

# **Mastering Dyalog APL**

## **A Complete Introduction to Dyalog APL**

**Bernard Legrand**

**With most grateful acknowledgements to the contributors:**

Kim S.	Andreasen
Daniel	Baronet
Gitte	Christensen
Peter	Donnelly
Morten	Kromberg
John	Scholes
Adrian	Smith
Tim JA.	Smith

*Dyalog is a trademark of Dyalog Limited  
Copyright © 1982-2009 by Dyalog Limited  
Published by Dyalog Limited*

*All rights reserved.*

## **First Edition November 2009**

*No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.*

*Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.*

### **TRADEMARKS**

*SQAPL is copyright of Insight Systems ApS.*

*UNIX is a trademark of X/Open Ltd.*

*Windows, Windows Vista, Visual Basic and Excel are trademarks of Microsoft Corporation.*

*All other trademarks and copyrights are acknowledged.*

**Dyalog Limited**

<http://www.dyalog.com>

**ISBN : 978-0-9564638-0-7**

---

# Contents

<b>INTRODUCTION - WILL YOU PLAY APL WITH ME?</b>	<b>1</b>
<b>Will You Follow Us?</b>	<b>1</b>
<b>Our First Steps into APL's Magic World</b>	<b>4</b>
<b>Array Processing</b>	<b>5</b>
<b>More Symbols</b>	<b>7</b>
<b>Most Symbols Have a Double Meaning</b>	<b>8</b>
<b>Reduction Unifies Traditional Notations</b>	<b>9</b>
<b>Let's Write Our First Programs</b>	<b>10</b>
<b>Indexing</b>	<b>11</b>
<b>Calculating Without Writing Programs</b>	<b>12</b>
<b>Friendly Binary Data</b>	<b>14</b>
<b>A Touch of Modern Math</b>	<b>16</b>
<b>A Powerful Search Function</b>	<b>17</b>
<b>After Values, Let Us Process Shapes</b>	<b>20</b>
<b>Back to Primary School</b>	<b>22</b>
<b>There Is a Lot to Discover Yet</b>	<b>25</b>
<b>FAQ</b>	<b>28</b>

<b>CHAPTER A: GETTING STARTED</b>	<b>31</b>
<b>1 - Installing the Software</b>	<b>31</b>
1.1 Installation	31
1.2 First Contact	33
1.3 Demonstration Files	36
<b>2 - Working with This Tutorial</b>	<b>40</b>
<b>CHAPTER B: DATA AND VARIABLES</b>	<b>43</b>
<b>1 - Simple Numeric Values</b>	<b>43</b>
1.1 Our First Operations	43
1.2 Variables	44
1.3 Operations on Variables	46
<b>2 - Arrays of Items</b>	<b>47</b>
2.1 Create a List or a Matrix	47
2.2 Special Cases with Reshape	48
2.3 Multi-dimensional Arrays	49
<b>3 - Shape, Rank, and Vocabulary</b>	<b>50</b>
3.1 Shape and Rank	50
3.2 Scaling Down the Ranks	51
3.3 Vocabulary	51
3.4 Beware!	52
<b>4 - Simple Character Values</b>	<b>53</b>
4.1 Character Vectors and Scalars	53
4.2 Character Arrays	55
<b>5 - Indexing</b>	<b>56</b>
5.1 Traditional Vector Indexing	56
5.2 The Shape of the Result	57
5.3 Array Indexing	58
5.4 Convention	60
5.5 Warnings	61
5.6 The Index Function	62

---

<b>6 - Mixed and Nested Arrays</b>	<b>63</b>
6.1 Mixed Arrays	63
6.2 Four Important Remarks	64
6.3 Nested Arrays	64
6.4 DISPLAY	66
6.5 Be Simple!	68
6.6 That's <i>Not</i> All, Folks!	70
<b>7 - Empty Arrays</b>	<b>70</b>
<b>8 - Workspaces and Commands</b>	<b>71</b>
8.1 The Active Workspace	72
8.2 The Libraries	73
8.3 Load a WS	74
8.4 File Extensions	75
8.5 Merge Workspaces	76
8.6 Exiting APL	78
8.7 Contents of a WS	78
8.8 Our First System Commands	79
Exercises	81
<b>The Specialist's Section</b>	<b>83</b>
Spe-1 Variable Names	83
Spe-2 Representation of Numbers	83
Spe-3 The Shape of the Result of Indexing	84
Spe-4 Multiple Usage of an Index	86
Spe-5 A Problem With Using Reshape ( $\rho$ )	86
Spe-6 Monadic Index ( $\uparrow$ )	87
<b>CHAPTER C: SOME PRIMITIVE FUNCTIONS</b>	<b>89</b>
<b>1 - Definitions</b>	<b>89</b>
<b>2 - Some Scalar Dyadic Functions</b>	<b>90</b>
2.1 Definition and Examples	90
2.2 Division By Zero	92
2.3 Power	92
2.4 Maximum & Minimum	92
2.5 Relationship	93
2.6 Residue	94
<b>3 - Order of Evaluation</b>	<b>94</b>

<b>4 - Monadic Scalar Functions</b>	<b>96</b>
4.1 The Four Basic Symbols	96
4.2 Other Scalar Monadic Functions	97
<b>5 - Processing Binary Data</b>	<b>99</b>
5.1 Membership	99
5.2 Binary Algebra	100
5.3 Without	102
<b>6 - Processing Nested Arrays</b>	<b>102</b>
6.1 Scalar vs. Non-scalar Functions	102
6.2 Be Careful With Shape/Type Compatibility	103
<b>7 - Reduction</b>	<b>104</b>
7.1 Presentation	104
7.2 Definition	105
7.3 Reduction of Binary Data	106
7.4 Reduction of Nested Arrays	107
7.5 <i>Application 1</i>	107
7.6 <i>Application 2</i>	108
<b>8 - Axis Specification</b>	<b>109</b>
8.1 Totals in an Array	109
8.2 The Shape of the Result	111
8.3 Special Notations	111
<b>9 - Our First Program</b>	<b>112</b>
<b>10 - Concatenation</b>	<b>113</b>
10.1 Concatenating Vectors	113
10.2 Concatenating Other Arrays	114
10.3 Concatenating Scalars	117
10.4 Special Notations	117
<b>11 - Replication</b>	<b>118</b>
11.1 Basic Approach: Compression	118
11.2 Replication	120
11.3 Scalar Left Argument	120
11.4 Special Notations	121
<b>12 - Position (<i>Index Of</i>)</b>	<b>121</b>
12.1 Discovery	121
12.2 <i>Application 3</i>	123

---

<b>13 - Index Generator</b>	<b>125</b>
13.1 Basic Usage	125
13.2 <i>Application 4</i>	126
13.3 Comparison of <i>Membership</i> and <i>Index Of</i>	127
13.4 Idioms	130
13.5 <i>Application 5</i>	131
13.6 <i>Application 6</i>	132
<b>14 - Ravel</b>	<b>132</b>
<b>15 - Empty Vectors and Black Holes</b>	<b>134</b>
Exercises	136
<b>The Specialist's Section</b>	<b>140</b>
Spe - 1 Division Control - <code>□DIV</code>	140
Spe - 2 Derived Functions	141
Spe - 3 Nor & Nand	141
Spe - 4 Index Generator of Arrays	142
Spe - 5 Ravel With Axis	143
Spe - 6 Residue	145
<b>CHAPTER D: USER DEFINED FUNCTIONS</b>	<b>147</b>
<b>1 - Landmarks</b>	<b>147</b>
1.1 Some Definitions	147
1.2 Configure Your Environment	148
<b>2 - Single-Line Direct Functions</b>	<b>152</b>
2.1 Definition	152
2.2 Unnamed D-Fns	153
2.3 Modifying The Code	153
<b>3 - Procedural Functions</b>	<b>154</b>
3.1 A First Example	154
3.2 Local Names	156
3.3 Miscellaneous	159
3.4 Second Example	161
Exercises	164
3.5 Calls to Sub-Functions	166

<b>4 - Flow Control</b>	<b>167</b>
4.1 Overview	167
4.2 Conditional Execution	169
4.3 Disparate Conditions	174
4.4 Predefined Loops	176
4.5 Conditional Loops	178
4.6 Exception Control	181
4.7 Endless Loops	182
<b>5 - Traditional Flow Control</b>	<b>186</b>
5.1 Conditional Execution	186
5.2 Multiple Conditions	190
5.3 Modern and Traditional Controls Cooperate	192
<b>6 - Input, Output, and Format</b>	<b>193</b>
6.1 Some Input and Output Methods	193
6.2 Format	194
6.3 Displaying Intermediate Results	196
6.4 Using Global Variables	197
6.5 Exchanging Data With an Excel Worksheet	198
6.6 Reading or Writing a Text File	199
6.7 Printing Results on a Printer	201
6.8 Using a Graphical User Interface	202
6.9 Requesting Values From the Keyboard	203
<b>7 - Syntax Considerations</b>	<b>205</b>
7.1 Comments & Statement Separators	205
7.2 Why Should a Function Return a Result?	206
7.3 Different Types of Functions	207
7.4 Nested Argument and Result	211
7.5 Choice of Names	212
<b>8 - Multi-Line Direct Functions</b>	<b>213</b>
8.1 Characteristics	213
8.2 Guards	215
8.3 Syntax Considerations	215
<b>9 - Recursion</b>	<b>217</b>
<b>10 - Synonyms</b>	<b>218</b>



---

<b>11 - About the Text Editor</b>	<b>220</b>
11.1 What Can You Edit?	220
11.2 What Can You Do?	221
11.3 Undo, Redo, Replay	222
11.4 Miscellaneous	224
<b>12 - SALT</b>	<b>225</b>
Exercises	227
<b>The Specialist's Section</b>	<b>230</b>
Spe-1 Shadowed Names	230
Spe-2 Loop Control	231
Spe-3 Labels and the Branch Arrow	231
Spe-4 Other Conditional Execution	233
Spe-5 Name Category of Synonyms	234
Spe-6 Bare Output	235
Spe-7 :InEach	236
<b>CHAPTER E: FIRST AID KIT</b>	<b>239</b>
<b>1 - When an Error Occurs</b>	<b>240</b>
1.1 Our First Error	240
1.2 Cascade of Errors	243
1.3 Information and Actions	249
1.4 Why Should You Reset Your State Indicator?	250
<b>2 - Most Frequent Error Messages</b>	<b>252</b>
2.1 Execution Errors	252
2.2 Some Other Errors	257
<b>3 - Trace Tools</b>	<b>258</b>
3.1 Invoke and Use the Tracer	258
3.2 Choose Your Configuration	261
3.3 Break-points and Trace-controls	262
3.4 System Functions	265
Exercises	267
<b>The Specialist's Section</b>	<b>268</b>
Spe-1 Value Errors	268
Spe-2 )SINL	269
Spe-3 Namespaces and Indicators	269

---

<b>CHAPTER F: EXECUTE &amp; FORMAT CONTROL</b>	<b>273</b>
<b>1 - Execute</b>	<b>273</b>
1.1 Definition	273
1.2 Some Typical Uses	274
1.3 Make Things Simple	276
<b>2 - The Format Primitive</b>	<b>276</b>
2.1 Monadic Format	276
2.2 Dyadic Format	277
<b>3 - The <code>FORMAT</code> System Function</b>	<b>280</b>
3.1 Monadic Use	280
3.2 Dyadic Use	281
3.3 Qualifiers and Affixtures	288
<b>The Specialist's Section</b>	<b>292</b>
Spe-1 Execute	292
Spe-2 Formatting data	295
<b>CHAPTER G: WORKING ON DATA SHAPE</b>	<b>299</b>
<b>1 - Take and Drop</b>	<b>299</b>
1.1 Take and Drop Applied to Vectors	299
1.2 Three Basic Applications	302
1.3 Take and Drop Applied to Arrays	303
<b>2 - Laminate</b>	<b>305</b>
2.1 Application to Vectors and Scalars	307
2.2 Applications	308
<b>3 - Expand</b>	<b>310</b>
3.1 Basic Use	310
3.2 Extended Definition	310
3.3 Expand Along First Axis	311
<b>4 - Reverse and Transpose</b>	<b>312</b>
<b>5 - Rotate</b>	<b>314</b>
5.1 Rotate Vectors	314
5.2 Rotate Higher-Rank Arrays	315

<b>6 - Dyadic Transpose</b>	<b>316</b>
Exercises	319
<b>The Specialist's Section</b>	<b>322</b>
Spe - 1 More About Laminate	322
Spe - 2 Dyadic Transpose	322
<b>CHAPTER H: SPECIAL SYNTAX</b>	<b>325</b>
<b>1 - Modified Assignment</b>	<b>325</b>
<b>2 - Multiple Assignment</b>	<b>326</b>
<b>3 - Selective Assignment</b>	<b>327</b>
3.1 Quick Overview	327
3.2 Available Primitives	328
<b>CHAPTER I: NESTED ARRAYS (CONTINUED)</b>	<b>331</b>
<b>1 - First Contact</b>	<b>331</b>
1.1 Definitions	331
1.2 Enclose & Disclose	332
1.3 More About DISPLAY	336
<b>2 - Depth &amp; Match</b>	<b>338</b>
2.1 Enclosing Scalars	338
2.2 Depth	339
2.3 Match & Natch	341
<b>3 - Each</b>	<b>342</b>
3.1 Definition and Examples	342
3.2 Three Compressions!	345
<b>4 - Processing Nested Arrays</b>	<b>346</b>
4.1 Scalar Dyadic Functions	346
4.2 Juxtaposition vs. Catenation	346
4.3 Characters and Numbers	348
4.4 Some More Operations	350
Exercises	353

<b>5 - Split and Mix</b>	<b>354</b>
5.1    Basic Use	354
5.2    Axis Specification	355
<b>6 - First &amp; Type</b>	<b>357</b>
<b>7 - Prototype, Fill Item</b>	<b>358</b>
<b>8 - Pick</b>	<b>361</b>
8.1 - Definition	361
8.2 - Beware!	362
8.3 - Important	363
8.4 - Selective Assignment	364
8.5 - An Idiom	365
<b>9 – Partition &amp; Partitioned Enclose</b>	<b>365</b>
9.1    The Dyalog Definition	366
9.2    The IBM Definition	367
<b>10 - Union &amp; Intersection</b>	<b>369</b>
<b>11 - Enlist</b>	<b>369</b>
Exercises	371
<b>The Specialist's Section</b>	<b>372</b>
Spe-1    Compatibility and Migration Level	372
Spe-2    The IBM Partition on Matrices	375
Spe-3    Ambiguous Representation	376
Spe-4    Pick Inside a Scalar	376
 <b>CHAPTER J: OPERATORS</b>	 <b>377</b>
<b>1 - Definitions</b>	<b>377</b>
1.1    Operators & Derived Functions	377
1.2    Sequences of Operators	378
1.3    List of Built-in Operators	379
<b>2 - More About Some Operators You Already Know</b>	<b>379</b>
2.1    Reduce	379
2.2 <i>n</i> -Wise Reduce	380
2.3    Axis	382

<b>3 - Scan</b>	<b>383</b>
3.1 Definition	383
3.2 Scan with Binary Values	384
3.3 Applications	385
<b>4 - Outer Product</b>	<b>386</b>
4.1 Definition	386
4.2 Extensions	387
4.3 <i>Applications</i>	389
Exercise	393
<b>5 - Inner Product</b>	<b>394</b>
5.1 A Concrete Situation	394
5.2 Definitions	396
5.3 Typical Uses of Inner Products	396
5.4 Other Uses of Inner Product	405
5.5 <i>Application</i>	406
Exercises	408
<b>6 - Compose</b>	<b>410</b>
6.1 Form 1	411
6.2 Form 2	412
6.3 Form 3	412
6.4 Form 4	413
<b>7 - Commute</b>	<b>414</b>
<b>8 - Power Operator</b>	<b>415</b>
8.1 - Elementary Use (Form 1)	415
8.2 - Conditional Execution (Form 1)	416
8.3 - Left Argument (All Forms)	417
8.4 - Inverse Function	417
8.5 - Fixpoint, and Use with Defined Operators	418
<b>9 - Spawn</b>	<b>418</b>
9.1 Main Features	418
9.2 Special Syntax	420
<b>10 - User-Defined Operators</b>	<b>421</b>
10.1 Definition Modes	421
10.2 Some Basic Examples	422

<b>The Specialist's Section</b>	<b>424</b>
Spe-1 Reduction Applied to Empty Vectors	424
Spe-2 Index Origin and Axis operator	426
Spe-3 The Power Operator	427
Spe-4 Defined Operators	429
Spe-5 The Result of an Inverse Function	429
<b>CHAPTER K: MATHEMATICAL FUNCTIONS</b>	<b>431</b>
<b>1 - Sorting and Searching Data</b>	<b>431</b>
1.1 Sorting Numeric Data	431
1.2 Sorting Characters	433
1.3 Finding Values	435
<b>2 - Encode and Decode</b>	<b>436</b>
2.1 Some Words of Theory	436
2.2 Using Decode & Encode	438
2.3 Applications	441
<b>3 - Randomised Values</b>	<b>444</b>
3.1 Deal: Dyadic Usage	445
3.2 Roll: Monadic Use	445
3.3 Derived Uses	446
<b>4 - Some More Maths</b>	<b>447</b>
4.1 Logarithms	447
4.2 Factorial & Binomial	448
4.3 Trigonometry	449
4.4 GCD and LCM	450
4.5 Set Union and Intersection	451
<b>5 - Domino</b>	<b>452</b>
5.1 Some Definitions	452
5.2 Matrix Inverse	453
5.3 Matrix Division	455
5.4 Two or Three Steps in Geometry	455
5.5 Least Squares Fitting	457
Exercises	461

---

<b>The Specialist's Section</b>	<b>463</b>
Spe - 1   Encode and Decode	463
Spe - 2   Random Link	466
Spe - 3   Gamma and Beta Functions	468
Spe - 4   Domino and Rectangular Matrices	468
<b>CHAPTER L: SYSTEM INTERFACES</b>	<b>473</b>
<b>1 - Overview</b>	<b>473</b>
1.1   Commands, System Variables, and System Functions	473
1.2   Common Properties	474
1.3   Organisation	475
<b>2 - Workspace Management</b>	<b>475</b>
2.1   )WSID & []WSID Workspace Identification	476
2.2   []LX Startup Expression	477
2.3   )LOAD, )XLOAD & []LOAD Load a Workspace	478
2.4   )COPY, )PCOPY & []CY Import Objects	479
2.5   )LIB Explore a Workspace Library	480
2.6   )CLEAR & []CLEAR Clear the Active Workspace	480
2.7   )SAVE & []SAVE Save a Workspace	481
2.8   []WA Memory Space Available	482
<b>3 - Object Management</b>	<b>482</b>
3.1   )VARS, )FNS, )OPS, )OBS & []NL Object Lists	482
3.2   []NC Name Category	485
3.3   )ERASE & []EX Delete Objects	486
3.4   []SIZE Object Size	487
<b>4 - Environment Control &amp; Information</b>	<b>488</b>
4.1   []TS Current Date & Time	488
4.2   []PP Print Precision	488
4.3   []IO Index Origin	489
4.4   []AI Account Information	490
4.5   []PFKEY Programmable Function Keys	491

<b>5 - Function Definition and Processing</b>	<b>493</b>
5.1 )ED & □ED Edit Objects	493
5.2 □CR, □NR, □VR & □OR Function Representations	493
5.3 □FX Function Creation	496
5.4 □SHADOW Name Shadowing	497
5.5 □LOCK Locking a Function	497
5.6 □REFS Internal References	498
5.7 □AT Function Attributes	498
<b>6 - Debugging and Event Trapping</b>	<b>500</b>
<b>7 - Calculation Control</b>	<b>501</b>
7.1 Already Studied	501
7.2 □CT Comparison Tolerance	501
7.3 □DL Delay	503
<b>8 - Character Processing, Input/Output</b>	<b>503</b>
8.1 □AV & □AVU Atomic Vectors	503
8.2 □UCS Unicode Conversions	504
8.3 □TC Terminal Control	504
8.4 □A & □D Alphabet & Digits	505
8.5 □NULL Null Item	505
<b>9 - Miscellaneous</b>	<b>507</b>
9.1 □OFF & )OFF Quit APL	507
9.2 □SH, □CMD, )SH & )CMD Host System Commands	507
9.3 □PW Page Width	508
<b>The Specialist's Section</b>	<b>509</b>
Spe-1 Commands vs. System Functions	509
Spe-2 □SAVE	510
Spe-3 )CONTINUE Save & Continue	511
Spe-4 □OR	511
Spe-5 □VFI Verify and Fix Input	512
Spe-6 □RTL Response Time Limit	513
Spe-7 □MONITOR Execution Monitoring	514
Spe-8 System Variables vs. System Functions	516



---

<b>CHAPTER M: EVENT HANDLING</b>	<b>517</b>
<b>1 - Diagnostic Tools</b>	<b>518</b>
<b>2 - Event Trapping</b>	<b>518</b>
2.1 Event Numbers / Event Messages	519
2.2 :Trap / :Else / :EndTrap	520
2.3 □TRAP	522
2.4 Beware of These Errors	527
2.5 Neutralise the Traps	530
<b>3 - Event Simulation</b>	<b>530</b>
3.1 □SIGNAL Example	532
<b>CHAPTER N: FILE PROCESSING</b>	<b>535</b>
<b>1 - Component Files</b>	<b>536</b>
1.1 First Steps	536
1.2 Utility Functions	540
1.3 Shared Files	544
1.4 How to Queue File Operations	551
<b>2 - Data Representation</b>	<b>554</b>
2.1 Representation of Values	554
2.2 Representation of Variables	557
<b>3 - Native Files</b>	<b>559</b>
3.1 Similarities and Differences	559
3.2 Basic Operations	561
<b>4 - External Variables</b>	<b>566</b>
<b>The Specialist's Section</b>	<b>569</b>
Spe-1 Component Files	569
Spe-2 Native Files	572
<b>CHAPTER O: NAMESPACES</b>	<b>577</b>
<b>1 - Simple Namespaces</b>	<b>577</b>
1.1 Introduction	577
1.2 Use the Contents of a Namespace	583

<b>2 - More about References</b>	<b>588</b>
2.1 Namespace References	588
2.2 Display Form	591
<b>3 - Arrays of Refs</b>	<b>592</b>
3.1 Create an Array	592
3.2 Indexing Arrays of Refs	594
<b>4 - The Session Namespace</b>	<b>594</b>
<b>The Specialist's Section</b>	<b>597</b>
Spe - 1 The Dot as a Syntactic Element	597
Spe - 2 State Indicators	598
Spe - 3 Evaluation of Statements	598
Spe - 4 The Dyalog Workspace Explorer	600
Spe - 5 Control of Exported Functions	601
Spe - 6 Retrieving a Namespace Source	602
 <b>CHAPTER P: GRAPHICAL USER INTERFACE</b>	 <b>603</b>
<b>1 - Guidelines</b>	<b>603</b>
1.1 Terminology and Options	603
1.2 Create a Simple Dialog Box	607
1.3 Get Information	610
1.4 Changing Properties	611
1.5 Make It Work	612
<b>2 - Call-Back Functions</b>	<b>613</b>
2.1 Discovery	613
2.2 The Arguments of a Call-Back Function	618
2.3 The Result of a Call-Back Function	622
2.4 Improve It	625
2.5 Tracing Call-Back Functions	628
<b>3 - Selection Tools</b>	<b>628</b>
3.1 List	628
3.2 Combo	631
<b>4 - Colours, Fonts, and Root</b>	<b>633</b>
4.1 Colours	633
4.2 Fonts	633
4.3 Properties of the Root Object	636

<b>5 - Improve Your User Interface</b>	<b>639</b>
5.1    Default Keys	639
5.2    Enqueuing Events and Using Methods	640
5.3    Activating Objects	641
5.4    Form Appearance	642
<b>6 - Menus</b>	<b>644</b>
<b>7 - The Grid Object</b>	<b>646</b>
7.1    Geometry & Titles	647
7.2    Cell Types	648
7.3    Interaction with a Grid	653
7.4    Example	654
7.5    Multi-Level Titles	658
7.6    Some Additional Properties	660
<b>8 - Using Printers</b>	<b>661</b>
8.1    The Printer Object	661
8.2    Printer Management	664
<b>9 - And Also ...</b>	<b>667</b>
<b>The Specialist's Section</b>	<b>669</b>
Spe-1    Lists of Properties, Methods, Events	669
Spe-2    Different Syntaxes	671
Spe-3    Using Classes	672
<b>CHAPTER Q: INTERFACES</b>	<b>675</b>
<b>1 - Introduction</b>	<b>675</b>
<b>2 - OLE Interface with Excel</b>	<b>676</b>
2.1    Introduction	676
2.2    Create, Fill, and Save a Workbook	677
2.3    Open and Process a Workbook	680
2.4    A Simple Example	683
<b>3 - Name Association</b>	<b>686</b>
3.1    Introduction	686
3.2    Detailed Syntax	688
3.3    See How It Works	690

<b>CHAPTER R: SALT</b>	<b>693</b>
<b>1 - Introduction</b>	<b>693</b>
1.1 Why a Source Code Management System?	693
1.2 Using Script Files	697
1.3 Updating a Script From the APL Session	700
<b>2 - Version Management</b>	<b>702</b>
2.1 Creating and Using Versions	702
2.2 File Management	705
2.3 Comparing Scripts	707
<b>3 - Settings</b>	<b>709</b>
<b>The Specialist's Section</b>	<b>711</b>
<b>CHAPTER S: PUBLISHING TOOLS</b>	<b>713</b>
<b>1 - NewLeaf</b>	<b>714</b>
1.1 Getting Started	714
1.2 Frames and Text	715
1.3 Fonts	720
1.4 Tables	722
1.5 The Page Designer	726
1.6 More Tools, Better Quality	735
<b>2 - RainPro</b>	<b>738</b>
2.1 Getting started	738
2.2 Multiple Bar Chart	740
2.3 Scattered Points	744
2.4 Min-Max Vertical Lines	750
2.5 Polar Representations	753
2.6 Multiple Charts	754
2.7 There is Much More To Explore!	756

---

<b>CHAPTER X: SOLUTIONS</b>	<b>757</b>
Chapter B	757
Chapter C	758
Chapter D	761
Chapter G	765
Chapter I	767
Chapter J	768
Chapter K	771
<b>APPENDICES</b>	<b>773</b>
<b>Appendix 1 : Scalar Functions</b>	<b>773</b>
<b>Appendix 2 : Invoking the Editor</b>	<b>774</b>
<b>Appendix 3 : Selective Assignment</b>	<b>775</b>
<b>Appendix 4 : Dyalog APL Operators</b>	<b>776</b>
<b>Appendix 5 : Identity Items</b>	<b>777</b>
<b>Appendix 6 : Event Numbers</b>	<b>778</b>
<b>Appendix 7 : System Variables and Functions</b>	<b>780</b>
<b>Appendix 8 : System Commands</b>	<b>783</b>
<b>Appendix 9 : Symbolic Index</b>	<b>784</b>
<b>INDEX</b>	<b>M-789</b>



# Introduction - Will You Play APL With Me?

## Will You Follow Us?

We would like to have you discover a new land, a land where people who may or may not be specialists in programming can process their data, build computerised applications, and take pleasure in using a programming language which is an extremely elegant and powerful tool of thought.

### **Beware: Dyalog APL is Addictive!**

Among the hundreds of programming languages which have been created, most of them share the same fundamentals, the same basic instruction set, approximately the same functions, and by and large the same methods to control the logic of a program. This greatly influences the way people imagine and build solutions to computing problems. Because the languages are so similar, the solutions are similar. Does it mean that these are the only ways of solving problems? Of course not!

Dyalog APL is there to open doors, windows, and minds, prove that original new methods do exist, and that mathematics is not limited to four basic operations. Using APL will expand and extend the range of mental models that you use to solve problems, but beware:

Once you are hooked on APL, there is a real risk that you will no longer accept the limitations of "traditional" programming languages.

### **Installation and Keyboard**

If you do not have access to a computer with Dyalog APL installed, you should still be able to gain an appreciation of the language from these pages and, we hope, enjoy the experience.

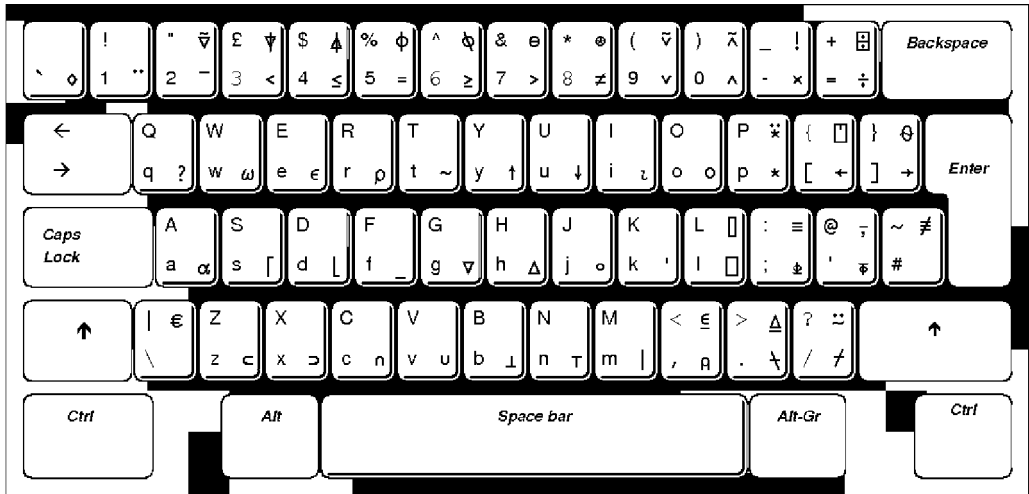
If you have installed Dyalog, not only can you read this book, but you can also experiment on your own computer using the examples below, and invent your own data and calculations.

If you have a copy of Dyalog APL, install it as explained in the User Guide. Just run the installation program and accept all the defaults; there is no need to change anything.

You might like to refer to section A-1 for additional installation hints.

As you will see in the following pages, APL uses special symbols, like  $\phi$ ,  $\rho$ , and  $\downarrow$ , which you enter using a special keyboard layout.

You will not need all of the special symbols to read the following pages. The picture below illustrates a cut-down version of the UK keyboard layout, with only the APL symbols that are referred to in this introduction. They are shown on a grey background. The US keyboard is slightly different, but the symbols we need are positioned identically. The full keyboard layout is shown in the User Guide.



Here is how the keyboard is to be used:

- All the standard English letters, numerals and symbols are typed as usual.
- The majority of the APL symbols are obtained by pressing the "*Ctrl*" key, in conjunction with another key. For example, to obtain  $\rho$ , you must press *Ctrl* and R. From now on, this keystroke will be identified as "*Ctrl*+R". On the keyboard layout illustrated above, the symbols that you enter this way are printed at the bottom-right corner of each key.
- Most other APL symbols are obtained by simultaneously pressing "*Ctrl*" and "*Shift*" and then the appropriate key. For example, to obtain  $\phi$ , you must press *Ctrl* and *Shift* and 6. From now on, this keystroke will be identified as "*Ctrl*+*Shift*+6". On the keyboard layout illustrated above, the symbols that you enter this way are printed at the top-right corner of each key.
- A few APL symbols are obtained by pressing the "*Alt Gr*" key (or *Ctrl*+*Alt* if your keyboard does not have an "*Alt Gr*" key), simultaneously with another key, but you don't have to bother about them here: We won't be using them in the examples in this introduction.



- In case you initially have any difficulty with the keyboard, there is a "language bar" on top of the session screen, with all the APL symbols on it. When you need a symbol, you just have to click on it and the symbol will appear wherever your cursor is positioned.

You may notice that some symbols appear twice on the keyboard. This is the case for example for the symbols `< = >`. These symbols are all part of a normal keyboard (the black ones), but they have been repeated on the APL keyboard, mostly in order to group the symbols used for *comparison functions* together (the red ones). Do not worry: No matter which key you use to produce one of the duplicated symbols, you'll obtain the same result.

## Utilities and Data

For most of the examples, you can just type what you read in the following pages, but sometimes you will need some data which we have prepared for you.

This data is contained in a special file (called a Workspace) named `DyalogTutor_EN.dws` which accompanies this book.

If you don't have the file, please refer to section A-1 for instructions on how to download it.

The file can only be opened by Dyalog APL. You can open it as follows:

- In Windows Explorer, double-click on the file's name. APL will be started, and it will then open the file.
- Or, start APL by double clicking on the Dyalog APL icon. Then, using the "File/Open" menu, search for the file and open it.

Once the workspace has been loaded, a welcome message is displayed, and you can check that the workspace contains the variables we shall be using in the following pages:

```
)vars
Actual Ages AlphLower AlphUpper Area Big Category etc...
```

You can display the contents of any variable by hovering over its name with the mouse-pointer, by double-clicking on its name, or just by typing its name and pressing the *Enter* key, like this:

```
Forecast
150 200 100 80 80 80
300 330 360 400 500 520
100 250 350 380 400 450
50 120 220 300 320 350
```

**Now, you are ready, fasten your seatbelts, we're off!**

# Our First Steps into APL's Magic World

## Simple Operations

In APL, what you type starts 6 characters right from the left margin (we say it is "*indented*"), whereas the computer's response begins at the left margin. For additional clarity, in the following pages the characters typed by the user are printed in red, the response given by the computer being in black.

You will notice that in the examples given in the book we very often put a blank space between a symbol and the surrounding names or values. This is in most cases unnecessary; we only do so in order to improve readability. Later on, we will gradually cease to insert the blank spaces in expressions that you should become familiar with along the way.

An expression gets evaluated and the result shown in the session when you press the *Enter* key. Let's try some simple expressions:

```

      27 + 53
80
      1271 - 708
563
      86 ÷ 4           The Divide sign is obtained using Ctrl+ =
21.5
      59 × 8           The Multiply sign is obtained using Ctrl+ -
472

```

You can see that APL behaves like any hand-held calculator with, however, a small difference; multiplication is represented by the multiplication symbol (  $\times$  ) which is used in schools in many countries; likewise for division (  $\div$  ).

In most other computer languages, a star  $*$  is used for Multiply and  $/$  for Divide. This is a legacy of the early days of computers, when the character set was limited to the one available on a typewriter. At the time it was decided to use  $*$  and  $/$  in place of  $\times$  and  $\div$ . But it is now possible to display any type of symbol on a screen and on a printer, and this transposition is no longer justifiable. The use of common symbols, which are taught all over the world, aids the understanding of APL by non programmers.

If you are familiar with other programming languages, you may occasionally and erroneously use  $*$  instead of  $\times$ . Let's see what might happen then:

```

      7 * 3
343

```

⇐ In APL the star means "Power" so that  $7*3$  is equivalent to  $7 \times 7 \times 7$

## Variables

As in any programming language, it is possible to create variables. Just choose a name and use the left arrow to assign it a value. In APL a numeric value can consist of a single number, or several numbers separated by at least one blank space. The arrow can be obtained using *Ctrl+ [*

```
VAT ← 19.6           ⇔ Read it as: VAT gets 19.6
Years ← 1952 1943 1986 2007
```

The names are "*case sensitive*". It means that three variables named respectively VAT, Vat, and vat, would be distinct, and may contain different values.

To ask for the contents of a variable, just type its name and press *Enter*, like this:

```
VAT
19.6
Years
1952 1943 1986 2007
```

## Array Processing

APL is able to operate on two sets of numbers, provided those two sets have the same "shape". For the moment, understand this as "the same number of items". For example, suppose that you have a list of prices of 5 products, and the quantity bought of each:

```
Prices      5.20  11.50  3.60  4.00  8.45
Quantities  2      1      3      6      2
```

You can create two variables like this:

```
Price ← 5.2 11.5 3.6 4 8.45
Qty   ← 2 1 3 6 2
```

When multiplied together, the variables are multiplied item by item, and produce a result of the same length. That result can be assigned to a new variable.

```
Costs ← Price × Qty
Costs
10.4 11.5 10.8 24 16.9
```

This *array processing* capability eliminates most of the "loops" which are common to other programming languages. This remains true even if the data is not a simple list but a multi-dimensional array, of almost any size and number of dimensions.

To make it clear, imagine that a Sales Director makes forecasts for sales of 4 products over the coming 6 months, and assigns them to the variable `Forecast`. At the end of the 6 months, he assigns the real values to the variable `Actual`. Here they are:

Forecast						Actual					
150	200	100	80	80	80	141	188	111	87	82	74
300	330	360	400	500	520	321	306	352	403	497	507
100	250	350	380	400	450	118	283	397	424	411	409
50	120	220	300	320	350	43	91	187	306	318	363

We have not yet explained how you can build such arrays of data, but if you have APL installed, these variables are provided in the Workspace file named "DyalogTutor\_EN.dws". Refer to the "*Utilities and Data*" section above to see how you can load the workspace and access the data.

It is clear that the first idea of any Sales Director will be to ask for the differences between what he expected and what he has really got. This can be done easily by typing:

Actual - Forecast											
-9	-12	11	7	2	-6	⇒ Note that to distinguish the sign attached to negative values from subtraction, negative values are shown with a high minus sign.					
21	-24	-8	3	-3	-13						
18	33	47	44	11	-41						
-7	-29	-33	6	-2	13						

To enter negative values, this high minus sign can be obtained by pressing `Ctrl+2`.

In most traditional programming languages an operation like the one above requires two embedded loops. See what is needed in PASCAL:

```
DO UNTIL I=4
  DO UNTIL J=6
    DIFF(I,J):=ACTUAL(I,J)-FORECAST(I,J)
  END
END.
```

Even if this may seem obvious to a programmer, it is worth noting that most of the code has nothing to do with the user requirement. The only important thing (subtract forecasts from actual values) is hidden behind the detailed workings of the computer program.

To have a calculation done by a machine, one must translate our human wording into something that the computer can understand. With traditional languages, most of that effort is made by the man, to produce a program like the PASCAL example above. The great advantage of APL is that the man has generally much less effort to make, and the machine does the rest.

We have seen that APL will work on two variables of the same shape; it also works if one of the variables is a single item, which is called a *scalar*. If so, the other variable may be of any shape.

For example, if we want to calculate the amount of 19.6% VAT applied to the variable `Price` above, we can type `Price × VAT ÷ 100` (or `VAT × Price ÷ 100` as well), as shown here:

```
Price × VAT ÷ 100
1.0192 2.254 0.7056 0.784 1.6562 ⇐ This result would require some rounding
                                     but this is not important for now
```

## More Symbols

Most programming languages represent only a very small subset of the mathematical functions using symbols (typically `+`, `-`, `*` and `/`). The creator of APL, Kenneth E. Iverson, chose to include many traditional mathematical symbols in his language, and also added some new symbols to the set that we already know so well.

E.g.: Many functions which in other programming languages are library routines with names like "Maximum" have their own symbols in APL.

The function "*Maximum*" (`∩`) returns the greater of two numbers, or of two arrays of numbers compared item by item.

There is also, as one might expect, a symbol for "*Minimum*" (`∪`).

```
75.6 ∩ 87.3 ⇐ Maximum (Ctrl+S)
87.3
11 28 52 14 ∩ 30 10 50 20 ⇐ Comparison item by item
30 28 52 20
11 28 52 14 ∪ 20 ⇐ Minimum (Ctrl+D)
11 20 20 14
```

APL supports about 70 symbols. Since some symbols have more than one meaning one could argue at length about the exact number.

This is nothing to worry about: Some of the symbols are familiar; such as `×` or `>` or again `÷` and `-`, but also `!` and a good many others.

## Most Symbols Have a Double Meaning

This is not a peculiarity of APL; in algebra we are familiar with the use of symbols as common as the minus sign being used in two different ways.

In the expression  $a = x - y$  the minus sign means subtract  
Whereas in  $a = -y$  the minus sign indicates the negation of  $y$ , that's different

The first form is called the "*dyadic*" use of the symbol.

The second form is called the "*monadic*" use of the symbol.

It is the same in APL, where most of the symbols can have two meanings.

For example, to find the shape (the dimensions) of an array, one uses the Greek letter Rho ( $\rho$ ), which can be read "*shape of ...*", in its monadic use. It is produced using *Ctrl*+*R*.

```

      ρ Price
5
      ρ Forecast
4 6
      ↵ Monadic use
      Price has 5 items
      Forecast has 4 rows of 6 items

```

Used dyadically, the same symbol will organise items into a specified shape. For example, suppose that we want to create the matrix below:

```

25 60
33 47
11 44
53 28

```

We must give the computer two pieces of information:

- First the *shape* to give to the matrix: 4 2 (4 rows of 2 columns)
- Next the *contents* of the matrix: 25 60 33 47 11 44 53 28

It is the symbol  $\rho$  (Rho) which makes the connection between the shape and the contents:

```

      Tab ← 4 2 ρ 25 60 33 47 11 44 53 28
      Tab
25 60
33 47
11 44
53 28

```

A new variable `Tab` is thereby created, and this is also how the variables `Forecast` and `Actual` above were made.

## Conventions

In APL, we give special names to certain shapes of data:

- *Scalar* is used for a single value, a number like 456.18 or a single letter like 'Q'.
- *Vector* is a plain list of values  
It may be composed of numbers like Price and Qty,  
or of letters like 'Once upon a time' within single quotes
- *Matrix* is an array with two dimensions, like Forecast or Tab
- *Array* is a generic word for any set of values, whatever the number of its dimensions
- *Table* is a common word used for arrays with 2 dimensions (matrices)
- *Cube* is a common word used for arrays with 3 dimensions

## Reduction Unifies Traditional Notations

Perhaps you remember the variable *Costs*:           10.4 11.5 10.8 24 16.9

So what must we do to work out the total? Mathematicians are creative people who long ago devised the symbol  $\sum$ , always with a pretty collection of indices above and below, which make it complex to understand and to type on a typewriter.

In APL, the operation is written like this:

```
+/ Costs
73.6
```

Simple isn't it? This gives the total of all the items of the array.

You can read this as "*Plus Reduction*" of the variable *Costs*.

To gain a better understanding of the process:

```
When we write an instruction such as      +/ 21 45 18 27 11
- it works as if we had written           21 + 45 + 18 + 27 + 11
- and we obtain the sum                   122
```

In fact, it works as if we had "inserted" the symbol + between the values.

```
But then, if we write                     ×/ 21 45 18 27 11
- it is as if we had written             21 × 45 × 18 × 27 × 11
- so, we get the product                 5051970
```

Similarly, if we write  $\lceil / 21\ 45\ 18\ 27\ 11$   
 - it is as if we had written  $21\ \lceil\ 45\ \lceil\ 18\ \lceil\ 27\ \lceil\ 11$   
 - so, we obtain the largest term  $45$

Reduction, represented by the symbol  $/$ , belongs to a special category of symbols called *Operators*. All the other symbols ( $+ - \times \lceil \rho \Phi \dots$ ) are called *Functions* (addition, subtraction, multiplication, maximum, shape, etc.).

The arguments of a function are data (arrays):  $\text{Price} \times \text{Qty}$

Whereas at least one of the arguments of an operator is a function:  $+ / \text{Qty}$

The left argument of *Reduction* can be one of many of the APL symbols, and it can also be the name of a user-defined program. This may give you an idea of the generality and power of the concept.

Dyalog APL contains 10 such powerful operators. If that is not enough, you can even write your own operators, just like you can write your own functions!

## Let's Write Our First Programs

Imagine that we want to calculate the average of the following numbers:

$\text{Val} \leftarrow 22\ 37\ 41\ 19\ 54\ 11\ 34$

We must:

- first calculate the sum of the values:  $+ / \text{Val}$  giving  $218$
- next calculate the number of values:  $\rho \text{Val}$  giving  $7$
- and finally divide one result by the other

The calculation can be written as the single formula:  $(+ / \text{Val}) \div (\rho \text{Val})$

As it is quite likely that we shall often want to make this sort of calculation, it is preferable to store this expression in the form of a program.

In APL we prefer the name *defined function* to the name "program".

Defined functions may be used in the same way as the built-in functions represented by special symbols like  $+ - \times - > \rho \dots$ , which are called *primitive functions*.

To define a simple function like this one, here is the easiest way:

$\text{Average} \leftarrow \{ (+ / \omega) \div (\rho \omega) \}$



Average is the program name  
 $\omega$  is a generic symbol which represents the array passed on the right.  
 $\alpha$  would be the generic symbol for the array passed on the left, if any

The definition of the function is delimited by a set of curly braces { and }. For more complex functions it is also possible to use a text editor, but this is beyond the scope of this short introduction.

Once defined, this function may be invoked in a very simple way:

```

Average Val                                     ⇐ For execution,  $\omega$  will get the values
31.1428571428                                     contained in Val
Average 12 74 56 23
41.25
  
```

Let us also write two little dyadic functions, the left argument of which is  $\alpha$ , and the right is  $\omega$ :

```

Plus ← { $\alpha + \omega$ }
Times ← { $\alpha \times \omega$ }
(3 Plus 6) Times (5 Plus 2)
63
  
```

As you can see, these functions behave exactly as if we had written  $(3+6) \times (5+2)$

We said in the preceding section that a user-defined program could be used by the *Reduce* operator; let us try:

```

Plus/ Val                                       ⇐ It works!
218
  
```

## Indexing

Returning to our vector of numbers Val:            22 37 41 19 54 11 34

In order to extract the 4th item, we just write:    Val[4]

In many other programming languages one uses parentheses instead of brackets; this is not very different.

What is new is that one can extract several items in one instruction.

```

Val
22 37 41 19 54 11 34
Val[2 4 7 1 4]                                     ⇐ One may extract the same item twice or more
37 19 34 22 19
  
```

And of course, in the same way, one may modify one or more items of `Val` using their indexes. Naturally, one must provide as many values as there are items to modify, or a single value for all:

```

      Val[3 5 1] ← 0
      Val
0 37 0 19 0 11 34
      Val[3 5 1] ← 300 77 111
      Val
111 37 300 19 77 11 34  ⇐ You can check that the 3rd item is now 300, the 5th is 77, etc.

```

It is often necessary to extract the first few items from a list of values, for example the first 5. Nothing could be easier:

```

      Val[1 2 3 4 5]
111 37 300 19 77

```

But if one needs to extract the first 500 items from a long list, typing the integers from 1 to 500 would of course be very inconvenient.

This is why APL has been given the symbol  $\iota$  (*Iota*), which produces the set of the first  $n$  integers ( $\iota$  can be obtained using *Ctrl+I*)

Thus, instead of writing `1 2 3 4 5 6 7 8`, it is sufficient to write  `$\iota$ 8`.

And to extract the first 500 terms of a large vector, one may write: `Big[ $\iota$ 500]`

We shall discover later an even simpler method.

## Calculating Without Writing Programs

The employees of a company are divided into three hierarchical categories, denoted simply 1, 2, and 3. One assigns to two variables the salaries and the categories of these employees; as partly shown here:

```

Salaries   ← 4225 1619 3706 2240 2076 1389 3916 3918 4939 2735 ...
Categories ←   3   1   3   2   2   1   3   3   3   2 ...

```

Do they never want to increase these salaries? (what has our poor world come to!).

A rumour reaches us about their plans: They want a different percentage increase for each category, according to the following scale:

Category	Suggested increase
1	8%
2	5%
3	2%

How much is this going to cost the company?

We create a variable containing the above three rates:

```
Rates ← 8 5 2 ÷ 100    ⇔ APL allows us to divide three numbers by a single one
```

```
Rates
```

```
0.08 0.05 0.02
```

The first employee is in category 3, so the rate that applies to him is:

```
Rates[3]
```

```
0.02
```

It follows that the first 5 employees, being in categories 3 1 3 2 2 respectively, are entitled to the following increases:

```
Rates[3 1 3 2 2]
```

```
0.02 0.08 0.02 0.05 0.05
```

More generally, the rates applied to all of our employees could be obtained like this:

```
Rates[Categories]
```

```
0.02 0.08 0.02 0.05 0.05 0.08 0.02 0.02 0.02 0.05 0.05 0.02 etc.
```

Having the rates, one has just to multiply by the salaries to obtain the individual increases:

```
Salaries × Rates[Categories]
```

```
84.5 129.52 74.12 112 103.8 111.12 78.32 78.36 98.78 136.75 etc.
```

Finally, by adding them all, one will know how much it will cost the company:

```
+ / Salaries × Rates[Categories]
```

```
2177.41
```

You may note that:

- The expression remains valid whatever the number of employees or categories,
- the result has been obtained without writing any program,
- and this expression can be read as the simplest possible English, like this:

***Sum the Salaries multiplied by Rates according to Categories***

Clever, no?

This illustrates how the expression of a solution in APL can be very close to the way that the solution could be phrased in everyday language. This also shows clearly that the ways of reasoning induced by traditional programming languages are not the only possible ones. This difference and originality, introduced by APL, are among the major features of the language.

## Friendly Binary Data

APL makes much use of binary data. It is most often created by means of relational functions like = or >, which give the answer 1 or 0, depending whether the relation is true or not:

```
Salaries > 3000
1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0 0
```

```
Actual > Forecast
0 0 1 1 1 0
1 0 0 1 0 0
1 1 1 1 1 0
0 0 0 1 0 1
```

⇐ One can see the favourable results instantly

APL offers the conventional mathematical form of the 6 relational functions:

< ≤ = ≥ > ≠

Naturally one can operate on this binary data using all the functions of Boolean algebra, and moreover, the symbols used are those familiar to mathematicians of all nationalities around the world:

Function *AND* is represented by the symbol  $\wedge$  (represented by the word AND in many programming languages)

Function *OR* is represented by the symbol  $\vee$  (represented by the word OR in these languages)

Thus, if I am looking for people in category 3 whose salary is less than 4000 euros, I can write:

```
(Categories = 3) ^ (Salaries < 4000)
0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1
```

In fact APL offers all the functions of Boolean algebra, including some perhaps less familiar functions like *NOR* and *NAND* (Not-OR and Not-AND), but they are very useful in finance and electronic automation.

There is, however, no special symbol for the function *Exclusive OR* (often called *XOR*). This is because it is not needed: The function *Not Equal*  $\neq$  gives the same result as *Exclusive OR* when it is used with Boolean values, as you can see below:

```
0 0 1 1  $\neq$  0 1 0 1
0 1 1 0
```

Finally, not only can these binary vectors be used as we have described but also for novel purposes, such as counting and selecting.

## Counting

Having found which salaries are less than 2500 euros by means of the following expression:

```
Salaries < 2500
0 1 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0
```

It is easy to add all the 1s and 0s to calculate how many people earn less than 2500 euros:

```
+ / Salaries < 2500
8
```

## Selection

One can also use the binary vector as a "mask" to select the items corresponding to the binary "1"s from another array:

```
1 1 0 1 0 0 1 / 23 55 17 46 81 82 83
23 55 46 83
```

The procedure is identical for character data:

```
1 0 1 0 0 0 0 1 1 / 'Drumstick'
Duck
```

This function, called *Compress*, is particularly useful for extracting the items conforming to a given criterion from a variable. For example, to display the salaries of people in Category 2, one writes:

```
(Categories = 2) / Salaries
2240 2076 2735 3278 1339 3319 ⇔ Powerful, isn't it?
```

## Discovery

To practise our skills some more, let us find in our variable `Val` the positions of numbers greater than 35. Here are the necessary steps:

```
Val      ←      22 37 41 19 54 11 34
Val>35   is      0  1  1  0  1  0  0
ρVal     is      7
ιρVal    is      1 2 3 4 5 6 7 ⇔ All possible positions
```

Let us compare two of these results

```
Val>35   ⇨      0 1 1 0 1 0 0
ιρVal    ⇨      1 2 3 4 5 6 7
```

You can see that that if you eliminate (using *Compress*) the items which correspond to zeros in order to retain only those corresponding to 1, you easily get the positions required: 2 3 5

Thus the job may be done as follows:

```
(Val>35) / ⍉Val
2 3 5
```

This expression is applicable in many different situations.

Here is a similar use, but applied to character data: To find the positions of "a" within a phrase; the method is the same.

```
Phrase ← 'Panama is a canal between Atlantic and Pacific'
(Phrase = 'a') / ⍉Phrase
2 4 6 11 14 16 30 36 41      ⇨ You can check it!
```

## A Touch of Modern Math

Proudly having found all the "a"s, we may wish to find all the vowels.

Alas, although we can write `Phrase = 'a'`, because a vector can be compared with a single value, one cannot write `Phrase = 'aeiouy'`<sup>(1)</sup>, because that would require the item by item comparison of a phrase of 46 letters and "aeiouy" which has only 6.

In other words: You may compare 46 letters with 46 other letters, or compare them with one letter, but not with 6.

So we shall use a new function: *Membership* which is represented by the symbol  $\epsilon$ , also used in mathematics. ( $\epsilon$  can be obtained by pressing *Ctrl+E*)

The expression `A  $\epsilon$  B` returns a Boolean result which indicates which items of the variable A appear in the variable B, wherever they may be. And it works no matter what are the shapes, the dimensions or the type (numeric or character) of A and B, a pure marvel!

For example:

```
5 7 2 8 4 9  $\epsilon$  3 4 5 6
1 0 0 0 1 0      ⇨ Only 5 and 4 are found in 3 4 5 6
'dandelion'  $\epsilon$  'garden'
1 1 1 1 1 0 0 0 1      ⇨ The letters "lio" do not appear in "garden"
```

---

<sup>1</sup> "Y" is considered to be a vowel in many European languages.

So in pursuit of our enquiry we shall write:

```
(Phrase ∈ 'aeiouy') / ιPhrase
2 4 6 8 11 14 16 20 23 24 30 33 36 41 43 45
```

One can also use membership between a vector and a matrix, as shown below, assuming that the list of towns is a variable created earlier.

We have represented side by side the variable itself and the result of using *Membership*:

Towns	Towns ∈ 'aeiouy'
Canberra	0 1 0 0 1 0 0 1 0 0
Paris	0 1 0 1 0 0 0 0 0 0
Washington	0 1 0 0 1 0 0 0 1 0
Moscow	0 1 0 0 1 0 0 0 0 0
Martigues	0 1 0 0 1 0 1 1 0 0
Mexico	0 1 0 1 0 1 0 0 0 0

We can reverse the expression, but the result has always the same shape as the left argument:

```
'aeiouy' ∈ Towns
1 1 1 1 1 0      ⇔ None of the town names contains a "y"
```

## A Powerful Search Function

We have harnessed a very useful method to look for the positions of letters or numbers in a vector, but the answer obtained does not provide a one to one correspondence between the search values and the resultant positions:

```
List ← 15 40 63 18 27 40 33 29 40 88      ⇔ Vector of values
Where ← 29 63 40 33 50                    ⇔ We want to find these
(List ∈ Where) / ιList                    ⇔ Let's apply our method
2 3 6 7 8 9                                ⇔ Positions found
```

The positions are correct, but 29 is not in position 2, and 40 is not in position 6.

The question we have answered using the expression above is: "In which positions in *List* do we find a number that also appears somewhere in *Where*?"

If we want to answer the slightly different question: "Where in *List* do we find each number in *Where*?" we need to use a different method.

This new method uses the dyadic form of the symbol  $\iota$  (*Iota*).

```
List ← 15 40 63 18 27 40 33 29 40 88      ⇔ Same vector of values
Where ← 29 63 40 33 50                    ⇔ Where are these?
List ι Where                                ⇔ New method using dyadic ι
8 3 2 7 11                                  ⇔ Positions found
```

It is true that 29, 63, 40 and 33, occur respectively in positions 8, 3, 2 and 7. It's much better!

But, first surprise: The value 40 occurs 3 times in `List`, but only the first one is reported in the result. This is because, by definition, dyadic *Iota* returns only the first occurrence of a given item. If the response for each value sought has to match a position; how may one, looking for 5 numbers, obtain 7 results?

Second surprise: The value 50 is reported as being found in position 11 in a vector comprising only 10 items! This is how the function *IndexOf* (dyadic  $\iota$ ) reports that a value is absent.

At first sight this seems a bit surprising, but in fact it is a property which makes this function so generally powerful, as we shall soon see.

## An Example

A car manufacturer decides that he will offer his customers a discount on the catalogue price (you can see how this example is imaginary!)

The country has been divided into 100 areas, and the discount rate will depend on the area according to the following table:

Area	Discount
17	9 %
50	8 %
59	6 %
84	5 %
89	4 %
Others	2 %

The problem is to calculate the discount rate that may be claimed for a potential customer who lives in given area  $D$ ; for example  $D \leftarrow 84$ .

Let us begin by creating two variables:

```
Area      ← 17 50 59 84 89
Discount ← 9  8  6  5  4  2
```

Let us see if 84 is in the list of favoured areas:

```
D ∈ Area
1
Area ι D
4
⇒ Yes, it's there
⇒ 84 is the 4th item in the list
```



Null Item	505	<i>Q</i>	
N-Wise Reduce	380	Quad (Evaluated Input)	203
<b>O</b>		Quad (symbol)	196
Object Representation	495, 511	Quiet Load	478
Object Size	487	Quit APL	78, 507
OLE	675	Quote (delimiter)	53
Operators	105, 377	Quote-Quad (Character Input)	204
Or (function)	100	<b>R</b>	
Orange	449	RainPro	738
Order of Evaluation	94	Random Link	466
OrIf (flow control)	170	Randomised Values	444
Outer Product	386	Rank	50
Overtaking	301	Rank Error	254
<b>P</b>		Ravel	85, 132
Page Designer (NewLeaf)	726	Ravel with Axis	143
Page Width	508	Read/Write Text Files	199
Partition	365, 374	Reciprocal	97
Pass Numbers	547, 570	Recursion	217
Pass-Through Value	62	Reduction	104, 350
PDF Format	714	Reduction of Empty Vectors	425
Perimeter	406	Reformat a Function	224
Pervasive Functions	103, 346	Refs	588
PFK	491	Refs in a function	498
Pi (trigonometry)	449	Relationship Functions	93
Pick	361, 376	Remainder	94
Pie Chart (RainPro)	755	Repeat (flow control)	178
PNG Output	739	Replay Input	223
Polar Representation (RainPro)	753	Replication (Replicate)	120
Polygon Area & Perimeter	406	Representation of Values	554
Polynomials	442	Representation of Variables	557
Position (function)	121	Reset the State Indicator	250
Power Function	44, 92	Reshape	47
Power Operator	415	Residue	94, 145
Primitive Functions	89	Reverse	312
Print Precision	488	Rho	47
Print your experiments	37	Right Arrow	167
Printer Object (GUI)	661	Right-align Text	443
Printers (GUI)	637, 661	Right-to-Left Evaluation	95
Procedural Functions	147, 154	Roll (monadic ?)	445
Programmable Function Keys	491	Root Object	604
Properties (GUI)	604	Root Object (GUI)	636
PropList (GUI)	669	Rotate	314
Protected Copy	76, 480	Round Up/Down	98
Prototype	358		
Pseudo Right-Inverse of a Matrix	470		

<b>S</b>		Styles (NewLeaf)	718
SALT	693	Synonyms	218
Save a WS	74, 481	Syntax Error	256
Scalar	51	System Commands	72, 473
Scalar Dyadic Functions	90	System Interfaces	473
Scan	383	System Variables/Functions	473
Scattered Points (RainPro)	744	<b>T</b>	
Scientific Representation	83, 295	Table	51
Script Files (SALT)	694	Take	299
Scroll Back/Forward	39	Target (SALT)	698
Search / Find	435	Terminal Control	374, 504
Search Path	74, 586	Text Editor	160, 220
Search Tool	253	Threads	205, 418
Select (flow control)	174	Time Limit	513
Selective Assignment	327, 364	Time Stamp	488
Semi-colon	59	Trace Points	265
Session Log	38	Tracing Call-Back Functions	628
Session Namespace	594	Transpose (dyadic)	316, 322
Set Union/Intersection	451	Transpose (monadic)	312
Sets of Equations	454	Trap	520
Settings (SALT)	698, 709	Trigonometry	449
Shadowed Names	230, 497	Type	302, 358, 374
Shape	50	<b>U</b>	
Shape of a Result	84, 111	UK APL Keyboard	35
Shape of an Array	47	Underscore	45
Shared Component Files	544	Underscored Letters	45
Shortest Route in a Graph	401	Undo / Redo	222
Show/Hide Line Numbers	224	Unicode Conversions	504
Shy Result	210, 216	Unicode Edition	31, 433, 504
Signum	97	Union	369, 451
Size of Objects	487	Unique (function)	132
Sorting Data	431	Universal Character Set	504
Source-Code Management	693	Unnamed D-Fns	153
Spawn	418	Unnamed Namespace	579
Special Notations	111, 117, 121	Until (flow control)	178
Special Syntax	420	US APL Keyboard	36
Split	354, 374	User Identity	545
Squad	62, 678	User-Defined Events (GUI)	640
Startup Expression	477	User-Defined Functions	89
State Indicator	241	User-Defined Operators	421
Statement Separator (Diamond)	205	<b>V</b>	
Stop (Trap action code)	523	Valence of a Function	207, 499
Stops	262	Value Error	252
Stops (Break points)	262, 265	Variable/Function Names	45
Strand Notation	64, 331		
Strong Interrupt	183		

		<b>Index</b>	795
Vector	51	Workspace Explorer	600
Vector Notation	64, 331	Workspace Identification	77, 476
Vector Representation	495	Workspace Management	475
Verify & Fix Input	512	Workspace Search Path	74, 478
Version Management (SALT)	702	WS	72
Visual Representation	159	WS Full (error)	256
<b><i>W</i></b>		<b><i>X</i></b>	
Weak Interrupt	183	Xor (function)	100
While (flow control)	178	<b><i>Z</i></b>	
Windows Language Bar	32	Zilde	126
With (control structure)	591, 609		
Without (function)	102		
Workspace	36, 72		

