.5+PFT+.×IND ⋄ →CONT
TRAP: ⍝REMOVE THOSE OUTSIDE THE WINDOW
 C←(IND[1;]>0)∧(IND[1;]≤1↑⍴PICTURE)∧(IND[2;]>0)∧IND[2;]≤¯1↑⍴PICTURE
 I←↓[1]¯1 0↓C/IND
CONT:→(0=⍴I)/0        ⍝ NO PIXELS, EXIT
⍝ IS INTENSITY 0 OR 1?
 →(~∨/0 1=INTEN)/GREY ⋄ PICTURE[I]←INTEN ⋄ →0
```

```
GREY: ⍝ WHICH PIXELS ARE BLACK?
 BLK←MASK[1+(⊂⍴MASK)|I]≤⌊0.5+INTEN××/⍴MASK
 PICTURE[BLK/I]←1 ⋄ PICTURE[(~BLK)/I]←0
 R←A AND B;COMP
⍝R IS AN OBJECT WHICH CONTAINS THE COMPONENTS OF BOTH A AND B
 A[1]←⊂(↓' ',↑⊃A[1]),⊂'AND' ⋄ B[1]←⊂↓' ',↑⊃B[1]
 COMP←⊃B[3] ⋄ COMP[5 6;]+←¯1↑⍴⊃A[2] ⋄ B[3]←⊂COMP
 R←A,¨B
```

```
 R←A AND B;COMP
⍝R IS AN OBJECT WHICH CONTAINS THE COMPONENTS OF BOTH A AND B
 A[1]←⊂(↓' ',↑⊃A[1]),⊂'AND' ⋄ B[1]←⊂↓' ',↑⊃B[1]
 COMP←⊃B[3] ⋄ COMP[5 6;]+←¯1↑⍴⊃A[2] ⋄ B[3]←⊂COMP
 R←A,¨B
```

```
 R←BLOCK TEXT
 R←(16,8×⍴,TEXT)⍴2 1 3⍉IMAGE[CHAR⍳,TEXT;;]

 R←A CROSS B
⍝ CALCULATES THE CROSS PRODUCT OF A AND B
 R←((1⌽A)×¯1⌽B)-(¯1⌽A)×1⌽B

 R←CS B
⍝ RETURNS R AS B EXCEPT FOR V. LARGE NOS. WHOSE SIGNS ARE REVERSED
 B←,R←B ⋄ B[(1E12<|B)/⍳⍴B]×←¯1 ⋄ R←(⍴R)⍴B

 DRAW OBJECT;ID;COOR;COMP;C;I;N;ORDER;P;Q
⍝ DRAWS AN X-Y PLANE PROJECTION ONTO THE BOOLEAN PICTURE MATRIX
⍝ IT IS A COVER FUNCTION FOR ADDCIRCLE AND ADDLINE
 SETTINGS
 ID COOR COMP←OBJECT ⋄ ⎕←((⍴ID),1)⍴ID
 ORDER←⍋COMP[2;] ⋄ I←1
LOOP: ⍝ PROCESS NEXT COMPONENT
 N←ORDER[I] ⋄ C←COMP[;N] ⋄ P←÷⌿COOR[2 2⍴1 2 4 4;C[5]]
 →C[1]⌽CIRCLE,LINE
CIRCLE:ADDCIRCLE P,C[3 4] ⋄ →CONT
LINE:Q←÷⌿COOR[2 2⍴1 2 4 4;C[6]]
 ADDLINE P,Q,C[3 4]
CONT:⍞←1↑C[1]⌽'CL' ⋄ →((⍴ORDER)≥I←I+1)/LOOP
 ⎕←''

 R←HIGHLIT OBJECT;ID;COOR;COMP;F;I;CCOOR;CCOMP;LCOOR;LCOMP
⍝ PRODUCES THE HIGHLIGHTS FOR OBJECT
 F←3     ⍝ FACTOR DIA OF OBJECT ÷ DIA OF HIGHLIGHT
 ID COOR COMP←OBJECT
 ID←ID,⊂'HIGHLIGHTED'
⍝ CIRCLES
 I←(0=COMP[1;])/⍳¯1↑⍴COMP
 CCOMP←COMP[;I] ⋄ CCOMP[3;]÷←F ⋄ CCOMP[4;]←0 ⋄ CCOMP[5;]←⍳⍴I
 CCOOR←COOR[;COMP[5;I]]
 CCOOR[⍳3;]+←(3,⍴I)⍴(÷4×3*0.5)×CCOOR[4;]×COMP[3;I]
```

```
⍝ LINES
 I←(1=COMP[1;])/⍳¯1↑⍴COMP
 LCOMP←COMP[;I] ⋄ LCOMP[3;]÷←F ⋄ LCOMP[4;]←0
 LCOMP[5 6;]←⍉((⍴I),2)⍴(¯1↑⍴CCOOR)+⍳2×⍴I
 LCOOR←COOR[;,⍉COMP[5 6;I]]
 LCOOR[⍳3;]+←(3,2×⍴I)⍴(÷4×3*0.5)×LCOOR[4;]×,⍉COMP[3 3;I]
⍝ PACKAGE THE RESULT
 R←ID(CCOOR,LCOOR)(CCOMP,LCOMP)

 R←F MAGNIFY A
⍝ MAGNIFIES THE BOOLEAN MATRIX A BY AN INTEGER FACTOR F
 R←(F×⍴A)⍴2 4 1 3⍉(F,F,⍴A)⍴A

 R←MOD A
 R←(+/A*2)*0.5

 R←OZ B
⍝ RETURNS R AS B EXCEPT FOR ITS ZEROS, WHICH GO BACK VERY SMALL
 B←,R←B ⋄ B[(1E¯12>|B)/⍳⍴B]←1E¯12 ⋄ R←(⍴R)⍴B

 R←PROJECT OBJECT;ID;COOR;COMP;C;L;M;I
⍝ PROJECTS THROUGH A STATION POINT,STN,ONTO THE X-Y PLANE
 ID COOR COMP←OBJECT ⋄ ID←ID,⊂'PROJECTED THROUGH ',⍕STN
 COOR[5;]←÷/COOR[3 4;]                      ⍝ CALC. DISTANCES
⍝ CALCULATE TRANSFORMATION MATRIX
 M←12⍴0 ⋄ M[1 3 6 7 11 12]←(STN,1)[3 1 3 2 4 3]
 M←3 4⍴M ⋄ M[;3]×←¯1
 COOR[1 2 4;]←M+.×COOR[⍳4;]                 ⍝ TRANSFORM COOR
⍝ CALCULATE COMPONENT DISTANCES
 C←(0=COMP[1;])/I←⍳¯1↑⍴COMP ⋄ L←(1=COMP[1;])/I
 COMP[2;C]←COOR[5;COMP[5;C]]                 ⍝ CIRCLES
 COMP[2;L]←0.5×+/COOR[5;COMP[5 6;L]]  ⍝ LINES
⍝ MODIFY THICKNESSES
 COMP[3;]×←STN[3]÷STN[3]-COMP[2;]
⍝ MODIFY INTENSITIES
 COMP[4;]×←(1,HAZE)[2;1++/COMP[2;]∘.>HAZE[1;]]
⍝ PACKAGE THE RESULT
 R←ID COOR COMP

 R←THETA ROTATE OBJECT;ID;COOR;COMP;S;T;MODT
⍝ ROTATES AN OBJECT THROUGH THE PSEUDOVECTOR THETA
 S←3 3⍴7 6 2 3 7 4 5 1 7
 S←(THETA,(-THETA),0)[S]                        ⍝ AUXILIARY MATRIX
 MODT←MOD THETA
 T←(3 3⍴1 0 0 0)+((1○MODT)÷MODT)×S
 T+←0.5×(((1○0.5×MODT)÷0.5×MODT)*2)×S+.×S  ⍝ TRANSFN MATRIX
 ID COOR COMP←OBJECT
 ID←ID,⊂'ROTATED BY ',⍕THETA
 COOR[⍳3;]÷←COOR[3⍴4;]
 COOR[⍳3;]←T+.×COOR[⍳3;]
 R←ID COOR COMP
 R←SETTINGS;M;N;X1;X2;Y1;Y2
⍝ CHECK WINDOW: X2>X1 AND Y2>Y1
 WINDOW←(⌊/WINDOW),[1.5]⌈/WINDOW
⍝ SHOW SETTINGS
 R←'WINDOW ='(⍕WINDOW)
 R,←'    ⍴PICTURE ='(⍕⍴PICTURE)
⍝ CALCULATE TRANSFORMATION MATRICES  XFP AND PFX
 X1 X2 Y1 Y2←,WINDOW ⋄ M N←⍴PICTURE
 XFP←2 3⍴0
 XFP[1;1 3]←¯0.5 1×(X2-X1)÷N
 XFP[2;1 2]←0.5 ¯1×(Y2-Y1)÷M
 XFP[;1]+←X1,Y2
 PFX←¯2 3↑⌹1 0 0⍪XFP
⍝ CALCULATE APERTURE
 APERTURE←WINDOW+(2 2⍴1 ¯1)×⍉2 2⍴0.5×((X2-X1)÷N),(Y2-Y1)÷M

 R←D TRANSLATE OBJECT;ID;COOR;COMP
⍝ TRANSLATES THE OBJECT BY THE 3 COMPONENTS OF D
 ID COOR COMP←OBJECT
 ID←ID,⊂'TRANSLATED BY ',⍕D
 COOR[⍳3;]+←COOR[3⍴4;]×⍉((¯1↑⍴COOR),3)⍴D
 R←ID COOR COMP
```

The PRINT function files its output which may be flushed to the printer by KYOFLUSH.

```
 {F}PRINT A;CMD;SIZE;X;Y;⎕IO
⍝ FUNCTION TO PRINT TO KYOCERA P-2000 PRINTER
⍝ IF A IS A MATRIX, IT IS TREATED AS BOOLEAN, AND PRINTED AT 300÷F DPI
⍝ 0 AND 1 REPRESENT WHITE AND BLACK       ⎕IO IS LOCAL AND SET TO ZERO
 ⎕IO←0
 ⍎(0=⎕NC'prt')/'KYOINIT'                 ⍝ INITIALISE PRINTER
 ⍎(A≡'')/'prt ⎕FMT''FF''⋄→EXIT'          ⍝ FORM FEED IF A IS EMPTY
 →(2=⍴⍴A)/BOOM ⋄ A←POST_esc¨↓⎕FMT⊃A     ⍝ ESCAPE SPECIAL CHARACTERS
 A←(⊂'NL (')'¨A,¨⊂') show' ⍝ NL FOLLOWED BY EACH LINE IN PARENTHESES
 prt¨⎕FMT¨A ⋄ →EXIT
BOOM:⍎(0=⎕NC'F')/'F←2'        ⍝ SET DEFAULT VALUE OF F AS 2
 A←I2H A                      ⍝ TRANSFORM BOOLEAN TO HEX CHAR MATRIX
 prt(1,⍴CMD)⍴CMD←'/bitmap <',(,A),'> def' ⍝ DEFINE bitmap
 X Y←4 1×⌽⍴A                              ⍝ DIMS OF bitmap
 CMD←⊂'gsave'                             ⍝ BEGIN SCRIPT
 CMD,←⊂'/DEPTH ',(⍕F×Y),' 300 div INCH PITCH add def'
 CMD,←⊂'VPOS DEPTH sub BM lt { FF } if'
 CMD,←⊂'/VPOS VPOS DEPTH sub def'
 CMD,←⊂'LM VPOS translate'
 CMD,←⊂(⍕X F),' mul 300 div INCH'
 CMD,←⊂(⍕Y F),' mul 300 div INCH scale'
 CMD,←⊂(⍕X Y),' 1 [',(⍕X,0 0 ¯1 0 1×Y),'] {bitmap} image'
 CMD,←⊂'grestore'
 CMD,←⊂'NL'
 SIZE←⍴CMD←↑CMD ⋄ CMD←,CMD
 CMD[(CMD='¯')/⍳⍴CMD]←'-'
```

```
  prt SIZEρCMD
 EXIT:→0

  KYOINIT;X;TM;LM;BM;PGHEIGHT;PITCH;FONTSIZE
 ⍝ INITIALISES KYOCERA P-2000 PRINTER
  'prt'⎕SH'prt'
  prtargs'postscript' '' 'APLPOST.PRN'
  TM←1.1                  ⍝ Top margin      (in.)
  LM←1                    ⍝ Left margin     (in.)
  BM←1                    ⍝ Bottom margin   (in.)
  PGHEIGHT←11.65          ⍝ Page height     (in.)
  PITCH←13                ⍝ Pitch in points (1/72 in.)
  FONTSIZE←11
  X←⊂'/INCH {72 mul} def'
  X,←⊂'/TM ',(⍕TM),' INCH def'
  X,←⊂'/LM ',(⍕LM),' INCH def'
  X,←⊂'/BM ',(⍕BM),' INCH def'
  X,←⊂'/PGHEIGHT ',(⍕PGHEIGHT),' INCH def'
  X,←⊂'/PITCH ',(⍕PITCH),' def'
  X,←⊂'/TOF { /VPOS PGHEIGHT TM sub def LM VPOS moveto} def'
  X,←⊂'/FF { showpage TOF } def'
  X,←⊂'/NL { /VPOS VPOS PITCH sub def'       ⍝ 2 LINE DEFN
  X,←⊂'VPOS BM gt {LM VPOS moveto} { FF } ifelse } def'
  X,←⊂'/APL-2741 findfont ',(⍕FONTSIZE),' scalefont setfont'
  X,←⊂'TOF'
  prt↑X

  KYOFLUSH
 ⍝ Flush information to printer
  prt 1 2ρ'FF'
  prtoff
  0 0ρ⎕SH'copy c:\dyalog\fonts\apl2741.fnt+aplpost.prn lpt1'

  R←I2H B;SHH;SHR
 ⍝ Return boolean as hex array, 1 bit per sample.
 ⍝ N.B. <FF> is white in PostScript!!
  B←0=((1↑ρB),8×⌈0.125×¯1↑ρB)↑B ⋄ SHR←(ρB)÷1 4
  ⍎(0=⎕NC'hex')/'''xutils'' ⎕SH '''''     ⍝ Invoke xutils ap if no hex
  SHH←((1+ρR←36↓hex B)÷9),9 ⋄ R←SHRρ(SHH-0 1)↑SHHρR

  R←POST_esc R              ⍝ Set each row of text for printing
  ((R∊'()\')/R),¨←'\'       ⍝ Escape all of '(', ')', '\'  with '\'
  R←⊃,/R                    ⍝ Return simple vector
```

Lower-case functions such as prt and hex come with Dyalog APL. ADDLINE requires a global variable RND. Execute RND←?100ρ1000 before first calling ADDLINE.

For the past five years I have been sponsored by the Venture Research Unit of BP International Ltd, to whom I am most grateful. I also wish to thank the authors of Dyalog APL and many friends within the APL community for their support and encouragement.

GENERAL ARTICLES

This section of VECTOR is oriented towards readers who may neither know APL, nor be interested in learning it. However we hope you are curious about how, under the right conditions, such impressive results can emerge so quickly from APL programmers

Writing Assembly Language Functions for ⎕NA

by Allan Gay

Look. I know the Reverend. I've read his sermons on ⎕NA in Vector and trembled. I mean, don't get me wrong - I'm a religious man myself - but I prefer to do my crusading in carpet slippers and with a good book in hand. Well - not that Good Book exactly. Not APL2 Programming: System Services Reference for heaven's sake!

Sigh.

I'm not making much sense, am I? I suppose I'd better start at the beginning.

It is an ancient Programmer and he stoppeth one ...

Evening was drawing in and I was sprawled in a chair with my heels up on the mantelpiece, toasting the old buns, when the 'phone rang. With one convulsive leap, I switched off the budgie, catapulted myself to the desk and snatched up the handset before the second ring. "Hi", I said, as nonchalantly as I could manage.

"My son," said a voice, "how dost thou fancy writing some ⎕NA Assembler functions for APL2?" It was the Reverend.

Well, it sounded like work and you know me and work (I could watch it for hours, etc.) But the old annual fitness rating was imminent and I knew the Reverend was big in the Synod, so I gave his challenging (read gruesome) proposal the big hello. "Your Grace," I said, "you know me and work".

I thought he sniffed, but it may have been line noise. "Thy mission - shouldst thou choose to accept it - is to produce three partitioned functions, namely Or Reduce, Or Scan and LessThan Scan. They will be dyadic, handle only Boolean vectors and will produce Boolean vector results. And," he said, "one assumes thou wilt also supply a FirstGroup routine if time permits."

Next morning, I tried some deep thinking but it was no good. I'd already taken compassionate leave to bury every elderly aunt which I possessed - some of them three times - and there had been barbed comments in the office of late about the marked and ongoing reduction in the population of northern England. It had to be faced: I was cornered.

So I dug out a pile of old hymnals, fired up the Shareware 370/Assembler simulator on the old PS/2 and did some heavy coding. The Reverend seemed to be looking for some pretty startling reductions in CPU time and I for one wasn't going to disappoint him.

Piece of cake ...

Well, you know, after reading an old sermon or two, dashed if things seemed all that bad. I'd a fair idea of what the inputs and outputs would have to look like in CDR format, the interface details between APL2 and me seemed a pretty straightforward piece of groined cross-vaulting apart from the odd ECV or so, the actual logic to do the bit manipulations would all be done in the cathedral organ's registers for reasons of speed and tone quality and - most important of all - I had faith.

So, when I presented myself at the tradesman's entrance a couple of days later I was feeling pretty holy. I had the relevant tracts with me, the Assembler prayers were on diskette and all in all I thought it was going to be a piece of cake.

While I used a magnetic card at the holy water vending machine in the south transept, an acolyte asperged and fumigated my diskette and checked it for impure thoughts. Then the prayers were copied into the bowels of the cathedral organ and battle commenced.

Now I don't know if you've ever groped blindly down a drain in an attempt to pick up a lead weight with a piece of limp string and a sweaty sock. Take it from me, the TSO TEST facility isn't like that at all. Heavens no! When you are running APL2 under TEST, it's more like using a B-52 to strangle a chicken. What I mean is there's a strong chance that you'll overshoot the target area and have to go round again. And that's no joke after a 7,000 mile flight from the initial LOAD command, believe me.

I had some initial troubles. I admit that. But I knew that once I'd decoded the cryptic words of my guide, the aforementioned System Servi es Reference, I'd crack the interface and it would be all plain sailing. After all, the algorithms were implemented entirely within the registers. I mean, what could possibly go wrong with that?

The troubles mounted until, after two days, I came to doubt that my guide was - how shall I put it? - strictly au fait with all the nuances. I based this suspicion on the fact that whenever I followed its advice my prayers, instead of winging heavenwards, tumbled on the mat with their paws in the air. Consequently, it came as something of a relief to learn from the Reverend that my guide had been excommunicated in 1987. Yes, the true faith was being expounded nowadays by APL2 Programming: Processor Interface Reference and, with its help, I got my prayers to implement FUNCTION linkage (the fanciest, detailedest and most sanctified type) to the point where they were returning explicit APL results at last.

The real stuff...

Now at this point, some of you fellow pilgrims out there will want to know the magic formula. Frankly, it seems pretty unfair that you chaps should be purified by my suffering but, heck, I'm going to tell you. In a word: it's services, and I don't mean Matins. The services I'm talking about are the interpreter's VP, VQ, XC and XG services. Look, will someone at the back push the kids and those crazy BASIC fans out of the room? This is grownups' talk. Thanks.

Okay, here's the real stuff. First off, every time the APLer runs the name-associated function, we get called with a perfectly normal MVS parmlist hanging off register 1. Dangling off the parmlist, there are pointers to the System Services routine, the ECV, and a gash, zero-filled, persistent doubleword which can be used to remember things between successive runs of our function.

It's good style to be re-entrant so that we can serve multiple communicants from the one chalice. Of course, cutting down on the crockery means we've got to watch the hygiene aspect. We aren't allowed to modify ourselves, so we can't embed work [illegible] in our routine but must instead acquire storage for them dynamically. We use [illegible] VP service for this and APL2 makes it easy for us by bolting a VP parameterlist into the initial parameterlist. Just pop in the byteage required, call System Services and bingo! After VP has delivered the goods by giving us a slice of the area reserved by APL2's FREESIZE parameter, we store the slice's address into the gash doubleword. Every time we're called, we check the current instance of that doubleword. If it's zero, we do a VP; if it's not, we use it to reach the area we acquired in a previous life. We may not be able to take it with us, but we certainly can get it back afterwards; our camel slinks around the eye of the needle.

Whenever we get called, there's a request code in the ECV to tell us what's going on. If it's a delete-linkage request and we previously got an area from VP, we use the VQ service to free it and then we grab a couple of hotel towels, use our credit cards and check out. The holiday is over. This is either because the APLer has just)OFFed himself or because he has severed our name association. Otherwise, it's a function-linkage request, meaning the APLer is on the blower and there's work to be done.

Although I've just given you the goods on getting and managing work area, I wouldn't like you to think I knew it all along. Golly, no! I'd read and misunderstood a remark to the effect that storage acquired via VP is automatically reclaimed when the process terminates. Believing, erroneously as it turned out, that my prayers were processes within the meaning of the act and that termination meant returning control to APL2, I initially refrained from any use of the VQ service, nor did I use the gash doubleword to remember where my area was. Instead, assuming that each previous execution's storage had been released, I blithely requested storage afresh at each invocation of my prayers. The eventual effect was to consume all the freespace and wreck the application.

You at the back ...

Right, I'll pause here and take a few questions. You, sir. Whassa what? Whassa ECV? Thassa big control block which contains all sorts of information about the function arguments, together with a parameterlist for the XG service and various items of session info. Using its contents in conjunction with the XC and XG services, we can access the input arguments, get room in the workspace for the result and, when we've constructed it, return it to APL by connecting it to this ECV.

Whassa CDR, did I hear you say? That stands for Common Data Representation and it's just the descriptive information for an APL variable. You know, its rho, rank, type, and so forth. By examining a variable's CDR, we can find our way to its data and also know how to interpret that data when we get there. It can get pretty complex but there are ways of forcing input variables into a straitjacket so that we don't have to program every conceivable possibility. My prayers handled Boolean vectors exclusively. We also have to construct a CDR for our result variable.

Yes, madam? How do you find out about the ECV's format? There are diagrams in the Processor Interface Reference and there's an AP2ECV macro which maps it and gives mountains of documentation. Speaking of macros, you should all note that there's an AP2CDR macro, too. You'll probably find those macros in an APL2.AP2MACS library on one of the DLIBs; check with the sysprogs. Oh, and if I suddenly start to speak in tongues, the first three letters of each word will be CDR or ECV and you'll know I'm talking about fields defined by the macros.

It's good to see you writing all this down, because there's going to be a short catechism afterwards.

The moving target ...

That covers the working storage aspect, so let's now take a look at creating variables. There are various ways of playing it but I'm just going to tell you the way I did it. You see, there are lots of choices because the APL2 environment isn't static. As APL objects are created, modified and deleted, things get moved around by the interpreter to reclaim released areas and generally defrag the workspace. This means you can start creating a variable, go back later to wrap it up, and find it has moved elsewhere. Obviously, while your ⎕*NA* routine is actually running, this doesn't happen. But, when you make certain calls to the interpreter, any workspace addresses you are currently working with will probably be invalidated. ECVXRLOC contains a relocation count which will have changed if a garbage-collect has occurred.

This mutability aspect is a pest but APL2 gets round it by using tokens. Now when APLers talk of tokens they are usually discussing APL statement parsing, but that's not what we mean here. Just get the idea that every separately addressable object in the workspace has a unique badge. That means each variable's descriptor (CDR) and each related string of data has a unique token which is unaffected by workspace defrag activities. APL2's XC service exists to supply the current virtual storage address for any valid token we care to supply. APL2's XG service, which allocates space in the workspace (as opposed to VP, which allocates it in the freespace) returns not only the address of the allocated space but also its token. APL2 can distinguish between addresses and tokens, so you can pass either to APL2 interchangeably. What APL2 will pass to you varies according to circumstances and I preferred to inspect such pointers and find out for myself.

The rule for pointers is: if the high-order bit is on, we have a virtual storage address which we can use as it stands. If the bit is off, we have a token and we need to use XC to get the equivalent virtual storage address. This finds application when we process the input variables. Their respective CDRs' tokens reside in ECVXCDRL (left-argument) and ECVXCDRR (right-argument). Within each of those CDRs is a pointer to the argument data.

If you are just creating a simple result, the strategy is straightforward. First, you inspect the input arguments, determine the maximum possible size your result could be, use XG to get a suitably-sized piece of the workspace in which to build the result data, and note its token. Second, you manufacture the data in that area, using the virtual storage address from XG to access it. Third, you use XG again to get more workspace in which to manufacture the result's CDR. Fourth, you build the CDR and you put the retained token of the data area into CDRPTR, in effect hooking the data to the descriptor. Finally, you put the CDR area's token into ECVXCDRZ to make the result available to APL2 when you cede control.

Now if you look at that carefully, you'll see that we always complete using one area before we call XG to get another. That means the virtual storage addresses supplied by XG are never invalidated until we've finished using them. And because we retain the tokens supplied by XG, we're able to connect everything together without needing to use XC at all. In fact we don't even need to look at ECVXRLOC to see whether things have moved, either.

Getting it wrong ...

A while back, I observed that the algorithms were all implemented in the cathedral organ's registers and I - rather unwisely as it turned out - used the expression "piece of cake". I also asked, rhetorically, "what could possibly go wrong with that?" Read on, brothers and sisters! Read on!

When I at last got the prayers to produce explicit results, I thought my troubles were over. So when I heard from the Reverend that those results were occasionally wrong I was not at all gruntled. If they had been wrong all the time, I could have handled that, but subtle intermittent errors are bad news. I tried putting the prayers in the fridge for a few hours in case the problems were temperature-related but it was the Reverend who spotted the cause: early bunkout.

Okay, I have a letter here from a listener who asks "What the heck is early bunkout for goodness sake?" I won't read the rest of the letter because the style deteriorates and he gets abusive, (cheese it, pal!) but I'd better explain. To do so, I need to describe the way in which the algorithms were implemented and I also need to explain where we get our performance improvements. Bear with me a moment.

Why and how ...

The whole idea of writing these routines was to get things to go faster. If you look at the Reverend's partitioned functions they are extremely terse but, inevitably, they use several APL2 primitive functions. Those primitive functions don't know that the Reverend intends to deal only with simple Boolean vectors, so they have to be prepared to field anything that comes along, which means they are overqualified for the job. Also, they produce intermediate results along the way which have to be allocated and deallocated, with the potential for a workspace defrag at any stage. Finally, APL2 is not aware that they are being used to do, say, a Partitioned OR Scan.

Conversely, our Assembler prayers are highly-specific to the expected data, are atomic, and have inside knowledge about the task. Consequently, they are smaller, they have less overhead and they can cheat. (Early bunkout is a form of cheating.) Even better, we can do almost everything in registers and avoid much of the overhead of accessing virtual storage.

The organists among you will know that the top-end IBM cathedral organs have sixteen 32-bit general purpose registers which, unlike those in the 80x86 series harmonium, lack almost all flavour and colour. By this, I mean that you can use any of them for almost anything, so there's a freedom that you just don't find on smaller instruments. In some of the prayers, I even preloaded constant comparands into spare registers for use within loops. As you can imagine, by eliminating repeated storage fetches of comparands, this expedited comparisons no end.

The basic processing technique, once the input arguments and the output result area had been connected up, was to lope along the inputs 32-bits (a fullword) at a time. Each pass of the loop entailed loading a fullword of left-argument data into one register, a fullword of right-argument data into another, and using a third register in which to generate result data. Apart from the Partitioned OR Reduce prayer, which could generally be expected to produce a result which contained fewer bits than the input, all prayers produced results of exactly the same length as their inputs. Consequently, at the end of each pass of the loop, another fullword of result bits would be stored in the output variable's data area.

My big mistake ...

Let's take a look at one of the algorithms - Partitioned LessThan Scan - in detail. A LessThan Scan produces a result which is a copy of the input argument in which all onbits except the first have been turned off. Partitioned LessThan Scan does this for each partition in the right-argument input data; the partitions are defined by onbits in the left-argument mask data. For instance:

```
      1 0 0 1 0 0 0 1 0 PARTITIONED_LT_SCAN 0 1 1 0 0 1 1 1 1
0 1 0 0 0 1 0 1 0
```

In this example, the left-argument mask data specifies three partitions of lengths 3, 4 and 2 bits respectively. In real life, however, we're talking about thousands of partitions each of which could be anything from one to thousands of bits long. Because the cathedral organ's registers go like greased lightning, we can happily loop along, shifting register contents to isolate each bit in turn, inspecting it and deciding what to do with it. We have no alternative. Or do we? The cathedral organ's repertoire includes many facilities which act on a whole fullword at once.

Imagine what would happen if, having already output the first onbit in the current partition, we loaded the next fullword of left-argument mask data and found it to be all zeroes. It would mean that the current partition continued for at least another 32 bits and, since we'd already output the single onbit permitted for it, we'd have to output 32 offbits. The nature of the corresponding fullword of right-argument data would be completely immaterial. Instead of looping through the 32 bits one at a time, we could store a zeroed fullword with a single instruction. This trick of segment-skipping can pay performance dividends when long partitions occur frequently in the input.

What's more, at any time during the processing of a nonzero maskword on a bit-by-bit basis, if a test reveals that all remaining bits are off, we can ignore the remaining dataword bits and store offbits for them in the result. This trick of loop-truncation can pay performance dividends when data onbits are infrequent.

This sort of trickery is the early bunkout to which I previously alluded.

There's another loop-truncation trick we can play, and this one is where I tripped myself up. Let's suppose that we've looked at the maskword and we find that there are some onbits in it. This means we've got a new partition starting within the next 32 bits so we can't do a segment skip. But suppose all the databits were off? We can't produce onbits in the result if the corresponding databits are off, so couldn't we store zeroes and bunk out early in that case?

The answer is that we could, but with one important proviso: first we must reset the partition metabit, that status flag which records whether we have yet found an onbit in the current partition's data. If we were to go into the next partition with the flag still on, we'd produce an all-zero output partition even if the input partition data contained some onbits. Guess who forgot to reset the partition metabit.

Connecting things up ...

Okay, we're on the homewards stretch now. All that remains is to recount how to make the prayers available to the APLer in his pew and to put some numbers on the resulting CPU-time savings.

The first thing we have to do is to package our prayers as one or more loadmodules. How many we put in a loadmodule is immaterial because the packaging process entails coding and assembling a frontend table which lists all the entrypoints in the individual routines. With this table in place, the usual limit (imposed by the Linkage Editor) of 16 entrypoints within a single loadmodule is eliminated. Instead, the loadmodule ends up with a single entrypoint via which APL2 enters to look at the table when it wants to name-associate a given function. The number of routines which may be declared in the table is potentially huge.

In the project under discussion, there were only four routines and the application used all four, so they were packaged as a single loadmodule.

To announce each routine to APL2, we create a member in a partitioned dataset termed the Names file. Each member names its corresponding routine and gives a DDname and membername to enable the package loadmodule to be found. Each member also uses a simple shorthand to declare the format of the input arguments and the result. Argument data supplied by the APLer to our functions will be coerced to suit if at all possible; otherwise, a DOMAIN ERROR occurs and we aren't called. This saves us from the chore of writing complex input-validations.

Before starting the application, we preallocate the Names file and the loadmodule library (the latter using the predetermined DDname coded in the Names file members). Then we hook each routine up as an APL function by entering

```
  '(NamesfileDDname)' 11 ⎕NA 'function'
```

After this, the function runs exactly as if it were written in APL. But a lot faster.

The score ...

So how did they do? Partitioned OR Reduce was the worst. Overall, it averaged a CPU-time saving of 44%. This means that, on average, for every 100 CPU seconds taken by the all-APL function, the Assembler version took 56 CPU seconds on the same data.

The other three functions did much better. They each saved 92%. Altogether, replacing these four much-loved utility routines knocked nearly 10% off the overall CPU-time cost for the entire application. Since it was a very large and heavily used application, this represented a very substantial reduction in machine costs.

All four functions were pretty much the same size. The average was 870 bytes and the total size of the package loadmodule was 3,592 bytes. The source code per function came to about 320 lines, excluding comments and macro expansions.

Development time averaged about 35 man hours per function. This reflects the learning curve and initial blind man's buffery. The last function (Partitioned FirstGroup) was taken from inception to completion in only four hours by taking a copy of the Partitioned LessThan Scan function and writing a new logic core, the argument handling and environmental processing being the same. If it's got Boolean vectors, I'm your man.

The aftermath ...

Hosannas ringing in my ears, I totter across the cathedral close. Back home, curtains drawn, I slump in the gloom with an icepack on my head. The 'phone rings. Wincing, I lift the handset. "Hi", I croak feebly.

"My son," says a brightly enthusiastic voice, "how dost thou fancy doing a Partitioned Plus Reduce?"

Mandelbrot Sets

by Ray Cannon

Let me say first of all, that this article has very little to do with APL. I did write some Mandelbrot code in APL for an IBM PC compatible, but it took too long to calculate even the simplest areas.

The reason it takes so long is that a single image 480 x 480 pixels in size, requires 1,200,000,000 odd floating points instructions.

Question: What is the Mandelbrot set? Answer: It is the set of all the connected Julian sets. (Ask a silly question and...)

OK, think of a number, square it, add the number you originally thought of (2,4,6), square it, add first number again (36,38), square it , add the first number (1444,1446)... Nothing much strange there, the result rapidly rises. Try -1 (-1,1,0,0,-1,1,0...), or zero (0,0,0,0...). All numbers seem to fall into these 2 groups. Either the number shoots off towards infinity, or the result stabilises in a cycle (which may only have one element).

Now try it with complex numbers, plotting each starting point on the complex plane (x/y graph) as either black (cycles) or white (shoots off). The result is a Mandelbrot diagram. (To speed things up, please note that ALL points outside the circle of radius 2 starting at the origin 0,0 are white, so only points INSIDE this circle need be considered.)

Thus the Mandelbrot set can be described via the transformation of z goes to z squared plus c, where c is a complex number.

To produce a picture of part of the Mandelbrot set, follow the following steps:

1) Decide on the number of (pixels) points in your picture (say 480x480) and map these onto a part of the "complex plane" within the region -2.0 to +0.5 on the real (x) axis, and -1.25 to +1.25 on the imaginary (y) axis.

2) Each pixel represents a point within a small section on the complex plane. Loop for each point in turn, taking its co-ordinates as the value for "c", with a starting value for z of zero.

3) For each "c", calculate the value of z1 by squaring z and adding c.

4) Reset z to the value of z1 and repeat step 3).

5) Continue doing steps 3) and 4) until either the absolute value of z is greater than 2, or the loop count is greater than an arbitrary value (say 256).

6) Set the colour of the pixel dependent on the loop count. Plot the point in black if it repeated, or reached the maximum value chosen.

7) Go on to the next point.

See the function called *MANDOT* for a specification in APL of steps 3, 4 and 5. (I have written *MANDOT* in a very simple manner which may become apparent later.)

One method of speeding things up is to utilise the fact that large areas in any one picture contain the same value, and as the MANDELBROT set is "connected" there are no "local maxima or minima". (Please don't ask me to prove that statement.)

From this it became apparent to me, that if all the points on the boundary of an area have the same value, then all the points within the boundary have the SAME value as the boundary. This can be used in a kind of "binary search" type algorithm.

So, define a square within the area to be mapped, (say 128 x 128 pixels). Calculate the "values" on the boundary, and if they are all the same, fill the complete area with that value.

If boundary has not the same value, divide the square into 4 smaller squares (64 x 64) and repeat the process on each of the smaller squares. Although the process can be repeated until the square is 2x2, due to the overheads involved with this process, squares of about 8 x 8 may not be worth sub-dividing.

This method reduces the number of calculations required in the BLACK area from an X squared to an X times 2 (plus) where X is the size of the square "filled". (I know the square has 4 sides, not 2, but boundaries between adjacent squares are common and only need to be calculated once.)

As I have said, APL is to slow to do this work in a reasonable length of time. So I wrote my MANDELBROT calculator in C with the MANDOT function written in PC assembler using the floating point maths co-processor.

I did find it useful however, to model the 80387 floating point co-processor assembler code in APL. This let me write and test the basic assembler code within an APL environment. The top level function *MAN* can be compared with the resulting assembler listing.

(NOTE on 80387 maths chip. The 80387 chip works on the principle of a floating stack, and has 8 internal stack registers. Most instructions act between the top of the stack and other stack registers. Programming it is a bit like programming a pocket calculator in "reverse polish notation".)

The APL listing and the resulting assembler code are shown below.

Notes on Functions

```
     Functions      Comment
     ---------      -------
     MANDOT         Normal (but very simple) APL
     MAN            Produces same result as MANDOT, calls FP fns
     FADD           Floating point FADD machine code instruction simulator
     FLD            Floating point FLD (load from stack)
     FLDM           Floating point FLD (load from memory) etc.
```

```
;assembler to do the MANDELBROT calculation. called from C
;C calling syntax  "man(x,y)"
;return mandot value for given U and V values
@CODE      SEGMENT BYTE PUBLIC 'CODE'
MAN      PROC near
      push bp
      mov bp,sp
      fninit             ;clear stack
      xor ax,ax          ;clear AX
;stack constant 4
      fld q[val4]        ;Push 4 into stack                     4
;get x,y u and v from parameter string
      fld q[bp+0c]       ;            stack                    y4
      fld q[bp+04]       ;            stack                   xy4
      fld 1              ;            stack                  vxy4
      fld 1              ;            stack                 uvxy4
      xor cx,cx          ;Clear registers
      xor dx,dx
      mov dl,0ff         ;Initialise loop control
      mov bx,1
MLOOP:                   ;Main loop
;;;;;;;;;;;;;;;;;;;;;;;;;;calc y2       stack               uvxy4
      fld 3              ;push y        stack              yuvxy4
      fmul 0,0           ;y2            stack           (y2)uvxy4
;;;;;;;;;;;;;;;;;;;;;;;;;;calc x2
      fld 3              ;push x        stack          x(y2)uvxy4
      fmul 0,0           ;x2            stack       (x2)(y2)uvxy4
      fxch 4             ;exchange      stack       x(y2)uv(x2)y4
;;;;;;;;;;;;;;;;;;;;;;;;;;new Y value calculations
      fadd 0,0           ;calc 2x       stack   (2x)(y2)uv(x2)y4
      fmul 0,5           ;calc 2xy      stack  (2xy)(y2)uv(x2)y4
      fadd 3             ;calc 2xy+v    stack       Y(y2)uv(x2)y4
      fstp 5             ;store new Y   stack        (y2)uv(x2)Y4
;;;;;;;;;;;;;;;;;;;;;;;;;;new X value calculations
      fld 3              ;push x2       stack    (x2)(y2)uv(x2)Y4
      fsub 0,1           ;calc x2-y2    stack (x2-y2)(y2)uv(x2)Y4
      fadd 2             ;calc x2-y2+u  stack       X(y2)uv(x2)Y4
      fxch 4             ;store new X   stack       (x2)(y2)uvXY4
;;;;;;;;;;;;;;;;;;;;;;;;;;compare  x2+y2 > 4.0
      faddp 1,0          ;calc y2+x2    stack        (y2+x2)uvXY4
      fcomp 5            ;comp vs 4.0   stack               uvXY4
;;;;;;;;;;;;;;;;;;;;;;;;;;check answer and loop or exit
      fnstsw ax
      and ax,04100
      jz FOUND            ;Found to be 4 or more
      inc bx              ;increment loop control
      cmp bx,dx           ;test against loop max
      JNE CS:MLOOP        ;next loop
;;;;;;;;;;;;;;;;;;;;;;;;;;Not found so return 0
BLACK:
      xor ax,ax
      jmp FIN
;;;;;;;;;;;;;;;;;;;;;;;;;;Found so return count
FOUND:
      xor ax,ax           ;clear return value
      mov AL,BL           ;ignore top byte
FIN:
      mov sp,bp
      pop bp
      ret
```

APL Functions

```
      ∇ FADD ARGS
[1]   ⍝30387 FADD
[2]    ARGS←¯2↑ST[0],ST[ARGS]
[3]    STACK[0]←+/ARGS
[4]    ST←8↑STACK
      ∇
```

```
      ∇ FADDP ARGS;RES
[1]   ⍝30387 FADDP
[2]    RES←+/ST[ARGS]
[3]    STACK[ARGS[0]]←RES
[4]    STACK←1↓STACK
[5]    ST←8↑STACK
      ∇
```

```
      ∇ FLD MEM
[1]   ⍝ 30387 FLD
[2]    STACK←ST[MEM],STACK
[3]    ST←8↑STACK
      ∇
```

```
      ∇ FLDM VAL
[1]   ⍝30387 LOAD FROM MEMORY
[2]    STACK←VAL,STACK
[3]    ST←8↑STACK
      ∇
```

```
      ∇ FLDZ
[1]   ⍝30387 FLDZ
[2]    STACK←0,STACK
[3]    ST←8↑STACK
      ∇
```

```
      ∇ FMUL ARGS
[1]   ⍝30387 FMUL
[2]    ARGS←¯2↑ST[0],ST[ARGS]
[3]    STACK[0]←×/ARGS
[4]    ST←8↑STACK
      ∇
```

```
      ∇ FNINIT
[1]   ⍝30387 FINIT
[2]   ⍝INIT FLOATING POINT
[3]    STACK←⍳0
[4]    ST←8↑STACK
      ∇
```

```
      ∇ FSTP ARG
[1]   ⍝30387 FSTP
[2]    STACK[ARG]←STACK[0]
[3]    STACK←1↓STACK
[4]    ST←8↑STACK
      ∇
```

```
     ∇ FSTPM NAME
[1]    ⍝30397 FSTP STR TO MEMORY
[2]     ⍎NAME,'←ST[0]'
[3]     STACK←1↓STACK
[4]     ST←8↑STACK
     ∇
```

```
     ∇ FSUB ARGS
[1]    ⍝30387 FSUB
[2]     ARGS←¯2↑ST[0],ST[ARGS]
[3]     STACK[0]←-/ARGS
[4]     ST←8↑STACK
     ∇
```

```
     ∇ FXCH ARGS
[1]    ⍝30387 FXCH
[2]     STACK[ARGS[0]]←ST[ARGS[1]]
[3]     STACK[ARGS[1]]←ST[ARGS[0]]
[4]     ST←8↑STACK
     ∇
```

```
∇ R←U MAN V;MAT;COUNT;ST;STACK
[1]    ⍝SIMULATE ASS MANDELBROT
[2]     R←0
[3]     MAT←0
[4]     COUNT←0
[5]     FNINIT
[6]     FLDM 4
[7]     FLDZ
[8]     FLD 0
[9]     FLDM U
[10]    FLDM V
[11]   LP:
[12]    FLD 3
[13]    FMUL 0 0
[14]    FLD 3
[15]    FMUL 0 0
[16]    FXCH 0 4
[17]    FADD 0 0
[18]    FMUL 0 5
[19]    FADD 3
[20]    FSTP 5
[21]    FLD 3
[22]    FSUB 0 1
[23]    FADD 2
[24]    FXCH 0 4
[25]    FADDP 1 0
[26]    FSUB 5
[27]    FSTPM'MAT'
[28]    →(MAT≥0)/FOUND
[29]    COUNT←COUNT+1
[30]    →(COUNT<256)/LP
[31]   FOUND:R←COUNT
     ∇
```

```
∇ R←V MANDOT U;X;Y;X2;Y2;AL
[1]    ⍝RETURN MBT VALUE FOR U V
[2]    ⍝LONG WINDED LISTING -
[3]    ⍝COMPARE AGAINST MAN
[4]     AL←1
[5]     X←X2←U
[6]     Y←Y2←V
[7]    LP:X2←X2×X
[8]     Y2←Y2×Y
[9]     Y←Y×X
[10]    X←Y
[11]    Y←Y+X
[12]    Y←Y+V
[13]    X←X2
[14]    X←X-Y2
[15]    X←X+U
[16]    X2←X2+Y2
[17]    X2←X2-4
[18]    →(X2≥0)/END
[19]    AL←AL+1
[20]    →(AL>255)/END
[21]    X2←X
[22]    Y2←Y
[23]    →LP
[24]   END:R←AL
     ∇
```

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

A Nesting Editor for STSC's APL*PLUS/PC

by Olle Evero (Evestic AB)

Though nested variables were first introduced in 2nd generation APL, nested functions had been there all the time. The concept of nesting is actually central to control flow in APL. Typical APL code contains functions calling functions, calling yet other functions - all the activity that is monitored by the State Indicator.

This boxes-within-boxes approach is of course absolutely trivial, almost a natural law of computing. Maybe the triviality accounts for its lack of support in the average APL development platform.

However, do not fear, rescue is at hand... I give you (drum roll, please): *NED*, the Nesting Editor! *NED* is a simple cover to the STSC ⎕*EDIT* command, that will help you explore and maintain an application in a natural, 'nested' way.

Picture yourself using ⎕*EDIT* to browse through the main function of a system, called *MAINMENU*. Suddenly (in the middle of line 13, say), you stop dead in your tracks. Before you is a reference to another function *FOO44*. Of course, you need to know what it contains.

Normally, you would have to exit the editor and start another editing session on *FOO44* to find out. But since you are using *NED*, you just place the cursor over *FOO44* and press ctrl-E (or ctrl-Q). This takes you immediately into ⎕*EDIT* with *FOO44*. After finishing browsing *FOO44*, simply place the cursor on an empty space in the function and press ctrl-E once more. This returns you to *MAINMENU*, at the exact spot where you left (line 13, remember?).

Well, there you have the whole idea: navigating the code structure of an application in a fashion similar to the application itself. Here follows the APL:

```
    ∇ A NED B;C;D;E;F;G;H;K;N;⎕ELX;⎕WINDOW

       ⍝------------------------------------------------------------
       ⍝ Nesting Editor for APL objects. Example call: NED 'MAINMENU'
       ⍝ A = ambivalent argument, unassigned at invocation
       ⍝ B = valid APL object name
       ⍝ ⎕IO assumed to be 1                      Olle Evero, Evestic AB
       ⍝------------------------------------------------------------

        ⎕ELX←'⎕SOUND 140 200⋄→LX' ⍝      beep and exit at error
        ⎕WINDOW← 0 0 24 80 ⍝             set window
        →(2=⎕NC 'A')/L0 ⍝                skip init status string if A exists
        A←B ⋄ →L1 ⍝                      init status string, skip append
    L0:A←B,' ',A ⍝                       append status string if A assigned
    L1: 24 0 1 80 ⎕WPUT 80↑A ⍝           put status string onto status line
        →(0≠⎕NC B)/L15 ⋄ ⎕INBUF 262 70 ,¯1+⎕AV⍳B ⍝ new object changed to fn
   L15:⎕EDIT B ⋄ →(0 2 3 =⎕NC B)/LX,L2,L3 ⍝ edit and branch acc to type
    L2:C←⍎B ⋄ C←(¯2↑ 1 1 ,⍴C)⍴C ⋄ →L4 ⍝      if variable,make matrix
    L3:C←⎕CR B ⍝                             if function,canonize
    L4:D←1+(⎕PEEK 191 192),256⊥⎕PEEK 194 193 ⍝ last cursor pos in ⎕EDIT
        D[1]←D[1]-6×⎕PEEK 179 ⍝                  subtract 6 if line numbers
        N←¯1++/D[2 3] ⍝                          calculate row in object
        →(N>1↑⍴C)/LX ⍝                           if out of bounds, exit
        E←'∆_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789' ⍝ char candidates
        E←E,'abcdefghijklmnopqrstuvwxyz⍙' ⍝             for APL name
        F←C[N;] ⋄ G←F∊E ⍝  produce row, and bit vector of char candidates
        K←(3=⎕NC B)∧~(F[1]='⍝')∨':'=1↑(~G)/F ⍝ nonlabel,noncomment rows in fn
        G←(K⍴0),G ⍝ allow for indentation of nonlabel,noncomment fun rows
        H←K↓(⌽∧\⌽D[1]↑G),1↓∧\(¯1+D[1])↓G ⍝ find name string
        H←H/F ⋄ →(0∊⍴H)/LX ⍝                 if no name, exit
        A NED H ⍝                            else recursive edit session
        ⎕INBUF(¯1+D[3 2 1])/ 421 420 389 ⍝ buffer cursor moves
        →L1 ⍝                                return to 'old' edit session
    LX: ⍝                                    exit
    ∇
```

Some Usage Notes:

- *NED* is recursive; you can follow a whole chain of function calls from top to bottom, then return to the top again.

- In the absence of programmable function keys in ⎕*EDIT*, *NED* operation is controlled by where the user puts the cursor. If there is a character under the cursor at exit, this character and the adjacent characters will be interpreted as a valid APL name, and a new ⎕*EDIT* session on this name will be invoked. If there is no character under the cursor at exit, you will return to the calling environment. This protocol may take some getting used to.

- You can use *NED* to build a system from scratch. After filling your main function with references to subfunctions, just place the cursor on any one of them, press ctrl-E and start coding it. No need to memorize a plethora of names - and it saves typing too!

- Screen line 25 is used to line up the names that are passed to *NED*. Think of this as ⎕*SI* raveled, if you will.
- The input buffer (used by ⎕*INBUF*) does not normally hold more than 256 characters. This could be a problem if you tend to write functions exceeding 200 lines or so. On the other hand, if you write such huge functions, this may well be the least of your problems...
- *NED* operates on functions and text matrices alike.
- Peeks and pokes are used as indicated in documentation pertaining to the somewhat obsolete version 7.1 of APL*PLUS/PC. You might want to check their applicability to the version you have got.

At Evestic, we have compiled a library of utilities related to the STSC APL*PLUS/PC product. The focus is on novel ideas and items otherwise hard to come by. There is guaranteed to be no assembler or locked code, just short stand-alone APL functions. Please drop a line to the following address, if you want more information on this.

Evestic AB, Attn: Olle Evero
Frejgatan 6/B-133
114 79 Stockholm
SWEDEN

Ed: This is a super idea, and in most cases it works very well, however there is one interesting snag: if any lines in the edit stack are wrapped by ⎕*EDIT, it fails to return to the right place! This is a particular problem if you write functions with lots of local variables, and hence long header lines.*

Resolution of this little dilemma is left as an exercise for Vector readers ... answers on the back of a postcard please in time for 8.1, and we will print the best in the conference special. Thanks.

Thoughts on *J←f g h*

by Maurice Jordan

Introduction

These notes arise from ideas first stimulated by Iverson's presentation at APL88 on yoke (which allows among other things infinite sets to be handled simply in APL), and function assignment e.g.

```
        sum←+/
```

Yoke has been superseded by the phrasal forms hook and fork (APL89) and these are now implemented in J (APL90). These phrasal forms assign meaning to phrases such as

```
        J←f g h
```

where f g and h are functions. Powerful though the concepts are, are they the best use of this phrasal form?.

Experiments using Dyalog APL

Function assignment has been available in Dyalog APL since their introduction of defined operators in 1986. When used with operators it becomes possible to build quite complex functions in a simple sequence of assigned functions.

As soon as I got my hands on a Beta-release version of Dyalog/386, I began experimenting with fork and hook. Hook is simply expressed as an operator:

```
 ∇ Z←{A}(f hook g)B ⍝ Elidable left args must be enclosed in {}
  ⍝∇ Iverson/McDonnell's hook phrasal form.
  ⍝ Z←→A f g B (dyadic); B f g B (monadic)
   →(0≠⎕NC 'A')↓L1 ⋄ A←B
  L1:Z←A f g B
 ∇
```

But the trouble with fork is that it needs a mechanism for deriving an operator or a way of passing more than two functions to an operator. I quickly found that I could use Bob Smith's NARS composition operator as implemented in Dyalog (analogous to & in J) to bind 3 functions so that they looked like one

```
        f∘g∘h
```

and thus fool the interpreter. Once inside the operator, the internal representation can be examined:

```
        fgh←+/∘÷∘⍴
        ∆DISP ⎕CR 'fgh'  ⍝ ∆DISP is a utility to display nested
                         ⍝ structure

 ┌─────────┬─┬─┐
 │┌──┬─┬─┐ │ │ │
 ││+/│∘│÷│ │∘│⍴│
 │└──┴─┴─┘ │ │ │
 └─────────┴─┴─┘
```

and in simple cases (where f g and h are primitive) the original f g and h can be reconstructed easily:

```
  ∇ Z←{A}(fns fork)B;f;g;h;FNS
   ⍝ Iverson/McDonnell's phrasal forms idea.
   ⍝      fns←→f∘g∘h; Z←({A} f B) g ({A} h B)
   ⍝ f g h are primitives here.
    FNS←⎕CR'fns'
    f←⍎⊃⊃FNS ⋄ g←⍎3⊃⊃FNS ⋄ h←⍎ 3⊃FNS ⍝ Monadic ⊃ is ↑ in APL2
    →(0=⎕NC'A')↑Monad ⋄ Z←(A f B)g A h B ⋄ →0
   Monad:Z←(f B)g h B
  ∇
```

This allows

```
        mean←+/∘÷∘⍴ fork
        mean 1 2 3 4 5
  3
```

I was able to experiment enough with these operators to realise that their combination with function assignment leads to a powerful tool.

A Better Way

This representation of fork depends on a fudge. The fudge is clumsy and difficult to generalise so that it deals with defined, derived-from-defined, and assigned functions. Instead of the composition primitive, relying on undocumented internals (therefore liable to change), why not regain control and use a defined operator?

```
∇ Z←(f j_ g)B
 ⍝∇ A Join operator to stack functions together for operator
 ⍝  calls and unstack them within the operator.
  →(B=0)↓L1 ⋄ Z←f ⋄ →0
 L1:Z←g
∇
```

This enables two (or more) functions to be joined together for syntactic binding in operator calls, and provides an easy means of unscrambling them again within the operator. This is used in the next generation of the fork operator:

```
∇ Z←{A}(fgh J_)B;fg;f;g;h
 ⍝∇ Performs J's fork construct when fgh is of form f j_ g j_ h
  ⍎(0=⎕NC 'A')/'A←right'
  fg←fgh 0 ⍝ fgh must be of form f j_ g j_ h
  f←fg 0
  g←fg 1
  h←fgh 1
  Z←(A f B)g(A h B)
 ⍝:⍺ MHJ 07Sep90
∇


      mean←+/ j_ ÷ j_ ⍴ J_
      mean 1 2 3 4 5
3
```

right is simply the right identity from Dictionary APL:

```
∇ B←{A}right B
 ⍝∇ Right identity (⊢ in Dictionary APL)
∇
```

as this is not a primitive in Dyalog APL. The idea of assigning an identity function for elided left arguments in ambivalent functions and operators comes from Phil Last. What a pity we can't write

```
 ∇ Z←{A←⊢}(fgh J_) B
```

to supply a default for an elided left argument and avoid clumsy conditional executes or computed gotos.

Even Simpler

A simpler construction, requiring only one call of j_ results from

```
 ∇ Z←{A}(fg Fork h)B;f;g
  ⍝∇ A simpler form of fork depending on the 'stack' operator j_
   ⍎(0=⎕NC 'A')/'A←right'
   f←fg 0 ⍝ fg must be of form f j_ g
   g←fg 1
   Z←(A f B)g A h B
  ⍝:⍺ MHJ 07Sep90
 ∇


        mean←+/ j_ ÷ Fork ⍴
        mean 1 2 3 4 5
 3
```

Other Forms of Fork

Other forms of fork can be derived from

```
 ∇ Z←{A}(f strand g)B
   ⍎(0=⎕NC'A')/'A←right'
   Z←(A f B)(A g B)
 ∇


        mean←⊃∘(÷/)∘(+/ strand ⍴) ⍝ rather unlovely
```

or

```
        mean←⊃∘(÷/)∘(+/ j_ ⍴ fnarray)
```

where fnarray is an operator to simulate Benkard's function array ideas but lacks the elegance to be shown here.

Back to the Yoke

Although forks may be simulated in these ways their usage lacks the elegance achieved by simply, but tediously, building a series of defined operators to cover the *g* functions in the `f  g  h` fork (as in the original yoke idea) e.g.

```
∇ Z←{A}(f and g)B
 ⍝∇ AND operator for proposition functions f and g
  ⍎(0=⎕NC'A')/'A←right'
  Z←(A f B)∧A g B
∇


∇ Z←{A}(f or g)B
 ⍝∇ OR operator for proposition functions f and g
  ⍎(0=⎕NC'A')/'A←right'
  Z←(A f B)∨A g B
∇
```

so e.g.

```
        integer_matrix←integer and matrix
```

This approach has the advantage that with very little added complication an early exit and/or error handling may be provided, allowing `DOMAIN ERROR` to be avoided in functions such as

```
        probability←(≤∘1) and (≥∘0) and numeric
```

Alternative Meaning for `f  g  h`

The experimentation with fork, hook and function assignment thus made possible confirm the expressive power of these constructs. However, much is due to the composition operator, this operator very soon starts to get in the way with simple functions such as matrix:

```
        matrix←⊃∘2∘=∘⍴∘⍴
```

which can be applied to any array:

```
        any_array←1∘left
```

APL allows me to assign from the right in a simple expression:

```
        Z←f g h A

        Z←f Y←g X←h A
```

When assigning functions, why can't I write expressions such as

```
        Z←(fgh←f (gh←g h))A
```

In other words, why shouldn't APL allow me to write

```
        matrix←⊃2=⍴⍴
        any_array←1⊣
```

In this context dyadic functions with left arguments are regarded as deriving monadic functions. Commute can be used for the rarer cases where we want a monadic function derived from a dyadic function and right argument e.g.

```
        spe←∊⍨ ⍝ 'spe'←⌽'eps'
```

For example, rewriting some J assignments from Vector 7.1 p 72 with APL equivalents:

```
  cos=. 2o.              ⍝ cos←2○
  g=. %&180              ⍝ g←÷∘180 ⍝ or g←180÷⍨
  rfd=. o.g              ⍝ rfd←○g
  cosd=. cos rfd         ⍝ cosd←cos rfd
  sind=. 1o.o.(%&180)    ⍝ sind←1○○180÷⍨
```

This last expression was originally

```
  sind=. 1&o.&(o.&(%&180))  ⍝ sind←1∘○∘(○∘(÷∘180)) ]
```

I have found that insertion of composition (and parentheses) into phrases for function assignment is often non-trivial. Yet, with the exception of cases like the definition of g above where an alternative is available through the commute operator, the expressions can be read simply if composition and parentheses are ignored. Why not provide the facility to write them without composition? Unfortunately this is in conflict with fork and hook. My experience so far is that fork and hook are a (not insignificant) minority of useful function assignments. For my purpose, it is more useful to use the yoke concept for fork, even if it means writing a family of operators which could have been derived from yoke.

Conclusion

Although composition (∘ in Dyalog and NARS, or & and other operators in J) is the basis for a lot of useful assigned functions and other constructions, it quickly ends up in the way. It should be possible to reduce it to an elided operator with a suitable definition of APL syntax. My impression is that there is a language of exceptional beauty simplicity and power struggling to emerge. Which constraint on our current thinking must be broken to release it?

References and Further Reading

[1] Iverson *A Commentary on APL development* Presentation at APL88 published in QuoteQuad 19.1

[2] Iverson, McDonnell *Phrasal Forms* APL89 Proceedings

[3] Hui,Iverson,McDonnell,Whitney *APL \ ?* APL90 Proceedings

[4] Iverson *J* Vector 7.1

[5] Cherlin *APL Trivia* APL90 Proceedings

[6] Benkard *Nonce Functions* APL90 Proceedings

[7] Benkard *Structural Experiments with Arrays of Functions* APL85 Proceedings

Maurice Jordan
91 Coldershaw Road,
West Ealing,
London W13 9DU

Full Screen Methods with APL2

by Peter Branson

Introduction

This is mostly concerned with APL2 on the mainframe and, although there is a widespread tendency to move APL applications down to PC's or the RS 6000, etc., there is still plenty of mainframe code out there. In most cases, APL mainframe systems are likely to have been around a while, and the full screen techniques well established, possibly with a set house-style. Nevertheless, mainframe APL development does still go on, and a review of some of the full screen methods available may be of interest to relative newcomers.

My primary interest is alphamerics for menus, reports, tables, etc. but graphics will get some mention. I will cover four approaches: (1) Character Matrix, (2) GDDM (AP126), (3) ISPF and (4) AP124 (yes, AP124!). As always, everything is in origin-1. I use TSO, but most of what I have to say also applies to VM.

Character Matrix

For simple menu systems, this is the oldest, the easiest and the cheapest method. Obviously, all you need to do is clear the screen (AP100 and TSO CLEAR), construct and display the matrix, have a little prompt/response function and use ⎕*TC*[2]. By the way, APL2 provides a rather neat way for that prompt function:

```
      ∇IPX[⎕]∇
[0]    Z←IPX MSG;⎕PR
[1]   ⍝ APL2 prompt/response fn
[2]    ⎕PR←' '
[3]    ⍞←MSG
[4]    Z←⍞~¨' '

       Z←⎕WGET 1
```

(This won't work on APL2/PC since ⎕*PR* is not yet implemented)

I can anticipate screams of horror at the thought that a 'professional' APL-er might still do full screens this way, but it does actually work! Of course, a prime objection is that it is too easy to interrupt back to raw APL, so there are many systems where you should NEVER use it. In some of our systems, though, for historical reasons we still have menus created this way, and with several hundred users over many years I am not aware of many complaints. I think the essential point is that our users are relatively sophisticated and would not be put off by accidentally hitting a wrong key. Of course, you shouldn't nest these simple menus too deeply since this makes backing out frustrating.

Graphical Data Display Manager

I suppose that GDDM (assessed via AP126) is regarded as the main IBM tool for full screen work, both text and graphics. However, it is expensive, and buffering of the GDDM calls where possible should be used to minimise CPU time. There is a good paper by Dick Bowman on how to set about this (Bo89).

As Dick points out, the APL2-supplied GDMX function (in workspace 2 GDMX) has very complex internal logic and seems to be designed on the basis that it will still work and access any new GDDM calls that IBM might invent in the future. However, I live in the here and now, and Dick's simpler type of code is far preferable. For serious full screen work a design tool is essential. Many sites will still have IBM's 2 FSDESIGN which does this, but this is obsolescent; in any case it does not support buffered GDDM calls. Dick Bowman's paper, though, does give the bones of an alternative design tool which works quite well. Be warned however! If you start copying his code, by the time you have nearly finished you come across that most helpful of all APL lines:

```
  FIELDPOS[26] etc, etc, etc.
```

Yes, you need about another 60 lines here, so you will have to understand the code after all - which is always a good idea anyway!

FIELDPOS does all the work to add, delete, move, resize, etc. screen fields and uses a couple of unexplained functions (*NOTOVER* and *PMAP*) which are not listed. Their purpose, though, is pretty clear if you remember that you cannot overlap screen fields in GDDM. The way *FIELDPOS* works is to generate the new format row and column details each time a change is made and put these with *PFMT* which holds the parameters for the fields already set. *PMAP* starts with a screen-size 'map' of 0's and goes round *PFMT* adding a 1 in each 'position' of each field. If we end up with only 0's and 1's it's O.K., otherwise an overlap so throw it away and start again. This is a much simpler method than trying to detect entry errors and putting up messages, etc. and the user soon learns to be careful with cursor positioning.

For what it's worth, my origin-1 code for those functions (combined together) is given at the end. (I wonder if anyone will complain about that loop?). Finally, before leaving *FIELDPOS*, there is a nice little idiom there for turning a vector into a 1-column matrix, which I don't remember seeing before:-

```
,[⍳0]VECTOR ↔ ((⍴VECTOR),1)⍴VECTOR
```

Interactive System Productivity Facility

Or ISPF for short. Many people routinely use this without realising that it is a full screen design system and that you can use it with APL2. Useful documentation is, however, scattered about.

The APL2 manual (IBM1 pp 72, 73) is not very helpful, although it does tell you how to use the ISPF editor to edit APL2 functions. For long functions (and, yes, they are sometimes necessary, viz. *FIELDPOS*) this is much better than *)EDITOR 2* because of its powerful insert, move, copy, delete etc. facilities for both single lines and blocks of lines. Worth looking at even if you don't use ISPF for panels. Minor restrictions: (1) You can't ring-edit - a pity! (2) You cannot have function lines longer than around 230 characters - but who wants to, anyway?

Where next then? You will need up-to-date ISPF manuals (IBM2), (IBM3) and there is a reasonable amount of APL2 there, including some example code (more on which later). You can learn how to write ISPF panels from these sources (and it is not difficult) but it is still not easy to see how to set up a reliable APL2-ISPF bond.

The best starting reference that I have come across is the paper by Loren Mayhew (Ma86) which gives a very clear and detailed account (with code) of one method of proceeding, including the sort of logic needed to incorporate ISPF split-screen mode. (You must not allow APL2 to be called twice! It has, as they say, 'unpredictable results'!) The key to the success of this method is the use of an AP101 stack to do a *)RESET* and then re-call the original (*⎕LX*) calling function.

One small point; the full-stops (periods) don't always show clearly in the code as printed and there are names like '.HELP', which is not the same as 'HELP'. Look for suspicious indentations if in doubt. All in all, I recommend this paper as a good place to start.

One thing the above paper doesn't touch on is ISPF tables, but (IBM3) is quite good here and has a worked telephone book example with two APL2 functions

TELEBK and *TELESV*. This code can be tidied up somewhat but it is worth coding up to learn about ISPF tables. One thing to be aware of is that it is not fully robust. If, for example, you try from the panel to delete a report before you have created it, you will have a nasty ABEND. There are a few like this, all sequence dependent, but they are easily covered by initialising some flags in *TELEBK* and trapping/switching them in *TELESV* to enforce correct sequencing.

The telephone book and (Ma86) between them cover almost everything you need to know about APL2 with ISPF. One exception is perhaps dynamic table building, a useful technique for large tables. (IBM2) has a selection on this with a worked PL/1 example. Even if you don't know PL/1 this is easy to translate into APL and gives a good insight, including how to use the excellent supplied scrolling facilities e.g. PF7, PF8. Also things like DOWN 500 or M for MAX on the command line, then PF8 to jump straight to the bottom of the scrolled data, etc.

With a properly set up ISPF system, there is no need to back out of menus; you can jump from almost anywhere to almost anywhere else. For example, '=3.4' will go straight to option 3.4, and '=X' will get you home wherever you are. If you want to do this sort of thing yourself, perhaps using GDDM, the logic is not exactly trivial and with ISPF it is already done for your.

Amongst the things I like about ISPF is the excellent help tutorial; you may need to write some application-specific help panels, but all the stuff about scroll commands, etc. is already there. Another thing is toggling the PF key display. A beginner can type PFSHOW ON on the command line to get them displayed and, when they are familiar, the PFSHOW OFF will get rid of them. Apart from the default PF key settings, application-specific ones can readily be set dynamically.

One piece of advice is to never waste time typing in a complete ISPF panel; instead, copy one of your own, or someone else's, and change it. For example, for your first 'primary' panel, pick the ISPF primary panel itself and modify a copy of that. If you see a panel you like somewhere just type PANELID on the command line to get its member name, then go look in the SYS...ISPPLIB libraries, or wherever, to grab a copy.

Someone mentioned to me a while back that a possible problem with ISPF-APL2 was competition for attention interrupts. Our relations with our end users are such that we can simply tell them not the use the attention key and they meekly obey. However, if this matters to you, I think it can be handled with an 'attention exit routine' in the calling CLIST; this is discussed in (IBM2) but I have not tried it yet.

I like ISPF panels. The APL code is a little messy because everything (even numbers) has to be passed as a character string, but the panels are easy to set up (almost WYSIWYG) and a whole raft of excellent data validation routines comes ready-supplied.

ISPF is also available on the PC, I gather, but I have not tried it because you cannot link it to APL2/PC. IBM tell me that I am the first person to request this option, and I hope that they can find time to build it in.

AP124

This was IBM's precursor to AP126 and allowed full-screen panels without the GDDM overhead. Well, IBM don't supply this any more, so why bother to mention it? Firstly, old VSAPL systems converting to APL2 (if there are any left who haven't done so already) either have to rewrite the full screen work or simulate this, perhaps even using GDDM (Yuk!). Fortunately I don't have this problem. What I am more interested in are commercial packages like FSM124 from Interprocess Systems which simulates AP124 without using GDDM, and is reportedly cheaper to use. I am hoping to test run this in the not too distant future.

Coming down from the mainframe for a moment, I am also interested in APL2/PC which doesn't have AP126 but does have AP124 so I am also getting involved there. The supplied workspace for AP124 (called AP124) does have a modest full-screen design function, FSDEF, but this requires all the individual field parameters to be hard-coded inside application functions, which is not very satisfactory for serious design work. Also, the code has been brought down from VSAPL and is distinctly APL1, with a plethora of global variables, etc.

If you want an AP124 full-screen design kit, which itself exploits the full-screen capabilities, then it's (almost) all there in Adrian Smith's book (Sm82), if you can beg, steal or borrow a copy. You will need some modifications for the PC (see the IBM PC manual) and, interestingly, you can now use overlapping fields for things like adding a uniform background colour. Finally, you will need to get rid of the embedded ⎕*AV* code, and (I'm sure Adrian would agree now that APL2 is available) might nowadays prefer using a nested array to avoid those global names.

Alternatively, if you prefer Dick Bowman's GDDM approach, it is not too difficult to modify this for AP124.

Graphics

Of the above four methods, only GDDM will do graphics on the mainframe, but as one keeps repeating, it is expensive. If you need mixed text and graphics GDDM can do this but, if your work is mostly text with a small graphics content, consider using ISPF and calling GDDM from there, which is possible with the latest release of ISPF.

I think, though, that these days many people would recommend downloading the data to a PC and using one of the excellent PC graphics packages like Harvard Business Graphics or Lotus Freelance, etc. This is the approach that we now use.

Is GDDM obsolescent?

References

[Bo89] Bowman, R. *'APL and GDDM - A High Performance Toolkit'* APL89 Conference Proceedings pp 43-53

[Ma86] Mayhew, L. *'Increasing Productivity with ISPF/APL2'* APL86 Conference Proceedings pp 243-251

[IBM1] *'APL2 Programming: System Services Reference'* IBM Publication SH20-9218-2 (Nov 87)

[IBM2] *'ISPF: Dialog Management Guide: MVS'* IBM Publication SC34-4112-00 (Jun 87)

[IBM3] *'ISPF: Dialog Management Services and Examples: MVS'* IBM Publication SC34-4113-00 (Jun 87)

[Sm82] Smith, A. *'APL - A Design Handbook for Commercial Systems'* (1982) - Out of print

Appendix

```
∇OVER[⎕]∇
[0] R←TL_NM OVER PFMT;FF;M;I;F;RS;CS
[1] ⍝ Check for overlapping fields
[2] ⍝-----------------------------------------------------------⍝
[3] ⍝ M - 'Bitmap' matrix of screen. M[J;K] is :-
[4] ⍝ 0 if unused, 1 if OK fld posn, >1 if overlap
[5] ⍝-----------------------------------------------------------⍝
[6] FF←(4↑[2]1↓[2]PFMT),[1]TL_NM ⍝ Add new fld
[7] M←(SS- 0 1)⍴0 ⍝ SS is global screen size
[8] I←0
[9] LP:→((1↑⍴FF)<I←I+1)/OLAP
[10] F←FF[I;]
[11] (RS CS)←(¯1+F[1]+⍳F[3])(¯1+F[2]+⍳F[4])
[12] M[RS;CS]←1+M[RS;CS] ⍝ Note addition
[13] →LP
[14] ⍝-----------------------------------------------------------⍝
[15] OLAP:R←1∊1<,M ⍝ R - 1 if overlap, 0 if OK
```

A Note on the Match Function in APL

by Joseph L.F. De Kerf

Abstract

In SHARP APL, empty arrays of the same structure always match. In APL2-like implementations, empty arrays only match if they are of the same structure and prototype. As a compromise, the APL Working Group - designing the new standard ISO APL Extended - defines the match function such that when the arguments are empty arrays of the same structure a domain error is reported. It is suggested in this note that the definition of the match function for arrays with the same structure should be consistent with the definition of the primitive scalar function equal.

Introduction

In ISO APL 8485 (1), the dyadic scalar function equal *Z←A=B* is defined such that, if the arguments *A* and *B* are arrays of the same structure, the explicit result *Z* is a boolean array of the same structure as the arguments, containing a 1 where the corresponding elements of *A* and *B* are equal, and 0 otherwise. If one argument is a scalar or one-item array, or if both arguments are one-item arrays, so-called scalar extension is applied. If the arguments *A* and *B* do not conform, an appropriate error message is reported. Comparison Tolerance *⎕CT* is an implicit argument.

Apparently there is no problem with this definition. With the introduction of arrays of arrays however, a controversy has arisen about the new dyadic function match *Z←A≡B*. In some implementations, empty arrays of the same structure always match. In some other implementations, empty arrays of tl.e same structure only match if they are of the same prototype. Confronted with this controversy, the APL Working group ISO-IEC/JTC1/SC22/WG3 - designing the new standard ISO APL Extended - defines as a compromise the match function such that when the arguments are empty arrays of the same structure a domain error is reported.

The SHARP APL Approach

In SHARP APL (2), based on the grounded array concept, the dyadic function *Z←A≡B*, if the arguments *A* and *B* are of the same structure and data, returns as explicit result *Z* a boolean scalar 1, and 0 otherwise. Two empty arrays of the same structure such as the empty character vector `''` and the empty numeric vector `⍳0` match, as they have no elements; `''≡⍳0  ↔  1`. The definition conforms to Ken Iverson's APL Dictionary (3).

The NARS Approach

In APL*PLUS NARS (4) and IBM APL2-like implementations (5), based on the floating array concept, two arguments *A* and *B* match if they are of the same structure and data, and if empty, they are of the same structure and prototype. Two empty arrays of the same structure but different prototype, such as the empty character vector `''` and the empty numeric vector `⍳0`, do not match: `''≡⍳0  ↔  0`.

The basic idea seems to be that two arrays match if and only if they behave in the same way. For example, the first function `↑B` applied to the empty character vector `''` gives a blank, while applied to the empty numeric vector `⍳0` gives a zero, and as such they do not match. However, match doesn't guarantee the same behaviour. For instance (the example is from a private communication by Eugene McDonnell):

```
      ⎕CT←1E¯13
      A←1
      B←0.999999999999999
      A≡B
1
      (1-A)≡1-B
0
```

The reason is of course that, such as for the dyadic scalar function equal, comparison tolerance `⎕CT` is an implicit argument. If comparison tolerance is set to zero, or the match function is defined as independent of comparison tolerance, two non-empty arrays of the same structure which match, always behave in the same way. And in the example given above, *A* and *B* no longer match:

```
      ⎕CT←0
      A←1
      B←0.999999999999999
      A≡B
0
      (1-A)≡1-B
0
```

Defining the dyadic function match as independent of comparison tolerance however would produce situations which do not conform with the definition of the dyadic scalar function equal and are unacceptable. For instance, the product of an integer with its inverse could give a result which never matches the value one:

```
        A←100⍴1
        B←B×÷B←⍳100
        ⎕CT←1E¯13
        A≡B
  1
        ⎕CT←0
        A≡B
  0
```

the discrepancies being dependent from the particular implementation used. For the implementation used in preparing this note (VAX APL Version 3 - DEC):

```
        A←100⍴1
        B←100
        ⎕CT←0
        (A≠B×÷B)/B
 23 27 46 54 89 92
```

which means that for this implementation the identity `B×÷B←→1` does not hold for the integers listed (23-27-46-54-89-92). Consistency of the concept match with the basic idea of assuring the same behaviour seems not to be feasible.

The ISO APL Approach

In the meantime, the APL Working Group ISO-IEC/JTC1/SC22/WG3 is preparing a new APL standard, provisionally called ISO APL Extended. At its Copenhagen Meeting in August 1990, the APL Working Group decided that "APL Extended must include the generalized array facility by defining the functions enclose ⊂ and disclose ⊃ (as defined by APL2)". As far as the inclusion of the dyadic function match ≡ is concerned however, it was clear that it would be very difficult to come to a concensus about the controversy described.

Finally, based on an earlier proposition from Eugene McDonnell (6), it was decided to define the dyadic function match in such a way that, when the arguments are empty arrays of the same structure, a domain error is reported. This gives the implementor the opportunity to choose - as a consistent extension - whether in this case the explicit is 1 or 0. This has no sense. In addition, as Jean-

Jacques Girardot stated at the APL 90 Conference (7), "it means that what the standard defines as a conforming program should not use *A*≡*B* without first checking the case of empty operands of the same dimensions" - an unacceptable situation.

Suggestions

There is however a plausible solution. APL, as every higher level programming language, even assembler, contains a lot of redundancy. Redundancy increases the power of a language, but redundancy must be consistent. For the dyadic function match *A*≡*B* for instance, this means that its definition must be consistent with the definition of the dyadic scalar *A*=*B* as given above; if two arrays *A* and *B* have the same structure, they should match if and only if the conjunction ∧ of the enlist of *A*=*B* is 1 or *A*≡*B* ↔ ∧/*ENLIST* *A*=*B* (*ENLIST* *R* being a monadic function that returns as explicit result a simple vector whose elements are all the single scalars in the right argument *R* taken in eventually nested row major order, as for instance the primitive function ∊R in some APL2-like implementations and the monadic system function ⎕*ENLIST* *R* in VAX APL Version 3 of DEC). This gives for two empty arrays with the same structure as for instance the empty character vector '' and the empty numeric vector ⍳0:

```
      ∧/⎕ENLIST ''=⍳0
1
```

which means that those empty arrays match. Maybe the APL Working Group ISO-IEC/JTC1/SC22/WG3 could reconsider its decision taken at the Copenhagen Meeting of August 1990. and discuss this proposal of consistency which, as far as the author of this note is concerned, is a must.

References

[1] *ISO 8485; 1989 (E); Programming Language - APL;First Edition - 1989 11-01;* Edited by A. Morrow; International Organization for Standardization - ISO, Geneva, Switzerland, 1989. See also : ISO 8485 (F).

[2] K.E. Iverson; *Composition and Enclosure; SHARP APL Technical Note SATN-41;* I.P.Sharp Associates Limited, Toronto, Ontario, 20 June 1981.

[3] K.E. Iverson; *A Dictionary of APL;* Publication No. 0402 8703 E3; I.P.Sharp Associates Limited, Toronto, Ontario, March 1987. See also: APL Quote Quad, Vol. 18, No. 1, September 1987, pp.5-40.

[4] C.M. Cheney; *APL*PLUS Nested Arrays System - Reference Manual;* Publication No. P046-0381; STSC, Bethesda, Maryland, March 1981.

[5] D. Rabenhorst; *APL2 Language Manual - Installed User Program 5798-DJB; Publication No. SB21-3015-0 - First edition;* IBM Corporation, APL Development Department, San Jose, California, June 1982.

[6] E.E.McDonnell; *Rationale for the Match Function - Proposed text for ISO Document;* APL Working Group ISO-IEC/JTC1/SC22/WG3-N200; Dated 19 May 1988.

[7] J.J. Girardot; *Arrays and References; APL 90 Conference Proceedings,* Copenhagen, Denmark, 13-17 August 1990; Edited by P Gjerlov; APL Quote Quad, Vol. 20, No.4, July 1990, pp.161-172

The Steam-Hammer and the Fly

by Gerard A Langlet

A French proverb says: "Never use a steam-hammer to smash a fly". It (or its English counterpart) should be repeated on every page of any book or tutorial devoted to APL.

Vigorously attacked in Vector 7.1 [1], I shall answer in a way that may not fulfil Bob Bykerk's criticisms completely, but as much as I can in a few pages; indeed a whole book would be better - that may come in the future.

First, I have been programming in APL for almost 20 years, writing sophisticated applications for research and industry, i.e. some important subset of the "real world". Quite often, I also have to translate my algorithms into other compiled languages for various reasons; execution speed, compatibility with other existing programs or systems, absence of APL implementations, hatred or incompetence of colleagues about APL, or simply the still high cost of good APL interpreters on some hardware.

The real world has many many constraints of all kinds, so, when programming in APL, which still is the best language ever designed to test a new idea and build a prototype, it does not seem to me superfluous to think of what will come next. The "RISC" programming style is the fruit of hard experiments. It may not correspond to what is taught in manuals about data structures and/or the usage of APL notation. But I am nothing of a masochist, and I would gently conform to any other so-called "healthy coding style" if I were convinced of its real superiority. APL should not remain like an ivory tower, the real world is indeed a multi-language environment, full of traps e.g. ambiguous and fuzzy (sometimes erroneous) data.

Many people often believe that the beauty and the expressive power of the APL notation on paper, especially in extended implementations, will lead them to write easily nice and efficient code. Just try `R←∊⍳¨V` with, for example, `V←10000⍴3 4 5 0 2 3` on some APL2 compatible interpreter and compare the CPU time with `R←V/V-+/V ⋄ R←R+⍳⍴R`.

Let us now talk about data structures; one is taught to represent sales data in 3 dimensions - salesman × month × year. Why not, if you just want to use the power of +/ on any of the major axes? But this is in fact a simplistic case; one might want to extract much more sophisticated information from this array - see below. The ideal data structure has not been discovered yet. It is certainly not an array. It may be a "fuzzy fractal". But some APL implementations (e.g. APL.68000, APL90, and to a certain extent, Dyalog-APL and APL*PLUS), offer fast reversible matrix to vector conversion tools such as *⎕BOX*.

In general, a text page is shorter when kept as a vector. This is also true for numeric data with heterogeneous length, but *⎕BOX* accepts them only in the first two implementations. Most of my data are kept in vectors, and if and only if it is necessary to handle them in matrices, or in vectors of vectors, or in vectors of matrices, they are TEMPORARILY, rapidly and internally reshaped with automatic mechanisms which accept ANY of the possible data structures as input.

Another item of my short letter to Vector [2] has been misunderstood. I never wrote not to use locals; I think that locals should be avoided when they are unnecessary, except e.g. for voluntary didactic purposes. I would appreciate an extended APL implementation in which all one character names would be automatically localised; this would save typing time and space. Since I deprecate the use of parentheses, APL expressions are shorter and do not lead so frequently to the *WS FULL* message; then, I do collect the intermediate results within the minimum of locals. With "RISC" programming style, I don't burn my bridges. My programs are easier to debug than before. Nobody is obliged to believe me; just try, if you wish, and criticise only in six months from now.

Try to write ex nihilo with branches, the *PERM* function listed in [3]. It is not an easy way of doing it. Then, try to write a screen manager using some combinations of the 3 buttons of a mouse as well as the key-pad, in order to handle several recursive pop-up menu-windows with lifts, travelators, automatic clipping, shadows and help files. I did that once, in APL*PLUS PC, and I doubt that I could have succeeded with branches and labels [4].

The result is a short general-purpose RISC-APL function which respects several other rules of my anathematised programming habits; the display of a function should not exceed the screen size, so that you never have to scroll up or down to see what it does; every procedure (simply a character string or vector) is defined before the one which uses it; the resulting code executes quickly in APL and can be translated easily i.e. by hand or with another small APL program into any of the important programming languages in the real world outside APL (yes! it does exist).

Are Salesmen Fly-smashers?

Suppose we have 20 salesmen who have worked for 10 years from 1981 to 1990. *Y* is the vector of the number of days in each year, and *N* its sum:

```
      1 20×N←+/⎕←Y←10⍴3 1/365 366
365 365 365 366 365 365 365 366 365 365
3652 73040
```

Let us introduce the following two functions. *PSUM* which produces the partial sums of *B* according to *A*, and *EXECR* which EXECutes ⍙ with Rank ∆

```
      ∇R←A PSUM B;D;N;⎕IO
      ⎕IO←1 ⋄ D←¯1↑⍴B←+\B ⋄ N←⌈D÷1⌈+/A ⋄ R←⍳N←|N×⍴,A ⋄ A←D⌊+\N⍴A ⋄ N←⍴⍴B
      N EXECR 'B←0,-B[A]⋄R←B[R]-B[R+1]'
      ∇

      ∇∆ EXECR ⍙                        ⍝ APL.68000 Level II
      ∆←'[',1↓∆⍴';' ⋄ ⍎⎕SS ⍙ '[' ∆      ⍝ ⎕SS accepts strand notation: A B C
      ∇

      ∇∆ EXECR ⍙                        ⍝ APL.68000 Level I
      ⍎⎕SS(⍙;'[';∆←'[',1↓∆⍴';')         ⍝ ⎕SS has 3 arguments (A;B;C)
      ∇
```

Note: ⎕*SS* substitutes every occurrence of *B* by *C* in character vector *A*.

Function *PSUM* is programmed "thinking of vectors", but it also works, reducing any rank array along the last dimension, due to *EXECR* which is a general purpose tool (JUST LOCK IT IF YOU DISLIKE IT), or adapt it to your own APL implementation if you have no ⎕*SS*. Rub it if you never use arrays.

Now, let us suppose that you want the results per week, per month, and per semester. What will you do if you have organised your data in an array and if you cannot use nested arrays? With vectors, heterogeneous groupings are easy. The number of full weeks in these 10 years is `W←W⌊N÷7`, i.e. 521 and the number of extra days in the same period is `E←7|N` i.e. 5. You may try: `RW←7 PSUM 20 3652⍴T` if you just want to start "as the data are" and love reshaping, but what happens if you want full weeks starting on Sundays - in the English way - or on Mondays - in the French one? Incidentally, January 1st, 1981 was Thursday, so that the first group of days is 3, and the last one is `E-3` when the week starts on Sunday.

```
        K←20×ρV←1 521 1/3 7,E-3
        G←KρV
        RSW←G PSUM T ⍝ for "Sunday weeks"
```

Now let *D* be the day/month-vector in a normal year, reshaped for 10 years:

```
  D←120ρ31 28 31 30 31 30 31 31 30 31 30 31 ⋄ I←120ρ12↑0 1
  M←2400ρD+I∧12/Y=366 ⋄ S←G PSUM M ⋄ RM←M PSUM T ⋄ RS←S PSUM T
```

RM is the result per Month, and *RS* the result per Semester.

Note: "Replicate" is frequently used here. Although absent from The ISO-Standard APL it will be in the next standard and is available in most present implementations. If you have APL2, all this will also work; then, try to measure CPU time using enclosed arrays instead... Will the fly survive?

References

[1] Bykerk, B. *The Dangers of APL RISC programming*, Vector, 7.1, 112.

[2] Langlet, G.A. *APL RISC Programming Style*, Vector, 6.2, 23-24.

[3] Langlet, G.A. *The Travelling Salesman Problem revisited with APL*, APL90 Conference, Copenhagen. APL Quote Quad, 20.4, 228-232 (July 1990).

[4] Langlet, G.A. *Presentation of GLOS, all-purpose software integrator on PC*, in "Modelling of Molecular Structures and Properties", Elsevier Science Publishers, ISBN 0-444-88714-8, 767 (1990).

Ed: The above is a shortened version of Gerard's original article, which was too long for the space available in Vector.

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 (with diskette as appropriate) to the Editor:

Jonathan Barman,
Hill Top House,
East Garston,
NEWBURY, Berks RG16 7HD
Tel: 048839-575 (not after 10.00pm please!)

Camera-ready artwork (e.g. advertisements) and diskettes of 'standard' material (e.g. sustaining members news) should be sent to Vector Production, c/o Adrian Smith, Brook House, Gilling East, YORK Tel: 04393-385 (6.00pm - midnight).

Product Guide updates should continue to go to Alison Chatterton, as should requests for advertising space.

MicroAPL for ...

Consultancy and Training

MicroAPL has 10 years' experience with small-computer APL systems - stand alone, networked or multi-user - and using all the major versions of APL. We provide a full range of consultancy services including feasibility studies, project management, programming, documentation and support services. Our programming skills cover all areas of APL and systems programming. MicroAPL also offer a comprehensive range of APL training courses.

For more details, contact David Eastwood at:

M I C R O
A P L

**MicroAPL Ltd.
South Bank Technopark
90 London Road
LONDON, SE1 6LN
Telephone: 071 922 8866**

Index to Advertisers

APL People (half)	66
Dyadic Systems Ltd	2,6
MicroAPL	82,142
APL Booklist (Renaissance Data Systems)	32

All queries regarding advertising in VECTOR should be made to Alison Chatterton, at the address on the inside back cover.

BAA: Membership Application Form

Membership of the British APL Association is open to anyone interested in APL. The membership year runs from 1st May to 30th April.

Name: ____________________

Address Line 1: ____________________

Address Line 2: ____________________

Address Line 3: ____________________

Post or zip code: ____________________

Country: ____________________

Telephone Number: ____________________

Membership category (please tick box):	91/92	
UK private membership	£12	☐
Overseas private membership	£20	☐
Airmail supplement (not needed for Europe)	£8	☐
Corporate membership	£100	☐
Corporate membership overseas	£155	☐
Sustaining membership	£430	☐
Non-voting student membership (UK only)	£6	☐

I authorise you to debit my Visa/Mastercard account

Number: ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiry date: ☐☐☐☐

for the membership category indicated above,

☐ annually, at the prevailing rate, until further notice
☐ one year's subscription only

(please tick the required option above)

Signature: ____________________

PAYMENT

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 Access or Visa number. Please send the completed form to:

BAA Administration, Alison Chatterton, 9 Oak Grove, HERTFORD SG13 8AT, England

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.

Typeset by APL-385 with MS Word 5.0 and GoScript

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 many timesharing bureaux and on most mainframe, mini and micro 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.

APL People
The Old Malthouse
Clarence St. BATH, BA1 5NS
Tel:0225-462602

Intelligent Programs Ltd
Unit 7, Hermitage Court
6-10 Sampson St
LONDON, E1 9NA
Tel:071-481 4813

Compass R&D Ltd
15 Frederick Sanger Rd
Surrey Research Park
GUILDFORD, Surrey GU2 5YD
Tel:0483 302249
Fax:0483 302279

Dyadic Systems Ltd
Riverside View, Basing Road,
Old Basing, BASINGSTOKE,
Hants, RG24 0AL
Tel:0256 811125
Fax:0256 811130

HMW Computing Ltd
Hamilton House,
1 Temple Avenue,
LONDON EC4Y 0HA
Tel:071-353 4212
Fax:071-353 3325

APL Impetus Ltd
Rusper, Sandy Lane
Ivy Hatch, SEVENOAKS
Kent TN15 0PD
Tel:0732-885126

Cocking & Drury Ltd
180 Tottenham Court Rd
LONDON, W1P 9LE
Tel:071-436 9481

MicroAPL Ltd
South Bank Technopark
90 London Road
LONDON SE1 6LN
Tel:071-922 8866

REUTER:FILE
1-4 Singer St.
LONDON EC2A 4BQ
Tel:071-867 1166
Fax:071-867 9792

Peter Cyriax Systems
213 Goldhurst Terrace
LONDON NW6 3ER
Tel:071-624-7013
Mobile:0860-377963

The British Computer Society, 13 Mansfield Street, London W1M 0BD.