Dyalog APL Language Reference Guide

Dyalog version 20.0

DYALOG

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2025 by Dyalog Limited
All rights reserved.

Dyalog APL Language Reference Guide

Dyalog version: 20.0
Document Revision: 2026-01-12 main:1fdd0d6ff9

Unless stated otherwise, all examples in this document assume that `⎕IO ⎕ML ← 1`

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.

email: support@dyalog.com
https://www.dyalog.com

TRADEMARKS:

Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle®, MySQL, and Java™ are registered trademarks of Oracle and/or its affiliates.
JavaScript™ is a trademark of Oracle Corporation.
Unicode is a registered trademarks of Unicode, Inc. in the U.S. and other countries.
UNIX® is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the U.S. and other countries.
macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Except where otherwise noted, this content is licensed under a Creative Commons Attribution 4.0 International licence.

Contents

1 Key to Notation 18

2 Glyphs 19

3 Primitive Functions 30

3.1 Primitive Functions 31
3.2 Notes 34
3.3 Primitive Functions (by Category) 36
3.4 Functions (A-Z) 48

3.4.1 Binomial 48
3.4.2 Catenate Laminate 48
3.4.3 Catenate First 50
3.4.4 Ceiling 50
3.4.5 Circular Functions 51
3.4.6 Conjugate 52
3.4.7 Deal 52
3.4.8 Decode 53
3.4.9 Depth 55
3.4.10 Direction 56
3.4.11 Divide 56
3.4.12 Drop 57

3.4.12.1 Drop 57
3.4.12.2 Drop with Axes 58

3.4.13 Dyadic Execute 60
3.4.14 Dyadic Grade Down 62
3.4.15 Dyadic Grade Up 64
3.4.16 Dyadic Transpose 66
3.4.17 Enclose 68

3.4.17.1 Enclose 68
3.4.17.2 Enclose with Axes 69

3.4.18 Encode 71

3.4.19 Enlist 72
3.4.20 Equal To 73
3.4.21 Execute 75
3.4.22 Expand 76
3.4.23 Expand First 77
3.4.24 Exponential 77
3.4.25 Factorial 78
3.4.26 Find 78
3.4.27 First 79
3.4.28 Floor 80
3.4.29 Format 81
3.4.30 Format by Specification 85
3.4.31 Grade Down 87
3.4.32 Grade Up 89
3.4.33 Greater Than 92
3.4.34 Greater Than Or Equal To 92
3.4.35 Index 93
3.4.35.1 Index 93
3.4.35.2 Index with Axes 95
3.4.36 Index Generator 97
3.4.37 Index Of 98
3.4.38 Intersection 101
3.4.39 Interval Index 101
3.4.40 Left 109
3.4.41 Less Than 110
3.4.42 Less Than Or Equal To 110
3.4.43 Logarithm 110
3.4.44 Lowest Common Multiple/AND 111
3.4.45 Magnitude 112
3.4.46 Match 112
3.4.47 Materialise 113
3.4.48 Matrix Divide 115
3.4.49 Matrix Inverse 117
3.4.50 Maximum 118
3.4.51 Membership 119
3.4.52 Minimum 119
3.4.53 Minus 119
3.4.54 Mix 120
3.4.55 NAND 126
3.4.56 Natural Logarithm 127

3.4.57 Negate 127
3.4.58 Nest 127
3.4.59 NOR 128
3.4.60 NOT 128
3.4.61 Not Equal To 129
3.4.62 Not Match 129
3.4.63 Greatest Common Divisor/OR 130
3.4.64 Partition 131
3.4.65 Partitioned Enclose 134
3.4.66 Pi Times 135
3.4.67 Pick 136
3.4.68 Plus 136
3.4.69 Power 137
3.4.70 Ravel 138
3.4.70.1 Ravel 138
3.4.70.2 Ravel with Axes 138
3.4.71 Reciprocal 142
3.4.72 Replicate 142
3.4.73 Replicate First 143
3.4.74 Reshape 143
3.4.75 Residue 144
3.4.76 Reverse 145
3.4.77 Reverse First 145
3.4.78 Right 145
3.4.79 Roll 146
3.4.80 Rotate 147
3.4.81 Rotate First 148
3.4.82 Same 148
3.4.83 Shape 149
3.4.84 Split 150
3.4.85 Table 150
3.4.86 Take 151
3.4.86.1 Take 151
3.4.86.2 Take with Axes 152
3.4.87 Tally 154
3.4.88 Times 154
3.4.89 Transpose 154
3.4.90 Type 156
3.4.91 Union 156

3.4.92 Unique 156
3.4.93 Unique Mask 158
3.4.94 Where 158
3.4.95 Without 159

4 Primitive Operators 161

4.1 Operator Syntax 161
4.2 Function Composition 162
4.3 Operators Summarised 165
4.4 Primitive Operators (A-Z) 167

4.4.1 At 167
4.4.2 Atop 171
4.4.3 Behind 172
4.4.4 Beside 176
4.4.5 Bind 176
4.4.6 Commute 177
4.4.7 Constant 178
4.4.8 Each 179

4.4.8.1 Each with Monadic Operand 179
4.4.8.2 Each with Dyadic Operand 179

4.4.9 I-Beam 181

4.4.9.1 I-Beam 181
4.4.9.2 8: Inverted Table Index-of 185
4.4.9.3 13: Log Use of Deprecated Features 187
4.4.9.4 43: Monadic Operator Generator 189
4.4.9.5 85: Execute Expression 190
4.4.9.6 109: Log File for Deprecations 191
4.4.9.7 120: Generate UUID 192
4.4.9.8 127: Overwrite Free Pockets 193
4.4.9.9 180: Canonical Representation 194
4.4.9.10 181: Unsqueezed Type 194
4.4.9.11 200: Syntax Colouring 194
4.4.9.12 201: Syntax Colour Tokens 195
4.4.9.13 219: Compress/Decompress Vector of Short Integers 196
4.4.9.14 220: Serialise/Deserialise Array 198
4.4.9.15 400: Compiler Control 199
4.4.9.16 600: Disable Traps 202
4.4.9.17 739: Temporary Directory 203
4.4.9.18 900: Called Monadically? 203
4.4.9.19 950: List Loaded Libraries 204

4.4.9.20 1010: Set Shell Script Debug Options 205
4.4.9.21 1111: Number of Threads 206
4.4.9.22 1112: Parallel Execution Threshold 206
4.4.9.23 1159: Update Function Time and User Stamp 206
4.4.9.24 1200: Format Date-Time 207
4.4.9.25 1302: aplcore Parameters 217
4.4.9.26 1500: Hash Array 218
4.4.9.27 2000: Memory Manager Statistics 220
4.4.9.28 2002: Specify Workspace Available 224
4.4.9.29 2007: Disable Global Triggers 225
4.4.9.30 2010: Update DataTable 226
4.4.9.31 2011: Read DataTable 229
4.4.9.32 2014: Remove Data Binding 233
4.4.9.33 2015: Create Data Binding Source 234
4.4.9.34 2016: Create .NET Delegate 245
4.4.9.35 2017: Identify .NET Type 246
4.4.9.36 2022: Flush Session Caption 246
4.4.9.37 2023: Close all Windows 247
4.4.9.38 2035: Set Dyalog Pixel Type 247
4.4.9.39 2041: Override COM Default Value 248
4.4.9.40 2100: Export To Memory 249
4.4.9.41 2101: Close .NET AppDomain 249
4.4.9.42 2250: Verify .NET Interface 249
4.4.9.43 2400: Set Workspace Save Options 250
4.4.9.44 2401: Expose Root Properties 251
4.4.9.45 2501: Discard Thread on Exit 252
4.4.9.46 2502: Discard Parked Threads 252
4.4.9.47 2503: Mark Thread as Uninterruptible 253
4.4.9.48 2520: Use Separate Thread for .NET 253
4.4.9.49 2704: Continue Autosave 254
4.4.9.50 3002: Disable Component Checksum Validation 254
4.4.9.51 3012: Enable Compression of Large Components 255
4.4.9.52 3500: Send Text to Ride-embedded Browser 255
4.4.9.53 3501: Connected to Ride? 256
4.4.9.54 3502: Manage Ride Connections 256
4.4.9.55 3535: Scan For Deprecated Files 257
4.4.9.56 4000: Fork New Task 258
4.4.9.57 4001: Change User 259
4.4.9.58 4002: Reap Forked Tasks 260
4.4.9.59 4007: Signal Counts 262
4.4.9.60 5171: Discard Source Information 263
4.4.9.61 5172: Discard Source Code 263
4.4.9.62 5176: List Loaded Files 264

4.4.9.63 5177: List Loaded File Objects ... 265
4.4.9.64 5178: Remove Loaded File Object Info ... 266
4.4.9.65 5179: Loaded File Object Info ... 267
4.4.9.66 5581: Unicode Normalisation ... 268
4.4.9.67 7162: JSON Translate Name ... 270
4.4.9.68 8373: Shell Process Control ... 271
4.4.9.69 8415: Singular Value Decomposition ... 272
4.4.9.70 8659: List Shared Code Files/Attached Names ... 273
4.4.9.71 8666: Attach/Assimilate/Detach Shared Code Files ... 274
4.4.9.72 8667: Save Shared Code Files ... 276
4.4.9.73 16808: Sample Probability Distribution ... 276
4.4.9.74 50100: Line Count ... 280
4.4.10 Inner Product ... 282
4.4.11 Key ... 283
4.4.12 Outer Product ... 287
4.4.13 Over ... 288
4.4.14 Power ... 289
4.4.15 Rank ... 292
4.4.16 Reduce ... 296
4.4.16.1 Reduce ... 296
4.4.16.2 Reduce N Wise ... 298
4.4.17 Reduce First ... 300
4.4.17.1 Reduce First ... 300
4.4.17.2 Reduce First N Wise ... 300
4.4.18 Scan ... 301
4.4.19 Scan First ... 302
4.4.20 Spawn ... 302
4.4.21 Stencil ... 303
4.4.22 Variant ... 309
5 Other Syntax ... 314
5.1 Abort ... 314
5.2 Assignment ... 315
5.2.1 Assignment ... 315
5.2.2 Assignment Modified ... 318
5.2.3 Assignment Indexed ... 319
5.2.4 Assignment Indexed Modified ... 324
5.2.5 Assignment Selective ... 325
5.2.6 Assignment Selective Modified ... 328

5.3 Axis 330
5.3.1 Axis with Monadic Operand 330
5.3.2 Axis with Dyadic Operand 331
5.4 Branch 335
5.5 Indexing 337
5.6 Zilde 342
6 System Functions and Variables 343
6.1 Introduction 343
6.2 System Functions 344
6.3 System Functions Categorised 346
6.4 System Functions (A-Z) 360
6.4.1 Character Input Output 360
6.4.2 Evaluated Input Output 362
6.4.3 Underscored Alphabetic Characters 364
6.4.4 ⎕A: Alphabetic Characters 365
6.4.5 ⎕AI: Account Information 365
6.4.6 ⎕AN: Account Name 365
6.4.7 ⎕ARBIN: Arbitrary Input 366
6.4.8 ⎕ARBOUT: Arbitrary Output 367
6.4.9 ⎕AT: Attributes 368
6.4.10 ⎕ATX: Extended Attributes 372
6.4.11 ⎕AV: Atomic Vector 375
6.4.12 ⎕AVU: Atomic Vector Unicode 376
6.4.13 ⎕BASE: Base Class 379
6.4.14 ⎕C: Case Convert 379
6.4.15 ⎕CLASS: Class 381
6.4.16 ⎕CLEAR: Clear Workspace 383
6.4.17 ⎕CMD: Execute Windows Command 384
6.4.18 ⎕CMD: Start Windows Auxiliary Processor 389
6.4.19 ⎕CR: Canonical Representation 390
6.4.20 ⎕CS: Change Space 392
6.4.21 ⎕CSV: Comma Separated Values 395
6.4.22 ⎕CT: Comparison Tolerance 409
6.4.23 ⎕CY: Copy Workspace 409
6.4.24 ⎕D: Digits 412
6.4.25 ⎕DCT: Decimal Comparison Tolerance 412
6.4.26 ⎕DF: Display Form 412
6.4.27 ⎕DIV: Division Method 415

6.4.28 ⎕DL: Delay 416
6.4.29 ⎕DM: Diagnostic Message 416
6.4.30 ⎕DMX: Extended Diagnostic Message 417
6.4.31 ⎕DQ: Dequeue Events 421
6.4.32 ⎕DR: Data Representation Dyadic 424
6.4.33 ⎕DR: Data Representation Monadic 426
6.4.34 ⎕DT: Date-time 427
6.4.35 ⎕ED: Edit Object 438
6.4.36 ⎕EM: Event Message 441
6.4.37 ⎕EN: Event Number 441
6.4.38 ⎕EXCEPTION: Exception 442
6.4.39 ⎕EX: Expunge Object 443
6.4.40 ⎕EXPORT: Export Object 445
6.4.41 ⎕FAPPEND: File Append Component 446
6.4.42 ⎕FAVAIL: File System Available 447
6.4.43 ⎕FCHK: File Check and Repair 447
6.4.44 ⎕FCOPY: File Copy 450
6.4.45 ⎕FCREATE: File Create 452
6.4.46 ⎕FDROP: File Drop Component 455
6.4.47 ⎕FERASE: File Erase 456
6.4.48 ⎕FHIST: File History 457
6.4.49 ⎕FHOLD: File Hold 459
6.4.50 ⎕FIX: Fix Script 460
6.4.51 ⎕FLIB: Component File Library 469
6.4.52 ⎕FMT: Format Monadic 470
6.4.53 ⎕FMT: Format Dyadic 470
6.4.54 ⎕FNAMES: File Names 478
6.4.55 ⎕FNUMS: File Numbers 479
6.4.56 ⎕FPROPS: File Properties 480
6.4.57 ⎕FR: Floating Point Representation 484
6.4.58 ⎕FRDAC: File Read Access 487
6.4.59 ⎕FRDCI: File Read Component Information 487
6.4.60 ⎕FREAD: File Read Component 488
6.4.61 ⎕FRENAME: File Rename 489
6.4.62 ⎕FREPLACE: File Replace Component 490
6.4.63 ⎕FRESIZE: File Resize 491
6.4.64 ⎕FSIZE: File Size 492
6.4.65 ⎕FSTAC: File Set Access 493
6.4.66 ⎕FSTIE: File Share Tie 494
6.4.67 ⎕FTIE: Exclusive File Tie 496

6.4.68 ⎕FUNTIE: File Untie 498
6.4.69 ⎕FX: Fix Definition 499
6.4.70 ⎕INSTANCES: Instances 500
6.4.71 ⎕IO: Index Origin 501
6.4.72 ⎕JSON: JSON Convert 502
6.4.73 ⎕KL: Key Label 516
6.4.74 ⎕LC: Line Count 517
6.4.75 ⎕LOAD: Load Workspace 518
6.4.76 ⎕LOCK: Lock Definition 519
6.4.77 ⎕LX: Latent Expression 520
6.4.78 ⎕MAP: Map File 520
6.4.79 ⎕MKDIR: Make Directory 522
6.4.80 ⎕ML: Migration Level 526
6.4.81 ⎕MONITOR: Set Monitor 527
6.4.82 ⎕MONITOR: Query Monitor 528
6.4.83 ⎕NA: Name Association 529
6.4.84 ⎕NAPPEND: Native File Append 562
6.4.85 ⎕NC: Name Classification 563
6.4.86 ⎕NCOPY: Native File Copy 577
6.4.87 ⎕NCREATE: Native File Create 581
6.4.88 ⎕NDELETE: Native File Delete 583
6.4.89 ⎕NERASE: Native File Erase 585
6.4.90 ⎕NEW: New Instance 585
6.4.91 ⎕NEXISTS: Native File Exists 588
6.4.92 ⎕NGET: Read Text File 589
6.4.93 ⎕NINFO: Native File Information 592
6.4.94 ⎕NL: Name List 600
6.4.95 ⎕NLOCK: Native File Lock 603
6.4.96 ⎕NMOVE: Native File Move 606
6.4.97 ⎕NNAMES: Native File Names 610
6.4.98 ⎕NNUMS: Native File Numbers 610
6.4.99 ⎕NPARTS: File Name Parts 610
6.4.100 ⎕NPUT: Write Text File 612
6.4.101 ⎕NQ: Enqueue Event 615
6.4.102 ⎕NR: Nested Representation 618
6.4.103 ⎕NREAD: Native File Read 619
6.4.104 ⎕NRENAME: Native File Rename 621
6.4.105 ⎕NREPLACE: Native File Replace 621
6.4.106 ⎕NRESIZE: Native File Resize 623
6.4.107 ⎕NS: Namespace 623

6.4.108 ⎕NSI: Namespace Indicator 628
6.4.109 ⎕NSIZE: Native File Size 629
6.4.110 ⎕NTIE: Native File Tie 629
6.4.111 ⎕NULL: Null Item 631
6.4.112 ⎕NUNTIE: Native File Untie 632
6.4.113 ⎕NXLATE: Native File Translate 632
6.4.114 ⎕OFF: Sign Off APL 633
6.4.115 ⎕OPT: Variant 634
6.4.116 ⎕OR: Object Representation 634
6.4.117 ⎕PATH: Search Path 638
6.4.118 ⎕PFKEY: Program Function Key 639
6.4.119 ⎕PP: Print Precision 641
6.4.120 ⎕PROFILE: Profile Application 641
6.4.121 ⎕PW: Print Width 648
6.4.122 ⎕R: Replace 649
6.4.123 ⎕REFS: Cross References 672
6.4.124 ⎕RL: Random Link 673
6.4.125 ⎕RSI: Space Indicator 677
6.4.126 ⎕RTL: Response Time Limit 678
6.4.127 ⎕S: Search 679
6.4.128 ⎕SAVE: Save Workspace 679
6.4.129 ⎕SD: Screen Dimensions 680
6.4.130 ⎕SE: Session Namespace 680
6.4.131 ⎕SH: Execute UNIX Command 681
6.4.132 ⎕SH: Start UNIX Auxiliary Processor 682
6.4.133 ⎕SHADOW: Shadow Name 683
6.4.134 ⎕SHELL: Execute external program 684
6.4.135 ⎕SI: State Indicator 699
6.4.136 ⎕SIGNAL: Signal Event 700
6.4.137 ⎕SIZE: Size of Object 704
6.4.138 ⎕SM: Screen Map 705
6.4.139 ⎕SR: Screen Read 708
6.4.140 ⎕SRC: Source 712
6.4.141 ⎕STACK: State Indicator Stack 713
6.4.142 ⎕STATE: State of Object 715
6.4.143 ⎕STOP: Set Stop 716
6.4.144 ⎕STOP: Query Stop 718
6.4.145 ⎕SVC: Set Access Control 718
6.4.146 ⎕SVC: Query Access Control 720
6.4.147 ⎕SVO: Shared Variable Offer 720

6.4.148 □SVO: Query Degree of Coupling 723
6.4.149 □SVQ: Shared Variable Query 723
6.4.150 □SVR: Shared Variable Retract Offer 724
6.4.151 □SVS: Shared Variable State 724
6.4.152 □TALLOC: Allocate Token Range 725
6.4.153 □TC: Terminal Control 727
6.4.154 □TCNUMS: Thread Child Numbers 728
6.4.155 □TGET: Get Tokens 728
6.4.156 □THIS: This Space 730
6.4.157 □TID: Current Thread Identity 731
6.4.158 □TKILL: Kill Thread 732
6.4.159 □TNAME: Current Thread Name 732
6.4.160 □TNUMS: Thread Numbers 732
6.4.161 □TPOOL: Token Pool 733
6.4.162 □TPUT: Put Tokens 733
6.4.163 □TRACE: Set Trace 734
6.4.164 □TRACE: Query Trace 736
6.4.165 □TRAP: Trap Event 736
6.4.166 □TREQ: Token Requests 741
6.4.167 □TS: Timestamp 742
6.4.168 □TSYNC: Wait for Threads to Terminate 742
6.4.169 □UCS: Unicode Convert 744
6.4.170 □USING: Using Microsoft Net Search Path 746
6.4.171 □VGET: Value Get 748
6.4.172 □VR: Vector Representation 754
6.4.173 □VSET: Value Set 755
6.4.174 □VFI: Verify Fix Input 758
6.4.175 □WA: Workspace Available 759
6.4.176 □WC: Windows Create Object 760
6.4.177 □WG: Windows Get Property 763
6.4.178 □WN: Windows Child Names 764
6.4.179 □WS: Windows Set Property 765
6.4.180 □WSID: Workspace Identification 766
6.4.181 □WX: Window Expose 767
6.4.182 □XML: XML Convert 768
6.4.183 □XSI: Extended State Indicator 784
6.4.184 □XT: Set External Variable 785
6.4.185 □XT: Query External Variable 788

7 System Commands 789

7.1 System Commands 789
7.2 System Commands (A-Z) 792

7.2.1 CLASSES: List Classes 792
7.2.2 CLEAR: Clear Workspace 793
7.2.3 CMD: Windows Command Processor 793
7.2.4 CONTINUE: Save Continuation 795
7.2.5 COPY: Copy Workspace 795
7.2.6 CS: Change Space 799
7.2.7 DROP: Drop Workspace 800
7.2.8 ED: Edit Object 800
7.2.9 ERASE: Erase Object 801
7.2.10 EVENTS: List Events 802
7.2.11 FNS: List Global Defined Functions 802
7.2.12 HOLDS: Display Held Tokens 803
7.2.13 LIB: List Workspace Library 804
7.2.14 LOAD: Load Workspace 804
7.2.15 METHODS: List Methods 805
7.2.16 NS: Create Namespace 806
7.2.17 OBJECTS: List Global Namespaces 806
7.2.18 OBS: List Global Namespaces Short 807
7.2.19 OFF: Sign Off APL 807
7.2.20 OPS: List Global Defined Operators 807
7.2.21 PCOPY: Protected Copy 807
7.2.22 PROPS: List Properties 808
7.2.23 RESET: Reset State Indicator 809
7.2.24 SAVE: Save Workspace 809
7.2.25 SH: Execute UNIX Command 812
7.2.26 SI: State Indicator 813
7.2.27 SIC: Clear State Indicator 815
7.2.28 SINL: State Indicator Name List 815
7.2.29 TID: Thread Identity 815
7.2.30 VARS: List Global Defined Variables 817
7.2.31 WSID: Workspace Identification 817
7.2.32 XLOAD: Load without Latent Expression 818

8 Appendices 819

8.1 PCRE Specifications 819

1 Key to Notation

The following definitions and conventions apply throughout this section:

`f`	A function, or an operator's left operand (function or array).
`g`	A function, or an operator's right operand (function or array).
`A`	An operator's left argument when an array.
`B`	An operator's right argument when an array.
`X`	The left argument of a function.
`Y`	The right argument of a function.
`R`	The explicit result of a function.
`[K]`	Axis specification.
`[I]`	Index specification.
`{X}`	The left argument of a function is optional.
`{R}←`	The function may or may not return a result, or the result may be suppressed.

function may refer to a primitive function, a system function, a defined (canonical, dfn or assigned) function or a derived (from an operator) function.

2 Glyphs

2.1 Arrows: ← → ↑ ↓

Glyph	Glyph Name	Uses
←	Left Arrow	*Assignment*
…←	Left Arrow	*Modified Assignment*
[…]←	Brackets with Left Arrow	*Indexed Assignment*
[…]…←	Brackets with Left Arrow	*Modified Indexed Assignment*
(…)←	Parentheses with Left Arrow	*Selective Assignment*
(…)…←	Parentheses with Left Arrow	*Modified Selective Assignment*
→	Right Arrow	*Abort*, *Branch*
↑	Up Arrow	*Mix*, *Take*
↓	Down Arrow	*Split*, *Drop*

2.2 Circles: ⍜ ∘

Glyph	Glyph Name	Uses
⍜	Circle	*Pi Times*, *Circular Functions*
∘	Jot	*Beside*, *Bind*
∘.	Jot Dot	*Outer Product*
⍤	Jot Diaeresis	*Atop*, *Rank*
⍟	Hydrant	*Execute*
⍣	Thorn	*Format*, *Format by Specification*
⍲	Lamp	*Programming: Statements*
⍜	Jot Underbar	*Behind*
⍥	Circle Diaeresis	*Over*
⊖	Circle Bar	*Reverse First*, *Rotate First*
⍉	Circle Backslash	*Transpose*, *Dyadic Transpose*
⌽	Circle Stile	*Reverse*, *Rotate*
⍟	Circle Star	*Natural Logarithm*, *Logarithm*

2.3 Diaereses: ¨

Glyph	Glyph Name	Uses
¨	Diaeresis	*Each with Dyadic Operand*, *Each with Monadic Operand*
⍨	Tilde Diaeresis	*Commute*, *Constant*
⍤	Jot Diaeresis	*Atop*, *Rank*
⍣	Star Diaeresis	*Power*
⍥	Circle Diaeresis	*Over*

2.4 Dots and Commas: . ,

Glyph	Glyph Name	Uses
.	Dot	*Inner Product*, *Programming: Namespaces And Localisation*
,	Comma	*Ravel*, *Catenate/Laminate*
∘.	Jot Dot	*Outer Product*
:	Colon	*Programming: Statements*, *Programming: Control Structures Introduction*, *Programming: Guards*, *Programming: Array Notation*
: :	Colon Colon	*Programming: Error Guards*
÷	Divide	*Reciprocal*, *Divide*
⍠	Variant	*Variant*
⍈	Domino	*Matrix Inverse*, *Matrix Divide*
!	Exclamation Mark	*Factorial*, *Binomial*
?	Question Mark	*Roll*, *Deal*
⍪	Comma Bar	*Table*, *Catenate First/Laminate*
;	Semicolon	*index separator*, *Programming: Global Local Names*, *Programming: Locals Lines*

2.5 Enclosures: `[] () {} ''`

Glyph	Glyph Name	Uses
`[…]`	Brackets	*Indexing*, *Programming: Array Notation*
`…[…]`	Brackets	*Axis with Dyadic Operand*, *Axis with Monadic Operand*
`[…]←`	Brackets with Left Arrow	*Indexed Assignment*
`[…]…←`	Brackets with Left Arrow	*Modified Indexed Assignment*
`]`	Right Bracket	*User Interface: User Commands*
`]…`	Right Bracket	*User Interface: User Commands*
`()`	Parentheses	*Programming: Array Notation*
`(…)`	Parentheses	*Programming: Expressions*, *Programming: Namelists*, *Programming: Array Notation*
`(…)←`	Parentheses with Left Arrow	*Selective Assignment*
`(…)…←`	Parentheses with Left Arrow	*Modified Selective Assignment*
`)…`	Right parenthesis	*start a system command*
`{}`	Braces	*Programming: Idiom List*
`{…}`	Braces	*Programming: Model Syntax*, *Programming: Dynamic Functions And Operators*
`''`	Quote Quote	*Programming: Characters*
`'…'`	Quotes	*Programming: Characters*

2.6 Horseshoes: ⊂ ⊃ ∩ ∪

Glyph	Glyph Name	Uses
⊂	Left Shoe	*Enclose*, *Partitioned Enclose*
⊃	Right Shoe	*First*, *Pick*
∩	Up Shoe	*Intersection*
∪	Down Shoe	*Unique*, *Union*
⊆	Left Shoe Underbar	*Nest*, *Partition*

2.7 Letterforms

Glyph	Glyph Name	Uses
@	At	*At*
&	Ampersand	*Spawn*
⍺	Alpha	*Programming: Dynamic Functions And Operators*
⍺⍺	Alpha Alpha	*Programming: Dynamic Operators*
⍵	Omega	*Programming: Dynamic Functions And Operators*
⍵⍵	Omega Omega	*Programming: Dynamic Operators*
∆	Delta	*Programming: Names*
⍙	Delta Underbar	*Programming: Names*
∈	Epsilon	*Enlist*, *Membership*
⍷	Epsilon Underbar	*Find*
⍳	Iota	*Index Generator*, *Index Of*
⍸	Iota Underbar	*Where*, *Interval Index*
⍴	Rho	*Shape*, *Reshape*

2.8 Lines – Diagonal: / \

Glyph	Glyph Name	Uses
/	Slash	*Replicate*, *Reduce*, *Reduce N Wise*
\	Backslash	*Expand*, *Scan*
⫽	Slash Bar	*Replicate First*, *Reduce First*, *Reduce First N Wise*
⫻	Backslash Bar	*Expand First*, *Scan First*
≠	Equal Underbar Slash	*Tally*, *Not Match*
≠	Not Equal	*Unique Mask*, *Not Equal To*
#	Hash	*Programming: Namespaces*
##	Hash Hash	*Programming: Namespaces*
×	Times	*Direction*, *Times*
⍉	Circle Backslash	*Transpose*, *Dyadic Transpose*

2.9 Lines – Horizontal: ‾ - _

Glyph	Glyph Name	Uses
¯…	High Minus	*Programming: Numbers*
–	Minus	*Negate*, *Minus*
_	Underbar	*Programming: Names*
=	Equal	*Equal To*
≡	Equal Underbar	*Depth*, *Match*
≠	Not Equal	*Unique Mask*, *Not Equal To*
≠	Equal Underbar Slash	*Tally*, *Not Match*
⌸	Quad Equal	*Key*
#	Hash	*Programming: Namespaces*
##	Hash Hash	*Programming: Namespaces*
+	Plus	*Conjugate*, *Plus*
⌈	Upstile	*Ceiling*, *Maximum*
⊖	Circle Bar	*Reverse First*, *Rotate First*
⍪	Comma Bar	*Table*, *Catenate First/Laminate*
⌿	Slash Bar	*Replicate First*, *Reduce First*, *Reduce First N Wise*
⍀	Backslash Bar	*Expand First*, *Scan First*
⊢	Right Tack	*Same*, *Right*
⊣	Left Tack	*Same*, *Left*
⍟	Hydrant	*Execute*
⍝	Thorn	*Format*, *Format by Specification*
⊤	Down Tack	*Encode*
⌶	I-Beam	*I-Beam*
⊥	Up Tack	*Decode*
⍜	Jot Underbar	*Behind*
≤	Less Than Or Equal To	*Less Than Or Equal To*
≥	Greater Than Or Equal To	*Greater Than Or Equal To*

Glyph	Glyph Name	Uses
⍸	Left Shoe Underbar	*Nest*, *Partition*
⍳	Iota Underbar	*Where*, *Interval Index*
⍷	Epsilon Underbar	*Find*
⍙	Delta Underbar	*Programming: Names*
⌊	Downstile	*Floor*, *Minimum*

2.10 Lines – Vertical: ∣

Glyph	Glyph Name	Uses
∣	Stile	*Magnitude*, *Residue*
+	Plus	*Conjugate*, *Plus*
⌈	Upstile	*Ceiling*, *Maximum*
⌊	Downstile	*Floor*, *Minimum*
⊤	Down Tack	*Encode*
⌶	I-Beam	*I-Beam*
⊥	Up Tack	*Decode*
⌽	Circle Stile	*Reverse*, *Rotate*
⍋	Grade Up	*Grade Up*, *Dyadic Grade Up*
⍒	Grade Down	*Grade Down*, *Dyadic Grade Down*
⍎	Hydrant	*Execute*
⍕	Thorn	*Format*, *Format by Specification*

2.11 Rectangles: ⍯

Glyph	Glyph Name	Uses
⍯	Squad	*Materialise*, *Index*
⎕	Quad	*evaluated input/output*
⎕...	Quad	*system function*
⍞	Quote Quad	*character input/output*
⌸	Variant	*Variant*
⍇	Domino	*Matrix Inverse*, *Matrix Divide*
⍈	Quad Equal	*Key*
⌺	Quad Diamond	*Stencil*

2.12 Stars: ⋆

Glyph	Glyph Name	Uses
⋆	Star	*Exponential*, *Power*
⍣	Star Diaeresis	*Power*
⍟	Circle Star	*Natural Logarithm*, *Logarithm*

2.13 Tildes: ~

Glyph	Glyph Name	Uses
~	Tilde	*NOT*, *Without*
⍨	Tilde Diaeresis	*Commute*, *Constant*
⍲	Logical NAND	*NAND*
⍱	Logical NOR	*NOR*
⍬	Zilde	*empty numeric vector*

2.14 Triangles: ∆ ∇

Glyph	Glyph Name	Uses
∆	Delta	*Programming: Names*
∇	Del	*Programming: Recursion*, *Programming: Methods*, *Programming: Apl Line Editor*
∇∇	Del Del	*Programming: Recursion*
⍋	Grade Up	*Grade Up*, *Dyadic Grade Up*
⍒	Grade Down	*Grade Down*, *Dyadic Grade Down*
⍙	Delta Underbar	*Programming: Names*

2.15 Wedges: < > ∧ ∨

Glyph	Glyph Name	Uses
<	Less Than	*Less Than*
>	Greater Than	*Greater Than*
∧	Logical AND	*Lowest Common Multiple/AND*
∨	Logical OR	*Greatest Common Divisor/OR*
≤	Less Than Or Equal To	*Less Than Or Equal To*
≥	Greater Than Or Equal To	*Greater Than Or Equal To*
⍲	Logical NAND	*NAND*
⍱	Logical NOR	*NOR*
⋄	Diamond	*Programming: Statements*, *Programming: Array Notation*
⌺	Quad Diamond	*Stencil*

3 Primitive Functions

3.1 Primitive Functions

Glyph	Glyph Name	Monadic Function	Dyadic Function
+	Plus	*Conjugate*	*Plus*
–	Minus	*Negate*	*Minus*
×	Times	*Direction*	*Times*
÷	Divide	*Reciprocal*	*Divide*
|	Stile	*Magnitude*	*Residue*
*	Star	*Exponential*	*Power*
⍟	Log	*Natural Logarithm*	*Logarithm*
○	Circle	*Pi Times*	*Circular Functions*
⌈	Upstile	*Ceiling*	*Maximum*
⌊	Downstile	*Floor*	*Minimum*
!	Exclamation Mark	*Factorial*	*Binomial*
?	Question Mark	*Roll*	*Deal*
^	Logical AND		*Lowest Common Multiple/ AND*
∨	Logical OR		*Greatest Common Divisor/ OR*
⍲	Logical NAND		*NAND*
⍱	Logical NOR		*NOR*
<	Less Than		*Less Than*
≤	Less Than Or Equal To		*Less Than Or Equal To*
=	Equal		*Equal To*
≥	Greater Than Or Equal To		*Greater Than Or Equal To*
>	Greater Than		*Greater Than*
≠	Not Equal	*Unique Mask*	*Not Equal To*
~	Tilde	*NOT*	*Without*
∩	Up Shoe		*Intersection*

Glyph	Glyph Name	Monadic Function	Dyadic Function
∪	Down Shoe	*Unique*	*Union*
≡	Equal Underbar	*Depth*	*Match*
≢	Equal Underbar Slash	*Tally*	*Not Match*
∈	Epsilon	*Enlist*	*Membership*
⍷	Epsilon Underbar		*Find*
⍳	Iota	*Index Generator*	*Index Of*
⍸	Iota Underbar	*Where*	*Interval Index*
⍋	Grade Up	*Grade Up*	*Dyadic Grade Up*
⍒	Grade Down	*Grade Down*	*Dyadic Grade Down*
⌷	Squad	*Materialise*	*Index*
⌽	Circle Stile	*Reverse*	*Rotate*
⍉	Circle Backslash	*Transpose*	*Dyadic Transpose*
⊖	Circle Bar	*Reverse First*	*Rotate First*
⍴	Rho	*Shape*	*Reshape*
,	Comma	*Ravel*	*Catenate/Laminate*
⍪	Comma Bar	*Table*	*Catenate First/Laminate*
↑	Up Arrow	*Mix*	*Take*
↓	Down Arrow	*Split*	*Drop*
⊂	Left Shoe	*Enclose*	*Partitioned Enclose*
⊆	Left Shoe Underbar	*Nest*	*Partition*
⊃	Right Shoe	*First*	*Pick*
⊤	Down Tack		*Encode*
⊥	Up Tack		*Decode*
⌹	Domino	*Matrix Inverse*	*Matrix Divide*
⊢	Right Tack	*Same*	*Right*
⊣	Left Tack	*Same*	*Left*

Glyph	Glyph Name	Monadic Function	Dyadic Function
⍎	Hydrant	*Execute*	*Dyadic Execute*
⍕	Thorn	*Format*	*Format by Specification*
⌿	Slash		*Replicate*
⍀	Slash Bar		*Replicate First*
\	Backslash		*Expand*
⍁	Backslash Bar		*Expand First*

3.2 Notes

Conformability

The arguments of a dyadic function are said to be CONFORMABLE if the shape of each argument meets the requirements of the function, possibly after scalar extension.

Fill Elements

Some primitive functions may include fill elements in their result. The fill element for an array is the enclosed type of the disclose of the array (`⊂∊⊃Y` for array `Y` with `⎕ML←0`). The Type function (`∊` with `⎕ML←0`) replaces a numeric value with zero and a character value with `' '`.

The Disclose function (`⊃`) returns the first item of an array. If the array is empty, `⊃Y` is the PROTOTYPE of `Y`. The prototype is the type of the first element of the original array.

Primitive functions which may return an array including fill elements are Expand (`\` or `⍁`), Replicate (`⌿` or `⍀`), Reshape (`⍴`) and Take (`↑`).

Examples

```
      ML←0
      ∈⍳5
0 0 0 0 0

      ∈⊃(⍳3)('ABC')
0 0 0

      ⊂∈⊃(⍳3)('ABC')
 0 0 0

      ⊂∈⊃⊂(⍳3)('ABC')
  0 0 0

      A←'ABC' (1 2 3)
      A←0⍴A
      ⊂∈⊃A

      ' '=⊂∈⊃A
 1 1 1
```

Axis Operator

The axis operator may be applied to all scalar dyadic primitive functions and certain mixed primitive functions. An integer axis identifies a specific axis along which the function is to be applied to one or both of its arguments. If the primitive function is to be applied without an axis specification, a default axis is implied, either the first or last.

Example

```
      1 0 1/[1] 3 2⍴⍳6
1 2
5 6
```

```
      1 2 3+[2]2 3⍴10 20 30
11 22 33
11 22 33
```

Sometimes the axis value is fractional, indicating that a new axis or axes are to be created between the axes identified by the lower and upper integer bounds of the value (either of which might not exist).

EXAMPLE

```
      'NAMES',[0.5]'='
NAMES
=====
```

⎕IO is an implicit argument of an axis specification.

3.3 Primitive Functions (by Category)

The primitive functions can be grouped together into categories that exhibit common behaviours or goals. Functions can be in multiple categories.

Scalar Functions

Scalar functions apply to simple scalars and are pervasive.

Monadic Scalar Functions

Monadic scalar functions apply independently to each simple scalar in their argument.

The result has the same structure as the argument, and is numeric (exception: `+Y` preserves the types of the argument).

Syntax	Meaning
+Y	*Conjugate*
−Y	*Negate*
×Y	*Direction*
÷Y	*Reciprocal*
|Y	*Magnitude*
⋆Y	*Exponential*
⍟Y	*Natural Logarithm*
○Y	*Pi Times*
⌈Y	*Ceiling*
⌊Y	*Floor*
!Y	*Factorial*
?Y	*Roll*
~Y	*NOT*

Dyadic Scalar Functions

Dyadic scalar functions apply independently to corresponding pairs of simple scalars in their arguments. The result has the same structure as the arguments, subject to *scalar extensions*, and is always numeric.

If a simple scalar corresponds to a non-scalar, the simple scalar is replicated to the shape of the non-scalar.

A singleton (single-element array) is treated as a scalar for scalar extension purposes; if both arguments are singletons, then the lower-rank argument is extended to the rank of the higher-rank argument.

Syntax	Meaning
X+Y	*Plus*
X−Y	*Minus*
X×Y	*Times*
X÷Y	*Divide*
X|Y	*Residue*
X*Y	*Power*
X⍟Y	*Logarithm*
X○Y	*Circular Functions*
X⌈Y	*Maximum*
X⌊Y	*Minimum*
X!Y	*Binomial*
X∧Y	*Lowest Common Multiple*
X∨Y	*Greatest Common Divisor*
X~Y	*Without*
X∧Y	*AND*
X∨Y	*OR*
X⍲Y	*NAND*
X⍱Y	*NOR*
X<Y	*Less Than*
X≤Y	*Less Than Or Equal To*
X=Y	*Equal To*
X≥Y	*Greater Than Or Equal To*
X>Y	*Greater Than*
X≠Y	*Not Equal To*

Mathematical Functions

Mathematical functions perform computations on numeric arguments; all except `+Y`, `⍟Y`, and `⍕Y` will reject non-numeric arguments.

Several of the monadic forms are equivalent to the dyadic forms with a default left argument.

Syntax	Meaning
+Y	*Conjugate*
X+Y	*Plus*
−Y	*Negate*
X−Y	*Minus*
×Y	*Direction*
X×Y	*Times*
÷Y	*Reciprocal*
X÷Y	*Divide*
|Y	*Magnitude*
X|Y	*Residue*
⋆Y	*Exponential*
X⋆Y	*Power*
⍟Y	*Natural Logarithm*
X⍟Y	*Logarithm*
○Y	*Pi Times*
X○Y	*Circular Functions*
⌈Y	*Ceiling*
X⌈Y	*Maximum*
⌊Y	*Floor*
X⌊Y	*Minimum*
!Y	*Factorial*
X!Y	*Binomial*
?Y	*Roll*
X?Y	*Deal*
X⍑Y	*Encode*
X⊥Y	*Decode*
⌹Y	*Matrix Inverse*

Syntax	Meaning
X⌹Y	*Matrix Divide*
X∧Y	*Lowest Common Multiple*
X∨Y	*Greatest Common Divisor*
⍋Y	*Grade Up*
⍒Y	*Grade Down*
X⍕Y	*Format by Specification*

Logic Functions

Logic functions represent logic gates. Their arguments and results are Boolean (comprising only `1`s and `0`s).

Syntax	Meaning	Logic Gate
~Y	*NOT*	NOT
X∧Y	*AND*	AND
X∨Y	*OR*	OR
X⍲Y	*NAND*	NAND
X⍱Y	*NOR*	NOR
X<Y	*Less Than*	CNIMPLY
X≤Y	*Less Than Or Equal To*	IMPLY
X=Y	*Equal To*	XNOR
X≥Y	*Greater Than Or Equal To*	CIMPLY
X>Y	*Greater Than*	NIMPLY
X≠Y	*Not Equal To*	XOR

Comparison Functions

Comparison functions perform comparisons. They return Boolean results indicating whether the comparisons are true (`1`) or false (`0`).

Syntax	Meaning
`X<Y`	*Less Than*
`X≤Y`	*Less Than Or Equal To*
`X=Y`	*Equal To*
`X≥Y`	*Greater Than Or Equal To*
`X>Y`	*Greater Than*
`X≠Y`	*Not Equal To*
`X≡Y`	*Match*
`X≢Y`	*Not Match*
`X∊Y`	*Membership*

Selection Functions

Selection functions restrict the parts of their arguments that are propagated into their result (exception: `X⊢Y` and `X⊣Y` return one of their arguments).

Syntax	Meaning
`X⌷Y`	*Index*
`⊃Y`	*First*
`X⊃Y`	*Pick*
`X/Y`	*Replicate*
`X⌿Y`	*Replicate First*
`X\Y`	*Expand*
`X⍀Y`	*Expand First*
`X⍉Y`	*Dyadic Transpose*
`X↑Y`	*Take*
`X↓Y`	*Drop*
`X∩Y`	*Intersection*
`∪Y`	*Unique*
`X∪Y`	*Union*
`X⊢Y`	*Right*
`X⊣Y`	*Left*

Set Functions

Set functions operate on arrays representing collections. Within the set functions:

- `X∩Y`, `X∪Y`, and `X~Y` are restricted to vectors.
- `X~Y`, `X∩Y`, `∪Y`, and `X∪Y` return new collections.
- `≠Y`, `X≡Y`, `X≢Y`, and `X∈Y` return Boolean results (comprising only `1`s and `0`s).

Syntax	Meaning
X~Y	*Without*
X∩Y	*Intersection*
∪Y	*Unique*
X∪Y	*Union*
≠Y	*Unique Mask*
X≡Y	*Match*
X≢Y	*Not Match*
X∈Y	*Membership*

Search Functions

Search functions determine whether/where values are located.

Syntax	Meaning
X⍳Y	*Index Of*
⍸Y	*Where*
X⍸Y	*Interval Index*
X∈Y	*Membership*
X⍷Y	*Find*

Ordering Functions

Ordering functions return indices that can subsequently be used to select from within the original array.

Syntax	Meaning
⍳Y	*Index Generator*
X⍳Y	*Index Of*
X⍸Y	*Interval Index*
⍋Y	*Grade Up*
X⍋Y	*Dyadic Grade Up*
⍒Y	*Grade Down*
X⍒Y	*Dyadic Grade Down*

Index Generator Functions

Index generator functions return indices that can then be used directly or to select from within the original array.

Syntax	Meaning
?Y	*Roll*
X?Y	*Deal*
⍳Y	*Index Generator*
X⍳Y	*Index Of*
⍸Y	*Where*
X⍸Y	*Interval Index*
⍋Y	*Grade Up*
X⍋Y	*Dyadic Grade Up*
⍒Y	*Grade Down*
X⍒Y	*Dyadic Grade Down*

Structural Functions

Structural functions modify the structure or shape of their right argument (including omitting values) but they do not otherwise change any scalar values.

Syntax	Meaning
X⍴Y	*Reshape*
,Y	*Ravel*
X,Y	*Catenate/Laminate*
⍪Y	*Table*
X⍪Y	*Catenate First/Laminate*
⌽Y	*Reverse*
X⌽Y	*Rotate*
⍉Y	*Transpose*
X⍉Y	*Dyadic Transpose*
⊖Y	*Reverse First*
X⊖Y	*Rotate First*
↑Y	*Mix*
X↑Y	*Take*
↓Y	*Split*
X↓Y	*Drop*
⊂Y	*Enclose*
X⊂Y	*Partitioned Enclose*
⊆Y	*Nest*
X⊆Y	*Partition*
⊃Y	*First*
X⊃Y	*Pick*
∈Y	*Enlist*

Data Conversion Functions

Data conversion functions change between different representations of the same data.

Syntax	Meaning
⍞Y	*Materialise*
⍎Y	*Execute*
X⍎Y	*Dyadic Execute*
⍕Y	*Format*
X⍕Y	*Format by Specification*

Identity Functions

Identity functions make it easy to write ambivalent dfns and are useful in tacit programming, as generic operands, and to break up stranding. They do not modify structure, shape, or values (exception: ⍞Y, when applied to an object, extracts its default value).

Syntax	Meaning
⍞Y	*Materialise*
⊢Y	*Same*
X⊢Y	*Right*
⊣Y	*Same*
X⊣Y	*Left*

Array Property Functions

Array property functions provide information about a given array.

Syntax	Meaning
⍞Y	*Materialise*
⍴Y	*Shape*
≡Y	*Depth*
≢Y	*Tally*

3.4 Functions (A-Z)

3.4.1 Binomial R←X!Y

X and Y may be any numbers except that if Y is a negative integer then X must be an integer. R is numeric. An element of R is integer if corresponding elements of X and Y are integers.

Binomial is defined in terms of the function Factorial:

```
      X!Y ↔ (!Y)÷(!X)×!Y-X
```

Results are derived smoothly from the Beta function:

```
      Beta(X,Y) ↔ ÷Y×(X-1)!X+Y-1
```

For positive integer arguments, R is the number of selections of X things from Y things.

EXAMPLE

```
      1 1.2 1.4 1.6 1.8 2!5
5 6.105689248 7.219424686 8.281104786 9.227916704 10

      2!3j2
1J5
```

3.4.2 Catenate/Laminate R←X,[K]Y

Y may be any array. X may be any array. The axis specification is optional. If specified, K must be a numeric scalar or 1-element vector which may have a fractional value. If not specified, the last axis is implied.

The form R←X⍪Y may be used to imply catenation along the first axis.

Two cases of the function catenate are permitted:

1. With an integer axis specification, or implied axis specification.
2. With a fractional axis specification, also called **laminate**.

Catenation with Integer or Implied Axis Specification

The arrays X and Y are joined along the required axis to form array R. A scalar is extended to the shape of the other argument except that the required axis is restricted to a unit dimension. X and Y must have the same shape (after extension) except along the required axis, or one of the arguments may have rank one less than the other, provided that their shapes conform to the prior rule after augmenting the array of lower rank to have a unit dimension along the required axis. The rank of R is the greater of the ranks of the arguments, but not less than 1.

EXAMPLES

```
      'FUR','LONG'
FURLONG

      1,2
1 2

      (2 4⍴'THISWEEK')⍪'='
THIS
WEEK
====

      S,[1]+⌿S←2 3⍴⍳6
1 2 3
4 5 6
5 7 9
```

If, after extension, exactly one of X and Y have a length of zero along the joined axis, then the data type of R will be that of the argument with a non-zero length. Otherwise, the data type of R will be that of X.

Lamination with Fractional Axis Specification

The arrays X and Y are joined along a new axis created before the ⌈Kth axis. The new axis has a length of 2. K must exceed ⎕IO (the index origin) minus 1, and K must be less than ⎕IO plus the greater of the ranks of X and Y. A scalar argument is extended to the shape of the other argument. Otherwise X and Y must have the same shape.

The rank of R is one plus the greater of the ranks of X and Y.

EXAMPLES

```
      'HEADING',[0.5]'-'
HEADING
-------

      'NIGHT',[1.5]'*'
N*
I*
G*
H*
T*
      ⎕IO←0
      'HEADING',[¯0.5]'-'
HEADING
-------
```

3.4.3 Catenate First R←X⍪[K]Y

The form R←X⍪Y implies catenation along the first axis whereas the form R←X,Y implies catenation along the last axis (columns). See *Section 3.4.2*.

3.4.4 Ceiling R←⌈Y

Ceiling is defined in terms of Floor as ⌈Y←→-⌊-Y

Y must be numeric.

If an element of Y is real, the corresponding element of R is the least integer greater than or equal to the value of Y.

If an element of Y is complex, the corresponding element of R depends on the relationship between the real and imaginary parts of the numbers in Y.

EXAMPLES

```
      ⌈¯2.3  0.1  100  3.3
¯2 1 100 4

      ⌈1.2j2.5 1.2j¯2.5
1J3 1J¯2
```

For further explanation, see *Section 3.4.28*.

⎕CT is an implied argument of Ceiling.

3.4.5 Circular Functions R←X○Y

Y must be numeric. X must be an integer in the range ¯12 ≤ X ≤ 12. R is numeric.

X determines which of a family of trigonometric, hyperbolic, Pythagorean, and complex functions to apply to Y, from the following table. Note that when Y is complex, a and b are used to represent its real and imaginary parts, while θ represents its phase.

(-X)○Y	X	X○Y
(1-Y*2)*.5	0	(1-Y*2)*.5
Arcsin Y	1	Sine Y
Arccos Y	2	Cosine Y
Arctan Y	3	Tangent Y
Y=¯1:0Y≠¯1:(Y+1)×((Y-1)÷Y+1)*0.5	4	(1+Y*2)*.5
Arcsinh Y	5	Sinh Y
Arccosh Y	6	Cosh Y
Arctanh Y	7	Tanh Y
-8○Y	8	(-1+Y*2)*0.5
Y	9	a
+Y	10	|Y
Y×0J1	11	b
*Y×0J1	12	θ

EXAMPLES

```
      0 ¯1 ○ 1
0 1.570796327

      1○(PI←○1)÷2 3 4
1 0.8660254038  0.7071067812

      2○PI÷3
0.5
```

```
      9 11○3.5J¯1.2
3.5 ¯1.2

      9 11○.○3.5J¯1.2 2J3 3J4
 3.5 2 3
¯1.2 3 4

      ¯4○¯1
0
```

3.4.6 Conjugate R←+Y

If Y is complex, R is Y with the imaginary part of all elements negated.

If Y is real or non-numeric, R is the same array unchanged, although ⊢ is faster. See *Section 3.4.82*.

EXAMPLES

```
      +3j4
3J¯4
      +1j2 2j3 3j4
1J¯2 2J¯3 3J¯4

      3j4++3j4
6
      3j4×+3j4
25

      +A←⍳5
1 2 3 4 5

      +⎕EX'A'
1
```

3.4.7 Deal R←X?Y

Y must be a simple scalar or 1-element vector containing a non-negative integer. X must be a simple scalar or 1-element vector containing a non-negative integer and X≤Y.

R is an integer vector obtained by making X random selections from ⍳Y without repetition.

Examples

```
      13?52
7 40 24 28 12 3 36 49 20 44 2 35 1

      13?52
20 4 22 36 31 49 45 28 5 35 37 48 40
```

⎕IO and ⎕RL are implicit arguments of Deal. A side effect of Deal is to change the value of ⎕RL. See *Random Number Seed*.

3.4.8 Decode R←X⊥Y

Y must be a simple numeric array. X must be a simple numeric array. R is the numeric array which results from the evaluation of Y in the number system with radix X.

X and Y are conformable if the length of the last axis of X is the same as the length of the first axis of Y. A scalar or 1-element vector is extended to a vector of the required length. If the last axis of X or the first axis of Y has a length of 1, the array is extended along that axis to conform with the other argument.

The shape of R is the catenation of the shape of X less the last dimension with the shape of Y less the first dimension. That is:

```
      ⍴R ↔ (¯1↓⍴X),1↓⍴Y
```

For vector arguments, each element of X defines the ratio between the units for corresponding pairs of elements in Y. The first element of X has no effect on the result.

This function is also known as Base Value.

Examples

```
      60 60⊥3 13
193

      0 60⊥3 13
193

      60⊥3 13
193

      2⊥1 0 1 0
10
```

Polynomial Evaluation

If X is a scalar and Y a vector of length n, decode evaluates the polynomial (Index origin 1):

Y [1] X n - 1 + Y [2] X n - 2 + ... + Y [n] X 0

EXAMPLES

```
      2⊥1 2 3 4
26
      3⊥1 2 3 4
58
      1j1⊥1 2 3 4
5J9
```

For higher-rank array arguments, each of the vectors along the last axis of X is taken as the radix vector for each of the vectors along the first axis of Y.

EXAMPLES

```
      M
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1

      A
1 1 1
2 2 2
3 3 3
4 4 4

      A⊥M
0 1 1 2  1  2  2  3
0 1 2 3  4  5  6  7
0 1 3 4  9 10 12 13
0 1 4 5 16 17 20 21
```

Scalar extension may be applied:

```
      2⊥M
0 1 2 3 4 5 6 7
```

Extension along a unit axis may be applied:

```
      +A←2 1⍴2 10
 2
10
      A⊥M
0 1  2  3   4   5   6   7
0 1 10 11 100 101 110 111
```

3.4.9 Depth (⎕ML) R←≡Y

Y may be any array. R is the maximum number of levels of nesting of Y. A simple scalar (rank-0 number, character or namespace-reference) has a depth of 0.

A higher rank array, all of whose items are simple scalars, is termed a *simple array* and has a depth of 1. An array whose items are not all simple scalars is *nested* and has a depth 1 greater than that of its most deeply nested item.

Y is of *uniform depth* if it is simple or if all of its items have the same uniform depth.

If ⎕ML<2 and Y is not of uniform depth then R is negated (therefore, when ⎕ML<2, a negative value of R indicates non-uniform depth).

Examples

```
      ≡1
0
      ≡'A'
0
      ≡'ABC'
1
      ≡1 'A'
1
```

```
      ⎕ML←0

      ≡A←(1 2)(3 (4 5)) ⍝ Non-uniform array
¯3
      ≡¨A           ⍝ A[1] is uniform, A[2] is non-uniform
1 ¯2
      ≡⍤⍤A
 0 0  0 1
```

```
      ⎕ML←2

      ≡A
3
      ≡¨A
1 2
      ≡⍤4⊢A
 0 0  0 1
```

3.4.10 Direction (Signum) R←×Y

Y may be any numeric array.

Where an element of Y is real, the corresponding element of R is an integer whose value indicates whether the value is negative (¯1), zero (0) or positive (1).

Where an element of Y is complex, the corresponding element of R is a number with the same phase but with magnitude (absolute value) 1. It is equivalent to Y÷|Y.

Examples

```
      ×¯15.3 0 101
¯1 0 1

      ×3j4 4j5
0.6J0.8 0.624695047 6J0.7808688094

      {⍵÷|⍵}3j4 4j5
0.6J0.8 0.624695047 6J0.7808688094

      |×3j4 4j5
1 1
```

3.4.11 Divide R←X÷Y

Y must be a numeric array. X must be a numeric array. R is the numeric array resulting from X divided by Y. System variable ⎕DIV is an implicit argument of Divide.

If ⎕DIV=0 and Y=0 then if X=0, the result of X÷Y is 1; if X≠0 then X÷Y is a DOMAIN ERROR.

If ⎕DIV=1 and Y=0, the result of X÷Y is 0 for all values of X.

EXAMPLES

```
      2 0 5÷4 0 2
0.5 1 2.5

      3j1 2.5 4j5÷2 1j1 .2
1.5J0.5 1.25J¯1.25 20J25

      ⎕DIV←1
      2 0 5÷4 0 0
0.5 0 0
```

3.4.12 Drop

Drop R←X↓Y

Y may be any array. X must be a simple scalar or vector of integers. If X is a scalar, it is treated as a one-element vector. If Y is a scalar, it is treated as an array whose shape is (⍴X)⍴1. After any scalar extensions, the shape of X must be less than or equal to the rank of Y. Any missing trailing items in X default to 0.

R is an array with the same rank as Y but with elements removed from the vectors along each of the axes of Y. For the Ith axis:

- if X[I] is positive, all but the first X[I] elements of the vectors result
- if X[I] is negative, all but the last X[I] elements of the vectors result

If the magnitude of X[I] exceeds the length of the Ith axis, the result is an empty array with zero length along that axis.

Examples

```
      4↓'OVERBOARD'
BOARD

      ¯5↓'OVERBOARD'
OVER

      ⍴10↓'OVERBOARD'
0

      M
ONE
FAT
FLY
      0 ¯2↓M
O
F
F

      ¯2 ¯1↓M
ON
      1↓M
FAT
FLY
      M3←2 3 4⍴⎕A

      1 1↓M3
QRST
UVWX
      ¯1 ¯1↓M3
ABCD
EFGH
```

Drop with Axes R←X↓[K]Y

Y may be any non-scalar array. X must be a simple integer scalar or vector. K is a vector of zero or more axes of Y.

R is an array of the elements of Y with the first or last X[i] elements removed. Elements are removed from the beginning or end of Y according to the sign of X[i].

The rank of R is the same as the rank of Y:

```
      ⍴⍴R ↔ ⍴⍴Y
```

The size of each axis of `R` is determined by the corresponding element of `X`:

```
     (⍴R)[,K] ↔ 0⌈(⍴Y)[,K]-|,X
```

Examples

```
     □←M←2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

     1↓[2]M
 5  6  7  8
 9 10 11 12

17 18 19 20
21 22 23 24

     2↓[3]M
 3  4
 7  8
11 12

15 16
19 20
23 24

     2 1↓[3 2]M
 7  8
11 12

19 20
23 24
```

3.4.13 Dyadic Execute R←X⍎Y

> **Warning**
>
> If the right argument to *dyadic execute* could include user input, then there is a risk to data and systems. To reduce this risk, a system function might be more appropriate than the *dyadic execute* function. For example:
>
> - use *Section 6.4.171* or [⎕VSET](../system-functions/vset.md to get or set the value of one or more variables named within one or more character vectors.
> - use *Section 6.4.174*, *Section 6.4.72*, or *Section 6.4.21* to make numbers in text form into actual numbers.
> - use *Section 6.4.116* to call a function by name; exact usage depends on valency. For example, in the namespace with the reference `nsRef`:
> - niladic – `(nsRef.⎕OR fnName)nsRef.{⍺⍺}⍬`
> - monadic – `(nsRef.⎕OR fnName)nsRef.{⍺⍺ ⍵}Y`
> - dyadic – `X((nsRef.⎕OR fnName)nsRef.{⍺ ⍺⍺ ⍵})Y` A namespace name can be resolved to a reference with `nsRef←⎕VGET nsName`.

`Y` must be a simple character scalar or vector containing an APL expression to be executed. The expression may contain one or more sub-expressions separated by ⋄ (Diamond) characters.

`X` must be a namespace reference or a simple character scalar or vector representing the name of a namespace in which the expression is to be executed. If `X` is an empty character vector, the expression is executed in the current space.

If the result of the expression is used or is assigned to a name, `R` is the result (if any) of the last-executed sub-expression and the non-shy results of all preceding expressions (that are not assigned within the expression) are displayed. Otherwise the unassigned non-shy results of all of the sub-expressions are displayed.

If the expression is an empty vector or a vector containing only blanks or one that does not produce a result, then `X⍎Y` has no value and using or assigning it to a name will generate `VALUE ERROR`.

If `Y` contains a branch expression, the branch is effected in the environment from which the *dyadic execute* was invoked, and `X⍎Y` does not return.

Examples

Using a named namespace:

```
      space←'myspace'
      space ⎕NS''
      space⍎'nums←⍳6'
      space⍎'nums'
1 2 3 4 5 6
```

Using a reference:

```
      space←⎕NS''
      space⍎'nums←⍳6'
      space⍎'nums'
1 2 3 4 5 6
```

Hints and Recommendations

It is faster and (potentially) safer to use system functions instead of *dyadic execute* in the above examples as follows:

Using a named namespace:

```
      space←'myspace'
      space ⎕NS''
      space ⎕VSET⊂'nums'(⍳6)
      space ⎕VGET'nums'
1 2 3 4 5 6
```

Using a reference:

```
      space←⎕NS''
      space ⎕VSET⊂'nums'(⍳6)
      space ⎕VGET'nums'
1 2 3 4 5 6
```

3.4.14 Dyadic Grade Down R←X⍒Y

`Y` must be a simple character array of rank greater than 0. `X` must be a simple character array of rank 1 or greater. `R` is a simple integer vector of shape `1↑⍴Y` containing the permutation of `⍳1↑⍴Y` that places the sub-arrays of `Y` along the first axis in descending order according to the collation sequence `X`. The indices of any set of identical sub-arrays in `Y` occur in `R` in ascending order.

If `X` is a vector, the following identity holds:

```
      X⍒Y ↔ ⍒X⍳Y
```

A left argument of rank greater than 1 allows successive resolution of duplicate orderings in the following way.

Starting with the last axis:

- The characters in the right argument are located along the current axis of the left argument. The position of the first occurrence gives the ordering value of the character.
- If a character occurs more than once in the left argument its lowest position along the current axis is used.
- If a character of the right argument does not occur in the left argument, the ordering value is one more than the maximum index of the current axis - as with dyadic iota.

The process is repeated using each axis in turn, from the last to the first, resolving duplicates until either no duplicates result or all axes have been exhausted.

For example, if index origin is 1:

Left argument:	Right argument:
abc	ab
ABA	ac
	Aa
	Ac

Along last axis:

Character:	Value:	Ordering:
ab	1 2	3
ac	1 3	=1 <-duplicate ordering with
Aa	1 1	4
Ac	1 3	=1 <-respect to last axis.

Duplicates exist, so resolve these with respect to the first axis:

Character:	Value:	Ordering:
ac	1 1	2
Ac	2 1	1

So the final row ordering is:

```
      ab          3
      ac          2
      Aa          4
      Ac          1
```

That is, the order of rows is 4 2 1 3 which corresponds to a descending row sort of:

```
      Ac          1
      ac          2
      ab          3
      Aa          4
```

Examples

```
      ⍴S1
2 27
      S1
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
      S2
 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
      S3
 AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
      S4
 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
 abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The following results are tabulated for comparison:

```
X        X[S1⍒X;]    X[S2⍒X;]    X[S3⍒X;]    X[S4⍒X;]
FIRsT      TAPE          rAT           TAPE          TAPE
TAP        TAP           fIRST         TAP           TAP
RATE       RATE          TAPE          rAT           RATE
FiRST      rAT           TAP           RATE          rAT
FIRST      RAT           RATE          RAT           RAT
rAT        MAT           RAT           MAT           MAT
fIRST      fIRST         MAT           fIRST         FIRsT
TAPE       FiRST         FiRST         FiRST         FiRST
MAT        FIRsT         FIRsT         FIRsT         FIRST
RAT        FIRST         FIRST         FIRST         fIRST
```

⎕IO is an implicit argument of Grade Down.

3.4.15 Dyadic Grade Up R←X⍋Y

Y must be a simple character array of rank greater than 0. X must be a simple character array of rank 1 or greater. R is a simple integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays of Y along the first axis in ascending order according to the collation sequence X.

If X is a vector, the following identity holds:

```
      X⍋Y ↔ ⍋X⍳Y
```

If X is a higher-rank array, each axis of X represents a grading attribute in increasing order of importance (the first axis is the least significant and the last axis is the most significant). If a character is repeated in X, it is treated as though it were located at the position in the array determined by the lowest index in each axis for all occurrences of the character. The character has the same weighting as the character located at the derived position in X.

Examples

```
      (2 2⍴'ABBA') ≡ 'AB'[?5 2⍴2] ⍝ A and B are equivalent
1 2 3 4 5

        ]display A←2 14⍴' abcdegiklmnrt ABCDEGIKLMNRT'
┌→──────────────┐
↓ abcdegiklmnrt│
│ ABCDEGIKLMNRT│
└──────────────┘

      V←'Ab' 'AB' 'aba' 'ABA' 'abaca' 'abecedarian'
      V,←'Abelian' 'black' 'blackball' 'black belt'
      V,←'blacking' 'Black Mass'

      ]display M←↑V
┌→────────────┐
↓Ab           │
│AB           │
│aba          │
│ABA          │
│abaca        │
│abecedarian  │
│Abelian      │
│black        │
│blackball    │
│black belt   │
│blacking     │
│Black Mass   │
└─────────────┘
```

```
      ]display M (M[(,A)⍋M;]) (M[(,⍉A)⍋M;]) (M[A⍋M;])
┌→──────────────────────────────────────────────────────────────────────┐
│ ┌→──────────┐ ┌→──────────┐ ┌→──────────┐ ┌→──────────┐              │
│ ↓Ab         │ ↓aba        │ ↓aba        │ ↓Ab         │              │
│ │AB         │ │abaca      │ │abaca      │ │AB         │              │
│ │aba        │ │abecedarian│ │abecedarian│ │aba        │              │
│ │ABA        │ │black      │ │Ab         │ │ABA        │              │
│ │abaca      │ │black belt │ │Abelian    │ │abaca      │              │
│ │abecedarian│ │blackball  │ │AB         │ │abecedarian│              │
│ │Abelian    │ │blacking   │ │ABA        │ │Abelian    │              │
│ │black      │ │Ab         │ │black      │ │black      │              │
│ │blackball  │ │Abelian    │ │black belt │ │black belt │              │
│ │black belt │ │AB         │ │blackball  │ │Black Mass │              │
│ │blacking   │ │ABA        │ │blacking   │ │blackball  │              │
│ │Black Mass │ │Black Mass │ │Black Mass │ │blacking   │              │
│ └───────────┘ └───────────┘ └───────────┘ └───────────┘              │
└∊─────────────────────────────────────────────────────────────────────┘
```

3.4.16 Transpose (Dyadic) R←X⍉Y

Y may be any array. X must be a simple scalar or vector whose elements are included in the set ⍳⍴⍴Y. Integer values in X may be repeated but all integers in the set ⍳⌈/X must be included. The length of X must equal the rank of Y.

R is an array formed by the transposition of the axes of Y as specified by X. The I^{th} element of X gives the new position for the I^{th} axis of Y. If X repositions two or more axes of Y to the same axis, the elements used to fill this axis are those whose indices on the relevant axes of Y are equal.

⎕IO is an implicit argument of Dyadic Transpose.

Examples

```
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
```

```
      2 1 3⍉A
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24

      1 1 1⍉A
1 18

      1 1 2⍉A
 1  2  3  4
17 18 19 20
```

Alternative Explanation

Assign a distinct letter for each unique integer in `X` :

```
0 1 2 3 …
i j k l
```

If `R←X⍉Y`, then `R[i;j;k;…]` equals `Y` indexed by the letters corresponding to elements of `X` .

For example

```
      ⎕IO←0

      Y← ? 5 13 19 17 11 ⍴ 100

      X← 2 1 2 0 1
      ⍺  k j k i j
```

```
      R←X⍉Y
```

```
      i←?17 ⋄ j←?11 ⋄ k←?5
      R[i;j;k] = Y[k;j;k;i;j]
1
      R[i;j;k]=Y[⊂⍋⍨'ijk'[X]]
1
```

From the above it can be seen that:

- the rank of R is `0⌈1+⌈/X`
- the shape of R is `(⍴Y)⌊.+(⌈/⍴Y)×X∘.≠⍳0⌈1+⌈/X`

3.4.17 Enclose

Enclose `R←⊂Y`

Y may be any array. R is a scalar array whose item is the array Y. If Y is a simple scalar, R is the simple scalar unchanged. Otherwise, R has a depth whose magnitude is one greater than the magnitude of the depth of Y.

EXAMPLES

```
      ⊂1
1

      ⊂'A'
A

      ⊂1 2 3
 1 2 3

      ⊂1,⊂'CAT'
 1  CAT

      ⊂2 4⍴⍳8
 1 2 3 4
 5 6 7 8

      ⊂⍳0

      ⊂⊂⍳0

      ⊂⊂10
10
```

See also: *Section 3.4.17.2*.

Enclose with Axes R←⊂[K]Y

Y may be any array. K is a vector of zero or more axes of Y. R is an array of the elements of Y enclosed along the axes K. The shape of R is the shape of Y with the K axes removed:

```
      ⍴R ↔ (⍴Y)[(⍳⍴⍴R)~K]
```

The shape of each element of R is the shape of the Kth axes of Y:

```
      ⍴⊃R ↔ (⍴Y)[,K]
```

Examples

```
      ]display A←2 3 4⍴'DUCKSWANBIRDWORMCAKESEED'
┌┌→──┐
↓↓DUCK│
│└──┘│
│ SWAN│
│ BIRD│
│     │
│ WORM│
│ CAKE│
│ SEED│
└────┘
```

```
      ]display ⊂[3]A
┌→──────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │DUCK│ │SWAN│ │BIRD│ │
│ └──┘ └──┘ └──┘ │
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │WORM│ │CAKE│ │SEED│ │
│ └──┘ └──┘ └──┘ │
└∊──────────────────┘
```

```
      ]display ⊂[2 3]A
┌→──────────────┐
│ ┌→──┐ ┌→──┐  │
│ ↓DUCK│ ↓WORM│ │
│ │SWAN│ │CAKE│ │
│ │BIRD│ │SEED│ │
│ └───┘ └───┘  │
└∈──────────────┘

      ]display ⊂[1 3]A
┌→──────────────────┐
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ ↓DUCK│ ↓SWAN│ ↓BIRD│ │
│ │WORM│ │CAKE│ │SEED│ │
│ └───┘ └───┘ └───┘ │
└∈──────────────────┘
```

3.4.18 Encode R←X⊤Y

`Y` must be a simple numeric array. `X` must be a simple numeric array. `R` is the numeric array which results from the representation of `Y` in the number system defined by `X`.

The shape of `R` is `(⍴X),⍴Y` (the catenation of the shapes of `X` and `Y`).

If `X` is a vector or a scalar, the result for each element of `Y` is the value of the element expressed in the number system defined by radix `X`. If `Y` is greater than can be expressed in the number system, the result is equal to the representation of the residue `(×/X)|Y`. If the first element of `X` is 0, the value will be fully represented.

This function is also known as Representation.

Examples

```
      10⊤5 15 125
5 5 5

      0 10⊤5 15 125
0 1 12
5 5  5
```

If `X` is a higher-rank array, each of the vectors along the first axis of `X` is used as the radix vector for each element of `Y`.

Examples

```
      A
2 0  0
2 0  0
2 0  0
2 0  0
2 8  0
2 8  0
2 8 16
2 8 16
```

This example shows binary, octal and hexadecimal representations of the decimal number 75.

```
      A⍉75
0 0  0
1 0  0
0 0  0
0 0  0
1 0  0
0 1  0
1 1  4
1 3 11
```

Examples

```
      0 1⍉1.25 10.5
1    10
0.25  0.5

      4 13⍉13?52
 3 1 0  2 3 2 0 1  3 1 2 3 1
12 2 4 12 1 7 6 3 10 1 0 3 8
```

⎕IO is not an implicit argument of encode.

3.4.19 Enlist (⎕ML≥1) R←∈Y

Migration level must be such that ⎕ML≥1 (otherwise see *Section 3.4.90*).

Y may be any array, R is a simple vector created from all the elements of Y in ravel order.

Examples

```
      ⎕ML←1          ⍝  Migration level 1
      MAT←2 2⍴'MISS' 'IS' 'SIP' 'PI' ⋄ MAT
 MISS  IS
 SIP   PI
      ∈MAT
MISSISSIPPI

      M←1 (2 2⍴2 3 4 5) (6(7 8))
      M
1  2 3  6  7 8
   4 5
      ∈M
1 2 3 4 5 6 7 8
```

3.4.20 Equal To R←X=Y

`Y` may be any array. `X` may be any array. `R` is Boolean.

`⎕CT` and `⎕DCT` are implicit arguments of Equal To.

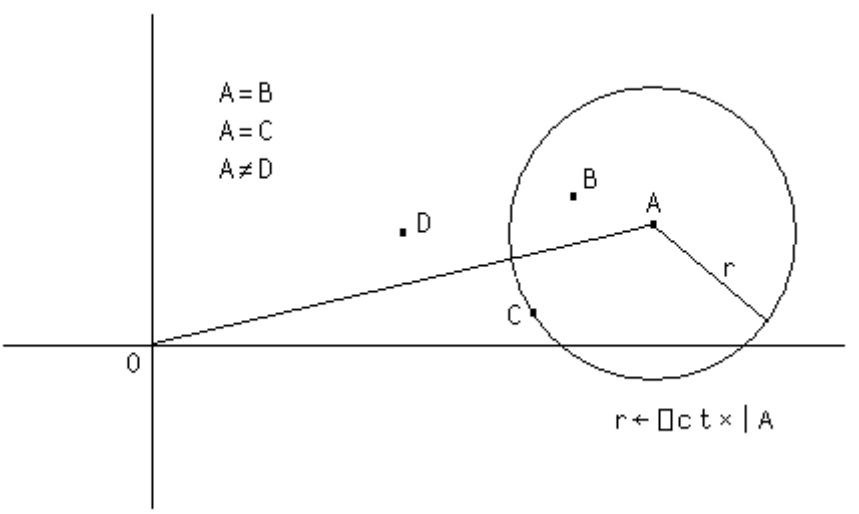
If `X` and `Y` are refs, then `R` is 1 if they are refs to the same object. If `X` is a ref and `Y` is not, or vice-versa, then `R` is 0.

If `X` and `Y` are character, then `R` is 1 if they are the same character. If `X` is character and `Y` is numeric, or vice-versa, then `R` is 0.

If `X` and `Y` are numeric, then `R` is 1 if `X` and `Y` are within comparison tolerance of each other.

For real numbers `X` and `Y`, `X` is considered equal to `Y` if `(|X-Y)` is not greater than `⎕CT×(|X)⌈|Y`.

For complex numbers `X=Y` is 1 if the magnitude of `X-Y` does not exceed `⎕CT` times the larger of the magnitudes of `X` and `Y`; geometrically, `X=Y` if the number smaller in magnitude lies on or within a circle centred on the one with larger magnitude, having radius `⎕CT` times the larger magnitude.



Examples

```
      3=3.1 3 ¯2 ¯3
0 1 0 0

      a←2+0j1×⎕CT
      a
2J1E¯14
      a=2j.00000000000001 2j.0000000000001
1 0

      'CAT'='FAT'
0 1 1

      'CAT'=1 2 3
0 0 0

      'CAT'='C' 2 3
1 0 0

      ⎕CT←1E¯10
      1=1.000000000001
1

      1=1.0000001
0
```

3.4.21 Execute R←⍎Y

> **Warning**
>
> If the argument to *execute* could include user input, then there is a risk to data and systems. To reduce this risk, a system function might be more appropriate than the *execute* function. For example:
>
> - use *Section 6.4.171* or [⎕VSET](../system-functions/vset.md to get or set the value of one or more variables named within one or more character vectors.
> - use *Section 6.4.174*, *Section 6.4.72*, or *Section 6.4.21* to make numbers in text form into actual numbers.
> - use *Section 6.4.116* to call a function by name; exact usage depends on valency. For example:
> - niladic – `(⎕OR fnName){⍺⍺}`
> - monadic – `(⎕OR fnName){⍺⍺ ⍵}YY`
> - dyadic – `X((⎕OR fnName){⍺ ⍺⍺ ⍵})Y`

`Y` must be a simple character scalar or vector containing an APL expression to be executed. The expression may contain one or more sub-expressions separated by ⋄ (Diamond) characters.

If the result of the expression is used or is assigned to a name, `R` is the result (if any) of the last-executed sub-expression and the non-shy results of all preceding expressions (that are not assigned within the expression) are displayed. Otherwise the unassigned non-shy results of all of the sub-expressions are displayed.

If the expression is an empty vector or a vector containing only blanks or one that does not produce a result, then `⍎Y` has no value and using or assigning it to a name will generate `VALUE ERROR`.

If `Y` contains a branch expression, the branch is effected in the environment from which *execute* was invoked, and `⍎Y` does not return.

Examples

```
      ⍎'2+2'
4
      ⍎'1+1 ⋄ 2+2'
2
4
      A← ⍎'1+1 ⋄ 2+2'
2
      A
4
      4=⍎'1+1 ⋄ 2+2'
2
1
      ⍎'A←2|¯1↑⎕TS ⋄ →0⍴⍨A ⋄ A'
0
      A
0
      A←⍎''
VALUE ERROR: No result was provided when the context expected one
      A←⍎''
        ^
```

3.4.22 Expand R←X\[K]Y

`Y` may be any array. `X` is a simple integer scalar or vector. The axis specification is optional. If present, `K` must be a simple integer scalar or 1-element vector. The value of `K` must be an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←X⍀Y` implies the first axis. If `Y` is a scalar, it is treated as a one-element vector.

If `Y` has length 1 along the K^{th} (or implied) axis, it is extended along that axis to match the number of positive elements in `X`. Otherwise, the number of positive elements in `X` must be the length of the K^{th} (or implied) axis of `Y`.

`R` is composed from the sub-arrays along the K^{th} axis of `Y`. If `X[I]` (an element of `X`) is the J^{th} positive element in `X`, then the J^{th} sub-array along the K^{th} axis of `Y` is replicated `X[I]` times. If `X[I]` is negative, then a sub-array of fill elements of `Y` is replicated `|X[I]` times and inserted in relative order along the K^{th} axis of the result. If `X[I]` is zero, it is treated as the value `¯1`. The shape of `R` is the shape of `Y` except that the length of the K^{th} axis is `+/1⌈|X`.

Examples

```
      0\⍳0
0

      1 ¯2 3 ¯4 5\'A'
A  AAA    AAAAA

      M
1 2 3
4 5 6

      1 ¯2 2 0 1\M
1 0 0 2 2 0 3
4 0 0 5 5 0 6

      1 0 1⌿M
1 2 3
0 0 0
4 5 6

      1 0 1\[1]M
1 2 3
0 0 0
4 5 6

      1 ¯2 1\(1 2)(3 4 5)
 1 2  0 0  0 0  3 4 5
```

3.4.23 Expand First R←X⌿Y

The form R←X⌿Y implies expansion along the first axis whereas the form R←X\Y implies expansion along the last axis (columns). See *Section 3.4.22*.

3.4.24 Exponential R←*Y

Y must be numeric. R is numeric and is the Yth power of *e*, the base of natural logarithms.

EXAMPLE

```
      *1 0
2.718281828 1

      *0j1 1j2
0.5403023059J0.8414709848 ¯1.131204384J2.471726672

      1+*○0j1 ⍝ Euler Identity
0
```

3.4.25 Factorial R←!Y

Y must be numeric excluding negative integers. R is numeric. R is the product of the first Y integers for positive integer values of Y. In general, !Y is gamma Y+1.

EXAMPLES

```
      !1 2 3 4 5
1 2 6 24 120

      !¯1.5 0 1.5 3.3
¯3.544907702 1 1.329340388 8.85534336

      !0j1 1j2
0.4980156681J¯0.1549498283 0.1122942423J0.3236128855
```

3.4.26 Find R←X⍷Y

X and Y may be any arrays. R is a simple Boolean array the same shape as Y which identifies occurrences of X within Y.

If the rank of X is smaller than the rank of Y, X is treated as if it were the same rank with leading axes of size 1. For example a vector is treated as a 1-row matrix.

If the rank of X is larger than the rank of Y, no occurrences of X are found in Y.

⎕CT and ⎕DCT are implicit arguments of Find.

Examples

```
      'AN'∈'BANANA'
0 1 0 1 0 0

      'ANA'∈'BANANA'
0 1 0 1 0 0

      'BIRDS' 'NEST'∈'BIRDS' 'NEST' 'SOUP'
1 0 0

      MAT
IS YOU IS
OR IS YOU
ISN'T
      'IS'∈MAT
1 0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0
      'IS YOU'∈MAT
1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

3.4.27 First (⎕ML) R←⊃Y or R←↑Y

The symbol chosen to represent First depends on the current Migration Level.

If ⎕ML<2, First is represented by the symbol: ⊃.

If ⎕ML≥2, First is represented by the symbol: ↑.

Y may be any array. R is an array. If Y is non-empty, R is the value of the first item of Y taken in ravel order. If Y is empty, R is the prototype of Y.

First is the inverse of Enclose. The identity R←→⊃⊂R holds for all R. First is also referred to as Disclose.

Examples

```
      ⊃1
1

      ⊃2 4 6
2

      ⊃'MONDAY' 'TUESDAY'
MONDAY

      ⊃(1 (2 3))(4 (5 6))
1  2 3

      ⊃⍳0
0

      ' '⊃''
1

      ⊃1↓⊂1,⊂2 3
0  0 0
```

3.4.28 Floor R←⌊Y

Y must be numeric.

For real numbers, R is the largest integer value less than or equal to Y within the comparison tolerance ⎕CT.

Examples

```
      ⌊¯2.3 0.1 100 3.3
¯3 0 100 3

      ⌊0.5 + 0.4 0.5 0.6
0 1 1
```

For complex numbers, R depends on the relationship between the real and imaginary parts of the numbers in Y.

```
      ⌊1j3.2 3.3j2.5 ¯3.3j¯2.5
1J3 3J2 ¯3J¯3
```

Complex Floor

The following (deliberately) simple function illustrates one way to express the rules for evaluating complex Floor.

```
     ∇ fl←CpxFloor cpxs;a;b
[1]   ⍝ Complex floor of scalar complex number (a+ib)
[2]    a b←9 11○cpxs
[3]    :If 1>(a-⌊a)+b-⌊b
[4]        fl←(⌊a)+0J1×⌊b
[5]    :Else
[6]        :If (a-⌊a)<b-⌊b
[7]            fl←(⌊a)+0J1×1+⌊b
[8]        :Else
[9]            fl←(1+⌊a)+0J1×⌊b
[10]       :EndIf
[11]   :EndIf
     ∇

      CpxFloor¨1j3.2 3.3j2.5 ¯3.3j¯2.5
1J3 3J2 ¯3J¯3
```

⎕CT and ⎕DCT are implicit arguments of Floor.

3.4.29 Format R←⍕Y

Y may be any array. R is a simple character array which will display identically to the display produced by Y. The result is independent of ⎕PW. If Y is a simple character array, then R is Y.

EXAMPLE

```
      +B←⍕A←2 6⍴'HELLO PEOPLE'
HELLO
PEOPLE

      B ≡ A
1
```

If Y is a simple numeric scalar, then R is a vector containing the formatted number without any spaces. A floating point number is formatted according to the system variable ⎕PP. ⎕PP is ignored when formatting integers.

Examples

```
      ⎕PP←5

      ⍴C←⍕⍳0
0

      ⍴C←⍕10
2

      C
10

      ⍴C←⍕12.34
5

      C
12.34

      ⍕123456789
123456789

      ⍕123.456789
123.46
```

Scaled notation is used if the magnitude of the non-integer number is too large to represent with ⎕PP significant digits or if the number requires more than five leading zeroes after the decimal point.

Examples

```
      ⍕123456.7
1.2346E5

      ⍕0.0000001234
1.234E¯7
```

If Y is a simple numeric vector, then R is a character vector in which each element of Y is independently formatted with a single separating space between formatted elements.

Example

```
      ⍴C←⍕¯123456 1 22.5 ¯0.000000667 5.00001
27

      C
¯1.2346E5 1 22.5 ¯6.67E¯7 5
```

If `Y` is a simple numeric array rank higher than one, `R` is a character array with the same shape as `Y` except that the last dimension of `Y` is determined by the length of the formatted data. The format width is determined independently for each column of `Y`, such that:

1. the decimal points for floating point or scaled formats are aligned.
2. the `E` characters for scaled formats are aligned, with trailing zeros added to the mantissae if necessary.
3. integer formats are aligned to the left of the decimal point column, if any, or right-adjusted in the field otherwise.
4. each formatted column is separated from its neighbours by a single blank column.
5. the exponent values in scaled formats are left-adjusted to remove any blanks.

Examples

```
      C←22 ¯0.000000123 2.34 ¯212 123456 6.00002 0

      ⍴C←⍕2 2 3⍴C
2 2 29

      C
  22    ¯1.2300E¯7  2.3400E0
¯212     1.2346E5   6.0000E0

   0     2.2000E1  ¯1.2300E¯7
   2.34 ¯2.1200E2   1.2346E5
```

If `Y` is non-simple, and all items of `Y` at any depth are scalars or vectors, then `R` is a vector.

Examples

```
      B←⍪A←'ABC' 100 (1 2 (3 4 5)) 10

      ⍴A
4
      ≡A
¯3

      ⍴B
26
      ≡B
1

      A
 ABC  100  1 2  3 4 5   10

      B
 ABC  100  1 2  3 4 5   10
```

By replacing spaces with ^, it is clearer to see how the result of ⍪ is formed:

```
^ABC^^100^^1^2^^3^4^5^^^10
```

If Y is non-simple, and all items of Y at any depth are not scalars, then R is a matrix.

Example

```
      D←⍪C←1 'AB' (2 2⍴1+⍳4) (2 2 3⍴'CDEFGHIJKLMN')

      C
1  AB  2 3  CDE
       4 5  FGH

            IJK
            LMN

      ⍴C
4

      ≡C
¯2

      D
1  AB  2 3  CDE
       4 5  FGH

            IJK
            LMN

      ⍴D
5 16

      ≡D
1
```

By replacing spaces with ^, it is clearer to see how the result of ⍪ is formed:

```
1^^AB^^2^3^^CDE^
^^^^^^^4^5^^FGH^
^^^^^^^^^^^^^^^^
^^^^^^^^^^^IJK^
^^^^^^^^^^^LMN^
```

⎕PP is an implicit argument of Format.

3.4.30 Format by Specification R←X⍕Y

Y must be a simple real (non-complex) numeric array. X must be a simple integer scalar or vector. R is a character array displaying the array Y according to the specification X. R

has rank `1⌈⍴⍴Y` and `¯1↓⍴R` is `¯1↓⍴Y`. If any element of `Y` is complex, dyadic `⍕` reports a `DOMAIN ERROR`.

Conformability requires that if `X` has more than two elements, then `⍴X` must be `2×¯1↑⍴Y`. If `X` contains one element, it is extended to `(2×¯1↑⍴Y)⍴0,X`. If `X` contains 2 elements, it is extended to `(2×¯1↑⍴Y)⍴X`.

`X` specifies two numbers (possibly after extension) for each column in `Y`. For this purpose, scalar `Y` is treated as a one-element vector. Each pair of numbers in `X` identifies a format width (`W`) and a format precision (`P`).

If `P` is 0, the column is to be formatted as integers.

Examples

```
      5 0 ⍕ 2 3⍴⍳6
 1     2     3
 4     5     6

      4 0⍕1.1 2 ¯4 2.547
1   2  ¯4   3
```

Example

If `P` is positive, the format is floating point with `P` significant digits to be displayed after the decimal point.

```
      4 1⍕1.1 2 ¯4 2.547
 1.1 2.0¯4.0 2.5
```

Example

If `P` is negative, scaled format is used with `|P` digits in the mantissa.

```
      7 ¯3⍕5 15 155 1555
5.00E0 1.50E1 1.55E2 1.56E3
```

Example

If `W` is 0 or absent, then the width of the corresponding columns of `R` are determined by the maximum width required by any element in the corresponding columns of `Y`, plus one separating space.

```
      3⍕2 3⍴10 15.2346 ¯17.1 2 3 4
 10.000 15.235 ¯17.100
  2.000  3.000   4.000
```

EXAMPLE

If a formatted element exceeds its specified field width when ⍵>0, the field width for that element is filled with asterisks.

```
      3 0 6 2 ⍕ 3 2⍴10.1 15 1001 22.357 101 1110.1
 10 15.00
*** 22.36
101******
```

EXAMPLE

If the format precision exceeds the internal precision, low order digits are replaced by the symbol '_'.

```
      26⍕2*100
1267650600228229_____________.__________________________

      ⍴26⍕2*100
59

      0 20⍕÷3
 0.3333333333333333____

      0 ¯20⍕÷3
 3.333333333333333____E¯1
```

The shape of R is the same as the shape of Y except that the last dimension of R is the sum of the field widths specified in X or deduced by the function. If Y is a scalar, the shape of R is the field width.

```
      ⍴5 2 ⍕ 2 3 4⍴⍳24
2 3 20
```

3.4.31 Grade Down R←⍧Y

Y may be any array of rank greater than 0 but may not contain namespaces. R is an integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays along the first axis in descending order. For the rules for comparing items of Y with one another, see *Section 3.4.32*.

⎕IO is an implicit argument of Grade Down.

Examples

```
      ⍒22.5 1 15 3 ¯4
1 3 4 2 5
```

```
      M
2 3 5
1 4 7

2 3 4
5 2 4

2 3 5
1 2 6
      ⍒M
1 3 2
```

> **Note**
>
> Character arrays sort differently in the Unicode and Classic Editions.

```
      M
Goldilocks
porridge
Porridge
3 bears
```

Unicode Edition	Classic Edition
⍋M 2 3 1 4	⍋M 3 1 4 2
M[⍋M;] porridge Porridge Goldilocks 3 bears	M[⍋M;] Porridge Goldilocks 3 bears porridge

```
      ⍴pb
6 3
      pb
┌────────┬─────┬───┐
│Rivers  │Jason│554│
├────────┼─────┼───┤
│Daintree│John │532│
├────────┼─────┼───┤
│Rivers  │Jason│543│
├────────┼─────┼───┤
│Foad    │Jay  │558│
├────────┼─────┼───┤
│Scholes │John │547│
├────────┼─────┼───┤
│Scholes │John │535│
└────────┴─────┴───┘
      ⍒pb
5 6 1 3 4 2
```

3.4.32 Grade Up R←⍋Y

Y may be any array of rank greater than 0 but may not contain namespaces. R is an integer vector being the permutation of ⍳1↑⍴Y that places the sub-arrays along the first axis in ascending order. The rules for comparing items of Y with one another are as follows:

Rules for comparing simple scalars

- Numeric comparisons are exact, as if ⎕CT←⎕DCT←0 and ⎕FR←1287
- Two real numbers are compared numerically, thus 1.2 precedes 3.
- In the Unicode Edition two characters are compared numerically according to their position in the Unicode table. Thus `'a'` (⎕UCS 97) precedes`'b'` (⎕UCS 98). In the Classic Edition characters are compared according to their index in ⎕AV.
- Complex numbers are ordered by first comparing their real parts. If these are equal, the order is determined by comparing their imaginary parts.Thus `1J¯2` precedes `1` which precedes `1J2`.
- ⎕NULL (which represents a null item obtained from an external source) precedes all numbers, and all numbers precede all characters.Thus ⎕NULL precedes 100, and 100 precedes `'A'`.

Rules for comparing non-scalar arrays

- Arrays are compared item by item in ravel order.
- For arrays of equal shape, the order is determined by the first pair of items which differ, thus `(1949 4 29)` precedes `(1949 4 30)`. Similarly `('April' 29)` precedes `('April' 30)`.
- Arrays with the same rank but different shape are ordered as if the shorter array were padded with items that precede all other types of item (negative infinity) including ⎕NULL. Thus `'car'` precedes `'carpet'`and `(1949 4)` precedes `(1949 4 30)`. An alternative model is to say that shorter arrays precede longer ones that begin the same way. For character vectors this is described as Lexicographical ordering, which is the order that words appear in a dictionary.
- Arrays with differing rank are ordered by first extending the shape of the lower-ranked array with 1s at the beginning, and then comparing the resultant equal-rank arrays as described above. So, to compare a vector (rank 1) with a matrix (rank 2), the vector is reshaped into a 1-row matrix.
- Empty arrays are compared first by type alone, so an empty numeric array precedes an empty character array, regardless of rank or shape.Thus `((0 3 2)⍴0)` precedes `''`. If the empty arrays are of the same type, they are sorted in order of their shape vector, working right to left.So `((0 5 2)⍴99)` precedes `((0 3 4)⍴0)` and`((0 3 4)⍴'')` precedes `((1 0 5 4)⍴'')`.

⎕IO is an implicit argument of Grade Up.

Examples

```
      ⍋22.5 1 15 3 ¯4
5 2 4 3 1
```

```
      M
2 3 5
1 4 7

2 3 4
5 2 4

2 3 5
1 2 6
      ⍋M
2 3 1
```

> **Note**
>
> Character arrays sort differently in the Unicode and Classic Editions.

```
      M
Goldilocks
porridge
Porridge
3 bears
```

Unicode Edition	Classic Edition
`      ⍋M` `4 1 3 2`	`      ⍋M` `2 4 1 3`

`      M[⍋M;]`
`3 bears`
`Goldilocks`
`Porridge`
`porridge`
	`      M[⍋M;]`
	`porridge`
	`3 bears`
	`Goldilocks`
	`Porridge`

```
      ⍴pb
6 3
      pb
┌────────┬─────┬───┐
│Rivers  │Jason│554│
├────────┼─────┼───┤
│Daintree│John │532│
├────────┼─────┼───┤
│Rivers  │Jason│543│
├────────┼─────┼───┤
│Foad    │Jay  │558│
├────────┼─────┼───┤
│Scholes │John │547│
├────────┼─────┼───┤
│Scholes │John │535│
└────────┴─────┴───┘
      ⍋pb
2 4 3 1 6 5
```

3.4.33 Greater Than R←X>Y

Y must be numeric. X must be numeric. R is Boolean. R is 1 if X is greater than Y and X=Y is 0. Otherwise R is 0.

⎕CT and ⎕DCT are implicit arguments of Greater Than.

EXAMPLES

```
      1 2 3 4 5 > 2
0 0 1 1 1

      ⎕CT←1E¯10

      1 1.00000000001 1.000000001 > 1
0 0 1
```

3.4.34 Greater Than Or Equal To R←X≥Y

Y must be numeric. X must be numeric. R is Boolean. R is 1 if X is greater than Y or X=Y. Otherwise R is 0.

⎕CT and ⎕DCT are implicit arguments of Greater Than Or Equal To.

EXAMPLES

```
      1 2 3 4 5 ≥ 3
0 0 1 1 1

      ⎕CT←1E¯10

      1≥1
1

      1≥1.00000000001
1

      1≥1.00000001
0
```

3.4.35 Index

Index R←X⌷Y

X must be a scalar or vector of depth ≤2 of integers each ≥⎕IO. Y may be any array. In general, the result R is similar to that obtained by square-bracket indexing in that:

```
      (I J ... ⌷ Y) ≡ Y[I;J;...]
```

The length of left argument X must be less than or equal to the rank of right argument Y. Any missing trailing items of X default to the index vector of the corresponding axis of Y.

Note that in common with square-bracket indexing, items of the left argument X may be of any rank and that the shape of the result is the concatenation of the shapes of the items of the left argument:

```
      (⍴X⌷Y) ≡ ↑,/⍴¨X
```

Index is sometimes referred to as *squad indexing*.

Note that index may be used with selective specification.

⎕IO is an implicit argument of index.

EXAMPLES

```
      ⎕IO←1

      VEC←111 222 333 444
      3⌷VEC
333
      (⊂4 3)⌷VEC
444 333
      (⊂2 3⍴3 1 4 1 2 3)⌷VEC
333 111 444
111 222 333

      ⎕←MAT←10⊥⍣¯1⍳3 4
11 12 13 14
21 22 23 24
31 32 33 34

      2 1⌷MAT
21
      2⌷MAT
21 22 23 24
```

```
      3(2 1)⌷MAT
32 31
      (2 3)1⌷MAT
21 31
      (2 3)(,1)⌷MAT
21
31
      ⍴(2 1⍴1)(3 4⍴2)⌷MAT
2 1 3 4
      ⍴⍬ ⍬⌷MAT
0 0
      (3(2 1)⌷MAT)←0 ⋄ MAT    ⍝ Selective assignment.
11 12 13 14
21 22 23 24
 0  0 33 34
```

Indexing Classes

If `Y` is a ref to an instance of a Class with a Default property, `⌷` is applied to the Default property. Similarly, `⌷` applied to a COM or .NET collection returns the appropriate item(s) of the collection.

Example

```
      ↑⎕SRC c
:Class c
    :Property Default p
    :Access Public Shared
        ∇ r←get
          r←2 3 4⍴⎕A
        ∇
    :EndProperty
:EndClass

      2⎕c
MNOP
QRST
UVWX
```

Index with Axes R←{X}⌷[K]Y

X must be a scalar or vector of depth ≤2, of integers each ≥⎕IO. Y may be any array. K is a simple scalar or vector specifying axes of Y. The length of K must be the same as the length of X:

```
      (⍴,X) ≡ ⍴,K
```

In general, the result R is similar to that obtained by square-bracket indexing with elided subscripts. Items of K distribute items of X along the axes of Y. For example:

```
      I J ⌷[1 3] Y  ←→  Y[I;;J]
```

Note that index with axis may be used with selective specification. ⎕IO is an implicit argument of index with axis.

Examples

```
      ⎕IO←1

      ⎕←CUBE←10⊥⍤⍳2 3 4
111 112 113 114
121 122 123 124
131 132 133 134

211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[1]CUBE
211 212 213 214
221 222 223 224
231 232 233 234

      2⌷[3]CUBE
112 122 132
212 222 232

      CUBE[;;2] ≡ 2⌷[3]CUBE
1
      (1 3)⍉⌷[2 3]CUBE
114 134
214 234

      CUBE[;1 3;⍉] ≡ (1 3)⍉⌷[2 3]CUBE
1
```

```
      (2(1 3)⍉⌷[1 3]CUBE)←0 ⋄ CUBE ⍝ Selective assignment.
111 112 113 114
121 122 123 124
131 132 133 134

  0 212   0 214
  0 222   0 224
  0 232   0 234
```

3.4.36 Index Generator R←⍳Y

Y must be a simple scalar or vector array of non-negative numbers. R is a numeric array composed of the set of all possible coordinates of an array of shape Y. The shape of R is Y and each element of R occurs in its self-indexing position in R. In particular, the following identity holds:

```
      ⍳Y ↔ (⍳Y)[⍳Y]
```

⎕IO is an implicit argument of Index Generator. This function is also known as Interval.

Examples

```
      ⎕IO
1
      ⍴⍳0
0
      ⍳5
1 2 3 4 5

      ⍳2 3
 1 1  1 2  1 3
 2 1  2 2  2 3

      ⊢A←2 4⍴'MAINEXIT'
MAIN
EXIT
      A[⍳⍴A]
MAIN
EXIT
```

```
      ⎕IO←0
      ⍳5
0 1 2 3 4

      ⍳2 3
 0 0  0 1  0 2
 1 0  1 1  1 2

      A[⍳⍴A]
MAIN
EXIT
```

3.4.37 Index Of R←X⍳Y

Y may be any array. X may be any array of rank 1 or more.

In general, the function locates the first occurrence of sub-arrays in Y which match major cells of X, where a major cell is a sub-array on the leading dimension of X with shape 1↓⍴X. The shape of the result R is (1-⍴⍴X)↓⍴Y.

If a sub-array of Y cannot be found in X, then the corresponding element of R will be ⎕IO+⊃⍴X.

In particular, if X is a vector, the result R is a simple integer array with the same shape as Y identifying where elements of Y are first found in X. If an element of Y cannot be found in X, then the corresponding element of R will be ⎕IO+⊃⍴X.

Elements of X and Y are considered the same if X≡Y returns 1 for those elements.

⎕IO, ⎕CT and ⎕DCT are implicit arguments of Index Of.

Examples

```
      ⎕IO←1

      2 4 3 1 4⍳1 2 3 4 5
4 1 3 2 6

      'CAT' 'DOG' 'MOUSE'⍳'DOG' 'BIRD'
2 4
```

```
      X←3 4⍴⍳12
```

```
      X
1  2  3  4
5  6  7  8
9 10 11 12
```

```
      X⍳1 2 3 4
1
```

```
      Y←2 4⍴1 2 3 4 9 10 11 12
      Y
1  2  3  4
9 10 11 12

      X⍳Y
1 3
      X⍳2 3 4 1
4
```

```
      X1←10 100 1000∘.+X
      X1
  11   12   13   14
  15   16   17   18
  19   20   21   22

 101  102  103  104
 105  106  107  108
 109  110  111  112

1001 1002 1003 1004
1005 1006 1007 1008
1009 1010 1011 1012
```

```
      X1⍳100 1000∘.+X
2 3
```

```
      x
United Kingdom
Germany
France
Italy
United States
Canada
Japan
Canada
France
      y
United Kingdom
Germany
France
Italy
USA

Canada
Japan
China
India
Deutschland
```

```
      ⍴x
9 14
      ⍴y
2 5 14
      x⍳y
1 2  3  4 10
6 7 10 10 10

      x⍳x
1 2 3 4 5 6 7 6 3
```

Note that the expression `y⍳x` signals a `LENGTH ERROR` because it looks for major cells in the left argument, whose shape is `5 14` (that is `1↓⍴y`), which is not the same as the trailing shape of `x`.

```
      y⍳x
LENGTH ERROR
      y⍳x
     ∧
```

For performance information, see *Programming: Search Functions And Hash*.

3.4.38 Intersection R←X∩Y

Y must be a scalar or vector. X must be a scalar or vector. A scalar X or Y is treated as a one-element vector. R is a vector composed of items occurring in both X and Y in the order of occurrence in X. If an item is repeated in X and also occurs in Y, the item is also repeated in R.

Items in X and Y are considered the same if X≡Y returns 1 for those items.

⎕CT and ⎕DCT are implicit arguments of Intersection.

EXAMPLES

```
      'ABRA'∩'CAR'
ARA

      1 'PLUS' 2 ∩ ⍳5
1 2
```

For performance information, see *Programming: Search Functions And Hash*.

3.4.39 Interval Index R←X⍸Y

> **Classic Edition**
>
> The symbol ⍸ (Iota Underbar) is not available in Classic Edition, and Interval Index is instead represented by ⎕U2378.

X is an ordered non-scalar array that represents a set of intervals or ranges.

Note that the i^{th} interval starts at X[i], then includes all subsequent values up to but not including X[i+1].

For example, if X is (1 3 5) it defines 4 intervals numbered 0 to 3 as follows.

0	less than 1	`<1`
1	between 1 and 3	`(≥1)∧(<3)`
2	between 3 and 5	`(≥3)∧(<5)`
3	greater than or equal to 5	`≥5`

If `X` is `'AEIOU'` it defines 6 intervals numbered 0 to 5 as follows:

0	before A	`⎕UCS 0,⍳64`
1	between A and E	`ABCD`
2	between E and I	`EFGH`
3	between I and O	`IJKLMN`
4	between O and U	`OPQREST`
5	U and after	`UVWXYZ...`

`Y` is an array of the same type (numeric or character) as `X`.

The result `R` is an integer array that identifies into which interval the corresponding value in `Y` falls.

Like dyadic `⍳` (see *Section 3.4.37*), Interval Index works with major cells. For a vector these are its elements; for a matrix its rows, and so forth.

`X` and `Y` are compared using the same logic as monadic `≤` (see *Section 3.4.32*) which is independent of `⎕CT` and `⎕DCT`.

`⎕IO` is an implicit argument of Interval Index. In all the following examples, `⎕IO` is 1.

Example

```
      10 20 30⍸11 1 31 21
1 0 3 2
```

In the above example:

- 11 is between `X[1]` and `X[2]` so the answer is 1.
- 1 is less than `X[1]` so the answer is 0
- 31 is greater than `X[⍴X]` so the answer is 3
- 21 is between `X[2]` and `X[3]` so the answer is 2.

```
      'AEIOU' ⍳ 'DYALOG'
1 5 1 3 4 2
```

And in the alphabetic example above:

- "D" is between `X[1]` and `X[2]`, so the answer is 1
- "Y" is after `X[⍴X]` so the answer is 5
- "A" is between `X[1]` and `X[2]`, so the answer is 1
- as so on ...

Example (Classification)

Commercially, olive oil is graded as follows:

- if its acidity is less than 0.8%, as "Extra Virgin"
- if its acidity is less than 2%, as "Virgin"
- if its acidity is less than 3.3%, as "Ordinary"
- otherwise, as "Lampante"

```
     grades←'Extra Virgin' 'Virgin' 'Ordinary' 'Lampante'
     acidity←0.8 2 3.3

     samples←1.3 1.9 0.7 4 .6 3.2
     acidity⍳samples
1 1 0 3 0 2
     samples,⍪grades[1+acidity⍳samples]
┌───┬──────────────┐
│1.3│Virgin        │
├───┼──────────────┤
│1.9│Virgin        │
├───┼──────────────┤
│0.7│Extra Virgin  │
├───┼──────────────┤
│4  │Lampante      │
├───┼──────────────┤
│0.6│Extra Virgin  │
├───┼──────────────┤
│3.2│Ordinary      │
└───┴──────────────┘
```

Example (Data Consolidation by Interval)

`x` represents some data sampled in chronological order at timestamps `t`.

```
      ⍴x
200000
      x
3984300 2020650 819000 1677100 3959200 2177250 3431800 ...
```

```
      ⍴t
200000 3
```

```
      (10↑t) (¯10↑t)
┌─────┬────────┐
│0 0 0│23 59 54│
│0 0 0│23 59 55│
│0 0 0│23 59 56│
│0 0 0│23 59 56│
│0 0 0│23 59 58│
│0 0 2│23 59 58│
│0 0 3│23 59 59│
│0 0 3│23 59 59│
│0 0 4│23 59 59│
│0 0 5│23 59 59│
└─────┴────────┘
```

`u` represents timestamps for 5-minute intervals:

```
      ⍴u
288 3
      (10↑u) (¯10↑u)
┌──────┬────────┐
│0  0 0│23 10 0│
│0  5 0│23 15 0│
│0 10 0│23 20 0│
│0 15 0│23 25 0│
│0 20 0│23 30 0│
│0 25 0│23 35 0│
│0 30 0│23 40 0│
│0 35 0│23 45 0│
│0 40 0│23 50 0│
│0 45 0│23 55 0│
└──────┴────────┘
```

Therefore, the expression `(u⌷t){+/⍵}⌸x` summarises `x` in 5-minute intervals.

```
      u ⍳ t
1 1 1 1 1 1 1 1 1 1 ... 288 288 288 288 288 288

      (u⍳t) {+/⍵}⌸ x
1339083050 1365108650 1541944750 1393476000 1454347100 ...

      (u⍳t) {(⍺⌷u),+/⍵}⌸ x
 0  0 0 1339083050
 0  5 0 1365108650
 0 10 0 1541944750
 0 15 0 1393476000
   ...
23 45 0 1388823150
23 50 0 1453472350
23 55 0 1492078850
```

Higher-Rank Left Argument

If `X` is a higher rank array, the function compares sub-arrays in `Y` with the major cells of `X`, where a major cell is a sub-array on the leading dimension of `X` with shape `1↓⍴X`. In this case, the shape of the result `R` is `(1-⍴⍴X)↓⍴Y`.

EXAMPLE

```
      x ← ↑ 'Fi' 'Jay' 'John' 'Morten' 'Roger'
      x
Fi
Jay
John
Morten
Roger
      ⍴x
5 6
```

```
      y ← x ⍪ ↑ 'JD' 'Jd' 'Geoff' 'Alpha' 'Omega' 'Zeus  '
      y
Fi
Jay
John
Morten
Roger
JD
Jd
Geoff
Alpha
Omega
Zeus
```

```
      x ⍳ y
1 2 3 4 5 1 2 1 0 4 5
      y ,⍪ x⍳y
Fi      1
Jay     2
John    3
Morten  4
Roger   5
JD      1
Jd      2
Geoff   1
Alpha   0
Omega   4
Zeus    5
```

Further Example

```
      ⍴x
5 6
      ⍴y
3 3 6
      x
Fi
Jay
John
Morten
Roger
      y
Fi
Jay
John

Morten
Roger
JD

Jd
Geoff
Alpha
      x⍪y
1 2 3
4 5 1
2 1 0
```

Nested Array Example

A card-player likes to sort a hand into suits spades, hearts, diamond, clubs (fortunately alphabetic) and high-to-low within each suit.

```
      suits←'Clubs' 'Diamonds' 'Hearts' 'Spades'
      pack←,(⊂¨suits)∘.,1↓14 ⍝ 11=Jack ... 14=Ace
      hand←↑(,pack)[7?52]
      hand←hand[⍒hand;]
      hand
┌────────┬──┐
│Spades  │12│
├────────┼──┤
│Hearts  │12│
├────────┼──┤
│Hearts  │7 │
├────────┼──┤
│Hearts  │2 │
├────────┼──┤
│Diamonds│11│
├────────┼──┤
│Diamonds│9 │
├────────┼──┤
│Clubs   │8 │
└────────┴──┘
```

Another card, the 10 of diamonds is dealt. Where must it go in the hand ?

```
      (⊖hand)⍸'Diamonds' 10 ⍝ left arg must be sorted up
2
      (¯2↓hand)⍪'Diamonds' 10⍪¯2↑hand
┌────────┬──┐
│Spades  │12│
├────────┼──┤
│Hearts  │12│
├────────┼──┤
│Hearts  │7 │
├────────┼──┤
│Hearts  │2 │
├────────┼──┤
│Diamonds│11│
├────────┼──┤
│Diamonds│10│
├────────┼──┤
│Diamonds│9 │
├────────┼──┤
│Clubs   │8 │
└────────┴──┘
```

Note that if `(∧/Y∈X)` and `X` is sorted and `⎕CT=0` ,then `x⍸y` is the same as `x⍳y`.

3.4.40 Left R←X⊣Y

X and Y may be any arrays.

The result R is the left argument X.

Example

```
      42⊣'abc' 1 2 3
42
```

Note that when ⊣ is applied using reduction, the derived function selects the first sub-array of the array along the specified dimension. This is implemented as an idiom.

Examples

```
      ⊣/1 2 3
1

      mat←↑'scent' 'canoe' 'arson' 'rouse' 'fleet'
      ⊣⌿mat  ⍝ first row
scent
      ⊣/mat  ⍝ first column
scarf
```

```
      ⊣/[2]2 3 4⍴⍳24 ⍝ first row from each plane
 1  2  3  4
13 14 15 16
```

Similarly, with expansion:

```
      ⊣\mat
sssss
ccccc
aaaaa
rrrrr
fffff
      ⊣⍀mat
scent
scent
scent
scent
scent
```

3.4.41 Less Than R←X<Y

Y may be any numeric array. X may be any numeric array. R is Boolean. R is 1 if X is less than Y and X=Y is 0. Otherwise R is 0.

⎕CT and ⎕⎕DCT are implicit arguments of Less Than.

Examples

```
      (2 4) (6 8 10) < 6
 1 1  0 0 0

      ⎕CT←1E¯10

      1 0.99999999999 0.9999999999<1
0 0 1
```

3.4.42 Less Than Or Equal To R←X≤Y

Y may be any numeric array. X may be any numeric array. R is Boolean. R is 1 if X is less than Y or X=Y. Otherwise R is 0.

⎕CT and ⎕⎕DCT are implicit arguments of Less Than Or Equal To.

Examples

```
      2 4 6 8 10 ≤ 6
1 1 1 0 0

      ⎕CT←1E¯10

      1  1.00000000001 1.00000001 ≤ 1
1 1 0
```

3.4.43 Logarithm R←X⍟Y

X and Y must be numeric arrays. X cannot be 1 unless Y is also 1. R is the base X logarithm of Y.

Note that Logarithm (dyadic ⍟) is defined in terms of Natural Logarithm (monadic ⍟) as:

```
      X⍟Y←→(⍟Y)÷⍟X
```

EXAMPLES

```
      10⊛100 2
2 0.3010299957

      2 10⊛0J1 1J2
0J2.266180071 0.3494850022J0.4808285788

      1 ⊛ 1
1
      2 ⊛ 1
0
```

3.4.44 Lowest Common Multiple/AND R←X∧Y

Case 1: Lowest Common Multiple – either or both X and Y are numeric (non-Boolean)

R is the lowest common multiple of X and Y. Note that in this case, ⎕CT and ⎕DCT are implicit arguments.

EXAMPLE

```
      15 1 2 7 ∧ 35 1 4 0
105 1 4 0

      2 3 4 ∧ 0j1 1j2 2j3
0J2 3J6 8J12

      2j2 2j4 ∧ 5j5 4j4
10J10 ¯4J12
```

Case 2: AND – X and Y are Boolean

R is Boolean is determined as follows:

```
             X   Y     R

             0   0     0
             0   1     0
             1   0     0
             1   1     1
```

Note that the ASCII caret (^) will also be interpreted as an APL **And** (∧).

EXAMPLE

```
      0 1 0 1 ^ 0 0 1 1
0 0 0 1
```

3.4.45 Magnitude R←|Y

Y may be any numeric array. R is numeric composed of the absolute (unsigned) values of Y.

Note that the magnitude of a complex number z = a + b i (where a and b are real numbers, and i is the imaginary unit) is defined to be:

| z | = a 2 + b 2

EXAMPLES

```
      |2 ¯3.4 0 ¯2.7
2 3.4 0 2.7

      |3j4
5
```

⎕IO is an implicit argument of magnitude.

3.4.46 Match R←X≡Y

Y may be any array. X may be any array. R is a simple Boolean scalar. If X is identical to Y, then R is 1. Otherwise R is 0.

Non-empty arrays are identical if they have the same structure and the same values in all corresponding locations. Empty arrays are identical if they have the same shape and the same prototype (disclosed nested structure).

⎕CT and ⎕DCT are implicit arguments of Match.

Examples

```
      ⍬≡⍳0
1
      ''≡⍳0
0
      A
THIS
WORD

      A≡2 4⍴'THISWORD'
1
      A≡⍳10
0
      +B←A A
 THIS  THIS
 WORD  WORD

      A≡⊃B
1

      (0⍴A)≡0⍴B
0

      ' '=⊃0⍴B
1 1 1 1
1 1 1 1

      ' '=⊃0⍴A
1
```

3.4.47 Materialise R←⌷Y

If Y is a ref to an instance of a Dyalog Class with a Default property:

- If the Default property is Simple, ⌷ returns its value.
- If the Default property is Numbered, ⌷ returns an array whose shape is determined by the PropertyShape function and whose items are constructed by calling the ProperyGet function repeatedly with the corresponding set of indices. However, see **Performance Considerations**.
- If the Default property is Keyed, NONCE ERROR is reported, because in this case APL has no way to determine the list of all the elements.

If Y is a ref to an instance of a COM or .NET Collection object, ⍞ returns a vector containing all its items.

Otherwise, if Y is an array, Y is returned.

Example

The following example uses the sample *Programming: Component File Class Example*.

```
      )LOAD ComponentFile
...\Samples\OO4APL\ComponentFile.dws saved ...
      F1←⍞NEW ComponentFile 'test1'

      F1
#.[ComponentFile]

      F1.Append¨(⍳5)×⊂⍳4
1 2 3 4 5
      ⍞F1
┌─────────┬─────────┬──────────┬──────────┬──────────┐
│1 2 3 4│2 4 6 8│3 6 9 12│4 8 12 16│5 10 15 20│
└─────────┴─────────┴──────────┴──────────┴──────────┘
```

The following example shows how ⍞obtains the items in an Excel Sheets collection .

```
      wb←ex.Workbooks.Open⊂'budget.xls'
      wb.Sheets
#.[OLEClient].[Workbooks].[_Workbook].[Sheets]
      ⍞wb.Sheets
 #.[_Worksheet]  #.[_Worksheet]  #.[_Worksheet]
      (⍞wb.Sheets).Name
┌────┬────┬────┐
│2012│2011│2010│
└────┴────┴────┘
```

Performance Considerations

Note that the *values* of the index set are obtained or assigned by calls to the corresponding PropertyGet and PropertySet functions. Furthermore, if there is a sequence of primitive functions to the left of the Index function, that operate on the index set itself (functions such as dyadic ⍴ , ↑ , ↓ , ⊃) as opposed to functions that operate on the *values* of the index set (functions such as + , ⌈ , ⌊ , ⍴¨), calls to the PropertyGet and PropertySet functions are deferred until the required index set has been completely determined. The full set of functions that cause deferral of calls to the

PropertyGet and PropertySet functions is the same as the set of functions that applies to selective specification.

If for example, `CompFile` is an Instance of the *Programming: Component File Class Example*:

```
      1↑⌽⍞CompFile
```

would only call the PropertyGet function (for `CompFile`) once, to get the value of the last element.

Note that similarly, the expression

```
     10000⍴⍞CompFile
```

would call the PropertyGet function 10000 times, on repeated indices if `CompFile` has less than 10000 elements. The deferral of access function calls is intended to be an optimisation, but can have the opposite effect. You can avoid unnecessary repetitive calls by assigning the result of `⍞` to a temporary variable.

3.4.48 Matrix Divide R←X⌹Y

`Y` must be a simple numeric array of rank 2 or less. `X` must be a simple numeric array of rank 2 or less. `Y` must be non-singular. A scalar argument is treated as a matrix with one-element. If `Y` is a vector, it is treated as a single column matrix. If `X` is a vector, it is treated as a single column matrix. The number of rows in `X` and `Y` must be the same. `Y` must have at least the same number of rows as columns.

`R` is the result of matrix division of `X` by `Y`. That is, the matrix product `Y+.×R` is `X`.

`R` is determined such that `(X-Y+.×R)*2` is minimised.

The shape of `R` is `(1↓⍴Y),1↓⍴X`.

Examples

```
      ⎕PP←5

      B
3 1 4
1 5 9
2 6 5

      35 89 79 ⌹ B
2.1444 8.2111 5.0889

      A
35 36
89 88
79 75

      A ⌹ B
2.1444 2.1889
8.2111 7.1222
5.0889 5.5778
```

If there are more rows than columns in the right argument, the least squares solution results. In the following example, the constants a and b which provide the best fit for the set of equations represented by P = a + bQ are determined:

```
      Q
1 1
1 2
1 3
1 4
1 5
1 6

      P
12.03 8.78 6.01 3.75 ¯0.31 ¯2.79

      P⌹Q
14.941 ¯2.9609
```

Example: linear regression on complex numbers

```
      x←j×¯50+?2 13 4⍴100
      y←(x+.×3 4 5 6) + j×0.0001×¯50+?2 13⍴100
      ⍴x
13 4
      ⍴y
13
      y ⌹ x
3J0.000011066 4J¯0.000018499 5J0.000005745 6J0.000050328
      ⍝ that is, y⌹x recovered the coefficients 3 4 5 6
```

Additional Information

```
      x⌹y ↔ (⌹(⍉⌽y)+.×y)+.×(⍉⌽y)+.×x
```

(Use `+⍉⌽` instead of `⍉⌽` for complex `y`.)

This equivalence, familiar to mathematicians and statisticians, explains

- the conformability requirements for ⌹
- how to compute the result for tall matrices from the better known square matrix case

3.4.49 Matrix Inverse R←⌹Y

`Y` must be a simple array of rank 2 or less. `Y` must be non-singular. If `Y` is a scalar, it is treated as a one-element matrix. If `Y` is a vector, it is treated as a single-column matrix. `Y` must have at least the same number of rows as columns.

`R` is the inverse of `Y` if `Y` is a square matrix, or the left inverse of `Y` if `Y` is not a square matrix. That is, `R+.×Y` is an identity matrix.

The shape of `R` is `⌽⍴Y`.

Examples

```
      M
2 ¯3
4 10

      +A←⌹M
 0.3125 0.09375
¯0.125  0.0625
```

Within calculation accuracy, `A+.×M` is the identity matrix.

```
      A+.×M
1 0
0 1



      j←{α←0 ⋄ α+0J1×ω}
      x←j⍣¯50+?2 5 5⍴100
      x
¯37J¯41  25J015  ¯5J¯09   3J020 ¯29J041
¯46J026  17J¯24  17J¯46  43J023 ¯12J¯18
  1J013  33J025 ¯47J049 ¯45J¯14   2J¯26
 17J048 ¯50J022 ¯12J025 ¯44J015  ¯9J¯43
 18J013   8J038  43J¯23  34J¯07   2J026
      ⍴x
5 5
      id←{∘.=⍨⍳ω}  ⍝ identity matrix of order ω
      ⌈/,| (id 1↑⍴x) - x+.×⌹x
3.66384E¯16
```

3.4.50 Maximum R←X⌈Y

`Y` may be any numeric array. `X` may be any numeric array. `R` is numeric. `R` is the larger of the numbers `X` and `Y`.

Example

```
      ¯2.01 0.1 15.3 ⌈ ¯3.2 ¯1.1 22.7
¯2.01 0.1 22.7
```

3.4.51 Membership R←X∈Y

Y may be any array. X may be any array. R is Boolean. An element of R is 1 if the corresponding element of X can be found in Y.

An element of X is considered identical to an element in Y if X≡Y returns 1 for those elements.

⎕CT and ⎕DCT are implicit arguments of Membership.

Examples

```
      'THIS NOUN' ∈ 'THAT WORD'
1 1 0 0 1 0 1 0 0

      'CAT' 'DOG' 'MOUSE' ∈ 'CAT' 'FOX' 'DOG' 'LLAMA'
1 1 0
```

For performance information, see *Programming: Search Functions And Hash*.

3.4.52 Minimum R←X⌊Y

Y may be any numeric array. X may be any numeric array. R is numeric. R is the smaller of X and Y.

Example

```
      ¯2.1 0.1 15.3 ⌊ ¯3.2 1 22
¯3.2 0.1 15.3
```

3.4.53 Minus R←X-Y

Y may be any numeric array. X may be any numeric array. R is numeric. The value of R is the difference between X and Y.

This function is also known as Subtract.

EXAMPLE

```
      3 ¯2 4 0 - 2 1 ¯2 4
1 ¯3 6 ¯4

      2j3-.3j5  ⍝ (a+bi)-(c+di) = (a-c)+(b-d)i
1.7J¯2
```

3.4.54 Mix (⎕ML) R←↑[K]Y or R←⊃[K]Y

The symbol chosen to represent Mix depends on the current Migration Level.

If ⎕ML<2, Mix is represented by the symbol: ↑.

If ⎕ML≥2, Mix is represented by the symbol: ⊃.

Y may be any array whose items may be uniform in rank and shape, or differ in rank and shape. If the items of Y are non-uniform, they are extended prior to the application of the function as follows:

1. If the items of Y have different ranks, each item is extended in rank to that of the greatest rank by padding its shape with leading 1s.
2. If the items of Y have different shapes, each is padded with the corresponding prototype to a shape that represents the greatest length along each axis of all items in Y.

For the purposes of the following narrative, y represents the virtual item in Y with the greatest rank and shape, with which all other items are extended to conform.

R is an array composed from the items of Y assembled into a higher-rank array with one less level of nesting. ⍴R will be some permutation of (⍴Y),⍴y.

K is an optional axis specification whose value(s) indicate where in the result the axes of y appear. There are three cases:

1. K may be a scalar or 1-element vector whose value is a fractional number indicating the two axes of Y between which new axes are to be inserted for y. The shape of R is the shape of Y with the shape ⍴y inserted between the ⌊Kth and the ⌈Kth axes of Y
2. K may be a scalar or 1-element vector integer whose value specifies the position of the first axis of y in the result. This case is identical to the fractional case where K (in this case) is ⌈K (in the fractional case).

3. `⍺` may be an integer vector, with the same length as `⍴⍵`, each element of which specifies the position in the result of the corresponding axis of the `⍵`.

If `⍺` is absent, the axes of `⍵` appear as the last axes of the result.

Simple Vector Examples

In this example, the shape of `⍹` is 3, and the shape of `⍵` is 2. So the shape of the result will be a permutation of 2 and 3, that is, in this simple example, either `(2 3)` or `(3 2)`.

If `⍺` is omitted, the shape of the result is `(⍴⍹),⍴⍵`.

```
      ↑(1 2)(3 4)(5 6)
1 2
3 4
5 6
```

If `⍺` is between 0 and 1, the shape of the result is `(⍴⍵),⍴⍹` because `(⍴⍵)` is inserted between the 0th and the 1st axis of the result, that is, at the beginning.

```
      ↑[.5](1 2.htm)(3 4)(5 6)
1 3 5
2 4 6
```

If `⍺` is between 1 and 2, the shape of the result is `(⍴⍹),⍴⍵` because `(⍴⍵)` is inserted between the 1st and 2nd axis of the result, that is, at the end. This is the same as the case when `⍺` is omitted.

```
      ↑[1.5](1 2.htm)(3 4)(5 6)
1 2
3 4
5 6
```

An integer `⍺` may be used instead. If `⎕ML≥2`, `⊃` is used instead of `↑`).

```
      ⎕ML←3
      ⊃(1 2)(3 4)(5 6)
1 2
3 4
5 6

      ⊃[1](1 2.htm)(3 4)(5 6)
1 3 5
2 4 6
      ⊃[2](1 2.htm)(3 4)(5 6)
1 2
3 4
5 6
```

Shape Extension

If the items of `Y` are unequal in shape, the shorter ones are extended:

```
      ⎕ML←3
      ⊃(1)(3 4)(5)
1 0
3 4
5 0

       ⊃[1](1.htm)(3 4)(5)
1 3 5
0 4 0
```

More Simple Vector Examples

```
      ]box on
Was OFF
       'Andy' 'Geoff' 'Pauline'
┌─────┬──────┬────────┐
│Andy │Geoff │Pauline │
└─────┴──────┴────────┘
      ↑'Andy' 'Geoff' 'Pauline'
Andy
Geoff
Pauline

        ⎕ML←3
        ⊃('andy' 19)('geoff' 37)('pauline' 21)
┌────────┬──┐
│andy    │19│
├────────┼──┤
│geoff   │37│
├────────┼──┤
│pauline │21│
└────────┴──┘
        ⊃[1]('andy' 19.htm)('geoff' 37)('pauline' 21)
┌─────┬──────┬────────┐
│andy │geoff │pauline │
├─────┼──────┼────────┤
│19   │37    │21      │
└─────┴──────┴────────┘
        ⊃('andy' 19)('geoff' 37)(⊂'pauline')
┌────────┬────────┐
│andy    │19      │
├────────┼────────┤
│geoff   │37      │
├────────┼────────┤
│pauline │        │
└────────┴────────┘
```

Notice that in the last statement, the shape of the third item was extended by catenating it with its prototype.

Example (Matrix of Matrices)

In the following examples, `Y` is a matrix of shape `(5 4)` and each item of `Y` (`y`) is a matrix of shape `(3 2)`. The shape of the result will be some permutation of `(5 4 3 2)`.

```
      Y←5 4⍴(⍳20)×⊂3 2⍴1
      Y
┌─────┬─────┬─────┬─────┐
│1 1  │2 2  │3 3  │4 4  │
│1 1  │2 2  │3 3  │4 4  │
│1 1  │2 2  │3 3  │4 4  │
├─────┼─────┼─────┼─────┤
│5 5  │6 6  │7 7  │8 8  │
│5 5  │6 6  │7 7  │8 8  │
│5 5  │6 6  │7 7  │8 8  │
├─────┼─────┼─────┼─────┤
│9 9  │10 10│11 11│12 12│
│9 9  │10 10│11 11│12 12│
│9 9  │10 10│11 11│12 12│
├─────┼─────┼─────┼─────┤
│13 13│14 14│15 15│16 16│
│13 13│14 14│15 15│16 16│
│13 13│14 14│15 15│16 16│
├─────┼─────┼─────┼─────┤
│17 17│18 18│19 19│20 20│
│17 17│18 18│19 19│20 20│
│17 17│18 18│19 19│20 20│
└─────┴─────┴─────┴─────┘
```

By default, the axes of `y` appear in the last position in the shape of the result, but this position is altered by specifying the axis `K`. Notice where the `(3 2)` appears in the following results:

```
      ⍴⊃Y
5 4 3 2
      ⍴⊃[1]Y
3 2 5 4
      ⍴⊃[2]Y
5 3 2 4
      ⍴⊃[3]Y
5 4 3 2
      ⍴⊃[4]Y
INDEX ERROR
      ⍴⊃[4]Y
     ^
```

Note that `⊃[4]Y` generates an `INDEX ERROR` because 4 is greater than the length of the result.

Example (Vector K)

The axes of `Y` do not have to be contiguous in the shape of the result. By specifying a vector `K`, they can be distributed. Notice where the `3` and the `2` appear in the following results:

```
      ⍴⊃[1 3]Y
3 5 2 4
      ⍴⊃[1 4]Y
3 5 4 2
      ⍴⊃[2 4]Y
5 3 4 2
      ⍴⊃[4 2]Y
5 2 4 3
```

Rank Extension

If the items of `Y` are unequal in rank, the lower rank items are extended in rank by prefixing their shapes with 1s. Each additional 1 may then be increased to match the maximum shape of the other items along that axis.

```
      ⎕ML←3
      Y←(1)(2 3 4 5)(2 3⍴10×⍳8)
      Y
┌─┬─────────┬──────────┐
│1│2 3 4 5  │10 20 30  │
│ │         │40 50 60  │
└─┴─────────┴──────────┘
       ⍴⊃Y
3 2 4
       ⊃Y
 1  0  0 0
 0  0  0 0

 2  3  4 5
 0  0  0 0

10 20 30 0
40 50 60 0
```

In the above example, the first item (1) becomes (`1 1⍴1`) to conform with the 3rd item which is rank 2. It is then extended in shape to become `(2 4↑1 1⍴1)` to conform with the 2-row 3rd item, and 4-column 2nd item.. Likewise, the 2nd item becomes a 2-row matrix, and the 3rd item gains another column.

3.4.55 NAND R←X⍲Y

`Y` must be a Boolean array. `X` must be a Boolean array. `R` is Boolean. The value of `R` is the truth value of the proposition "not both `X` and `Y`", and is determined as follows:

```
      X   Y     R

      0   0     1
      0   1     1
      1   0     1
      1   1     0
```

Example

```
      (0 1)(1 0) ⍲ (0 0)(1 1)
 1 1  0 1
```

3.4.56 Natural Logarithm R←⍟Y

Y must be a numeric array. R is numeric. R is the natural (or Napierian) logarithm of Y whose base is the mathematical constant *e=2.71828…*.

EXAMPLE

```
      ⍟1 2
0 0.6931471806

      ⍟2 2⍴0j1 1j2 2j3 3j4
0.000000000J1.570796327  0.8047189562J1.107148718
1.282474679J0.9827937232 1.609437912J0.927295218
```

3.4.57 Negate R←-Y

Y may be any numeric array. R is numeric and is the negative value of Y. For complex numbers both the real and imaginary parts are negated.

EXAMPLE

```
      -4 2 0 ¯3 ¯5
¯4 ¯2 0 3 5

      -1j2 ¯2J3 4J¯5
¯1J¯2 2J¯3 ¯4J5
```

3.4.58 Nest R←⊆Y

Classic Edition: the symbol ⊆ (Left Shoe Underbar) is not available in Classic Edition, and Nest is instead represented by □U2286.

Y may be any array.

If Y is simple, R is a scalar array whose item is the array Y. If Y is a simple scalar or is already nested, R is Y unchanged.

Examples

```
      ⊆1 2 3
┌─────┐
│1 2 3│
└─────┘
      ⊆ 1 (1 2 3)
┌─┬─────┐
│1│1 2 3│
└─┴─────┘
      ⊆'Dyalog'
┌──────┐
│Dyalog│
└──────┘
      ⊆'Dyalog' 'APL'
┌──────┬───┐
│Dyalog│APL│
└──────┴───┘
```

3.4.59 NOR R←X⍱Y

Y must be a Boolean array. X must be a Boolean array. R is Boolean. The value of R is the truth value of the proposition "neither X nor Y", and is determined as follows:

```
              X   Y     R

              0   0     1
              0   1     0
              1   0     0
              1   1     0
```

Example

```
      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0
```

3.4.60 NOT R←~Y

Y must be a Boolean array. R is Boolean. The value of R is 0 if Y is 1, and R is 1 if Y is 0.

EXAMPLE

```
      ~0 1
1 0
```

3.4.61 Not Equal To R←X≠Y

Y may be any array. X may be any array. R is Boolean. R is 0 if X=Y. Otherwise R is 1.

For Boolean X and Y, the value of R is the exclusive or (XOR) result, determined as follows:

X	Y	R
0	0	0
0	1	1
1	0	1
1	1	0

⎕CT and ⎕DCT are implicit arguments of Not Equal To.

EXAMPLES

```
      1 2 3 ≠ 1.1 2 3
1 0 0

      ⎕CT←1E¯10

      1≠1 1.00000000001 1.0000001
0 0 1

      1 2 3 ≠'CAT'
1 1 1
```

3.4.62 Not Match R←X≢Y

Y may be any array. X may be any array. R is a simple Boolean scalar. If X is identical to Y, then R is 0. Otherwise R is 1.

Non-empty arrays are identical if they have the same structure and the same values in all corresponding locations. Empty arrays are identical if they have the same shape and the same prototype (disclosed nested structure).

⎕CT and ⎕DCT are implicit arguments of Not Match.

EXAMPLES

```
      0≠⍳0
0
      ''≠⍳0
1
```

```
      ⊢A←⊂(⍳3) 'ABC'
  1 2 3  ABC
```

```
      A≠(⍳3)'ABC'
1
      A≠⊂(⍳3) 'ABC'
0
      0≠0⍴A
1
      (1↑0⍴A)≠⊂(0 0 0) '   '
1
```

3.4.63 Greatest Common Divisor/OR R←X∨Y

Case 1: Greatest Common Divisor – X and Y are numeric (non-Boolean)

R is the Greatest Common Divisor of X and Y. Note that in this case, ⎕CT and ⎕DCT are implicit arguments.

EXAMPLES

```
      15 1 2 7 ∨ 35 1 4 0
5 1 2 7

      rational←{↑⍵ 1÷⊂1∨⍵} ⍝ rational (⎕CT) approximation
                           ⍝ to floating array.
      rational 0.4321 0.1234 6.66, ÷1 2 3
 4321  617 333 1 1 1
10000 5000  50 1 2 3
```

Case 2: OR – X and Y are Boolean

R is Boolean and is determined as follows:

```
      X   Y     R

      0   0     0
      0   1     1
      1   0     1
      1   1     1
```

EXAMPLE

```
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1
```

3.4.64 Partition R←X⊆[K]Y

> **Classic Edition**
>
> The symbol ⊆ (Left Shoe Underbar) is not available in Classic Edition, and Partition is instead represented by ⍷U2286.

Y may be any non-scalar array.

X must be a simple scalar or vector of non-negative integers.

The axis specification is optional. If present, it must be a simple integer scalar or one element array representing an axis of Y. If absent, the last axis is implied.

R is an array of the elements of Y partitioned according to X.

A new partition is started in the result whenever the corresponding element in X is greater than the previous one. Items in Y corresponding to 0s in X are not included in the result.

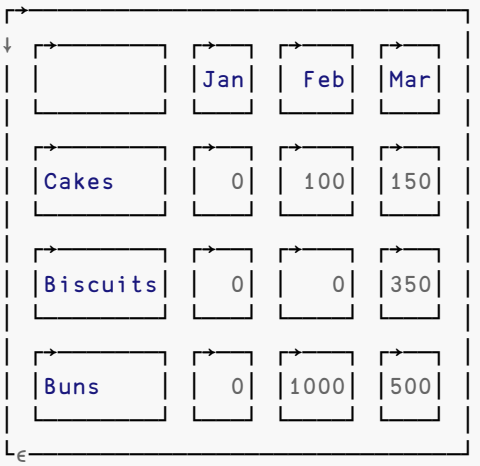
Note that if ⎕ML≥3, the symbol ⊂ means the same as ⊆.

EXAMPLES

```
      ⎕ML←3

      ]display 1 1 1 2 2 3 3 3⊆'NOWISTHE'
┌→──────────────┐
│ ┌→──┐ ┌→┐ ┌→──┐ │
│ │NOW│ │IS│ │THE│ │
│ └───┘ └──┘ └───┘ │
└∊──────────────┘

      ]display 1 1 1 0 0 3 3 3⊆'NOWISTHE'
┌→──────────┐
│ ┌→──┐ ┌→──┐ │
│ │NOW│ │THE│ │
│ └───┘ └───┘ │
└∊──────────┘

      TEXT←'   NOW     IS      THE      TIME    '
      ]display (' '≠TEXT)⊆TEXT
┌→──────────────────┐
│ ┌→──┐ ┌→┐ ┌→──┐ ┌→───┐ │
│ │NOW│ │IS│ │THE│ │TIME│ │
│ └───┘ └──┘ └───┘ └────┘ │
└∊──────────────────┘
```

```
      ]display CMAT←⎕FMT(' ',ROWS),COLS⍪NMAT
┌→──────────────────┐
↓           Jan  Feb  Mar │
│ Cakes         0  100  150 │
│ Biscuits      0    0  350 │
│ Buns          0 1000  500 │
└───────────────────┘
```

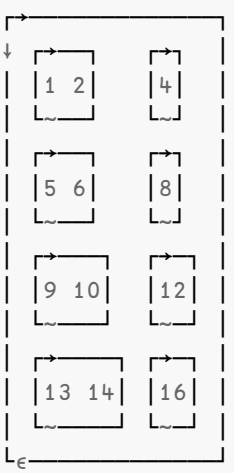
```
      ]display (∨≠' '≠CMAT)⊆CMAT   ⍝ Split at blank cols.
```




```
      ]display N←4 4⍴⍳16
```

```
┌→──────────┐
↓ 1  2  3  4│
│ 5  6  7  8│
│ 9 10 11 12│
│13 14 15 16│
└~──────────┘
```

```
      ]display 1 1 0 1⊆N
```



```
      ]display 1 1 0 1⊆[1]N
┌→──────────────────────────┐
↓ ┌→┐ ┌→┐ ┌→┐ ┌→┐          │
│ │1 5│ │2 6│ │3 7│ │4 8│  │
│ └~┘ └~┘ └~┘ └~┘          │
│ ┌→┐   ┌→┐   ┌→┐   ┌→┐   │
│ │13│  │14│  │15│  │16│   │
│ └~┘   └~┘   └~┘   └~┘   │
└∈──────────────────────────┘
```

3.4.65 Partitioned Enclose (⍁ML<3) R←X⊂[K]Y

Y may be any array. X must be a simple integer scalar or vector. If X is a scalar it is extended to (≢Y)⍴X.

The axis specification is optional. If present, it must be a simple integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied.

R is a vector of items selected from Y by inserting 0 or more dividers, specified by X, between its major cells.

Each element of X specifies the number of dividers to insert before the corresponding major cell of Y. The maximum length of X is 1+≢Y, when the last element of X specifies the number of trailing dividers. Note that major cells of Y that precede the first divider (identified by the first non-zero element of X) are excluded from the result.

The length of R is +/X (after possible extension).

Examples

```
      0 0 1 0 0 1 0⊂'abcdefg'
┌───┬──┐
│cde│fg│
└───┴──┘
      2 0 1 3 0 2 0 1⊂'abcdefg'
┌──┬─┬┬┬──┬┬──┬┐
│ab│c│││de│││fg│
└──┴─┴┴┴──┴┴──┴┘
      0 2 0 1⊂'abcdefg'
┌──┬────┐
│bc│defg│
└──┴────┘
```

The above examples may be explained pictorially by the diagram below.

Further Examples

```
      1 0 1⊂[1]3 4⍴⍳12
┌─────────┬─────────┐
│1 2 3 4│9 10 11 12│
│5 6 7 8│          │
└─────────┴─────────┘
      1 0 0 1⊂[2]3 4⍴⍳12
┌─────────┬──┐
│1  2  3│ 4│
│5  6  7│ 8│
│9 10 11│12│
└─────────┴──┘
```

3.4.66 Pi Times R←○Y

Y may be any numeric array. R is numeric. The value of R is the product of the mathematical constant *π=3.14159…* (Pi), and Y.

EXAMPLE

```
      ○0.5 1 2
1.570796327 3.141592654 6.283185307

      ○0J1
0J3.141592654

      *○0J1 ⍝ Euler
¯1
```

3.4.67 Pick R←X⊃Y

Y may be any array.

X is a scalar or vector of indices of Y.

R is an item selected from the structure of Y according to X.

Elements of X select from successively deeper levels in the structure of Y. The items of X are simple integer scalars or vectors which identify a set of indices, one per axis at the particular level of nesting of Y in row-major order. Simple scalar items in Y may be picked by empty vector items in X to any arbitrary depth.

⎕IO is an implicit argument of Pick.

EXAMPLES

```
      G←('ABC' 1)('DEF' 2)('GHI' 3)('JKL' 4)

      G←2 3⍴G,('MNO' 5)('PQR' 6)

      G
  ABC  1   DEF  2   GHI  3
  JKL  4   MNO  5   PQR  6
```

```
    ((⊂2 1),1)⊃G
JKL

      (⊂2 1)⊃G
 JKL  4

      ((2 1)1 2)⊃G
K

      (5⍴⊂⍳0)⊃10
10
```

3.4.68 Plus R←X+Y

Y must be numeric. X must be numeric. R is the arithmetic sum of X and Y. R is numeric. This function is also known as Add.

Examples

```
      1 2 + 3 4
4 6

      1 2 + 3,⊂4 5
4  6 7

      1J1 2J2 + 3J3
4J4 5J5

      ¯5 + 4J4 5J5
¯1J4 0J5
```

3.4.69 Power R←X*Y

Y must be a numeric array. X must be a numeric array. R is numeric. The value of R is X raised to the power of Y.

If Y is zero, R is defined to be 1.

If X is zero, Y must be non-negative.

In general, X*Y is defined as *Y×⊗X. If X is negative, the result R is likely to be complex.

Examples

```
      2*2 ¯2
4 0.25

      9 64*0.5
3 8

      ¯27*3 2 1.2 .5
¯19683 729 ¯42.22738244J¯30.67998919 0J5.196152423

      ¯8*÷3
1J1.732050808
```

3.4.70 Ravel

Ravel	R←,Y

Y may be any array. R is a vector of the elements of Y taken in row-major order.

EXAMPLES

```
      M
1 2 3
4 5 6

      ,M
1 2 3 4 5 6

      A
ABC
DEF
GHI
JKL
      ,A
ABCDEFGHIJKL

      ⍴,10
1
```

See also: *Section 3.4.70.2*.

Ravel with Axes	R←,[K]Y

Y may be any array.

K is either:

- A simple fractional scalar adjacent to an axis of Y, or
- A simple integer scalar or vector of axes of Y, or
- An empty vector

Ravel with axis can be used with selective specification.

R depends on the case of K above.

If `K` is a fraction, the result `R` is an array of the same shape as `Y`, but with a new axis of length 1 inserted at the `K`th position.

```
      ⍴⍴R ↔ 1+⍴⍴Y
      ⍴R  ↔ (1,⍴Y)[⍋⍋K,⍳⍴⍴Y]
```

Examples

```
      ,[0.5]'ABC'
ABC
      ⍴,[0.5]'ABC'
1 3
      ,[1.5]'ABC'
A
B
C
      ⍴,[1.5]'ABC'
3 1

      MAT←3 4⍴⍳12
      ⍴,[0.5]MAT
1 3 4
      ⍴,[1.5]MAT
3 1 4
      ⍴,[2.5]MAT
3 4 1
```

If `K` is an integer scalar or vector of axes of `Y`, then:

- `K` must contain contiguous axes of `Y` in ascending order
- `R` contains the elements of `Y` ravelled along the indicated axes

Note that if `K` is a scalar or single element vector, `R ↔ Y`.

```
      ⍴⍴R ↔ 1+(⍴⍴Y)-⍴,K
```

Examples

```
      M
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      ⍴M
2 3 4
```

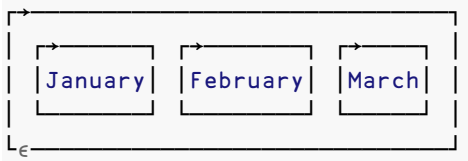
```
      ,[1 2]M
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
      ⍴,[1 2]M
6 4

      ,[2 3]M
 1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24

      ⍴,[2 3]M
2 12
```

If ⍵ is an empty vector a new last axis of length 1 is created.

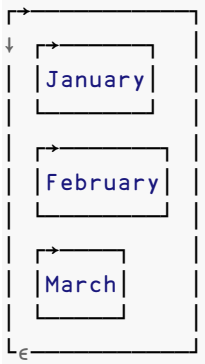
```
      ⍴R ↔ (⍴Y),1
```

Examples

```
      Q1←'January' 'February' 'March'
      ]display Q1
```




```
      ]display ,[⍳0]Q1
```



See also: *Section 3.4.70.1*.

3.4.71 Reciprocal R←÷Y

Y must be a numeric array. R is numeric. R is the reciprocal of Y; that is 1÷Y. If ⎕DIV=0, ÷0 results in a DOMAIN ERROR. If ⎕DIV=1, ÷0 returns 0.

⎕DIV is an implicit argument of Reciprocal.

EXAMPLES

```
      ÷4 2 5
0.25 0.5 0.2

      ÷0j1 0j¯1 2j2 4j4
0J¯1 0J1 0.25J¯0.25 0.125J¯0.125

      ⎕DIV←1
      ÷0 0.5
0 2
```

3.4.72 Replicate R←X/[K]Y

Y may be any array. X is a simple integer vector or scalar.

The axis specification is optional. If present, K must be a simple integer scalar or 1-element vector. The value of K must be an axis of Y. If absent, the last axis of Y is implied. The form R←X⌿Y implies the first axis of Y.

If Y has length 1 along the K^{th} (or implied) axis, it is extended along that axis to match the length of X. Otherwise, the length of X must be the length of the K^{th} (or implied) axis of Y. However, if X is a scalar or one-element vector, it will be extended to the length of the K^{th} axis.

R is composed from sub-arrays along the Kth axis of Y. If X[I] (an element of X) is positive, then the corresponding sub-array is replicated X[I] times. If X[I] is zero, then the corresponding sub-array of Y is excluded. If X[I] is negative, then the fill element of Y is replicated |X[I] times. Each of the (replicated) sub-arrays and fill items are joined along the Kth axis in the order of occurrence. The shape of R is the shape of Y except that the length of the (implied) Kth axis is +/|X (after possible extension).

This function is sometimes called Compress when X is Boolean.

Examples

```
      1 0 1 0 1/⍳5
1 3 5

      1 ¯2 3 ¯4 5/⍳5
1 0 0 3 3 3 0 0 0 0 5 5 5 5 5

      M
1 2 3
4 5 6

      2 0 1/M
1 1 3
4 4 6

      0 1⌿M
4 5 6

      0 1/[1]M
4 5 6
```

If `Y` is a singleton `(1=×/⍴,Y)` its value is notionally extended to the length of `X` along the specified axis.

```
      1 0 1/4
4 4
      1 0 1/,3
3 3
      1 0 1/1 1⍴5
5 5
```

3.4.73 Replicate First R←X⌿[K]Y

The form `R←X⌿Y` implies replication along the first axis of `Y`. See *Section 3.4.72*.

3.4.74 Reshape R←X⍴Y

`Y` may be any array. `X` must be a simple scalar or vector of non-negative integers. `R` is an array of shape `X` whose elements are taken from `Y` in row-major sequence and repeated cyclically if required. If `Y` is empty, `R` is composed of fill elements of `Y` (`⊂∊⊃Y` with `⎕ML←0`). If `X` contains at least one zero, then `R` is empty. If `X` is an empty vector, then `R` is scalar.

Examples

```
      2 3⍴⍳8
1 2 3
4 5 6

      2 3⍴⍳4
1 2 3
4 1 2

      2 3⍴⍳0
0 0 0
0 0 0
```

3.4.75 Residue R←X|Y

Y may be any numeric array. X may be any numeric array.

For positive arguments, R is the remainder when Y is divided by X. If X=0, R is Y.

For other argument values, R is given by the expression Y-X×⌊Y÷X+0=X. This expression also applies when X and/or Y are complex if the simple ⌊ is replaced by the CpxFloor function. See *Section 3.4.28*.

⎕CT and ⎕DCT are implicit arguments of Residue.

Examples

```
      3 3 ¯3 ¯3|¯5 5 ¯4 4
1 2 ¯1 ¯2

      0.5|3.12 ¯1 ¯0.6
0.12 0 0.4

      ¯1 0 1|¯5.25 0 2.41
¯0.25 0 0.41

      1j2|2j3 3j4 5j6
1J1 ¯1J1 0J1
```

Note that the ASCII Broken Bar (⎕UCS 166, U+00A6) is not interpreted as Residue.

3.4.76 Reverse R←⌽[K]Y

Y may be any array. The axis specification is optional. If present, K must be an integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis is implied. The form R←⊖Y implies the first axis.

R is the array Y reversed on the Kth or implied axis.

EXAMPLES

```
      ⌽1 2 3 4 5
5 4 3 2 1

      M
1 2 3
4 5 6
      ⌽M
3 2 1
6 5 4
      ⊖M
4 5 6
1 2 3
      ⌽[1]M
4 5 6
1 2 3
```

3.4.77 Reverse First R←⊖[K]Y

The form R←⊖Y implies reversal along the first axis. See *Section 3.4.76*.

3.4.78 Right R←X⊢Y

X and Y may be any arrays. The result R is the right argument Y.

EXAMPLES

```
      42 ⊢'abc' 1 2 3
 abc  1 2 3
```

```
      32+1.8×⊢0 100       ⍝ {32+1.8×⍵} 0 100
32 212
```

```
      (⊢÷+/) 4 3 0 1      ⍝ {⍵÷+/⍵} 4 3 0 1
0.5 0.375 0 0.125

      ↓⍤2⊢2 2 2 2⍴⎕A      ⍝ (↓⍤2)2 2 2 2⍴⎕A
  AB  CD    EF  GH
  IJ  KL    MN  OP
```

When ⊢ is applied using reduction, the derived function selects the last sub-array of the array along the specified dimension. This is implemented as an idiom.

EXAMPLES

```
      ⊢/1 2 3
3
      mat←↑'scent' 'canoe' 'arson' 'rouse' 'fleet'

      ⊢⌿mat  ⍝ last row
fleet
      ⊢/mat  ⍝ last column
tenet

      ⊢/[2]2 3 4⍴⍳24 ⍝ last row from each plane
 9 10 11 12
21 22 23 24
```

3.4.79 Roll R←?Y

Y may be any non-negative integer array. R has the same shape as Y at each depth.

For each positive element of Y the corresponding element of R is an integer, pseudo-randomly selected from the integers ⍳Y with each integer in this population having an equal chance of being selected.

For each zero element of Y, the corresponding element of R is a pseudo-random floating-point value in the range 0 - 1, but excluding 0 and 1, that is, (0<R[I]<1).

⎕IO and ⎕RL are implicit arguments of Roll. A side effect of Roll is to change the value of ⎕RL.

Note that different random number generators are available; see ⎕RL for more information.

Examples

```
      ?9 9 9
2 7 5
      ?3⍴0
0.3205466592 0.3772891947 0.5456603511
```

3.4.80 Rotate R←X⌽[K]Y

`Y` may be any array. `X` must be a simple integer array. The axis specification is optional. If present, `K` must be a simple integer scalar or one-element vector.

The value of `K` must be an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←X⊖Y` implies the first axis.

`X` must have the same shape as `Y`, excluding `Y`'s `K`th axis. Otherwise, if `X` is a one-element array, it will be extended to conform.

`R` is an array with the same shape as `Y`, with the elements of each of the vectors along the `K`th axis of `Y` rotated by the value of the corresponding element of `X`. If the value is positive, the rotation is in the sense of right to left. If the value is negative, the rotation is in the sense of left to right.

Examples

```
      3 ϕ 1 2 3 4 5 6 7
4 5 6 7 1 2 3
      ¯2 ϕ 1 2 3 4 5
4 5 1 2 3

      M
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

      I
0 1 ¯1 0
0 3  2 1

      I⌽[2]M
 1  6  7  4
 5  2  3  8

 9 14 11 16
13 10 15 12
```

```
      J
2 ¯3
3 ¯2
      J⌽M
 3  4  1  2
 6  7  8  5

12  9 10 11
15 16 13 14
```

3.4.81 Rotate First R←X⊖[K]Y

The form `R←X⊖Y` implies rotation along the first axis. See *Section 3.4.80*.

3.4.82 Same R←⊣YR←⊢Y

`Y` may be any array.

The result `R` is the argument `Y`.

Examples

```
      ⊣'abc' 1 2 3
 abc  1 2 3
```

```
      (⊢,⎕SIZE) 'a'⎕NL 4 ⍝ left tine of fork meaning "it"
acc      572
and      492
ascan    740
ascana   716
at      1764
avl    17476
```

3.4.83 Shape R←⍴Y

Y may be any array. R is a non-negative integer vector whose elements are the dimensions of Y. If Y is a scalar, then R is an empty vector. The rank of Y is given by ⍴⍴Y.

Examples

```
      ⍴10

      ⍴'CAT'
3

      ⍴3 4⍴⍳12
3 4

      +G←(2 3⍴⍳6)('CAT' 'MOUSE' 'FLEA')
 1 2 3   CAT  MOUSE  FLEA
 4 5 6

      ⍴G
2

      ⍴⍴G
1

      ⍴¨G
 2 3  3

      ⍴⍴⍴⍴G
          3  5  4
```

3.4.84 Split R←↓[K]Y

Y may be any array. The axis specification is optional. If present, K must be a simple integer scalar or one-element vector. The value of K must be an axis of Y. If absent, the last axis is implied.

The items of R are the sub-arrays of Y along the Kth axis. R is a scalar if Y is a scalar. Otherwise R is an array whose rank is ¯1+⍴⍴Y and whose shape is (K≠⍳⍴⍴Y)/⍴Y.

EXAMPLES

```
      ↓3 4⍴'MINDTHATSTEP'
MIND  THAT  STEP

      ↓2 5⍴⍳10
1 2 3 4 5  6 7 8 9 10

      ↓[1]2 5⍴⍳10
1 6  2 7  3 8  4 9  5 10
```

3.4.85 Table R←⍪Y

Y may be any array. R is a 2-dimensional matrix of the elements of Y taken in row-major order, preserving the shape of the first dimension of Y if it exists

Table has been implemented according to the Extended APL Standard (*ISO/IEC 13751:2001)*.

EXAMPLES

```
      ]display {⍵ (⍴⍵)} ⍪'a'
┌→─────────┐
│ ┌→┐ ┌→──┐ │
│ ↓a│ │1 1│ │
│ └─┘ └~──┘ │
└∈─────────┘

      ]display {⍵ (⍴⍵)} ⍪'hello'
┌→─────────┐
│ ┌→┐ ┌→──┐ │
│ ↓h│ │5 1│ │
│ │e│ └~──┘ │
│ │l│       │
│ │l│       │
│ │o│       │
│ └─┘       │
└∈─────────┘

      ]display {⍵ (⍴⍵)} ⍪2 3 4⍴⍳24
┌→──────────────────────────────────────────────────┐
│ ┌→──────────────────────────────────────┐ ┌→──┐ │
│ ↓ 1  2  3  4  5  6  7  8  9 10 11 12│ │2 12│ │
│ │13 14 15 16 17 18 19 20 21 22 23 24│ └~──┘ │
│ └~──────────────────────────────────────┘     │
└∈──────────────────────────────────────────────┘
```

3.4.86 Take

Take	R←X↑Y

Y may be any array. X must be a simple integer scalar or vector.

If Y is a scalar, it is treated as a one-element array of shape (⍴,X)⍴1. The length of X must be the same as or less than the rank of Y. If the length of X is less than the rank of Y, the missing elements of X default to the length of the corresponding axis of Y.

R is an array of the same rank as Y (after possible extension), and of shape |X. If X[I] (an element of X) is positive, then X[I] sub-arrays are taken from the beginning of the I^{th} axis of Y. If X[I] is negative, then X[I] sub-arrays are taken from the end of the I^{th} axis of Y.

If more elements are taken than exist on axis `I`, the extra positions in `R` are filled with the fill element of `Y` (`⊂∈⊃Y` with `⎕ML←0`).

Examples

```
      5↑'ABCDEF'
ABCDE

      5↑1 2 3
1 2 3 0 0

      ¯5↑1 2 3
0 0 1 2 3

      5↑(⍳3) (⍳4) (⍳5)
 1 2 3  1 2 3 4  1 2 3 4 5  0 0 0  0 0 0
```

```
      M
1 2 3 4
5 6 7 8

      2 3↑M
1 2 3
5 6 7

      ¯1 ¯2↑M
7 8
      M3←2 3 4⍴⎕A
      1↑M3
ABCD
EFGH
IJKL
      ¯1↑M3
MNOP
QRST
UVWX
```

Take with Axes — R←X↑[K]Y

`Y` may be any non-scalar array. `X` must be a simple integer scalar or vector. `K` is a vector of zero or more axes of `Y`.

`R` is an array of the first or last elements of `Y` taken along the axes `K` depending on whether the corresponding element of `X` is positive or negative respectively.

The rank of `R` is the same as the rank of `Y`:

```
      ⍴⍴R ↔ ⍴⍴Y
```

The size of each axis of `R` is determined by the corresponding element of `X`:

```
      (⍴R)[,K] ↔ ⌊,X
```

Examples

```
      □←M←2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      2↑[2]M
 1  2  3  4
 5  6  7  8

13 14 15 16
17 18 19 20

      2↑[3]M
 1  2
 5  6
 9 10

13 14
17 18
21 22


      2 ¯2↑[3 2]M
 5  6
 9 10

17 18
21 22
```

3.4.87 Tally R←≢Y

Y may be any array. R is a simple numeric scalar.

Tally returns the number of major cells of Y. See *Programming: Cells And Subarrays*.

This can also be expressed as the length of the leading axis or 1 if Y is a scalar. Tally is equivalent to the function {⊖⍴(⍴⍵),1}.

Examples

```
      ≢2 3 4⍴⍳10
2
      ≢2
1
      ≢⊖
0
```

Note that ≢⍵ is useful for returning the length of vector ⍵ as a scalar. (In contrast, ⍴⍵ is a one-element vector.)

3.4.88 Times R←X×Y

Y may be any numeric array. X may be any numeric array. R is the arithmetic product of X and Y.

This function is also known as Multiply.

Example

```
      3 2 1 0 × 2 4 9 6
6 8 9 0

       2j3×.3j.5 1j2 3j4 .5
¯0.9J1.9 ¯4J7 ¯6J17 1J1.5
```

3.4.89 Transpose (Monadic) R←⍉Y

Y may be any array. R is an array of shape ⌽⍴Y, similar to Y with the order of the axes reversed.

Examples

```
      M
1 2 3
4 5 6

      ⍉M
1 4
2 5
3 6
```

```
      cube    ⍝ 3D array
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      ⍉ cube
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      ⍴ cube
2 3 4
      ⍴ ⍉ cube    ⍝ transpose reverses shape
4 3 2
```

3.4.90 Type (⎕ML<1) R←∈Y

Migration level must be such that ⎕ML<1 (otherwise ∈ means Enlist. See *Section 3.4.19*).

Y may be any array. R is an array with the same shape and structure as Y in which a numeric value is replaced by 0 and a character value is replaced by ' '.

EXAMPLES

```
      ∈(2 3⍴⍳6)(1 4⍴'TEXT')
 0 0 0
 0 0 0

      ' '=∈'X'
1
```

3.4.91 Union R←X∪Y

Y must be a vector. X must be a vector. If either argument is a scalar, it is treated as a one-element vector. R is a vector of the elements of X catenated with the elements of Y which are not found in X.

Items in X and Y are considered the same if X≡Y returns 1 for those items.

⎕CT and ⎕DCT are implicit arguments of Union.

EXAMPLES

```
      'WASH' ∪ 'SHOUT'
WASHOUT

      'ONE' 'TWO' ∪ 'TWO' 'THREE'
 ONE  TWO  THREE
```

For performance information, see *Programming: Search Functions And Hash*.

3.4.92 Unique R←∪Y

Y may be any array. R is a vector of the unique major cells of Y (the unique items of a vector, the unique rows of a matrix and so forth), in the order in which they first appear in Y. For further information, see *Programming: Cells And Subarrays*.

⍁CT and ⍁⍁DCT are implicit arguments of Unique.

Examples

```
      ∪ 22 10 22 22 21 10 5 10
22 10 21 5

      ∪ v←'CAT' 'DOG' 'CAT' 'DUCK' 'DOG' 'DUCK'
┌───┬───┬────┐
│CAT│DOG│DUCK│
└───┴───┴────┘
      ⊢mat←↑v
CAT
DOG
CAT
DUCK
DOG
DUCK
      ∪mat
CAT
DOG
DUCK
```

```
      a←3 4 5⍴⍳20
      a
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20

 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20

 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
      ∪a
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
```

3.4.93 Unique Mask R←≠Y

Y may be any array.

R is a Boolean vector whose length is the number of major cells in Y. For each major cell of Y, the corresponding element of R is 1 if it is the first occurrence of that value, and 0 if it is a duplicate of an earlier major cell.

⎕CT and ⎕DCT are implicit arguments of Unique.

EXAMPLES

```
      ≠22 10 22 22 21 10 5 10
1 1 0 0 1 0 1 0
```

```
      ≠ v←'CAT' 'DOG' 'CAT' 'DUCK' 'DOG' 'DUCK'
1 1 0 1 0 0

      ⊢mat←↑v
CAT
DOG
CAT
DUCK
DOG
DUCK
      ≠mat
1 1 0 1 0 0
```

3.4.94 Where R←⍸Y

Classic Edition: the symbol ⍸ (Iota Underbar) is not available in Classic Edition, and Where is instead represented by ⎕U2378.

Y must be a simple Boolean or numeric array of non-negative integers.

The model for Where can be expressed as {(⍳⍵)/,⍳⍴⍵}.

If Y is Boolean, R is a vector of the indices of all the 1s in Y. If Y is all zeros, R is an empty vector.

⎕IO is an implicit argument of Where.

Examples

```
      ⎕IO
1
      ⍳ 1 0 1 0 0 0 0 1 0
1 3 8

      ⍳'e'='Pete'
2 4

      3 4⍴0 1 1
0 1 1 0
1 1 0 1
1 0 1 1

      ⍳ 3 4⍴0 1 1
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│1 2 │1 3 │2 1 │2 2 │2 4 │3 1 │3 3 │3 4 │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘

      ⍳2 3 4⍴0 0 0 0 1
┌─────┬─────┬─────┬─────┐
│1 2 1│1 3 2│2 1 3│2 2 4│
└─────┴─────┴─────┴─────┘

      ⍳ 0 1 0 2
2 4 4

      {⍵/⍤⍳,⍴⍵} 0 1 0 2
2 4 4
      ⍳2 2⍴0 1 2 3
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│1 2 │2 1 │2 1 │2 2 │2 2 │2 2 │2 2 │2 2 │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
```

3.4.95 Without R←X~Y

X must be a scalar or vector. R is a vector of the elements of X excluding those elements which occur in Y taken in the order in which they occur in X.

Elements of X and Y are considered the same if X≡Y returns 1 for those elements.

⎕CT and ⎕DCT are implicit arguments of Without. Without is also known as Excluding.

Examples

```
      'HELLO'~'GOODBYE'
HLL
      'MONDAY' 'TUESDAY' 'WEDNESDAY'~'TUESDAY' 'FRIDAY'
 MONDAY  WEDNESDAY

      5 10 15~⍳10
15
```

For performance information, see *Programming: Search Functions And Hash*.

4 Primitive Operators

4.1 Operator Syntax

Operators take one or two operands. An operator with one operand is monadic. The operand of a monadic operator is to the left of the operator. An operator with two operands is dyadic. Both operands are required for a dyadic operator.

Operators have long scope to the left. That is, the left operand is the longest function or array expression to its left (see *Programming: Operators*). A dyadic operator has short scope on the right. Right scope may be extended by the use of parentheses.

An operand may be an array, a primitive function, a system function, a defined function or a derived function. An array may be the result of an array expression.

An operator with its operand(s) forms a derived function. The derived function may be monadic or dyadic and it may or may not return an explicit result.

EXAMPLES

```
      +/⍳5
15
      (*∘2)⍳3
1 4 9

      PLUS ← + ⋄ TIMES ← ×
      1 PLUS.TIMES 2
2
```

Monadic Operators

Like primitive functions, monadic operators can be:

- named
- enclosed within parentheses
- displayed in the session

EXAMPLES

```
      ⍞ ← each ← (⍨)       ⍝ name and display
⍨
      shape←⍴
      shape each (1 2) (3 4 5)
 2  3

      slash←⌿
      +slash ⍳10
55
      swap←⍨⍨
      3 -swap 4
1
```

Right Operand Currying

A dyadic operator may be bound or *curried* with its right operand to form a monadic operator:

EXAMPLES

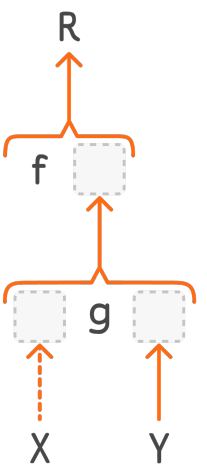
```
      ⍞ ← inv ← ⍣¯1  ⍝ produces monadic inverse operator
⍣ ¯1
      +\inv 1 2 3    ⍝ scan-inverse
1 1 1
      lim ← ⍣≡       ⍝ power-limit

      1 +⍥÷lim 1     ⍝ Phi
1.61803
```

4.2 Function Composition

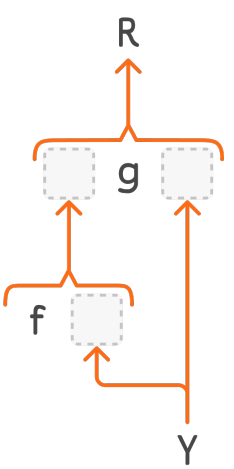
Function composition refers to the "gluing" together of two functions using a dyadic operator such that the functions are applied to the argument(s) as normal, but in a particular pattern specific to the operator that is being used. The term *function composition* comes from traditional mathematics, where it is used for a function `h(x)=f(g(x))` when written as `h(x)=(f∘g)(x)` APL generalises this idea to dyadic functions, allowing various patterns of application in addition to the simple application of one monadic function to the result of another monadic function. The four main patterns, represented by *atop*, *behind*, *beside*, and *over*, can be visualised as shown below.

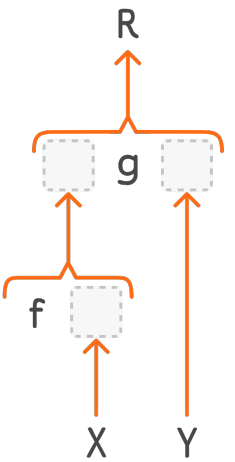
> **Information**
>
> In the diagrams below, the dotted branch falls away when the operator is applied monadically. Note that monadic *atop*, *beside*, and *over* are all equivalent to each other and to `h⁡(x)=(f∘g)(x)` of traditional mathematics.

Atop: `{R}←{X}f⍥g Y`

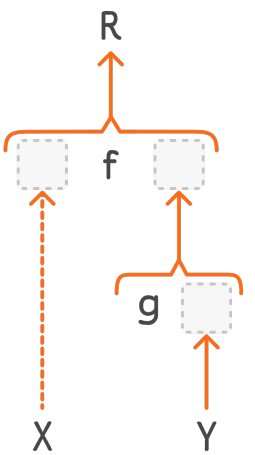


Behind: `{R}←{X}f⍤g Y`

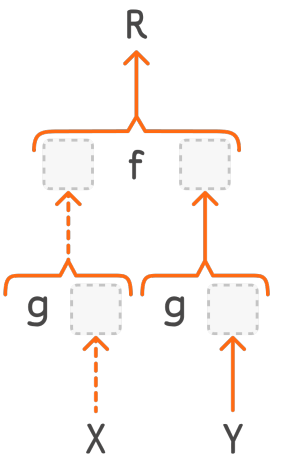




Beside: `{R}←{X}f∘g Y`



Over: `{R}←{X}f⍥g Y`



4.3 Operators Summarised

TABLE-REFERENCE and TABLE-REFERENCE below summarise the Monadic and Dyadic primitive operators whose detailed descriptions follow in alphabetical order in this section.

Some operators may include an axis specification (indicated `[]`in the tables). Note that in these case `⎕IO` is an implicit argument of the derived function.

Table: Monadic Primitive Operators { #monadic-operators }

Name	Producing Monadic derived function	Producing Dyadic derived function
Assignment (Modified)		`Xf←Y`
Assignment (Indexed Modified)		`X[I]f←Y`
Assignment (Selective Modified)		`(EXP X)f←Y`
Commute	`f⍨Y`	`Xf⍨Y`
Each	`f¨Y`	`Xf¨Y`
I-Beam	`A⌶Y`	`X(A⌶)Y`
Key	`f⌸Y`	`Xf⌸Y`
Reduction	`f/Y [ ]`	`Xf/Y [ ]`
Reduction First	`f⌿Y [ ]`	`Xf⌿Y [ ]`
Scan	`f\Y [ ]`	
Scan First	`f⍀Y [ ]`	
Spawn	`f&Y`	`Xf&Y`

Table: Dyadic Primitive Operators { #dyadic-operators }

Name	Producing Monadic derived function	Producing Dyadic derived function
At	f@gY	Xf@gY
Atop	f⍤gY	Xf⍤gY
Axis	f[B]Y	Xf[B]Y
Behind	f⍥gY	Xf⍥gY
Beside	f∘gY	Xf∘gY
Bind	A∘gY	
	(f∘B)Y	
Constant	(A⍨)Y	X(A⍨)Y
Inner Product		Xf.gY
Outer Product		X∘.gY
Over	f⍤gY	Xf⍤gY
Power	f⍣gY	Xf⍣gY
Rank	f⍤kY	Xf⍤kY
Stencil	f⌺gY	
Variant	f⌸gY	Xf⌸gY

4.4 Primitive Operators (A-Z)

4.4.1 At R←{X}(f@g)Y

This operator substitutes selected items in Y with new values or applies a function to modify selected items in Y.

The right operand g identifies which items of array Y are to be substituted or modified. It is either:

- an array that specifies a set of indices in Y. If g is a simple scalar or vector, it selects major cells in Y. If nested, it specifies indices for Choose or Reach indexing.

- or a function that when applied to `Y` returns a Boolean array of the same shape as `Y` (a *mask*) in which a 1 indicates that the corresponding item of `Y` is to be substituted or modified. Note that the *ravel* of the mask selects from the *ravel* of the right argument's index array.

The left operand `f` is either:

- an array that contains values to replace those items in `Y` identified by `g`
- or a function to be applied to those items, the result of which is used to replace them. If this function is dyadic, its left argument is the array `X`. Note that the function is applied to the sub-array of `Y` selected by `g` *as a whole* and not to each item separately.

The result `R` is the same as `Y` but with the items specified by `g` substituted or modified by `f`.

Examples (array @ array)

Replace the 2nd and 4th items of `⍳5`:

```
      (10 20@2 4)⍳5   ⍝ [^1]
1 10 3 20 5

      10 20@2 4⍳5
1 10 3 20 5
```

Replace the 2nd and 4th items of nested vector with `⍬`:

```
      (⊂⍬)@2 4 ⍳¨⍳5
┌─┐┌─────┐┌─────────┐
│1│⋄1 2 3⋄│1 2 3 4 5│
└─┘└─────┘└─────────┘
```

Replace the 2nd and 4th rows (major cells) of a matrix:

```
      (2 3⍴10 20)(@2 4)4 3⍴⍳12
 1  2  3
10 20 10
 7  8  9
20 10 20
```

Replace first and last elements with 0 using Choose Indexing:

```
      (0@(1 1)(4 3))4 3⍴⍳12
 0  2 3
 4  5 6
 7  8 9
10 11 0
```

Replace nested items using Reach Indexing:

```
       G
┌─────┬─────┬─────┐
│┌───┐│┌───┐│┌───┐│
│|ABC|1|||DEF|2|||GHI|3||
│└───┘│└───┘│└───┘│
├─────┼─────┼─────┤
│┌───┐│┌───┐│┌───┐│
│|JKL|4|||MNO|5|||PQR|6||
│└───┘│└───┘│└───┘│
└─────┴─────┴─────┘
       G[((1 2)1)((2 3)2)]
┌───┐
|DEF|6|
└───┘
       ('' '*' @((1 2)1)((2 3)2)) G
┌─────┬─────┬─────┐
│┌───┐│┌─┐  │┌───┐│
│|ABC|1|||2|  |||GHI|3||
│└───┘│└─┘  │└───┘│
├─────┼─────┼─────┤
│┌───┐│┌───┐│┌───┐│
│|JKL|4|||MNO|5|||PQR|*||
│└───┘│└───┘│└───┘│
└─────┴─────┴─────┘
```

Examples (function @ array)

Replace the 2nd and 4th items of ⍳5 with their reciprocals:

```
      ÷@2 4 ⍳5
1 0.5 3 0.25 5
```

Replace the 2nd and 4th items of ⍳5 with their reversal:

```
      ⌽@2 4 ⍳5
1 4 3 2 5
```

Multiply the 2nd and 4th items of ⍳5 by 10:

```
      10×@2 4⍳5
1 20 3 40 5
```

Replace the 2nd and 4th items by their totals:

```
       +/¨@2 4 ⍳¨⍳5
┌─┬─┬─────┬──┬─────────┐
│1│3│1 2 3│10│1 2 3 4 5│
└─┴─┴─────┴──┴─────────┘
```

Replace the 2nd and 4th rows (major cells) of a matrix with their accumulatives:

```
      (+\@2 4)4 3⍴⍳12
 1  2  3
 4  9 15
 7  8  9
10 21 33
```

Examples (array @ function)

Replace odd elements with 0:

```
      0@(2∘|)⍳5
0 2 0 4 0
```

Replace multiples of 3 (note that masked items are substituted in ravel order):

```
      'abcde'@{0=3|⍵} 4 4⍴⍳16
 1  2  a  4
 5  b  7  8
 c 10 11  d
13 14  e 16

      'abcde'@(0=3|⊢) 4 4⍴⍳16 ⍝ or using a train
 1  2  a  4
 5  b  7  8
 c 10 11  d
13 14  e 16
```

Examples (function @ function)

Replace odd elements with their reciprocals:

```
      ÷@(2∘|)⍳5
1 2 0.3333333333 4 0.2
```

Replace odd items of `⍳5` with themselves reversed:

```
      ⌽@(2∘|)⍳5
5 2 3 4 1
```

4.4.2 Atop {R}←{X}f⍤gY

Classic Edition: the symbol ⍤ is not available in Classic Edition, and the Atop operator is instead represented by □U2364.

`f` can be any monadic function. `Y` can be any array that is suitable as the right argument to function `g` with the result of `g` being appropriate to function `f`.

If `X` is omitted, `g` must be a monadic function. If `X` is specified, `g` must be a dyadic function and `X` can be any array that is suitable as the left argument to function `g`.

The derived function is equivalent to `fgY` or `fXgY` and need not return a result.

The Atop operator allows functions to be *glued* together to build up more complex functions. For further information, see *Function Composition*.

EXAMPLES

```
      -⍤÷ 4       ⍝ (  f⍤g y) ≡  f   g y
¯0.25
      12 -⍤÷ 4    ⍝ (x f⍤g y) ≡ (f x g y)
¯3
      3 1 4 1 5 ~⍤∈ 1 2 3
0 0 1 0 1
```

4.4.3 Behind {R}←{X}f⍥gY

> **Information**
>
> The `⍥` glyph is not available in Classic Edition, and the *behind* operator is instead represented by `⍁U235B`.

`f` can be any monadic function that returns a result; the result must be suitable as the left argument to the function `g`.

`g` can be any dyadic function; it does not need to return a result.

`Y` can be any array that is suitable as the right argument to the function `g`. If `X` is omitted, `Y` must also be suitable as the right argument to the function `f`.

`X` can be any array that is suitable as the right argument to the function `f`.

The derived function is equivalent to either `(f Y) g Y` or `(f X) g Y`, depending on whether `X` is specified or not.

The *behind* operator allows functions to be *glued* together to build up more complex functions. For further information, see *Section 4.1*.

Examples: Monadic Application of Derived Function

Are numbers in a sequence identical to the first number in that sequence?

```
      ⊃⍥= 2 7 1 8 2 8 1 8 2 8
1 0 0 0 1 0 0 0 1 0
```

Are the characters within each vector unique?

```
      ∪⍥≡¨ 'Hello' 'Helo'
0 1
```

Where in a Boolean mask does it match its reverse?

```
      ⌽⍥∧ 0 1 1 1 1 0 0 0 0
0 0 0 0 1 0 0 0 0
```

Prefix a vector with its final element:

```
      ⊢/⍥,'XYZ'
ZXYZ
```

Identify where the odd numbers are located within a vector, and add 1 to them:

```
      2∘| 3 1 4 1 5 9     ⍝ create mask of odd numbers
1 1 0 1 1 1
      2∘|⍥+ 3 1 4 1 5 9   ⍝ add one to the odd numbers
4 2 4 2 6 10
```

Identify the numbers in a vector greater than 3, and filter to return only those numbers:

```
      3∘< 2 7 1 8 2 8     ⍝ create mask
0 1 0 1 0 1
      3∘<⍥/ 2 7 1 8 2 8   ⍝ apply filter
7 8 8
```

Identify the integers in a vector of numbers, and filter to return only the integers:

```
      ⌊⍥= 1 3.2 ¯5 0 ¯3.2 8.1     ⍝ create mask
1 0 1 1 0 0
      ⌊⍥=⍥/ 1 3.2 ¯5 0 ¯3.2 8.1   ⍝ apply filter
1 ¯5 0
```

Identify the palindromes, and filter to return only the palindromes:

```
      ⌽⍥≡¨ 'racecar' 'racer' 'hanna' 'asa'   ⍝ create mask
1 0 0 1
      ⌽⍥≡¨⍥/ 'racecar' 'racer' 'hanna' 'asa' ⍝ apply filter
‘racecar’ ‘asa’
```

Find the mean value of a vector of numbers, identify the individual numbers in the vector that are greater than the mean value, and filter to retun only those numbers:

```
      (+⌿÷≢) 3 1 4 1 5 9 2 6       ⍝ calculate mean value
3.875
      (+⌿÷≢)⍥< 3 1 4 1 5 9 2 6     ⍝ create mask
0 0 1 0 1 1 0 1
      (+⌿÷≢)⍥<⍥/ 3 1 4 1 5 9 2 6   ⍝ apply filter
4 5 9 6
```

Examples: Dyadic Application of Derived Function

Some functions require their left arguments to be enclosed to achieve the desired result. Without *behind*, the left argument needs to be parenthesised to enclose the left argument. With *behind*, a new function can be derived that achieves the required enclosure by including ⊂⍤⍥ before the main function. For example:

Does the left argument exist within the right argument?

```
      (⊂'dog') ∈ 'cat' 'dog' 'bird'    ⍝ without behind
1
      'dog' ⊂⍤⍥∈ 'cat' 'dog' 'bird'    ⍝ with behind
1
```

Select elements using the index function:

```
      1 4 1 13 ⊂⍤⍥[] ⎕A
ADAM
```

Write a vector of vectors to a file:

```
      'abc' 'def' ⊂⍤⍥⎕NPUT 'myfile.txt'
```

You can compose functions to achieve useful functionality. For example:

Evaluate x^2+2 for multiple values of x:

```
      (⍨5 10)⊥ 1 0 2              ⍝ without behind
      5 10 ⍨⍤⍥⊥ 1 0 2             ⍝ with behind
27 102
```

Convert numbers to text as they are catenated:

```
      'there are ',2∘⍕⍤⍥,' things'
there are 2 things
```

Reshape an array to conform to the shape of another array:

```
      'abc' ⍴⍤⍥⍴ 'z'
zzz
```

Some functionality does not exist as a primitive but can easily be composed using *behind*. For example:

Identify the first occurrence of an element starting from the end of the source:

```
      'abracadabra' ⌽⍸⍤⍳ 'ab'
1 3
```

Construct a Boolean mask from a length and the indices of the `1`s (inverse of monadic `⍳`):

```
      10 ⍳⍸⍤∈ 1 3 5
1 0 1 0 1 0 0 0 0 0
```

Filter to return elements that correspond to `0` in a Boolean mask – the opposite of *reduce* (`/`), which returns elements that correspond to `1` in a Boolean mask.:

```
      0 1 0 1 1 0 ~⍤/ 'Dyalog'
Dag
```

Take elements from both ends of the source:

```
      3 (↑⍪-⍤↑) ⎕A
ABCXYZ
```

Sort a vector by values in another vector:

```
      30 40 20 ⊂⍤⍋⍤⍋⍤[] 'Abe' 'Bea' 'Carl'
Carl Abe Bea
```

Split a vector with a function that can be applied either monadically (in which case it uses the initial character as separator) or dyadically (in which case it uses the left argument as separator):

```
      ]Boxing on
      Split←⊃⍤≠⊆⊢
      Text←',I S,EAT ING,RATES'
      Split Text     ⍝ Split at occurences of first element
┌─┬─┬───────┬─────┐
│ │I│ S,EAT │ING,RATES│
└─┴─┴───────┴─────┘
      ' 'SPLIT TEXT  ⍝ Split at occurences of left argument
┌─┬─┬─────┬──────┐
│ │,I│S,EAT│ING,RATES│
└─┴─┴─────┴──────┘
```

4.4.4 Beside {R}←{X}f∘gY

g can be any monadic function which returns a result. Y can be any array appropriate to function g with gY being suitable as the right argument to function f.

If X is omitted, f must be a monadic function. If X is specified, f must be a dyadic function and X can be any array that is suitable as the left argument to function f.

The derived function is equivalent to fgY or XfgY and need not return a result.

The Beside operator allows functions to be glued together to build up more complex functions. For further information, see *Function Composition*.

Examples

```
      RANK ← ⍴∘⍴
      RANK ¨ 'JOANNE' (2 3⍴⍳6)
 1  2
```

```
      +/∘⍳¨2 4 6
3 10 21


      ⎕VR'SUM'
     ∇ R←SUM X
[1]    R←+/X
     ∇

      SUM∘⍳¨2 4 6
3 10 21
```

```
      +∘÷/40⍴1        ⍝ Golden Ratio! (Bob Smith)
1.618033989

      0,∘⍳¨⍳5
0 1  0 1 2  0 1 2 3  0 1 2 3 4  0 1 2 3 4 5
```

4.4.5 Bind {R}←A∘fY{R}←(f∘B)Y

The Bind operator binds an array A or B to a dyadic function f either as its left or its right argument respectively. The former may be described as left argument currying and the latter as right argument currying.

`A`, `B` and `Y` may be any arrays whose items are appropriate to function `f`. In the case where `B` is bound as the right argument of function `f`, the parentheses are required in order to distinguish between the operand `B` and the argument `Y`.

The derived function is equivalent to `AfY` or `YfB` and need not return a result.

EXAMPLES

```
      2 2∘⍴ ⍤ 'AB'
 AA  BB
 AA  BB

      SINE ← 1∘○

      SINE 10 20 30
¯0.5440211109 0.9129452507 ¯0.9880316241
```

```
      (*∘0.5)4 16 25
2 4 5

      SQRT ← *∘.5

      SQRT 4 16 25
2 4 5
```

The following example uses both forms of Bind to list functions in the workspace:

```
      ⎕NL 3
ADD
PLUS

      ⎕←∘⎕VR⍤↓⎕NL 3
     ∇ ADD X
[1]    →LAB⍴⍨0≠⎕NC'SUM' ⋄ SUM←0
[2]   LAB:SUM←SUM++/X
     ∇
     ∇ R←A PLUS B
[1]    R←A+B
     ∇
```

4.4.6 Commute {R}←{X}f⍨Y

`f` may be any dyadic function. `X` and `Y` may be any arrays whose items are appropriate to function `f`.

The derived function is equivalent to `Y f X`. The derived function need not return a result.

If left argument `X` is omitted, the right argument `Y` is duplicated in its place, that is:

```
      f⍨Y ↔ Y f⍨Y
```

EXAMPLES

```
      N
3 2 5 4 6 1 3

      N/⍨2|N
3 5 1 3

      ⍴⍨3
3 3 3



      mean←+/∘(÷∘⍴⍨) ⍝ mean of a vector
      mean ⍳10
5.5
```

The following statements are equivalent:

```
      F/⍨←I
      F←F/⍨I
      F←I/F
```

Commute often eliminates the need for parentheses

4.4.7 Constant R←{X}(A⍨)Y

`A`, `X` and `Y` are arrays. The Constant operator returns array `A`.

EXAMPLES

```
      'mu'⍨ 'any' ⍬NULL   ⍝ Always returns its operand
mu
      1E100 ('mu'⍨) 1j1
mu
      ¯1⍨⍤1 ⍳2 3
¯1 ¯1 ¯1
¯1 ¯1 ¯1
```

4.4.8 Each

Each (with Monadic Operand) {R}←f¨Y

f may be any monadic function. Y may be any array, each of whose items are separately appropriate to function f.

The derived function applies function f separately to each item of Y. The derived function need not return a result. If a result is returned, R has the same shape as Y, and its elements are the items produced by the application of function f to the corresponding items of Y.

If Y is empty, the prototype of R is determined by applying the operand function *once* to the prototype of Y.

EXAMPLES

```
      G←('TOM' (⍳3))('DICK' (⍳4))('HARRY' (⍳5))
      ⍴G
3
      ⍴¨G
 2  2  2

      ⍴⍴¨G
  3  3    4  4    5  5

      +⎕FX¨('FOO1' 'A←1')('FOO2' 'A←2')
 FOO1 FOO2
```

Each (with Dyadic Operand) {R}←XfY

f may be any dyadic function. X and Y may be any arrays whose corresponding items (after scalar extension) are appropriate to function f when applied separately.

The derived function is applied separately to each pair of corresponding elements of X and Y. If X or Y is a scalar or single-element array, it will be extended to conform with the other argument. The derived function need not produce an explicit result. If a result is returned, R has the same shape as Y (after possible scalar extension) whose elements are the items produced by the application of the derived function to the corresponding items of X and Y.

If X or Y is empty, the operand function is applied *once* between the prototypes of X and Y to determine the prototype of R.

Examples

```
      +G←(1 (2 3))(4 (5 6))(8 9)10
  1  2 3   4  5 6   8 9  10
      1⌽⍤G
 2 3  1   5 6  4  9 8  10

      1⌽⍤⍤G
1  3 2   4  6 5   8 9  10

      1⌽⍤⍤⍤G
1  2 3   4  5 6   8 9  10

      1 2 3 4↑⍤G
1  4  5 6   8 9 0  10 0 0 0

      'ABC',⍤'XYZ'
AX  BY  CZ
```

4.4.9 I-Beam

I-Beam	R←{X}(A⌶)Y

I-Beam is a monadic operator that provides a range of system-related services.

> **Warning**
>
> Any service provided using an I-beam should be considered as experimental and subject to change – without notice – from one release to the next. Any use of I-beams in applications should, therefore, be carefully isolated in cover-functions that can be adjusted if necessary.

`A` is an integer that specifies the type of operation to be performed – see the table below.

`X` (optionally, depends on `A`) and `Y` are arrays that supply further information about what is to be done and to what.

`R` is the result.

When attempting to use the I-beam operator with an unsupported operation value `A`, one of three different error messages will be reported:

- Invalid I-Beam function selection
- I-Beam function A has been withdrawn
- I-Beam function A is not supported by this interpreter

This allows the user to distinguish between operation values that have never been used, those that have been used in earlier versions but are no longer included in the current version, and those that are valid in other editions or on other platforms.

Key to restrictions in the following table:

- AIX: AIX only
- LiW: Linux/Windows only
- WF: Microsoft .NET Framework only
- WIN: Microsoft Windows only

- X: AIX/Linux/macOS only
- 64U: 64-bit Unicode only

A	Derived Function	Restrictions
8	*Inverted Table Index-of*	
13	*Log Use of Deprecated Features*	
43	*Monadic Operator Generator*	
85	*Execute Expression*	
109	*Log File for Deprecations*	
120	*Generate UUID*	
127	*Overwrite Free Pockets*	
180	*Canonical Representation*	
181	*Unsqueezed Type*	
200	*Syntax Colouring*	
201	*Syntax Colour Tokens*	
219	*Compress/Decompress Vector of Short Integers*	
220	*Serialise/Deserialise Array*	
400	*Compiler Control*	
600	*Disable Traps*	
739	*Temporary Directory*	
900	*Called Monadically?*	
950	*List Loaded Libraries*	
1010	*Set Shell Script Debug Options*	
1111	*Number of Threads*	
1112	*Parallel Execution Threshold*	
1159	*Update Function Time and User Stamp*	
1200	*Format Date-time*	
1302	*aplcore Parameters*	
1500	*Hash Array*	
2000	*Memory Manager Statistics*	
2002	*Specify Workspace Available*	

A	Derived Function	Restrictions
2007	*Disable Global Triggers*	
2010	*Update DataTable*	WF
2011	*Read DataTable*	WF
2014	*Remove Data Binding*	WF
2015	*Create Data Binding Source*	WF
2016	*Create .NET Delegate*	WF
2017	*Identify .NET Type*	WF
2022	*Flush Session Caption*	WIN
2023	*Close all Windows*	
2035	*Set Dyalog Pixel Type*	WIN
2041	*Override COM Default Value*	WIN
2100	*Export to Memory*	WIN
2101	*Close .NET AppDomain*	WF
2250	*Verify .NET Interface*	
2400	*Set Workspace Save Options*	
2401	*Expose Root Properties*	
2501	*Discard Thread on Exit*	WIN
2502	*Discard Parked Threads*	WIN
2503	*Mark Thread as Uninterruptible*	
2520	*Use Separate Thread For .NET*	WF
2704	*Continue Autosave*	
3002	*Disable Component Checksum Validation*	
3012	*Enable Compression of Large Components*	
3500	*Send Text to Ride-embedded Browser*	
3501	*Connected to Ride?*	
3502	*Manage Ride Connections*	
3535	*Scan For Deprecated Files*	

⎕A	Derived Function	Restrictions
4000	*Fork New Task*	AIX
4001	*Change User*	X
4002	*Reap Forked Tasks*	AIX
4007	*Signal Counts*	X
5171	*Discard Source Information*	
5172	*Discard Source Code*	
5176	*List Loaded Files*	
5177	*List Loaded File Objects*	
5178	*Remove Loaded File Object Info*	
5179	*Loaded File Object Info*	
5581	*Unicode Normalisation*	
7162	*JSON Translate Name*	
8373	*Shell Process Control*	
8415	*Singular Value Decomposition*	
8659	*List Shared Code Files/Attached Names*	64U
8666	*Attach/Assimilate/Detach Shared Code Files*	64U
8667	*Save Shared Code Files*	64U
16808	*Sample Probability Distribution*	LiW
50100	*Line Count*	

Inverted Table Index-of `R←X(⎕8⌶)Y`

This function computes `X` index-of `Y` (viz. `X⍳Y`) where `X` and `Y` are compatible inverted tables. `R` is the indices of `Y` in `X`.

An inverted table is a (nested) vector all of whose items have the same number of major cells. That is, `1=≢⍴⍵` and `(≢⊃⍵)=≢¨⍵`. An inverted table representation of relational data is more efficient in time and space than other representations.

The following is an example of an inverted table:

```
      X←(10 3⍴⎕A) (⍳10) 'metalepsis'
      X
┌───┬──────────────────────┬──────────┐
│ABC│0 1 2 3 4 5 6 7 8 9   │metalepsis│
│DEF│                      │          │
│GHI│                      │          │
│JKL│                      │          │
│MNO│                      │          │
│PQR│                      │          │
│STU│                      │          │
│VWX│                      │          │
│YZA│                      │          │
│BCD│                      │          │
└───┴──────────────────────┴──────────┘
```

Using inverted tables, it is often necessary to perform a table look-up to find the "row" indices of one in another. Suppose there is a second table `Y`:

```
      Y←(⊂⊂3 1 4 1 5 9)⌷⍤¯1⊢X
      Y
 GHI  3 1 4 1 5 9  tmamli
 ABC
 JKL
 ABC
 MNO
 YZA
```

To compute the indices of `Y` in `X` using dyadic `⍳`, it is necessary to first un-invert each of the tables in order to create nested matrices that `⍳` can handle.

```
      unvert ← {⍵↑⊂⍤¯1⍤1⍵}
      unvert X
┌───┬─┬─┐
│ABC│0│m│
├───┼─┼─┤
│DEF│1│e│
├───┼─┼─┤
│GHI│2│t│
├───┼─┼─┤
│JKL│3│a│
├───┼─┼─┤
│MNO│4│l│
├───┼─┼─┤
│PQR│5│e│
├───┼─┼─┤
│STU│6│p│
├───┼─┼─┤
│VWX│7│s│
├───┼─┼─┤
│YZA│8│i│
├───┼─┼─┤
│BCD│9│s│
└───┴─┴─┘

      (unvert X) ⍳ (unvert Y)
3 1 4 1 5 9
```

Each un-inverted table requires considerably more workspace than its inverted form, so if the inverted tables are large, this operation is potentially expensive in terms of both time and workspace.

`8⌶` is an optimised version of the above expression.

```
      X (8⌶) Y
3 1 4 1 5 9
```

Log Use of Deprecated Features `{R}←(13⌶)Y`

Controls which deprecated features are logged when logging is enabled. For an overview of deprecated features and this I-beam's role in identifying them within your code, see *Programming: Deprecated Functionality*.

`Y` is a character vector, or a vector of character vectors, each containing the name of a deprecated feature, or one of the names defined in the table below. The names are

release-dependent; for a list of valid names see the *v20.0 Release Notes: Deprecated Functionality*.

Subsequent uses of the selected deprecated features will be logged, provided that *Section 4.4.9.6* has been called to set the name of the log file.

If `13I` is called again, the list of features that are logged is replaced.

The result `R` is a vector of zero or more names, as described in the table below.

Value(s) in `Y`	**Meaning**	**Value(s) in** `R`
Feature names	Enable logging of the specified features	Names of features for which logging is enabled (shy)
`'All'`	Enable logging of all deprecated features	Names of features for which logging is enabled (non-shy)
`'None'`	Disable logging of deprecated features	
`'Enabled'`	List all features for which logging is enabled	
`'List'`	List names of all possible features	Names of all deprecated features (non-shy)

Before any logging information is created, the log file must also be configured using *Section 4.4.9.6*. Without selecting a log file, all logging is silently discarded.

Each log entry is a complete JSON5 object definition that includes the following items:

- `TS`: a timestamp.
- `Type`: a description of the entry type (always `'Warning'`).
- `Message`: a message associated with the log entry (always `'Use of deprecated feature'`).
- `Feature`: a description of the deprecated feature.
- `ExtraInfo`: feature-specific additional information (can be an empty string).
- `WSID`: the name of the workspace in which the feature was used.
- `Stack`: an array of strings indicating the SIstack at the point the feature was used.

EXAMPLE

```
      13⌶'List'
This  That  TheOther
      13⌶'Enabled'

      13⌶'That' 'This'
      13⌶'Enabled'
This  That
      13⌶'All'
      13⌶'Enabled'
This  That  TheOther
```

See also *Section 4.4.9.6*.

Monadic Operator Generator R←43⌶Y

Returns a monadic operator, with functionality determined by the value of `Y`.

`Y` is a scalar integer. Possible values for `Y` are shown in the table below. If an undefined value is specified, a `DOMAIN ERROR` is generated.

`R` is a monadic operator with functionality determined by the value of `Y`. As `R` is a monadic operator, Dyalog Ltd recommends that the evaluation of this I-Beam is either parenthesised or named.

Y	Returned Operator	Notes
632	Generics Operator	Requires .NET.

> **Warning**
>
> The set of supported values for `Y` might change, with existing values being withdrawn as well as new ones being added. This means that saving the resulting operator in a workspace and attempting to run it in a later release might result in a `DOMAIN ERROR` if support has been withdrawn.

632 - Generics Operator

The generics operator can create concrete versions of generic classes and execute generic methods. For more information, see the *.NET Interface Guide*.

EXAMPLE

```
      ⎕USING←''

      ⍝ I-Beam call is parenthesised
      IntList←System.Collections.Generic.List (43⌶632) System.Int32

      ⍝ Result of I-Beam call is named
      GenOp←43⌶632
      CharList←System.Collections.Generic.List GenOp System.Char
```

Execute Expression R←X(85⌶)Y

Executes an expression.

`Y` is a character vector containing an APL expression.

The function executes the expression in `Y` exactly as it would be executed by the monadic Execute primitive function `⍎`, but handles shy results of the execution rather differently.

The left argument `X` determines how a shy result from the execution of `Y` is treated and is either 0 or 1.

If `X` is 1, and the expression in `Y` returns an explicit result, `R` is that result. If the expression in `Y` returns no result or returns a shy result, the function signals `ERROR 85`. Effectively, a shy result is discarded.

If `X` is 0, and the expression in `Y` returns an explicit result or a shy result, `R` is that result (but is no longer shy). If the expression in `Y` returns no result, the function signals `ERROR 85`.

Examples

```
      ⍎'a←42'
      ⍬← ⍎'a←42'      ⍝ shy result
42
      0 (85⌶) 'a←42' ⍝ not shy
42
      1 (85⌶) 'a←42'
ERROR 85
      1(85⌶)'a←42'
     ^
```

Log File for Deprecations {R}←{X} 109⌶Y

Manages the file used to log the use of deprecated features. For an overview of deprecated features and this I-beam's role in identifying them within your code, see *Programming: Deprecated Functionality*.

`Y` indicates the action to perform.

Set or query the log file name (Y = 0)

If `X` is omitted, the result `R` is the name of the log file currently being written to. An empty character vector is returned if a name has not been set.

Otherwise, `X` is a simple character scalar or vector containing a valid filename, or is empty. Any existing log file is closed and, if `X` is not empty, the specified file is opened and subsequent log messages will be appended to it. The shy result `R` is the previous name of the log file.

An error will be signalled if the specified file cannot be opened in append mode. However, if messages cannot subsequently be written to it, the running application will continue uninterrupted and the messages will be silently discarded. The log file status can be queried at any time.

Query log file status (Y = 1)

X must be omitted. The result R is a two element vector consisting of a numeric status code (0 indicating no error) and a character vector containing text describing the error that was encountered (empty if no error).

EXAMPLE

```
      ⊢'logfile.txt'(109⌶)0
old_logfile.txt
      (109⌶)1
┌┬┐
│0││
└┴┘
```

See also *Section 4.4.9.3*.

Generate UUID R←120⌶Y

This function generates a UUID (Universally Unique IDentifier) according to the RFC 9562 specification.

A UUID is a label that uniquely identifies objects in computer systems; it does not depend on a central registration authority or co-ordination between the parties generating them. UUIDs are also known as GUIDs (Globally Unique IDentifiers).

Y specifies the UUID version required. Supported values of Y are:

Y	Version
4	UUIDv4: random values
7	UUIDv7: a combination of values that are time-ordered (based on Unix Epoch) and random values

The result R is a vector containing the generated 36-character UUID.

EXAMPLE

```
      120⌶4
32cd549f-eb33-4457-bf45-babf26dc2b53
```

Overwrite Free Pockets R←127⌶Y

Overwrites all free pockets in the workspace.

Some applications (cryptography for example) make use of secure data during execution. The nature of the APL workspace is such that remnants of this secure data may persist in the workspace (and thus the process memory) even after the relevant APL variables have been expunged. This function overwrites all unused data pockets in the workspace so that any potentially secure data is removed.

Y is any empty array, preferably ⍬(zilde). R is always 1.

It is the responsibility of the programmer to ensure that there are no USED pockets in the workspace that reference the data.

EXAMPLE

```
     ∇ foo;a
[1]    a←'my secure data'
[2]    ⎕EX'a'
[3]    ⍝ 'my secure data' is now in an
[4]    ⍝ UNUSED pocket in the workspace
[5]    a←127⌶0 ⍝ all unused pockets are overwritten,
[6]            ⍝ 'my secure data' is no longer present
     ∇
```

Whereas

```
     ∇ foo;a;b
[1]    a←'my secure data'
[2]    b←a
[3]    ⎕EX'a'
[4]    ⍝ 'my secure data' is now in an
[5]    ⍝ UNUSED pocket in the workspace
[6]    a←127⌶0 ⍝ all unused pockets are overwritten,
[7]            ⍝ but 'my secure data' is still present
[8]            ⍝ because it is referenced by b
     ∇
```

Canonical Representation R←180⌶Y

This function is the same as the system function ⎕CR except that it can be used to obtain the canonical representation of methods in classes. 180⌶ is used by]PROFILE.

EXAMPLE

```
      )LOAD ComponentFile
C:\Program Files\Dyalog\Dyalog APL-64 15.0 Unicode\...

      180⌶'ComponentFile.Close'
 Close
 :Implements Destructor
 :If tie∈⎕FNUMS
     :If temp ⋄ Name ⎕FERASE tie
     :Else ⋄ ⎕FUNTIE tie
     :EndIf
 :EndIf
```

Unsqueezed Type R←181⌶Y

Y is any array.

The result R is an integer scalar containing an integer value which indicates the type of the array. For further information see *Section 6.4.33*.

181⌶ is functionally identical to monadic ⎕DR, except that no attempt is made to squeeze the data into smaller data types. ⎕DR always attempts to squeeze the data; 181⌶ does not, but if a workspace compaction occurs during execution of 181⌶, the data may still be squeezed before the type is identified.

EXAMPLE

```
      ⎕DR 1↑1 1000
11
      (181⌶) 1↑1 1000
163
```

Syntax Colouring R←200⌶Y

This function obtains syntax colouring information for a function.

`Y` is a vector of character vectors containing the `⎕NR` representation of a function or operator.

`R` is a vector of integer vectors with the same shape and structure of `Y` in which each number identifies the syntax colour element associated with the corresponding character in `Y`.

```
      ∇foo∇
     ∇ foo;local
[1]    global
[2]    local←⍴⍴'hello'
     ∇
      ⎕NR 'foo'
  foo;local   global   local←⍴⍴'hello'

     {(↑⍵),↑200⌶⍵}⎕NR 'foo'
 foo;local       3 21 21 21 19 34 34 34 34 34 0 0 0 0 0 0
 global          3  7  7  7  7  7  7  0  0  0 0 0 0 0 0 0
 local←⍴⍴'hello' 3 34 34 34 34 34 19 23 23  4 4 4 4 4 4 4
```

In this example:

21	is the syntax identifier for "character constant"
19	is the syntax identifier for "primitive"
3	is the syntax identifier for "white space"
34	is the syntax identifier for "local name"
7	is the syntax identifier for "global name"
23	is the syntax identifier for "idiom"

The list of syntax colour elements supported by the current interpreter is given by `201⌶`. It is important to note that the values may change within a release, and are very likely to change across releases .. you should always call `201⌶` rather than relying the results from a different interpreter. See *Section 4.4.9.12*.

Syntax Colour Tokens `R←201⌶Y`

This function provides a description of the syntax colour tokens reported by `200⌶`. See *Section 4.4.9.11*.

`Y` is `⍬` (zilde).

`R` is a 4-column matrix that describes the syntax colouring tokens as follows:

`R[;1]`	Token type
`R[;2]`	Token Value
`R[;3]`	Internal description
`R[;4]`	Colour index

The 4th column is intended for the benefit of non-Windows users using the tty interface and indicates the video/foreground/background colour index. These indices appear in the output translate tables found in `$DYALOG/apltrans`, and are used to define the colours used in the tty interface.

Example

```
      ⍴201⌶⍬
207 4
     3 4↑201⌶⍬
┌──────────────┬─┬──────────────┬───┐
│Global token  │0│MINI_NULL     │129│
├──────────────┼─┼──────────────┼───┤
│Global token  │1│MINI_COMMENT  │137│
├──────────────┼─┼──────────────┼───┤
│Global token  │2│MINI_UCC      │129│
└──────────────┴─┴──────────────┴───┘
```

Compress/Decompress Vector of Short Integers R←X(219⌶)Y

In this section, the term *sint_vector* is used to refer to a simple integer vector whose items are all in the range `¯128` to `127`, that is, they are type 83. For further information see *Section 6.4.33*.

In most cases this I-Beam functionality will be used in conjunction with `220⌶` (Serialise/Deserialise Array). However, it may be possible to pass the raw compressed data to and from other applications.

`X` specifies the operation to be performed, either compression or decompression, the compression library to be used, and any optional parameters. `Y` contains the data to be operated on.

Compression

`Y` must be a *sint_vector*.

`R` is a two item vector, each of which is a *sint_vector*. `R[1]` describes the compression, and `R[2]` contains the raw data which is the result of applying the compression library to the input data `Y`.

`X` is specified as follows:

`X[1]`	`X[2]`	Compression Library
1	n/a	LZ4
2	0 .. 9	zlib
3	0 .. 9	gzip
4	n/a	LZ4 with frames (compresses arrays >2GB)

If LZ4 compression is required, then `X` must either be a scalar or a one element vector. Otherwise, `X[2]`, if present, specifies the compression level; higher numbers produce better compression, but take longer.

Decompression

`R` is a *sint_vector*, containing the output of applying the decompression library to the input data, `Y`.

If `X` is a scalar or a one item vector, and has the value 0, then `Y` must be a vector of two items which is the result of previously calling `219⌶` to compress a *sint_vector*.

Otherwise, `X` is a scalar or one or two element vector and `Y` must be a *sint_vector*.

The first element of `X` must be one of the following values.

`X[1]`	Compression Library
`¯1`	LZ4
`¯2`	zlib
`¯3`	gzip

The second, optional, element of X specifies the length of the uncompressed data. Its presence results in a more efficient use of the compression library.

X may not be a two item vector whose first item has the value 0.

Examples

```
      sint←{⍵-256×⍵>127}
      utf8←'UTF-8'⎕UCS
      str←'empty←⍬'
      ⎕←v←sint utf8  str
101 109 112 116 121 ¯30 ¯122 ¯112 ¯30 ¯115 ¯84
      ⎕←comp←1 (219⌶) v
8 ¯55 1 0 0 0 0 11  ¯80 101 109 112 116 121 ¯30 ¯122 ¯112 ¯30 ¯115
¯84

      utf8 256| 0(219⌶)comp
empty←⍬
      utf8 256| ¯1(219⌶)2⊃comp
empty←⍬
```

Serialise/Deserialise Array R←X(220⌶)Y

In this section, the term *sint_vector* is used to refer to a simple integer vector whose items are all in the range ¯128 to 127, that is, they are type 83. For further information see *Section 6.4.33*.

It is expected that in many cases this I-Beam functionality will be used in conjunction with 219⌶ - Compress/Decompress vector of short integers. It would also be possible to encrypt the serialised form and write to a file (either component or native), and reverse the process at a later date.

X is a scalar which can take the value 0 or 1.

When X is 1, Y can be any array. The result R is the serialised form of the array, presented as a *sint_vector*.

When X is 0, Y must be a *sint_vector*. The result R is an array whose serialised form is represented by Y.

Typically it is not possible to construct a vector which can be deserialised; it is expected that the only source of a vector which can be deserialised is the result of using 1(220⌶) to serialise an array.

The result of `1(220⌶)` will differ between interpreters of differing widths and editions, but the resulting vector can be deserialised in other interpreters, with the exception that, like arrays in component files, it may not be possible to deserialise an array which was serialised in a later interpreter

The following identity holds true:

```
      A≡ 0(220⌶) 0(219⌶) 1(219⌶) 1(220⌶) A
```

EXAMPLE

```
      a←'ab'
      b←1(220⌶)a
      b
¯33 ¯108 5 0 0 0 31 39 0 0 2 0 0 0 97 98 0 0
      c←0(220⌶)b
      c≡a
1
```

Compiler Control R←{X}(400⌶)Y

Controls the actions of the Compiler. For further information, see *Compiler User Guide*.

The optional left-argument `X` must be one of the following:

X	Description
`0`	Set automatic compilation options (default)
`1`	Determine whether the function/operator `Y` has been successfully compiled
`2`	Compile the function/operator `Y`
`3`	Discard compiled form of the function/operator `Y`
`4`	Show bytecode for the compiled function/operator `Y`
`nsref`	Compile the function/operator `Y` using user-defined callbacks in this namespace to provide information about global names

The nature of `Y` and `R` depend on the value of `X` as follows:

`X=0` : Control Automatic Compilation (default)

`Y` must be an integer 0, 1, 2, or 3.

`Y`	Description
`0`	disable automatic compilation (initial setting)
`1`	compile functions when they are fixed (with `⎕FX` or from the function editor)
`2`	compile operators the first time they are executed
`3`	compile functions when they are fixed (with `⎕FX` or from the function editor) and compile operators the first time they are executed

The result `R` is the previous value of `Y`.

The automatic compilation setting is maintained within the workspace, and is saved and loaded with the workspace.

`X=1`: Query Compilation State

`Y` must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result `R` is a Boolean scalar or vector, with the value 1 if the corresponding function/operator has been successfully compiled or 0 if it has not.

`X=2`: Compile

`Y` must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names that should be compiled.

The result `R` is a matrix of diagnostic information or, if `Y` was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

1. the APL error number
2. the line number in the function/operator
3. the column number (currently always 0)

4. the error message

If the matrix `R` has zero rows then the compilation was successful.

If this mechanism is used to compile operators, then the compiled bytecode will assume that the operator's operands are functions rather than arrays. At run time, the operands will be checked – if they are functions then the compiled bytecode will be used, otherwise the operator will be interpreted.

`X=3`: Discard Compiled Form

If `Y` is empty, discard any compiled bytecode for all functions and operators in the workspace. If `Y` is a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names, discard any compiled bytecode for the name(s) specified by `Y`. `R` is always 0

`X=4`: Show Bytecode

`Y` must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result `R` is a multi-line string (that is, a character vector with embedded newlines) or, if `Y` was either a matrix or a vector of vectors, a vector of such strings. Each string is a human-readable representation of the bytecode of a compiled function or operator.

This functionality is provided for information and diagnostic purposes only. The human-readable form of the bytecode is subject to change at any time.

`X` is a namespace reference: Compile With Callbacks

`Y` must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names. The specified functions or operators are compiled in the same way as when `X` = 2 except that the compiler uses the user-defined callback functions in the namespace `X` to obtain information about global names. The namespace `X` can contain any or all of following callback functions:

Callback	Description
`quadNC`	analogous to the system function `⎕NC` . When applied monadically to an enclosed character vector it should return the detailed name class of that name. For example, given the name of a global dfn it should return the value 3.2.
`quadAT`	analogous to the system function `⎕AT` . When applied monadically to an enclosed character vector it should return a 1 by 4 matrix whose first item is a vector of 3 integers describing (respectively) the result, function valence and operator valence of the name.
`getValue`	used to obtain the value of global constants. When applied monadically to a character vector that is a global constant it should return the enclose of the constant value, otherwise it returns `⍬` .

Each of these callback functions returns information about names that should be guaranteed to exist when the compiled functions are executed. The compiler assumes that the information returned by the callbacks is correct, and generates bytecode accordingly. In the case of `quadNC` and `quadAT`, if the information returned by the callbacks turns out not to be correct when the compiled function is executed, then a runtime error is generated.

The result `R` is a matrix of diagnostic information or, if `Y` was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

1. the APL error number
2. the line number in the function/operator
3. the column number (currently always 0)
4. the error message

If the matrix `R` has zero rows then the compilation was successful.

Disable Traps `R←600⌶Y`

This function is used to temporarily disable the error trapping mechanism used by `:Trap` and `⎕TRAP`. This can be useful in debugging applications.

`Y` is an integer 0, 1 or 2 as shown in the following table.

`R` is the previous value (0, 1, or 2) of the trap state.

Y	Effect
0	Enable all traps.
1	Disable all traps.
2	Disable traps in suspended functions from triggering when an error is generated in the Session.

Note that the *Disable traps in session* option of the Session *Options* menu performs the same tasks as `(600⌶0)` and `(600⌶2)`.

For error-guards in dfns `600⌶0` and `600⌶2` are equivalent; in neither case is an error generated in the session caught by an error guard in a suspended dfn.

Temporary Directory R←739⌶Y

Returns the name of a system temporary directory suitable for user files, as a character vector. The name reported does not include a trailing directory separator

`Y` is 0.

The result `R` is a character vector.

Example (Windows)

```
      739⌶0
C:/Users/Pete/AppData/Local/Temp
```

Example (non-Windows)

```
      739⌶0
/tmp
```

Called Monadically? R←900⌶Y

Identifies how the current function was called. It reports whether the nearest tradfn on the stack was called without a left argument or not.

`Y` may be any array.

The result R is Boolean. 1 means that the nearest tradfn was called monadically; 0 means that it wasn't. If there is no function on the stack, the result is 0.

EXAMPLE

```
     ∇ r←{left}foo right
[1]    r←900⌶0
     ∇
      foo 10
1
      0 foo 10
0
```

List Loaded Libraries R←950⌶Y

Reports the names of the dynamic link libraries that are currently loaded as a result of executing ⎕NA.

Y is an empty vector.

The result R is a vector of character vectors containing the names of all the DLLs or shared libraries that have been explicitly loaded by ⎕NA and are still loaded by virtue of the presence of at least one external function reference.

Examples

```
      )CLEAR
clear ws
      'Aloc'⎕NA'P kernel32|GlobalAlloc U4 P'
      'Free'⎕NA'P kernel32|GlobalFree P'
      'Lock'⎕NA'P kernel32|GlobalLock P'
      'Ulok'⎕NA'U4 kernel32|GlobalUnlock P'
      'Valu'⎕NA'U4 version|VerQueryValue* P <0T >U4 >U4'
      'copy'⎕NA'P msvcrt|memcpy >U4[] P U4'

      950⌶0
 KERNEL32.DLL  VERSION.DLL  MSVCRT.DLL
      )FNS
Aloc    Free    Lock    Ulok    Valu    copy

      )ERASE Aloc    Free    Lock    Valu
      950⌶0
 KERNEL32.DLL  MSVCRT.DLL
      )FNS
Ulok    copy

      )ERASE Ulok
      950⌶0
 MSVCRT.DLL

      )CLEAR
clear ws
      950⌶0
```

Set Shell Script Debug Options R←{X}(1010⌶)Y

Y is an integer that selects options as follows.

If Y is 1, the lines in the script to be echoed to stderr before they are executed.. The optional left argument X specifies a character scalar of vector that prefixes each line of output. If X is omitted, the default is '+'.

If Y is 2, the effect is as if ⎕TRACE was set for every line of every function in the script. In this case the left argument (if any) is ignored.

If Y is 3, it specifies a combination of the above.

The result R is the previous value of the debug options.

Number of Threads R←1111⌶Y

Specifies how many threads are to be used for parallel execution.

If Y has the value 0, R is the number of virtual processors in the machine.

Otherwise, Y is an integer that specifies the number of threads that are to be used henceforth for parallel execution. Prior to this call, the default number of threads is specified by the parameter APL_MAX_THREADS. See *Installation/Configuration: Configuration Parameters*.

Note that whatever the value of Y, Dyalog limits the number of threads to 64. So the effective number of threads is Y⌊64.

R is the previous value.

To set the number of threads to be the same as the number of virtual processors, execute the stamement:

```
      {}1111⌶ 1111⌶0
```

See *Programming: Parallel Execution* and *Section 4.4.9.22*.

Parallel Execution Threshold R←1112⌶Y

Y is an integer that specifies the array size threshold at which parallel execution takes place. If a parallel-enabled function is invoked on an array whose number of elements is equal to or greater than this threshold, execution takes place in parallel. If not, it doesn't.

Prior to this call, the default value of the threshold is specified by an environment variable named APL_MIN_PARALLEL. If this variable is not set, the default is 32768.

R is the previous value.

See *Programming: Parallel Execution* and *Section 4.4.9.21*.

Update Function Time and User Stamp {R}←X(1159⌶)Y

Y is an array of function names in the same format as the right argument of ⎕AT. For further information, see *Section 6.4.9*.

X is an array of function attributes in the same format as the output of ⎕AT.

The shy result R is a vector of numeric items, one per each specified function containing the following values:

0	No change was made; the name is not that of a function, or the function was locked
1	The time and user stamp were updated

Note that the last item of the function timestamp must be set to 0 otherwise `1159 ⌶` will generate a `DOMAIN ERROR`. Additionally, the timestamp must be greater than `1970 1 1 0 0 0 0`.

Example

```
      ]disp ⎕AT'Christmas'
┌→──────┬──────────────────────┬─┬─────────┐
│0 0 0│2013 3 1 11 14 58 0│0│Richard│
└~──→┴~────────────────────→┴─┴─────────┘

      x←⎕AT 'Christmas'
      x[2 4]←(2012 12 25 11 59 0 0)('Santa')
      x (1159⌶) 'Christmas'

      ]disp ⎕AT'Christmas'
┌→──────┬──────────────────────┬─┬──────┐
│0 0 0│2012 12 25 11 59 0 0│0│Santa│
└~──→┴~────────────────────→┴─┴────→┘
```

Format Date-time R←X(1200⌶)Y

Y is a numeric array of any shape, where every element contains a Dyalog Date Number that represents a date between 1 January 0001 and 31 December 9999 in the Proleptic Gregorian Calendar.

X is a character scalar or vector specifying a pattern with which the elements in Y should be formatted.

R is an array of the same shape as Y, whose elements are enclosed character vectors.

Formatting Pattern in `X`

The formatting pattern allows a time number to be converted to a highly user-configurable, plain text format. When a time number is formatted, elements in the result are copies of the format pattern with format sequences replaced by the elements they represent.

The format sequences are intended to be visually reminiscent of the generated text. They use alphabetic characters easily associated with the substitution (e.g. `D`, `M` and `Y` for Day, Month and Year respectively) repeated one or more times to indicate format. As noted below, some sequences allow the first character to be replaced by a _, or the casing to be altered.

Format letter	Length	Meaning	Variations	Example
Year	YY	Without century	YY	19
	YYYY	With century	YYYY	2019
Month	M	1 or 2 digit numeric	M	3
	MM	2 character numeric	MM _M	03 3
	MMM	Abbreviated name	MMM Mmm mmm _mm[1]	MAR Mar mar Mar
	MMMM	Full name	MMMM Mmmm mmmm _mmm[1]	MARCH March march March
Day of month	D	1 or 2 digit numeric	D	4
	DD	2 character numeric	DD _D	04 4
hours	h	1 or 2 digit numeric	h	8
	hh	2 character numeric	hh _h	08 8
minutes	m	1 or 2 digit numeric	m	5
	mm	2 character numeric	mm _m	05 5
seconds	s	1 or 2 digit numeric	s	0
	ss	2 character numeric	ss _s	00 0
fractional seconds	f	1 digit precision	f	5
	ff	2 digit precision	ff	55
	fff	3 digit precision	fff	555
	ffff	4 digit precision	ffff	5555
	fffff	5 digit precision	fffff	55555
	ffffff	6 digit precision	ffffff	555555

Format letter	Length	Meaning	Variations	Example
day of week	d	Numeric (1-7)	d	1
	ddd	Abbreviated name	DDD Ddd ddd _dd[1]	MON Mon mon Mon
	dddd	Full name	DDDD Dddd dddd _ddd[1]	MONDAY Monday monday Monday
ISO week number	w	1 or 2 digit numeric	w	10
	ww	2 character numeric	ww _w	10 10
year of ISO Week number[2]	WW	Without century	WW	19
	WWWW	With century	WWWW	2019
day of year	y	1 to 3 digit numeric	y	63
	yy	3 character numeric	yy _y	063 63
Ordinal indicator[3] for day of month	O	Short	O o	T t
	OO	Full	OO Oo oo	TH Th th
hours in twelve hour clock	t	1 or 2 digit numeric	t	8
	tt	2 character numeric	tt _t	08 8
AM/PM Indicator	P	Short	P p	A a
	PP	Full	PP pp	AM am

Footnotes

1. Natural sentence case, which may be specified for M(month name) and d(day name) only, causes the text to be substituted in the case which is natural for the language; some languages (for example, English) always capitalise the first letter of day and month names whereas others (for example, French) do not.
2. Dates at the start of the year may be in the final week of the previous year, and dates at the end of the year may be in the first week of the following year.
3. An ordinal indicator is a character or group of characters following a numeral, such as (in English) the suffixes -st, -nd, -rd, -th as in 1st, 2nd, 3rd, 4th.

The upper and lower case letters, underscore _, dollar `$` and percent `%` are all reserved for introducing format sequences, even though not all currently have meaning. The remaining, non-reserved, characters are copied to the result unchanged, thus the format string `hh:mm` represents the hour of the day and minute of the hour with a colon between (e.g. `12:00`). All characters or sequences of characters may be delimited by `"` or `'` at any point in the format string to prevent them being interpreted as a part of a format sequence, and, within these delimiters, two adjacent delimiter characters produce a single delimiter.

Note: The characters `AaaaBbbb` consist of two adjacent format sequences because there is a sequence of As followed by a sequence of Bs. The characters `AaaaAaaa` consist of one format sequence because it only contains `A`s. It can be separated into two format sequences by insering an empty `"` or `'` - delimited string, e.g. `Aaaa""Aaaa`.

Language

Unless overridden, English is used for text substitutions. Different languages may be selected using the Language variant option and/or the use of language specifiers within the format pattern. In either case, the language is specified as either a two letter ISO 639-1 language code in lower case (e.g. en) or as a five character language with an additional underscore and two character region in upper case (e.g. en_GB). Within the format pattern, **xx** (where xx is the two or five character specifier) will switch the language of the subsequent generated text. Dictionaries for the following languages are built in:

ISO 639-1	Language
da	Danish
de	German
el	Greek
en	English
es	Spanish
fi	Finnish
fr	French
it	Italian
ja	Japanese
nb	Norwegian Bokmål
nl	Dutch
nn	Norwegian Nynorsk
pl	Polish
pt	Portuguese
ru	Russian
sv	Swedish
zh	Chinese

Predefined patterns

Any pattern can contain (in part or in whole) a named predefined pattern, which allows common date and time formats to be specified in abbreviated form. Predefined patterns may be specified on a per-language basis, allowing patterns to be tailored for the selected language.

Predefined patterns are included in a pattern using `%` delimiters. For example, `%ISO%` includes the named predefined pattern `ISO`.

The following global predefined pattern is built in:

Name	Substitutes as
ISO	YYYY-MM-DD"T"hh:mm:ss

This list may be expanded in future.

Additional predefined patterns may be defined using the Dictionary variant option. Predefined patterns must not contain references to other predefined patterns.

Variant Options

The Language variant option specifies the language used for formatting datetimes and defaults to `'en'` (English). The option value is a two or five character name (e.g. `'en'` or `'en_GB'`). The setting may be explicitly overridden in the format pattern.

The Dictionary variant option specifies a namespace which contains additional or replacement names for the months etc. and/or predefined patterns, for languages and language regions.

At the top level there may be zero or more sub-namespaces with two or five character names, according to the rules for language and language regions. Within each of these, month names etc. are defined as follows:

Named item	Description
MonthNames	A twelve-element vector of character vectors containing the full names corresponding to January to December, respectively.
ShortMonthNames	A twelve-element vector of character vectors containing the short names corresponding to Jan to Dec, respectively.
WeekdayNames	A seven-element vector of character vectors containing the full names corresponding to Monday to Sunday, respectively.
ShortWeekdayNames	A seven-element vector of character vectors containing the full names corresponding to Mon to Sun, respectively.
MorningAfternoon	A two-element vector of character vectors containing the names corresponding to AM and PM, respectively.
Ordinals	A character vector containing the one ordinal used for all numbers in the range 1 to 31, or a thirty one-element vector of character vectors containing the ordinals for 1 to 31, respectively.

Also at the top level of the dictionary namespace there may be a sub-namespace named Patterns and within this further sub-namespaces named `Global` and/or two or five character language names, containing definitions of predefined patterns. Predefined patterns are defined in the same way as the formatting pattern except that they may not contain references to other predefined patterns.

If the namespace contains a definition which is supplied built into the interpreter, it replaces the built-in one.

If a dictionary is incomplete (e.g. is missing one of the expected named items, or one of the named items contains too few elements) an error is signalled only if the missing content would actually be needed.

Example dictionary

The following creates a dictionary defined by the namespace `dict` using JSON text. See the formatting examples below for uses of this dictionary.

```
dict←(
  Patterns: (
    Global: (
      ISOweek: 'YYYY-"W"ww'
      DateCompact: 'D-MMM-YYYY'
      DateVerbose: '"the date is" DD _mm YYYY'
    )
    fr: ( DateVerbose: '"la date est le" DD mmm YYYY' )
    en_US: ( DateVerbose: '"the date is" Mmm DD, YYYY' )
  )
  en_US: (
    ShortMonthNames: (
      'Jan.' ◇ 'Feb.' ◇ 'Mar.' ◇ 'Apr.' ◇ 'May' ◇ 'June'
      'July' ◇ 'Aug.' ◇ 'Sept.' ◇ 'Oct.' ◇ 'Nov.' ◇ 'Dec.'
    )
  )
  cy: (
    MonthNames: (
      'Ionawr' ◇ 'Chwefror' ◇ 'Mawrth' ◇ 'Ebrill' ◇ 'Mai' ◇ 'Mehefin'
      'Gorffennaf' ◇ 'Awst' ◇ 'Medi' ◇ 'Hydref' ◇ 'Tachwedd' ◇ 'Rhagfyr'
    )
    ShortMonthNames: (
      'Ion' ◇ 'Chw' ◇ 'Maw' ◇ 'Ebr' ◇ 'Mai' ◇ 'Meh'
      'Gor' ◇ 'Awst' ◇ 'Medi' ◇ 'Hyd' ◇ 'Tach' ◇ 'Rhag'
    )
    WeekdayNames: (
      'Dydd Sul' ◇ 'Dydd Llun' ◇ 'Dydd Mawrth' ◇ 'Dydd Mercher'
      'Dydd Iau' ◇ 'Dydd Gwener' ◇ 'Dydd Sadwrn'
    )
    ShortWeekdayNames: (
      'Sul' ◇ 'Llun' ◇ 'Maw' ◇ 'Mer' ◇ 'Iau' ◇ 'Gwen' ◇ 'Sad'
    )
    MorningAfternoon: ( 'yb' ◇ 'yh' )
    Ordinals: (
      'af' ◇ 'il' ◇ 'ydd' ◇ 'ydd' ◇ 'ed' ◇ 'ed' ◇ 'fed' ◇ 'fed' ◇ 'fed'
      'fed' ◇ 'eg' ◇ 'fed' ◇ 'eg' ◇ 'eg' ◇ 'fed' ◇ 'eg' ◇ 'eg' ◇ 'fed'
      'eg' ◇ 'fed' ◇ 'ain' ◇ 'ain' ◇ 'ain' ◇ 'ain' ◇ 'ain' ◇ 'ain'
      'ain' ◇ 'ain' ◇ 'ain' ◇ 'ain' ◇ 'ain'
    )
  )
)
```

Note the following

- In the example, the predefined pattern `ISOweek` is defined globally and is not redefined. It therefore has the same value for all languages. Similarly, `DateCompact` has the same value for all languages, but although the definition is global, it contains the pattern `MMM` and this will be substituted with the month name in the selected language.
- The predefined patterns `DateVerbose` is defined globally, and redefined for languages `fr` and `en_US`. The global definition will be used when any language other than `fr` and `en_US` is selected. If there was not global definition it would only be defined for `fr`, all regional variations of `fr`, and `en_US`.
- There is no explicit definition of patterns or names for language region `en_GB`. If this language is selected the definitions for `en` will be used.
- There is an explicit definition for `ShortMonthNames` for language region `en_US`. If this language is selected the definition of `ShortMonthNames` is as defined, and as for `en` for other names. As `en` is not defined in the dictionary, the built-in defaults are used.

In the following examples:

```
      tn←1 ⎕DT ⊂2019 2 13 10 16 56
      tn
43508.42843
```

English

```
      'Dddd, DDoo Mmmm YYYY; hh:mm:ss' (1200⌶) tn
Wednesday, 13th February 2019; 10:16:56

      '__en__Dddd, DDoo Mmmm YYYY; hh:mm:ss' (1200⌶) tn
Wednesday, 13th February 2019; 10:16:56

      '"ISO date": %ISO%' (1200⌶) tn
ISO date: 2019-02-13T10:16:56

      '%DateVerbose%'(1200⌶⍠'Dictionary'dict) tn
the date is 13 Feb 2019
```

English (US)

```
     fmt←'%DateVerbose%'
     fmt (1200⌶('Dictionary'dict)('Language' 'en_US'))tn
the date is Feb. 13, 2019
```

Danish

```
      '__da__Dddd, DDoo mmmm YYYY; hh:mm:ss' (1200⌶) tn
Onsdag, 13. februar 2019; 10:16:56

      fmt←'Dddd, DDoo mmmm YYYY; hh:mm:ss'
      fmt(1200⌶'Language' 'da') tn
Onsdag, 13. februar 2019; 10:16:56
```

Welsh (using the dictionary defined above)

```
      fmt←'__cy__Dddd, DDoo mmmm YYYY; hh:mm:ss'
      fmt (1200⌶'Dictionary' dict) tn
Dydd Mercher, 13eg chwefror 2019; 10:16:56


      '__cy__%DateVerbose%' (1200⌶'Dictionary' dict) tn
the date is 13 Chw 2019
```

aplcore Parameters R←1302⌶Y

Sets the aplcore parameters **AplCoreName** and/or **MaxAplCores** for the current process.

Y may be:

- a simple character vector that specifies AplCoreName
- a simple integer that specifies MaxAplCores
- a 2-element nested vector containing new values for AplCoreName and MaxAplCores in that order
- an empty vector

R is a 2-element nested vector containing the old values.

If Y is empty, the function simply returns the values of these parameters without changing them.

See also: *Installation/Configuration: Aplcorename* and *Installation/Configuration: Maxaplcores*

Hash Array R←{X}1500⌶Y

This function creates a hashed array, returns an unhashed copy of an array or reports the state of hashing of an array.

Y may be any array.

If X is omitted, the result R is a copy of Y that has been invisibly marked as hashed. R behaves the same as Y in all respects. The only difference is that dyadic ⍳ and related functions are expected to run faster when applied to a hashed array. The *hash* will be created the first time the array is used as an argument to ⍳ or other set functions. The *hashed* property is preserved across assignments and argument passing, but in general is not preserved by any primitive functions.

If X is 1, the result R returns an indication of whether Y has been marked for hashing or whether the hash has been created:

R	**State of** Y
0	Y has not been marked for hashing
1	Y has been marked for hashing, but the hash tables has not yet been created
2	Y has a hash table

If X is 2, the result R is the unhashed form of Y.

Examples

```
      R←1500⌶1 2 3    ⍝ R is marked for hashing

      1 (1500⌶)R
1
      S←R             ⍝ S is marked for hashing
      {⍵⍳2 3 5}R      ⍝ R is now hashed
      1 (1500⌶)R
2
      U←(⍴R)⍴R        ⍝ U is not hashed
      U←⊃⊂R           ⍝   ditto
      1 (1500⌶)U
0
```

If `R` is a hashed array then certain forms of modified assignment will preserve and efficiently update the hash table:

```
      R,←Y    ⍝ only for scalar or vector R
      R⍪←Y
      R↓⍨←Y   ⍝ only for negative singleton Y
```

Examples

```
      R←1500⌶1 2 3 ⍝ R is hashed

      R,←5    ⍝ ,← preserves and updates
              ⍝ the hash table
      R
1 2 3 5
      R⍳2 4 6
2 5 5

      R↓⍨←¯2  ⍝ ↓⍨← preserves and updates
              ⍝ the hash table
      R
1 2
      R⍳2 4 6
2 3 3
```

The *hashed* property survives `)SAVE`/`)LOAD` and `)SAVE`/`)COPY`. It does not currently survive writing to a component file and reading back again.

Memory Manager Statistics `R←{X}(2000⌶)Y`

This function returns information about the state of the workspace and provides a means to reset certain statistics and to control workspace allocation. This I-Beam is provided for performance tuning and is VERY LIKELY to change in the next release. See also *Installation/Configuration: Workspace Management*.

`Y` is a simple integer scalar or vector containing values listed in the table below.

If `X` is omitted, the result `R` is an array with the same structure as `Y`, but with values in `Y` replaced by the following statistics. For any value in `Y` outside those listed below, the result is undefined.

Value	Description
0	Workspace available (a "quick" ▯WA).
1	Workspace used.
2	Number of compactions since the workspace was loaded.
3	Number of garbage collections that found garbage.
4	Current number of garbage pockets in the workspace.
9	Current number of free pockets in the workspace.
10	Current number of used pockets in the workspace.
12	Sediment size.
13	Current workspace allocation, that is, the amount of memory that is actually being used.
14	Workspace allocation high-water mark, that is, the maximum amount of memory that has been allocated since the workspace was loaded or since this count was reset.
15	Limit on minimum workspace allocation.
16	Limit on maximum workspace allocation.
19	The number of calls to ▯WA or 2002I since the last time 2000I was called, or when the process started.
20	The requested size of the WS Full Buffer , that is, the amount of workspace requested for handling WS FULL errors.
21	The actual size of the WS Full Buffer .
22	The number of WS FULL handlers that are currently running.
23	The total number of WS FULL errors that have occurred.
24	The total number of WS FULL errors that have been trapped.

Note: While all other operations are relatively fast, the operation to count the number of garbage pockets (4) may take a noticeable amount of time, depending upon the size and state of the workspace.

See also *Section 4.4.9.28*.

Examples

```
      2000⌶0
55414796
      2000⌶0,⍳16   ⍝ with MAXWS=95G
1.02004292E11 1181312 1 1 0 ¯1 ¯1 ¯1 ¯1 78 13280 ¯1 1180800 1595016496 1
595042464 0 1.020054733E11
```

If `X` is specified, it must be either a simple integer scalar, or a vector of the same length as `Y`, and the result `R` is `⍬`. In this case, the value in `Y` specifies the item to be set and `X` specifies its new value according to the table below.

Value	Description
2	0 resets the compaction count; no other values allowed.
3	0 resets the count of garbage collections that found garbage; no other values allowed.
14	0 resets the workspace allocation high-water mark; no other values allowed. This should be called following a call to `⎕WA` (which compacts the workspace and returns unused memory to the operating system).
15	Sets the minimum workspace allocation to the corresponding value in `X` ; must be between 0 and the current workspace allocation.
16	Sets the maximum workspace allocation to the corresponding value in `X` ; 0 implies **MAXWS** otherwise must be between the current workspace allocation and **MAXWS** .
19	0 resets the compaction count; no other values allowed.
20	Sets the requested size of the `WS Full Buffer` to the value specified by `X` . The actual space allocated may be less than that requested.

Notes

- The workspace allocation high-water mark indicates a minimum value for **MAXWS**.
- Limiting the maximum workspace allocation can be used to prevent code that reserves as much workspace as it can from skewing the peak usage result.
- Limiting the minimum workspace allocation can avoid repeatedly committing and releasing memory to the Operating System when memory usage is fluctuating.

Examples

```
      2000⌶2 3
6 0 33216252
      0 (2000⌶)2 3 14 ⍝ Reset compaction count

      2000⌶2 3
0 0
      30000000 40000000(2000⌶)15 16 ⍝ Restrict min/max ws

      (2000⌶)15 16
30000000 40000000

      0 (2000⌶)15 16 ⍝ Reset min/max ws

      (2000⌶)15 16
0 65536000
```

```
      (2000⌶)13 14 ⍝ Current, peak WS allocation
4072532 4072532

      a←10e6⍴'x' ⍝ Increase WS allocation

      (2000⌶)13 14 ⍝ Current, peak WS allocation
15108580 15108580

      ⎕EX 'a' ⋄ {}⎕WA ⍝ Decrease current WS allocation

      (2000⌶)13 14 ⍝ Current, peak WS allocation
1962856 15108580

      0 (2000⌶) 14 ⍝ Reset High-water mark

      (2000⌶)13 14 ⍝ Current, peak WS allocation
1962856 1962856
```

WS Full Handling

Potentially, a `WS FULL` error represents a terminal condition that would prevent a program from continuing because the process has, quite literally, run out of memory.

To alleviate the problem,. Dyalog reserves a special *WS Full Buffer* for handling `WS FULL` errors. The default size of this buffer is `(1MB)⌊(0.01×⎕WA)`.

In simple terms, when a `WS FULL` error occurs that triggers a handler, that is, an expression executed via `⎕TRAP` or `:Trap`, the reserved workspace in the *WS Full Buffer* is released to provide additional memory space for that expression to execute. When the expression terminates, the system removes the memory that it had previously released, reserving it once more for another potential `WS FULL`.

Note that until a `WS FULL` handler starts, the memory allocated to the *WS Full Buffer* is unavailable and inaccessible for any other purpose, thereby reducing the amount of active workspace available (`⎕WA`).

Further considerations are:

- Multiple `WS FULL` handlers can run concurrently as a result of muti-threading or nesting (when a `WS FULL` handler itself generates a `WS FULL` error).
- When the `WS Full Buffer` is restored when the handler (more accurately, the last handler) terminates, or when a saved workspace is re-loaded, there may be insufficient memory available. In these circumstances, the system allocates a reduced amount, without reporting an error. However, the system will later try to reclaim more (up to the desired amount), if more workspace has become free. The desired and actual sizes of the `WS Full Buffer` are reported by `(2000⌶)20` and `(2000⌶)21` respectively.
- When a `WS FULL` handler is activated and the `WS Full Buffer` is freed, `(2000⌶)21` will return 0 until the handler terminates.

Specify Workspace Available R←2002⌶Y

This function is identical to the system function `⎕WA` except that it provides the means to specify the amount of memory [1] that is *committed* for the workspace rather than have it assigned by the internal algorithm. Committed memory is memory that is allocated to a specific process and thereby reduces the amount of memory available for other applications. See also *Installation/Configuration: Workspace Management*.

Like `⎕WA`, `2002⌶` compacts the workspace so that it occupies the minimum number of bytes possible, adds an *extra amount*, and then de-commits all the remaining memory that it is currently using, allowing it to be allocated by the operating system for use by other applications.

The argument `Y` is an integer which specifies the size, in bytes, of this *extra amount*.

1 The term memory here means virtual memory which includes memory mapped to disk.

The purpose of the *extra amount* is to reduce the likelihood that APL will immediately have to ask the operating system to re-commit memory that it has just de-committed, something that would have a deleterious effect on performance. At the same time, if the *extra amount* were to be excessively large, APL could starve other applications of memory which itself could reduce the effective performance of the system. Whereas `⎕WA` calculates the size of the *extra amount* using a simple internal algorithm, `2002⌶` uses a value specified by the programmer.

`R` is an integer which reports the size in bytes of the memory committed for the workspace, and is the sum of the minimum amount required by the workspace itself and the argument `Y`.

If the size of the committed workspace would be smaller than the minimum value (specified by `2000⌶`) or larger than the maximum value (which defaults to `MAXWS`), a `DOMAIN ERROR` is signalled.

See also *Section 4.4.9.27*.

Note that this function does not change the size of the *extra amount* that will be applied subsequently by `⎕WA` or by an automatic compaction.

Disable Global Triggers `R←2007⌶Y`

This function is used to temporarily disable and re-enable *Programming: Global Triggers*.

`Y` is Boolean.

`R` is the previous value setting.

`Y`	**Effect**
`0`	Enable Global Triggers.
`1`	Disable Global Triggers.

This function has effect only in the current thread and its effect is immediate. If there are pending triggers when triggers are disabled, those pending will be queued and fired when triggers are re-enabled.

Update DataTable `R←{X}2010⌶Y`

> **Note**
>
> **.NET Framework only**

This function performs a *block update* of an instance of the ADO.NET object System.Data.DataTable. This object may only be updated using an explicit row-wise loop, which is slow at the APL level. `2010⌶` implements an *internal* row-wise loop which is much faster on large arrays. Furthermore, the function handles NULL values and the conversion of internal APL data to the appropriate .NET datatype in a more efficient manner than can be otherwise achieved. These 3 factors together mean that the function provides a significant improvement in performance compared to calling the row-wise programming interface directly at the APL level.

`Y` is a 2, 3 or 4-item array containing:

1. A reference to an instance of System.Data.DataTable.
2. A matrix with the same number of columns as the table specified by `Y[1]`.
3. An optional vector which specifies for each column in the DataTable the values in `Y[2]`which should be converted to a System.DBNull.
4. An optional vector which specifies the indices (in zero origin) of the rows of the DataTable which are to be updated. If omitted, the matrix specified by `Y[2]` will be *appended* to the DataTable.

The optional argument `X` is Boolean vector, where a 1 indicates that the corresponding column of `Y[2]` is a string from which the new values should be converted according to that column's data type.

Example

Shown firstly for comparison is the type of code that is required to update a DataTable by looping:

```
      ⎕USING←'System' 'System.Data,system.data.dll'
      dt←⎕NEW DataTable
      ac←{dt.Columns.Add ⍺ ⍵}
      'S1' 'S2' 'I1' 'D1' ac¨String String Int32 DateTime
 S1  S2  I1  D1

      NextYear←DateTime.Now+{⎕NEW TimeSpan (4↑⍵)}¨⍳n←365
      data←(↕¨⍳n),(n⍴'odd' 'even'),(10|⍳n),⊂NextYear
      ¯2 4↑data
 364  even  4  18-01-2011 14:03:29
 365  odd   5  19-01-2011 14:03:29

      ar←{(row←dt.NewRow).ItemArray←⍵ ⋄ dt.Rows.Add row}
      t←3⊃⎕AI ⋄ ar¨↓data ⋄ (3⊃⎕AI)-t
449
```

This result shows that this code can only insert roughly 800 rows per second (`3⊃⎕AI` returns elapsed time in milliseconds), because of the need to loop on each row and perform a noticeable amount of work each time around the loop.

`2010⌶` does all the looping in compiled code:

```
      dt.Rows.Clear ⍝ Delete the rows inserted above
      SetDT←2010⌶
      t←3⊃⎕AI ⋄ SetDT dt data ⋄ (3⊃⎕AI)-t
4
```

So in this case, using `2010⌶` achieves over 90,000 rows per second.

DateTime columns

Creating large arrays of DateTime objects in the workspace takes additional resources, and unless the data is already stored that way, it is not necessary to convert it to .NET objects. Data in `⎕TS` format (7-element integer vector) or in a suitable character format may be used directly. The former is a specific Dyalog optimisation; the latter a feature of .NET Version 4.0. The following examples use numeric and character data for the dates:

```
   months←12⍴31 ⋄ months[2 4 6 9 11]←29 30 30 30 30
   n←⍴NextYear←7↑¨⊃,/(⍳12){(⊂2016,⍺),¨⍳⍵}¨months
   data←(↕¨⍳n),(n⍴'odd' 'even'),(10|⍳n),⊂NextYear
   SetDT dt data
```

```
n←⍴NextYear←⊃,/(⍳12){(⊂'2016/',(⍕⍺),'/'),∘⍕¨⍵}¨months
data←(⍕¨⍳n),(n⍴'odd' 'even'),(10|⍳n),⍪NextYear
SetDT dt data
```

Using Strings

In circumstances where .NET fails to accept character data automatically, it is possible to force conversion from character format to the corresponding .NET type.

If specified, the optional left argument `X` instructs the system to pass the corresponding columns of data to the Parse() method of the data type for those columns prior to performing the update.

In the following example, the left argument is not strictly necessary using .NET Version 4.0, but forces parsing for the data in the 4th column:

```
months←12⍴31 ⋄ months[2 4 6 9 11]←29 30 30 30 30
n←⍴NextYear←⊃,/(⍳12){(⊂'2016/',(⍕⍺),'/'),∘⍕¨⍵}¨months
data←(⍕¨⍳n),(n⍴'odd' 'even'),(10|⍳n),⍪NextYear
0 0 0 1 SetDT dt data
```

Handling Nulls

If applicable, `Y[3]` is a vector with as many elements as the DataTable has columns, indicating the value that should be converted to `System.DBNull` as data is written. For example, using the same DataTable as above:

```
      t
<null>  odd    1  21-01-2010 14:50:19
two     even   2  22-01-2010 14:50:19
three   odd   99  23-01-2010 14:50:19

      dt.Rows.Clear ⍝ Clear the contents of dt
      SetDT dt t ('<null>' 'even' 99 '')
```

Above, we have declared that the string `'<null>'` should be considered to be a null value in the first column, `'even'` in the second column, and the integer `99` in the third.

Updating Selected Rows

Sometimes, you may have read a very large number of rows from a DataTable, but only want to update a single row, or a very small number of rows. Row indices can be provided as the fourth element of the argument to `2010⌶`. If you are not using `Y[3]` explicitly, you can just use an empty vector as a placeholder. Continuing from the example above, we could replace the first row in our DataTable using:

```
      SetDT←2010⌶
      SetDT dt (1 4⍴'one' 'odd' 1 DateTime.Now) ⍬ 0
```

> **Note**
>
> - `Y[2]` must be provided as a matrix, even if you only want to update a single row,
> - `Y[4]` specifies row indices using zero origin (the first row has number 0).
> - `Y[2]` must be provided as a matrix, even if you only want to update a single row,
> - `Y[4]` specifies row indices using zero origin (the first row has number 0).

> **Warning**
>
> If you are experimenting with writing to a DataTable, note that you should call `dt.Rows.Clear` each time to clear the current contents of the table. Otherwise you will end up with a very large number of rows after a while.

Read DataTable `R←{X}2011⌶Y`

> **Note**
>
> **.NET Framework only**

This function performs a *block read* from an instance of the ADO.NET object System.Data.DataTable. This object may only be read using an explicit row-wise loop, which is slow at the APL level. `2011⌶` implements an *internal* row-wise loop which is

much faster on large arrays. Furthermore, the function handles NULL values and the conversion of .NET data types to the appropriate internal APL form in a more efficient manner than can otherwise be achieved. These 3 factors together mean that the function provides a significant improvement in performance compared to calling the row-wise programming interface directly at the APL level.

`Y` is a scalar or a 1 or 2-item array containing:

1. A reference to an instance of `System.Data.DataTable`.
2. An optional vector which specifies the values to which a `System.DBNull` should be mapped in the corresponding columns of the result

The result `R` depends upon the value of the Variant option Invert. This the primary option with a default value of 0.

Invert Option (Boolean)

0	The result `R` is a matrix with the same shape as the DataTable referenced by `⊃Y`.
1	The result `R` is vector whose length is the same as the number of columns in the DataTable referenced by `⊃Y`.

The optional left argument `X` is a numeric vector with the same length as the number of columns in the result in the DataTable referenced by `⊃Y`:

1	Specifies that the corresponding column of the result should be converted to a string using the `ToString` method of the data type of column in question.
2	Specifies that numbers of type `System.Int64` in the corresponding column of the result should be converted to DECFs (NOT into .NET objects, which is the default)
4	Specifies that if the type of the corresponding column is `System.String` the entire column should be returned as a character matrix rather than a vector of character vectors. Any nulls will be replaced with a row of spaces. This applies only when **Invert** is 1.
5	Combines 1 and 4.

EXAMPLES

```
      ⎕USING←'' 'System.Data,system.data.dll'

      dt←⎕NEW DataTable

      add_col←{col←⍺.Columns.Add ⍬ ⋄ col.DataType←⍵}
      dt add_col System.String
      dt add_col System.Int32
      dt add_col System.Int64
```

```
      in←⍉↑('One' 'Two')(1 2)(6401 6402)
      2010⌶ dt in

      ⎕←out←2011⌶ dt
┌───┬─┬──────┐
│One│1│ 6401 │
├───┼─┼──────┤
│Two│2│ 6402 │
└───┴─┴──────┘
      out[;3].GetType
System.Int64  System.Int64
```

```
      0 0 2(2011⌶) dt ⍝ Convert 3rd col to DECF
┌───┬─┬────┐
│One│1│6401│
├───┼─┼────┤
│Two│2│6402│
└───┴─┴────┘
```

```
      1 1 1(2011⌶)dt ⍝ Convert all values to text
┌───┬─┬────┐
│One│1│6401│
├───┼─┼────┤
│Two│2│6402│
└───┴─┴────┘
```

```
      ((2011⌶)⌸('Invert' 1)) dt
┌─────────┬─┬──────────────┐
│┌───┬───┐│1 2│ 6401  6402 │
│|One|Two|||   │            │
│└───┴───┘│   │            │
└─────────┴─┴──────────────┘
```

```
      4 0 0((2011⌶)⌹('Invert' 1))dt
┌─────┬─┬─┬──────┬──────┐
│One  │1│2│ 6401 │ 6402 │
│Two  │ │ │      │      │
└─────┴─┴─┴──────┴──────┘
```

```
      5 5 5((2011⌶)⌹('Invert' 1))dt ⍝ Convert to cmats
┌─────┬─┬────┐
│One  │1│6401│
│Two  │2│6402│
└─────┴─┴────┘
```

Handling Nulls

```
      2010⌶dt(1 3⍴⎕NULL) ⍝ Add a row of nulls
      ⎕←out←2011⌶ dt
┌─────┬─┬──────┐
│One  │1│ 6401 │
├─────┼─┼──────┤
│Two  │2│ 6402 │
├─────┼─┼──────┤
│     │ │      │
└─────┴─┴──────┘
      out[3;].GetType
System.DBNull  System.DBNull  System.DBNull
```

```
      2011⌶ dt ('this is null' 'this too' 'and this')
┌──────────┬────────┬─────────┐
│One       │1       │ 6401    │
├──────────┼────────┼─────────┤
│Two       │2       │ 6402    │
├──────────┼────────┼─────────┤
│this is null│this too│and this│
└──────────┴────────┴─────────┘
```

Performance Considerations

First for comparison is shown the type of code that is required to read a DataTable by looping:

```
      t←3⊃⎕AI ⋄ data1←↑(⎕dt.Rows).ItemArray ⋄ (3⊃⎕AI)-t
191
```

The above expression turns the `dt.Rows` collection into an array using `⎕`, and *mixes* the ItemArray properties to produce the result. Although here there is no explicit loop, involved, there is an implicit loop required to reference each item of the collection in succession. This operation performs at about 200 rows/sec.

`2011⌶` does the looping entirely in compiled code and is significantly faster:

```
      GetDT←2011⌶
      t←3⊃⎕AI ⋄ data2←GetDT dt ⋄ (3⊃⎕AI)-t
25
```

In the first case, `2011⌶` created 365 instances of System.DateTime objects in the workspace. If we are willing to receive the timestamps in the form of strings, we can read the data almost an order of magnitude faster:

```
      t←3⊃⎕AI ⋄ data3←0 0 0 1 GetDT dt ⋄ (3⊃⎕AI)-t
3
```

The left argument to `2011⌶` allows you to flag columns which should be returned as the `ToString()` value of each object in the flagged columns. Although the resulting array looks identical to the original, it is not: The fourth column contains character vectors:

```
      ¯2 4↑data3
 364  even  4  18-01-2011 14:03:29
 365  odd   5  19-01-2011 14:03:29
```

Depending on your application, you may need to process the text in the fourth column in some way – but the overall performance will probably still be very much better than it would be if DateTime objects were used.

Remove Data Binding R←2014⌶Y

> **Note**
>
> **.NET Framework only**

This function disassociates a data-bound variable from its data binding source.

Y is any array.

If Y or an element of Y is a character vector that contains the name of a data-bound variable, that variable is dissociated from its data binding source.

The result R is always 1.

EXAMPLE

```
      2014⌶'txtSource'
1
```

Create Data Binding Source R←{X}2015⌶Y

> **Note**
>
> **.NET Framework only**

Creates an object that may be used as a data source for WPF data binding. [2]

This function connects a *Binding Target* to a *Binding Source*. In WPF a Binding Target is a particular property of a user interface object; for example, the Text property of a TextBox object. A Binding Source is a *Path* to a value in a data object (which may contain other values). The value of the Binding Source determines the value of the Binding Target. If two-way binding is in place, a change in a user-interface component causes the bound data value to change accordingly. In the example of the TextBox, the value in the Binding Source changes as the user types into the TextBox.

Y is a character vector containing one of the following:

- the name of a variable
- the name of a namespace containing one or more variables
- the name of a variable containing a vector of refs to namespaces, each of which contains one or more variables.

If the name specified by Y doesn't exist or represents neither a variable nor a namespace, the function reports DOMAIN ERROR. Currently, no further validation of the

[2] It is beyond the scope of this document to fully explain the concepts of WPF data binding. See Microsoft Developer Network, Data Binding Overview.

structure and contents of `Y` is performed, but nothing other than the examples described herein is supported.

If the optional left argument `X` is given and `Y` is a variable other than a ref, `X` specifies the binding type for that variable. If `Y` specifies one or more namespaces, `X` specifies the names and binding types of each of the variables which are to be bound, contained in the namespaces specified by `Y`.

The structure of `X` depends upon the structure of `Y` and is discussed later in this topic.

If `X` is omitted, all of the variables specified by `Y` are bound with default binding types.

Here the term *bind variable* refers to any variable specified by `X` and `Y` to be bound, and the term *binding type* means the .NET data type to which the value of the bind variable is converted before it is passed to the .NET interface.

`2015⌶` creates a Binding Source object `R`. This is a .NET object which contains *Path*(s) to one or more bind variables. This object may then be assigned to a property of a WPF object or passed a `s` as a parameter to a WPF method that requires a Binding Source.

Bind Variables and Bind Types

A bind variable should be of rank 2 or less. Higher rank arrays are not supported.

If not specified by `X`, the binding type of a bind variable is derived from its content at the time `2015⌶` is executed. The binding type is then stored with the variable in the workspace. There is no mechanism to change a variable's binding type without erasing the variable and re-executing `2015⌶`. If you change the type or rank of a bind variable while it is bound (for example from a variable to a namespace), the behaviour of the system is unpredictable.

The default binding type is derived as follow:

If the bind variable is a simple scalar number the default binding type is System.Object. At the point when the value of the variable is passed to the .NET interface this will be cast to a numeric type such as System.Int16, System.Int32, System.Int64, or System.Double, depending upon the internal representation of the data. The .NET property to which it is bound will typically only accept a single Type (for example System.Int32), so to avoid unpredictable behaviour, it is recommended that the left argument `X` be used to specify the binding type for numeric data.

If the bind variable is a character scalar or vector, the default binding type is also System.Object, but at the point when the value of the variable is passed to the .NET interface it will always be passed as System.String, which is suitable for binding to any property that accepts a System.String, such as the Text property of a TextBox.

If the bind variable is a vector other than a simple character vector, such as a vector of character vectors, a simple numeric vector, or a vector of .NET objects, the bind type will be a collection. This is suitable for binding to any property that represents a collection (list) of items, for example the ItemsSource property of a ListBox.

If the bind variable is a matrix, the default binding type is System.Object.

All the examples that follow assume `⎕USING←'System'`.

Binding Single Variables

In this case, `Y` specifies the name of a variable which is one of the following:

- character vector (or scalar)
- numeric scalar
- scalar .NET object (not currently supported)
- vector of character vectors
- numeric vector
- vector of .NET objects
- matrix

`X` (if specified) defines the binding type for the bind variable named by `Y` and is a single .NET Type.

Note that in the following examples, the reason for expunging the name first is discussed in the section headed Rebinding a Variable.

Binding a Character Vector

This example illustrates how to bind a variable which contains a character vector.

```
      ⎕EX'txtSource'
      txtSource←HELLO WORLD'
      bindsource←2015⌶'txtSource'
```

In this example, the binding type of the variable `txtSource` will be System.String, suitable for binding to any property that accepts a String, such as the Text property of a `TextBox`.

Binding a Numeric Scalar

This example illustrates how to bind a variable which contains a numeric scalar value.

```
      ⎕EX'sizeSource'
      sizeSource←36
      bindSource←Int32(2015⌶)'sizeSource'
```

In this example, the left argument `Int32` specifies that the binding type for the variable sizeSource is to be System.Int32. This means that whenever APL passes the value of `sizeSource` to the control, it will first be cast to an Int32. This makes it suitable, for example, for binding to the FontSize property of a TextBox.

A number of controls have a Value property which must be expressed as a System.Double. The next example shows how to create a Binding Source for such a variable.

```
      ⎕EX'valSource'
      valSource←42
      bindSource←Double(2015⌶)'valSource'
```

Binding a Scalar .NET Object

This is currently not supported.

Binding a Vector of Character Vectors

WPF data binding provides the means to bind controls that display lists of items, such as the ListBox, ListView, and TreeView controls, to collections of data. These controls are all based upon the ItemsControl class. To bind an ItemsControl to a collection object, you use its ItemsSource property.

This example illustrates how to bind a variable which contains a vector of character vectors.

```
      ⎕EX'itemsSource'
      itemsSource←'beer' 'wine' 'water'
      bindsource←2015⌶'itemsSource'
```

In this example, the binding type of the variable `itemsSource` will be System.Collection, suitable for binding to the ItemSource property of an ItemsControl.

Binding a Numeric Vector

By default, a numeric vector is bound in the same way as a vector of character vectors, that is, as a System.Collection, suitable for binding to the ItemSource property of an ItemsControl.

```
      ⎕EX'yearsSource'
      yearsSource←2000+⍳20
      bindSource←2015⌶'yearsSource'
```

In principle, a numeric vector may alternatively be bound to a WPF property that requires a 1-dimensional numeric array, by specifying the appropriate data type (for example, Int32, Double) for the array as the left argument. For example:

```
      ⎕EX'arraySource'
      arraySource←42 24
      bindSource←Int32 (2015⌶)'arraySource'
```

Binding a Vector of .NET Objects

A vector of .NET objects is bound in the same way as a vector of character vectors, that is, as a System.Collection, suitable for binding to the ItemSource property of an ItemsControl.

```
      ↑Easter
2015 4 12
2016 5  1
2017 4 16
2018 4  8
2019 4 28
2020 4 19
2021 5  2
2022 4 24
2023 4 16
2024 5  5
      dt←{⎕NEW DateTime ⍵}¨Easter
      bindSource←2015⌶'dt'
```

Note that, as a specific optimisation for binding DateTime data, it is not necessary to create `DateTime` objects in the workspace. Instead, the data may be represented by 7-element integer vectors (`⎕TS` format) or character strings that can be parsed by the DateTime.Parse(String) method. However, in both cases it is necessary to explicitly specify the binding type to force the data to be converted to `DateTime`, as illustrated by the following examples:

```
      TSEaster←7↑¨Easter
      bindSource←DateTime (2015⌶) 'TSEaster'
```

```
      CharEaster←'2015/4/12' '2016/5/1' '2017/4/16'
      bindSource←DateTime (2015⌶) 'CharEaster'
```

Binding a Matrix

If the bind variable is a matrix, it is bound in a similar way to a vector of namespaces and is discussed below.

Rebinding a Variable

As mentioned earlier, when a variable is bound its binding type is stored with it in the workspace. If you subsequently attempt to rebind the variable there is no mechanism in place to alter the binding type. If the current binding type (whether specified by the left argument `X`, or by being the default) differs from the saved one, the function will generate a `DOMAIN ERROR`.

```
      num←42
      bs←2015⌶'num'

      bs←'Int32'(2015⌶)'num'
DOMAIN ERROR: You cannot redefine the binding types
      bs←'Int32'(2015⌶)'num'
     ^
```

In this example, perhaps the programmer realised after binding `num` (with a default binding type of System.Object) that the binding type should really be System.Int32, and simply was trying to correct the error. To avoid this problem, it is recommended that you expunge the name before using it.

```
      ⎕EX 'num'
      num←42
      bs←2015⌶'num'⍝ (default) binding type System.Object

      ⎕EX 'num'
      num←42
      bs←Int32(2015⌶)'num'
```

Binding A Namespace

In this case, `Y` specifies the name of a namespace that contains one or more variables. By default, each variable is bound using its default binding type as described above. Objects other than variables are ignored.

If it is required to specify the binding type of any of the variables, or if certain variables are to be excluded, the left argument is a 2-column matrix. The first column contains the names of the variables to be bound, and the second column their binding types.

EXAMPLE

The following code snippet binds a namespace containing two variables named `txtSource` and `sizeSource`. In this case, the name of each variable may be specified as the Path for a WPF property that requires a String or an Int32. For example, if bindSource were assigned to the DataContext property of a TextBox, its Text property could be bound to `txtSource` and its FontSize property to `sizeSource`.

```
      src←⎕NS''
      src.txtSource←'Hello World'
      src.sizeSource←36
      options←2 2⍴'txtSource'String'sizeSource'Int32
      bindSource←options(2015⌶)'src'
```

Binding a Vector of Namespaces

In this case, `Y` specifies the name of a variable that contains a vector of refs to namespaces. In this case, the result `R` is of type Dyalog.Data.DataBoundCollectionHandler which is suitable for binding to a WPF property that requires an IEnumerable implementation, such as the ItemsSource property of the DataGrid.

Each namespace in `Y` represents one of a collection of instances of an object, which exports a particular set of properties for binding purposes. For example, `Y` could specify a wine database where each namespace represents a different wine, and each namespace contains the same set of variables that contain the name, price (and so forth) of each wine.

EXAMPLE

```
winelist←⎕NS¨(⍴Wines)⍴⊂''
winelist.Name←Wines
winelist.Price←0.01×10000+?(⍴Wines)⍴10000

bindSource←2015⌶'winelist'
```

Binding a Matrix

Binding a matrix is like binding a vector of namespaces. Each row of `Y` represents one of a collection of instances of an object, which exports a particular set of properties for binding purposes. Each column of `Y` represents one of these properties.

Every row in the datasource will be of the same type (which might not be the case with an array of namespaces), and so the collection is a collection of specific things. The *specific thing* is a .NET type that is created dynamically and has a unique name.

Unlike variables in namespaces, the columns of an APL matrix do not have names which can be exported as properties, so this information must be provided in the left

argument to `(2015⌶)` which also specifies their data types. If the left argument is omitted, the default names are `Column1`, `Column2`, ... and so forth and the default data type is `System.Object`.

So if the right argument of `(2015⌶)Y` is the name of a matrix, the left argument `X` is a matrix with as many rows as there are columns in `Y`. `X[;1]` contains the names by which each of the columns of `Y` will be exported as a property, and `X[;2]` their data types.

Values in the matrix may be scalar numbers, character scalars or vectors, or nested vectors, but each column in the matrix must be uniform.

The result `R` is a specific type that is created dynamically and assigned a unique name of the form Dyalog.Data.DyalogCollectionNotifyHandler`1[Dyalog.Data.DataBoundRow_nnnnnnnn]. This is suitable for binding to a WPF property that requires an IEnumerable implementation, such as the ItemsSource property of the DataGrid.

Example

`mat` is a matrix of numbers and is bound with default property/column names `Column1`, `Column2`, ... `Column10` and the default data type of `System.Object`.

```
      mat←?20 10⍴100
      bindSource←2015⌶'mat'
```

Example

`winelist` is a matrix whose first column contains a list of wines, and whose second column their prices. The left argument is a matrix. Its first column specifies the property names by which the columns will be exported (`'Name'` and `'Price'`) and its second column, their data types (`System.Object`)

```
winelist←Wines,[1.5]0.01×10000+?(⍴Wines)⍴10000
info←(↑'Name' 'Price'),⊂Object

bindSource←info(2015⌶)'winelist'
```

Example

`emp` is a 3-column matrix which contains names, numbers and addresses. Each address is made up of two character vectors containing street and town

```
      emp
┌──────────────────┬──────────────────┬──────────────────┐
│John Smith        │Mary White        │T.W. Penk         │
├──────────────────┼──────────────────┼──────────────────┤
│1                 │2                 │3                 │
├──────────────────┼──────────────────┼──────────────────┤
│ ┌──────┬───────┐ │ ┌──────┬─────┐   │ ┌──────┬───────┐ │
│ │2 East│Headley│ │ │42 Hig│Alton│   │ │23 Wes│Farnham│ │
│ └──────┴───────┘ │ └──────┴─────┘   │ └──────┴───────┘ │
└──────────────────┴──────────────────┴──────────────────┘

      schema
┌───────┬──────────────────────┐
│Name   │(System.Object)       │
├───────┼──────────────────────┤
│Number │(System.Object)       │
├───────┼──────────────────────┤
│Address│ ┌──────┬─────────────┐ │
│       │ │Street│(System.Obje│ │
│       │ ├──────┼─────────────┤ │
│       │ │Town  │(System.Obje│ │
│       │ └──────┴─────────────┘ │
└───────┴──────────────────────┘


      bindSource←schema(2015⌶)'emp'
```

Notification Events

The object `R` generates notification events when the value(s) of the Binding Source are updated as the contents of the Binding Target are changed by the user. These events are generated after the data has changed and there is no mechanism to prevent the change from occurring.

There are two types of event; ElementChanged and CellsChanged. The CellsChanged event applies only to a data bound matrix; the ElementChanged event applies to all other types of binding.

The event message supplied as the right argument to your callback function, is a 2-element vector as follows :

`[1]`	Object	ref
`[2]`	EventArgs	ref

EventArgs is an instance of the internal class `Dyalog.Data.ElementChangedEventArgs` or `Dyalog.Data.CellsChangedEventArgs` whose fields are described below:

`Dyalog.Data.ElementChangedEventArgs` fields

`Indices`	An indication of which member has changed. Typically this will either be `¯1` to indicate that the indices are unavailable or a scalar value indicating (origin 0), which element of an array has been modified or added.
`Name`	The name of the variable that has been modified. This is especially useful when the datasource corresponds to a namespace.
`Path`	A path used to locate the variable that has been modified. This is especially useful when the datasource corresponds to a deeply nested namespace, where the value changed is an element of an array inside a namespace which is itself an element of an array within the datasource.

`Dyalog.Data.CellsChangedEventArgs` fields

`Path`	Identifies the cell or row that was changed. See below.
`SourceName`	The name of the matrix that was specified as the right argument to `2015⌶`.
`Reason`	A character vector that describes what in the matrix has changed is `'RowDeleted'`, `'CellChanged'` or `'RowInserted'`
`Value`	The new value in the cell or `⎕NULL`

If Reason is `'CellChanged'`, Path is the row and column number (in origin 0) of the cell that was changed and Value is its new value.

If Reason is `'RowDeleted'` or `'RowInserted'`, Path is the number of the row that has been added or removed (in origin 0) and Value is `⎕NULL`.

Create .NET Delegate R←2016⌶Y

> **Note**
>
> **.NET Framework only**

.NET methods (and properties) may specify a parameter to be a *delegate*. A delegate is a place holder for a function, normally with a particular signature and result type, that should be supplied when the method is called. Sometimes the signature of a .NET method that takes a delegate as a parameter does not provide enough information for Dyalog to determine automatically what type of delegate is required. `2016⌶` allows you to specify the type so that Dyalog can perform the necessary conversion(s) at run-time.

`Y` is a 2-element array. The first element is a .NET type that inherits from the abstract .NET Class System.Delegate. The second item is either the name of or the `⎕OR` of an APL function which is to be invoked via a .NET method or property.

The result `R` is a ref to an instance of a .NET type specified by the first element of `Y`, which internally is associated with the function identified by the second element of `Y`.

Example

```
      ∇foo∇
     ∇ foo(ev arg)
[1]    ⍝ Callback for .NET method
     ∇
      ⎕USING←'System'
      del←2016⌶ EventHandler'foo'
      del
System.EventHandler
```

Then, when calling a .NET method that requires a Delegate of type System.Eventhandler, but whose signature is imprecise in this respect, the object `del` should be used instead.

Identify .NET Type R←2017⌶Y

> **Note**
>
> **.NET Framework only**

Returns the .NET Type of a named .NET class that is loaded in the current AppDomain. Note that System.Type.GetType requires the fully qualified name, that is, prefixed by the assembly name, whereas (2017⌶) does not.

Y is a character string containing the name of a .NET object. Unless the fully qualified name is given, the namespaces in the current AppDomain are searched in the order they are specified by ⎕USING or :Using.

If the object is identified in the current AppDomain, the result R is its Type. If not, the function generates DOMAIN ERROR.

EXAMPLE

```
      ⎕USING←'System'
      2017⌶'DateTime'
System.DateTime
```

Flush Session Caption R←2022⌶Y

> **Note**
>
> **Windows only**

Under Windows, the Session Caption displays information such as the name of the current workspace. The contents of the Caption can be modified: see *Window Captions* in the *Installation and Configuration Guide* for more details.

However, the Caption is updated only at the six-space prompt; calling ⎕LOAD for example from within a function will not result in the Caption being updated at the end of the ⎕LOAD.

This I-Beam causes the Session Caption to be updated (flushed) when called. Note that this I-Beam does not alter the contents of the Caption.

EXAMPLE

```
      2022⌶0
```

Close All Windows R←2023⌶Y

Under Windows the option, *Windows -> Close All Windows* allows the user to close all open Editor and Tracer Windows, but does not reset the *state indicator*.

This I-beam mimics this behaviour, thus allowing the user to write code which can close all windows before attempting to save the workspace; with the exception of calling `0 ⎕SAVE` it is not possible to save a workspace if any editor or tracer windows are open.

Under non-Windows operating systems this is the only mechanism for closing all such windows. This I-beam is effective in Ride too.

EXAMPLE

```
      2023⌶0
```

Set Dyalog Pixel Type R←2035⌶Y

> **Note**
>
> **Windows only**

Determines how Coord `'Pixel'` is interpreted. This is determined initially by the value of the DYALOG_PIXEL_TYPE parameter and, when altered by this function, applies to all subsequent GUI operations.

`Y` is a character vector that is either `'ScaledPixel'` or `'RealPixel'`. Any other value will cause a `DOMAIN ERROR`.

The result `R` is the previous value.

Example

```
      2035⌶'ScaledPixel'
RealPixel
      2035⌶'RealPixel'
ScaledPixel

      2035⌶'realpixel'
DOMAIN ERROR
      2035⌶'realpixel'
     ^
```

Override COM Default Value R←{X}(2041⌶)Y

> **Note**
>
> **Windows only**

Certain COM objects, for example, VT_BLOBs, cannot be represented in APL or may be in error. By default Dyalog will generate a `DOMAIN ERROR` in these cases. For COM objects of type **VT_EMPTY** the interpreter by default returns `⎕NULL`.

`2041⌶` allows the APL programmer to specify what is returned by the interpreter in these cases.

`Y` may be `1` or `2`.

If `Y` is `1`, then `X` specifies the value that is returned instead of `⎕NULL` when the COM object is of type **VT_EMPTY**.

If `Y` is `2`, then `X` specifies the value that is returned when the COM object is in error, or is of a type that cannot be represented in APL.

In both cases, omitting `X` results in the default behaviour being restored.

`R` is the previous value specified; if there was no previous value then this function will perform its task but generate a `VALUE ERROR`.

Export To Memory R←2100⌶Y

> **Note**
>
> **Windows only**

This function exports the current active workspace as an in-memory .NET Assembly.

`Y` may be any array and is ignored.

The result `R` is 1 if the operation succeeded or 0 if it failed.

Close .NET AppDomain R←2101⌶Y

> **Note**
>
> **.NET Framework only**

This function closes the current .NET AppDomain.

`Y` may be any array and is ignored.

The result `R` is 0 if the operation succeeded or a non-zero integer if it failed.

This I-Beam is very likely to be changed in future.

Verify .NET Interface R←2250⌶Y

This function provides information about the Dyalog interface to .NET. The system attempts to load the Bridge DLL and reports the status of the .NET interface. It can be used to determine whether your .NET-related code can run, and also what sort of .NET support you have. It also means that you can suppress all messages that `⎕USING` would otherwise generate

The right argument `Y` is zero:

The result `R` is a 3-element nested array:

Item	Description
`R[1]`	Numeric. ¯1: the interface is not supported 0: the interface is not configured 1: the interface is configured to use .NET 2: the interface is configured to use the .NET Framework
`R[2]`	Boolean 0 or 1. 1 : the Bridge DLL was successfully loaded. 0 : the Bridge DLL failed to load, or was not attempted.
`R[3]`	A character vector containing error messages generated during load.

Examples (Windows)

```
      ⎕←2 ⎕NQ '.' 'GetEnvironment' 'Dyalog_NETCore'

      ⎕←2250⌶0
┌─┬─┐
│2│1│
└─┴─┘
```

```
      ⎕←2 ⎕NQ '.' 'GetEnvironment' 'Dyalog_NETCore'
1
      ⎕←2250⌶0
┌─┬─┐
│1│1│
└─┴─┘
```

Implementation Note

The underlying code is run once only and the results cached, so all subsequent calls to `2250⌶` will return the same result as the first time.

Set Workspace Save Options R←2400⌶Y

This function sets a flag in the workspace that determines what happens when it is saved. The flag itself is part of the workspace and is saved with it.

If the flag is set, all Trace, Stop and Monitor settings will be cleared whenever the workspace is saved, whether by `)SAVE`, `⎕SAVE` or by *File/Save* from the Session menubar.

`Y` must be 1 (set the flag) or 0 (clear the flag).

The result `R` is the previous value of the flag.

This function may be extended in the future and a left-argument may be added.

EXAMPLE

```
      (2400⌶)1
0
      )SAVE
0 Trace bits cleared.
3 Stop bits cleared.
0 Monitor bits cleared.
temp saved Sat Apr 05 17:01:30 2014
```

Expose Root Properties R←2401⌶Y

This function is used to expose or hide Root Properties, Event and Methods.

If `Y` is 1, Root Properties, Events and Methods are exposed.

If `Y` is 0, no further Root Properties, Events or Methods are exposed; however any that have already been exposed will remain so.

This functionality is available in Windows versions by selecting or unselecting the *Expose Root Properties* MenuItem in the *Options* Menu in the Session. Note that deselecting this MenuItem only affects future references to Root Properties, Events or Methods.

This function is the only mechanism available under non-Windows versions of Dyalog APL; the state of this setting is saved in the workspace, and therefore cannot be controlled by an environment variable.

EXAMPLE

```
      #.GetEnvironment'MAXWS'
VALUE ERROR
      #.GetEnvironment'MAXWS'
     ^

      2401⌶1
0
      #.GetEnvironment'MAXWS'
64M

      2401⌶0
1
      #.GetEnvironment'MAXWS'
64M
      #.GetCommandLine
VALUE ERROR
      #.GetCommandLine
     ^
```

Discard Thread on Exit R←2501⌶Y

APL threads that Dyalog creates to serve incoming .NET requests are not terminated when their work is done. They persist so that if another call comes in on the same .NET thread the same APL thread can handle it. In effect the thread is *parked* until it is needed again. If the thread is not required, there is a small performance cost in maintaining it in this state.

`(2501⌶0)` must be called from WITHIN one of these threads and tells the interpreter NOT to park the thread on termination, but to discard the thread completely.

Discard Parked Threads R←2502⌶Y

APL threads that Dyalog creates to serve incoming .NET requests are not terminated when their work is done. They persist so that if another call comes in on the same .NET thread the same APL thread can handle it. In effect the thread is *parked* until it is needed again. If the thread is not required, there is a small performance cost in maintaining it in this state.

`(2502⌶0)` removes all parked threads from the workspace.

Mark Thread as Uninterruptible R←2503⌶Y

This function marks the current thread (the thread in which it is called) as uninterruptible, and/or determines whether or not any child threads, subsequently created by the current thread, will be uninterruptible.

The right argument `Y` is an integer whose value is the sum of the following (bit-wise) values:

- 1 : mark thread as uninterruptible
- 2 : mark its children as uninterruptible

The result `R` is an integer value that indicates the previous state of the thread.

In many multi-threaded applications a large proportion of the threads are used for communication mechanisms (⎕DQ on TCPSockets, Conga, isolates); but most of the "real work" is done in thread zero.

It is undesirable that a weak interrupt interrupts a seemingly random thread. The mechanism to prevent a thread from being (weak) interrupted allows an application to be configured so that only specific threads would respond to a weak interrupt.

Use Separate Thread For .NET R←2520⌶Y

This function determines the way that .NET calls are executed in APL thread 0.

The right argument `Y` is a Boolean value:

- 1 : run .NET calls in a separate system thread
- 0 : run .NET calls in the same system thread

The result `R` is a Boolean value which indicates the previous behaviour.

When an APL **thread** first makes a .NET call, it creates a unique system thread in which that and subsequent .NET calls are made. If a .NET call results in the creation of a message queue, that queue is associated with that same system thread. So each message queue is also unique. This strategy successfully maintains separation between multiple Windows message queues being executed in different APL threads.

By default, the base APL thread (thread 0) runs .NET code in the same system thread as itself. This is a different system thread to that used to run .NET code from other APL threads, so the separation between message queues associated with APL thread 0 and those associated with other APL threads is maintained. However, in certain

circumstances, messages generated by .NET objects interfere with APL's internal message processing (and vice-versa), for example when handling exceptions.

For this reason, Dyalog recommends that APL code that creates instances of .NET objects that generate events (such as Windows Presentation Foundation objects) are run in a separate APL thread.

Where this is not possible, `2520⌶1` may be used to force Dyalog to use a unique system thread for .NET that is associated with APL thread 0. If so, it is recommended that `2520⌶1` is called at application start-up time.

Continue Autosave {R}←2704⌶Y

This function enables or disables the automatic saving of a `CONTINUE` workspace when Dyalog exits under certain circumstances.

By default this is disabled when Dyalog starts and must be explicitly enabled using this function.

`Y` is an integer defined as follows:

Value	Description
0	Disable the automatic saving of a `CONTINUE` workspace.
1	Enable the automatic saving of a `CONTINUE` workspace. This setting applies only to the current session or until disabled by `2704⌶0` .

The shy result `R` is the previous value of this setting.

Circumstances when Dyalog automatically saves a `CONTINUE` workspace include:

- a run-time violation. This is most frequently caused by an untrapped APL error which causes Dyalog to return to session-input mode (that is, an application programming fault), and APL was started with the "+q" command line option or input and output were redirected at the command line.

Disable Component Checksum Validation {R}←3002⌶Y

Checksums allow component files to be validated and repaired using *Section 6.4.43*.

From Version 13.1 onwards, components which contain checksums are also validated on every component read.

Although not recommended, applications which favour performance over security may disable checksum validation by ⎕FREAD using this function.

Y is an integer defined as follows:

Value	Description
0	⎕FREAD will not validate checksums.
1	⎕FREAD will validate checksums when they are present. This is the default.

The shy result R is the previous value of this setting.

Enable Compression of Large Components {R}←3012⌶Y

Specifies whether large components (>2GB) may be compressed.

Y is an integer defined as follows:

Value	Description
0	Large components will not be compressed.
1	Large components will be compressed if Z property is 1 (see *File Properties*), but versions of Dyalog prior to v19.0 will not be able to read them.

The shy result R is the previous value of this setting.

Send Text to Ride-embedded Browser R←{X}(3500⌶)Y

Optionally, X is a simple character vector or scalar, the contents of which are used as the caption for the tab in the Ride client that contains the embedded browser. If omitted, then the caption defaults to "3500⌶".

Y is a simple character vector the contents of which are displayed in the embedded browser tab.

The result R identifies whether the write to Ride was successful. Possible values are:

- 0 : the write to the Ride client was successful
- ¯1 : the write to the Ride client was not successful

Connected to Ride? R←(3501⌶)Y

`Y` can be any value and is ignored.

The result `R` identifies whether the Dyalog Session is running through Ride. Possible values are:

- `0` : the Session is not running through Ride
- `1` : the Session is running through Ride

This I-Beam may be extended in future.

Manage Ride Connections R←3502⌶Y

`3502⌶` gives control over Ride connections to the interpreter. More details about Ride can be found in the Ride User Guide.

`Y` may be either `0` or `1` or a simple character vector.

`R` has the value `0` if the call to `3502⌶` was successful; if unsuccessful the value may be either a positive or negative integer.

If `Y` is `0`, then any active Ride connections are disconnected, and no future connections may be made.

If `Y` is `1`, then the interpreter attempts to enable Ride, using the value of the initialisation string to determine the connection details. If the current initialisation string is ill-defined, `R` will be 64. If the Conga DLL/shared libraries are not available, `R` will be 32. In previous versions of Dyalog there were separate Ride and Conga DLLs/shared libraries; these have been merged into one set in 16.0.

If `Y` is a character vector and Ride is currently disabled, then the current initialisation string is unconditionally replaced by the contents of `Y`. If Ride is currently enabled, the initialisation string is not replaced, and `R` will have the value `¯2`.

The initialisation string has the same syntax as the value of the **RIDE_INIT** configuration parameter which is described in the Ride User Guide.

The configuration parameter **RIDE_INIT** can still be used to establish the initial value of the Ride initialisation string.

The runtime interpreter has Ride disabled by default, whether or not **RIDE_INIT** is set; the only method of enabling Ride in a runtime interpreter is to call `3502⌶1`.

If **RIDE_INIT** is set when a development interpreter is called, Ride will be enabled provided that the Ride DLL/shared library is available and the **RIDE_INIT** variable is properly formed. If the connection is of type SERVE the port must not be in use. If any of these conditions are not met, then the interpreter fails with a non-zero exit code. If **RIDE_INIT** is not set then the development interpreter will start, but with Ride disabled.

It is therefore possible to override the **RIDE_INIT** variable in the development interpreter with code similar to:

```
      r←3502⌶0              ⍝ Stop Ride
      r←3502⌶'SERVE::4511' ⍝ Update init string
      r←3502⌶1              ⍝ Start Ride
```

and similarly for altering the Ride settings in an active APL session.

> **Note**
>
> In 14.1 and earlier `3502⌶0` was used to enable Ride; this value is still valid, albeit deprecated: code should call `3502⌶1` instead.

Enabling Ride to access applications that use the run-time interpreter means that the APL code of those applications can be accessed. The I-beam mechanism described above means that the APL code itself must grant the right for a Ride client to connect to the run-time interpreter. Although Dyalog Ltd might change the details of this mechanism, the APL code will **always** need to grant connection rights. In particular, no mechanism that is only dependent on configuration parameters will be implemented.

Scan For Deprecated Files — R←{X}(3535⌶)Y

Scans a directory (and, optionally, sub-directories) for deprecated filetypes. For an overview of deprecated features, see *Programming: Deprecated Functionality* in the *Dyalog Programming Reference Guide*.

`Y` is the name of the directory to scan.

`X` specifies whether sub-directories within `Y` should also be scanned. Possible values are:

- `0` – sub-directories are not scanned (this is the default).

- `1` – sub-directories are scanned.

`R` is a two-column matrix identifying the files which are deprecated, with one filename per row.

The files in `Y` (and, optionally, sub-directories of `Y`) are examined, and only the names of files that are deprecated or cannot be checked are included in `R`. The first column contains the filenames, and the second contains a vector of one or more labels indicating why the file is deprecated. The labels are release-dependent; for a list of valid labels see the *v20.0 Release Notes: Deprecated Functionality*. The rows in `R` are not sorted.

EXAMPLE

```
      1(3535⌶)'.'
./J0C0.dcf              J0C0
./ws2000.dws            OLDWS
./XT.dxv                □XT
./subdir/S32J0C0.dcf    J0C0  S32
```

See also *Section 4.4.9.3*.

Fork New Task R←4000⌶Y

> **Note**
>
> **AIX only**

`Y` must be is a simple empty vector but is ignored.

This function *forks* the current APL task. This means that it initiates a new separate copy of the APL program, with exactly the same APL state indicator.

Following the execution of this function, there will be two identical APL processes running on the machine, each with the same state indicator and set of APL objects and values. However, none of the external interfaces and resources in the parent process will exist in the newly forked child process.

The function will return a result in both processes.

- In the parent process, `R` is the process id of the child (forked) process.

- In the child process, `R` is a scalar zero.

The following external interfaces and resources that may be present in the parent process are not replicated in the child process:

- Component file ties
- Native file ties
- Mapped file associations
- Auxiliary Processors
- .NET objects
- Edit windows
- Clipboard entries
- GUI objects (all children of `'.'`)
- I/O to the current terminal

Note that External Functions established using `⎕NA` are replicated in the child process.

The function will fail with a `DOMAIN ERROR` if there is more than one APL thread running.

The function will fail with a `FILE ERROR 11 Resource temporarily unavailable` if an attempt is made to exceed the maximum number of processes allowed per user.

Change User `R←4001⌶Y`

`Y` is a character vector that specifies a valid UNIX user name. The function changes the *userid (uid)* and *groupid (gid)* of the process to values that correspond to the specified user name.

Note that it is only possible to change the user name if the *effective uid* is 0 (that is, the process has root privileges).

If the operation is successful, `R` is the user name specified in `Y`. Note that the value of `⎕AN` will not be affected, but the value of `⎕SE.⎕AI` will be.

If the operation fails, the function generates a `FILE ERROR 1 Not Owner` error.

If the argument to `4001⌶` is other than a non-empty simple character vector, the function generates a `DOMAIN ERROR`.

If the argument is not the name of a valid user the function generates a `FILE ERROR 3 No such process`.

If the argument is the same name as the current effective user, then the function returns that name, but has no effect.

If the argument is a valid name other than the name of the effective user id of the current process, and that effective user id is not root the function generates a `FILE ERROR 1 Not owner`.

Reap Forked Tasks R←4002⌶Y

> **Note**
>
> **AIX only**

Under UNIX, when a child process terminates, it signals to its parent that it has terminated and waits for the parent to acknowledge that signal. `4002⌶` is the mechanism to allow the APL programmer to issue such acknowledgements.

`Y` must be a simple empty vector but is ignored.

The result `R` is a matrix containing the list of the newly-terminated processes which have been terminated as a result of receiving the acknowledgement, along with information about each of those processes as described below.

`R[;1]` is the process ID (PID) of the terminated child

`R[;2]` is `¯1` if the child process terminated normally, otherwise it is the signal number which caused the child process to terminate.

`R[;3]` is `¯1` if the child process terminated as the result of a signal, otherwise it is the exit code of the child process

The remaining 15 columns are the contents of the `rusage` structure returned by the underlying `wait3()` system call. Note that the two timevalstructs are each returned as a floating point number.

The current `rusage` structure contains:

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims */
    long   ru_majflt;        /* page faults */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* messages sent */
    long   ru_msgrcv;        /* messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

`4002⌶` may return the PID of an abnormally terminated Auxiliary Processor; APL code should check that the list of processes that have been reaped is a superset of the list of processes that have been started.

EXAMPLE

```
     ∇ tryforks;pid;fpid;rpid
[1]    rpids←fpids←⍬
[2]    :For i :In ⍳5
[3]        fpid←4000⌶'' ⍝ fork() a process
[4]    ⍝ if the child, hang around for a while
[5]        :If fpid=0
[6]            ⎕DL 2×i
[7]            ⎕OFF
[8]        :Else
[9]    ⍝ if the parent, save child's pid
[10]           +fpids,←fpid
[11]       :EndIf
[12]   :EndFor
[13]
[14]   :For i :In ⍳20
[15]       ⎕DL 3
[16]   ⍝ get list of newly terminated child processes
[17]       rpid←4002⌶''
[18]   ⍝ and if not empty, make note of their pids
[19]       :If 0≠⊃⍴rpid
[20]           +rpids,←rpid[;1]
[21]       :EndIf
[22]   ⍝ if all fork()'d child processes accounted for
[23]       :If fpids≡fpids∩rpids
[24]           :Leave  ⍝ quit
[25]       :EndIf
[26]   :EndFor
     ∇
```

Signal Counts R←4007⌶Y

> **Note**
>
> **UNIX, Linux and macOS only.**

Y must be a simple empty vector but is ignored.

The result `R` is an integer vector of signal counts. The length of the vector is system dependent. On AIX 32-bit it is 63 on AIX 64-bit it is 256 but code should not rely on the length.

Each element is a count of the number of signals that have been generated since the last call to this function, or since the start of the process. `R[1]` is the number of occurrences of signal 1 (SIGHUP), `R[2]` the number of occurrences of signal 2, and so forth.

Each time the function is called it zeros the counts; it is therefore inadvisable to call it in more than one APL thread.

Currently, only SIGHUP, SIGINT, SIGQUIT, SIGTERM and SIGWINCH are counted and all other corresponding elements of `R` are 0.

Discard Source Information `R←5171⌶Y`

This function discards source code and file information for scripted objects, namespaces, classes, functions, and operators that is saved in the workspace. See also *Section 4.4.9.61*.

`Y` is a vector or scalar containing zero or more references to `#` or `⎕SE`, and specifies from which namespaces the information is removed.

`R` is an integer. A non-zero value indicates that some information was removed. 0 means nothing was discarded.

- The expression `5171⌶ #` discards source code and file information from the workspace, but not from `⎕SE`.
- `5171⌶⎕SE` discards source code and file information from `⎕SE` but not from the workspace.
- `5171⌶ # ⎕SE` discards source code and file information from the workspace and from `⎕SE`.

For further information, see *Earlier Release Notes: Source As Typed*.

Discard Source Code `R←5172⌶Y`

This specifies whether source code is discarded for functions and operators when they are created by the editor or by `⎕FIX`. See also *Section 4.4.9.60*.

`Y` is 0 or 1.

If `Y` is 0 (the default), source code is retained in the workspace when an object is fixed.

If `Y` is 1, source code is not retained in the workspace when an object is fixed (source code already retained in the workspace is not discarded).

In all case the result `R` is the previous setting (0 or 1).

For further information, see *Earlier Release Notes: Source As Typed*.

List Loaded Files `R←5176⌶Y`

The editor may be used to edit Dyalog script files (*.dyalog* files) and general text files and to save the contents in the workspace. Additionally `⎕FIX` can be used to fix scripts held in files. This I-Beam returns a list of all of the files which are associated with objects in the workspace, together with information about each file.

`Y` may be any value.

`R` is a vector of vectors, one element per associated file. Each element is a 5 element vector:

Element	Contains
1	File name
2	Encoding
3	Checksum
4	Newline
5	Flags

Encoding, newline and flags are defined the same as for `⎕NGET`. See *Section 6.4.92*. Checksum is an 8-character hexadecimal value, see *Object Reference: Getbuildid* for more information.

Examples

```
      )CLEAR
clear ws
      ('' '' (8⍴' ') ⍬ 0)≡⊃5176⌶''
1
      dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      aedit←'/SALT/spice/aedit.dyalog'
      ⎕FIX 'file:///',dyalog,aedit
#.arrayeditor

      1↓⊃5176⌶⍬      ⍝ Ignore filename
 UTF-8-BOM  18507aa6  13 10  0
```

List Loaded File Objects R←5177⌶Y

The editor may be used to edit Dyalog script files (*.dyalog* files) and general text files and to save the contents in the workspace. Additionally ⎕FIX can be used to fix scripts held in files. This I-Beam returns details about all of the objects in the workspace that are associated with such files.

Y must be an empty array.

R is a vector of 8-element vectors, one vector per object in the workspace that is associated with a file.

Element	Contains
1	Object name or ref (refs are returned for all types of namespace)
2	Parent namespace
3	Name class (see ⎕NC)
4	File name
5	Start line (first line in file, 0 origin, of the object)
6	Line count (number of lines in file occupied by the object)
7	File Checksum
8	File modification time (⎕TS format)

If an object occupies a file in its entirety, both *Start line* and *Line count* will be 0.

Examples

```
      )CLEAR
clear ws
      ⊃5177⌶0
┌┬──────┬┬┬┬┬─────────┬─────────────┐
│┆[Null]┆0┆┆0┆0┆         ┆0 0 0 0 0 0 0┆
└┴──────┴┴┴┴┴─────────┴─────────────┘


      dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      aedit←'/SALT/spice/aedit.dyalog'
      ⊢⎕FIX 'file://',dyalog,aedit
#.arrayeditor

      1 1 1 0 1 1 1 1/↑5177⌶0 ⍝remove file names for brevity
┌──────────┬──────────────┬─┬──┬─┬────────┬────────────────────┐
│Run       │#.arrayeditor│3│38│4 │008fe4ed│2018 5 11 8 5 10 0│
├──────────┼──────────────┼─┼──┼─┼────────┼────────────────────┤
│Help      │#.arrayeditor│3│28│9 │008fe4ed│2018 5 11 8 5 10 0│
├──────────┼──────────────┼─┼──┼─┼────────┼────────────────────┤
│List      │#.arrayeditor│3│22│5 │008fe4ed│2018 5 11 8 5 10 0│
├──────────┼──────────────┼─┼──┼─┼────────┼────────────────────┤
│DESC      │#.arrayeditor│3│10│11│008fe4ed│2018 5 11 8 5 10 0│
├──────────┼──────────────┼─┼──┼─┼────────┼────────────────────┤
│#.arrayeditor│#          │9│0 │0 │008fe4ed│2018 5 11 8 5 10 0│
└──────────┴──────────────┴─┴──┴─┴────────┴────────────────────┘
```

Remove Loaded File Object Info R←5178⌶Y

The editor may be used to edit Dyalog script files (*.dyalog* files) and general text files and to save the contents in the workspace. Additionally ⎕FIX can be used to fix scripts held in files. This I-Beam removes the information held about an object in the workspace specified by Y that is associated with such a file.

Y is a character vector that specifies the name of a workspace object or a ref to an object.

R is Boolean. 1 means that the information was removed; 0 that it wasn't.

Note that the workspace object itself remains in the workspace; just the information about its associated file is removed.

Examples

```
      dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      aedit←'/SALT/spice/aedit.dyalog'
      ⊢⎕FIX 'file://',dyalog,aedit
#.arrayeditor
      5178⌶'arrayeditor'
1
      5178⌶'xyz' ⍝ unused name
0
```

Loaded File Object Info R←5179⌶Y

The editor may be used to edit Dyalog script files (*.dyalog* files) and general text files and to save the contents in the workspace. Additionally ⎕FIX can be used to fix scripts held in files. This I-Beam returns details about an object in the workspace specified by Y that is associated with such a file.

Y is a character vector that specifies the name of a workspace object or a ref to an object.

R is an 8-element vector containing the following information pertaining to the object and

Element	Contains
1	Object name or ref (Y)
2	Parent namespace
3	Name class (see ⎕NC)
4	File name
5	Start line (first line in file, 0 origin, of the object)
6	Line count (number of lines in file occupied by the object)
7	File Checksum
8	File modification time (⎕TS format)

If an object occupies a file in its entirety, both *Start line* and *Line count* are 0.

Examples

```
      dyalog←2 ⎕NQ '.' 'GetEnvironment' 'DYALOG'
      aedit←'/SALT/spice/aedit.dyalog'
      ⊢⎕FIX 'file://',dyalog,aedit
#.arrayeditor
      1 1 1 0 1 1 1 1⌿ 5179⌶'arrayeditor'
┌──────────────┬─┬─┬─┬─┬────────┬────────────────────┐
│#.arrayeditor│#│9│0│0│008fe4ed│2018 5 11 8 56 10 0│
└──────────────┴─┴─┴─┴─┴────────┴────────────────────┘
      1 1 1 0 1 1 1 1⌿ 5179⌶'arrayeditor.List'
┌────┬──────────────┬─┬──┬─┬────────┬────────────────────┐
│List│#.arrayeditor│3│22│5│008fe4ed│2018 5 11 8 56 10 0│
└────┴──────────────┴─┴──┴─┴────────┴────────────────────┘
      5179⌶'xyz' ⍝ unused name
┌┬──────┬┬┬┬┬────────┬──────────────┐
│ [Null] │0│0│0│        │0 0 0 0 0 0 0│
└┴──────┴┴┴┴┴────────┴──────────────┘
```

Unicode Normalisation — R←X(5581⌶)Y

Converts characters in an array to one of four Unicode Normal Forms.

`Y` must be one of the following:

- A character scalar or vector.
- A nested vector of character scalars/vectors.
- An array of any shape, in which each element is a vector character, a scalar numeric, or `⎕NULL`.

All characters in Y must be valid Unicode code points, that is, within the ranges U+0000 to U+D800 and U+E000 to U+10FFFF.

`X` must be a character scalar or vector describing the form of normalisation required, from the following:

X	Normalisation form
`'D'`	Canonical Decomposition
`'C'`	Canonical Decomposition followed by Canonical Composition
`'KD'`	Compatibility Decomposition
`'KC'`	Compatibility Decomposition followed by Canonical Composition

The result `R` is the same as `Y` except that the character arrays are normalised as required. If `Y` is a character scalar or a vector of character scalars and vectors, then scalars in `Y` are converted to vectors. Vectors can contain a different number of characters after normalisation.

Normalisation

The Universal Character Set allows multiple ways to represent certain glyphs and glyph sequences. For example, the character `Ç` has its own code point (U+00C7, or `⎕UCS 199`) but can also be formed from two separate code points – that for the character `C` (U+0043, or `⎕UCS 67`) followed by the combining cedilla (U+0327, or `⎕UCS 807`). These two representations are *canonically equivalent*. Some code points or combinations of code points more loosely represent the same character. For example, the characters `5` and `⁵` (superscript 5) both represent the same numeric digit but are visually distinct. These two have *compatibility equivalence*.

Unicode normalisation is used to transform all equivalent text to a single representation. The four different normal forms are generated by decomposing, or decomposing then composing, code points to either their canonically equivalent or compatibly equivalent sequences. Example use cases for each form are:

Form	Use For
`D`	Searching, linguistic analysis
`C`	Display, storage, user input
`KD`	Text comparison, data indexing, accessibility
`KC`	Search, login names, spoofing defence

For a full explanation of the different normalisation forms, see Unicode Standard Annex #15.

Examples

```
      COMBINING_CEDILLA←⎕UCS 807
      'C' (5581⌶) 5 '5⁵' ('C' COMBINING_CEDILLA)
┌→──────────┐
│  ┌→┐ ┌→┐ │
│ 5│5⁵│ │Ç│ │
│  └─┘ └─┘ │
└∊─────────┘
      'KC' (5581⌶) 5 '5⁵' ('C' COMBINING_CEDILLA)
┌→──────────┐
│  ┌→┐ ┌→┐ │
│ 5│55│ │Ç│ │
│  └─┘ └─┘ │
└∊─────────┘
      'C' COMBINING_CEDILLA ≡ 'D' (5581⌶) 'Ç'
1
      'BrandTM'≡'Brand™'
0
      'KD'(5581⌶)'Brand™'
BrandTM
      'BrandTM'≡⍤('KD'∘(5581⌶))'Brand™'
1
```

JSON Translate Name R←X(7162⌶)Y

Converts between JSON names and APL names.

When ⎕JSON imports an entity from JSON text whose name would be an invalid APL name, the function converts the invalid name into a valid APL name using a *name mangling* algorithm. When ⎕JSON exports an APL namespace as JSON text, the process is reversed.

This function performs the same *name mangling* allowing the programmer to identify JSON entities as APL names, and vice-versa.

Y is a character vector or scalar.

X is a scalar numeric value which must be 1 or 0.

When X is 0, R is the name in Y converted, if necessary, so that it is a valid APL name. It performs the same translation of JSON object names to APL names that is performed when importing JSON.

When X is 1, R is the name in Y which, if mangled, is converted back to the original form.. It performs the same translation of APL names to JSON object names that is performed when exporting JSON.

EXAMPLES

```
      0(7162⌶)'2a'
Δ2a
      1(7162⌶)'Δ2a'
2a
```

Note that the algorithm can be applied, even when mangling is not required. So:

```
      0(7162⌶)'foo'
foo
      1(7162⌶)'foo'
foo
```

For further details, see *Section 6.4.72*.

Shell Process Control R←{X}(8373⌶)Y

This function provides a way to determine the process IDs of processes started by *Section 6.4.134*, as well as enabling the sending of signals to any of those processes.

Monadic Use: Query Process IDs

If Y is a positive integer that identifies an APL thread number, and that thread is currently running ⎕SHELL, then R is the child process ID.

If Y is ⍬, then R is a numeric vector containing all the process IDs of the processes that remained running after ⎕SHELL completed (due to timeouts or interrupts for example - see *Section 6.4.134* for more information). Any call to 8373⌶ starts by removing the "dead" processes from the list; dead processes are ones that are no longer running but their ID is still reserved so that it is not reused by a different process.

EXAMPLE

```
    tid←{
        ...
        r←⎕SHELL ...
        ...
    }&0

    processID←(8373⌶)tid
    processID
84093
```

Dyadic Use: Signal Child Process

`X` must be a integer scalar representing a signal number that is to be sent to the child process.

`Y` must be an integer scalar that identifies the child process:

- Positive values (including `0`) are interpreted as APL thread numbers, and identify the child process of any currently running `⎕SHELL` call on that thread.
- Negative values are negated and treated as process IDs. The process ID *must* be one of the processes left behind by a finished call to `⎕SHELL`.

`R` is a Boolean scalar indicating whether the signal was successfully sent (`1`), as reported by the operating-system.

> **Dyalog on Microsoft Windows**
>
> On Microsoft Windows, the only valid value for `X` is `9`, which makes the I-beam call `TerminateProcess()` on the child process.

Singular Value Decomposition `R←(8415⌶)Y`

`Y` is a simple numeric matrix. `⎕FR` must be 645.

The result `R` is a 4 element vector whose elements are as follows.

`[1]`	`U`	a unitary matrix
`[2]`	`S`	a diagonal matrix
`[3]`	`V`	a unitary matrix
`[4]`	`f`	a Boolean flag indicating whether the algorithm converged (1)or not (0)

This function computes a factorisation of the matrix `Y` such that:

```
      Y ≡ U +.× S +.× ⍉+V
```

This can be useful for analysing matrices for which ⌹ cannot compute an inverse, because they are singular or nearly singular.

For further information, see Wikipedia's article on Singular Value Decomposition.

> **Note**
>
> This function signals `DOMAIN ERROR` if `⎕FR` is 1287. See *Section 6.4.57*.

List Shared Code Files/Attached Names {R}←{X} (8659⌶) Y

Restriction: 64-bit Unicode only

The value of `Y` specifies whether this returns a list of the shared code files that are attached to the current workspace or the names in a specific shared code file. For more information on shared code files, see the *Shared Code Files User Guide*.

List shared code files

`Y⍳⍳0`

`X` is ignored.

`R` is a 2-column matrix listing the shared code files that are attached to the current workspace:

- `[;1]` is the slot identifier for the fixed virtual memory address of the shared code file.
- `[;2]` is the name (including the filename extension) of the shared code file that was loaded.

The rows of the matrix (one row for each shared code file) are ordered to correspond to the order in which the shared code files were specified when they were originally attached, that is, in the right argument to `8666⌶` (see ADD XREF).

List attached names

`Y` is an integer vector that would be a valid right argument to `⎕NL`; it identifies the nameclasses and subclasses for which the names should be listed.

`X` is the slot identifier (an integer in the range 1-8) for the unique fixed virtual memory address of the shared code file.

`R` lists the names in the shared code file identified by slot. If any element of `Y` is negative, then positive values in `Y` are treated as if they were negative and `R` is a vector of character vectors. Otherwise, `R` is a simple character matrix.

Attach/Assimilate/Detach Shared Code Files `{R}←{X} (8666⌶) Y`

Restriction: 64-bit Unicode only

The value of `Y` specifies whether this attaches, assimilates, or detaches shared code files. For more information on shared code files, see the *Shared Code Files User Guide*.

Attach Shared Code Files

`Y` is a vector of character vectors of shared code files to load, or a single character vector (a character scalar is not acceptable). If filename extensions are not provided, then **.dwx** is used. The path can be absolute or relative to current location; there is no sensitivity to `WSPATH`.

Optionally, `X` is a list of nameclass identifiers to be brought over (cannot include sub-classes). The default is `2 3 4 9` (variables, functions, operators, and namespaces respectively).

The effect of attaching shared code files is analogous to performing a *Section 7.2.21* from the shared code files, that is:

- names that already have a definition are preserved unaltered; if the same name appears in more than one shared code file, then the files are searched in the specified order and the first occurrence of the name is used.
- names in attached files immediately affect the results of system functions that provide metadata, such as `⎕NL` or `⎕NC`.

If any shared code files are already attached, then they are detached from the active workspace before new shared code files are attached. Multiple shared code files cannot be attached using separate calls to `8666⌶`.

Assimilate Shared Code Files

`Y` is `⍬NULL`.

All referenced objects in the shared code files are copied into the active workspace. The active workspace then contains all the code and data that was visible to it when the shared code files were attached; it can then be saved and used independently of the shared code files. The shared code files that are attached to the active workspace are then disconnected from the active workspace.

Significant space might be required to assimilate all the code in the shared code files into the active workspace. If a `WSFULL` error occurs then the operation will fail; it cannot be rolled back, and leaves the workspace in an indeterminate (but consistent) state. In this situation, the shared code files are not disconnected from the active workspace as doing so could result in further errors.

Only things that are the current referent are copied. The process is driven from the data not the name; this means that if multiple shared code files that include the same names are attached, then only the first of these names is external and that is the one that gets copied.

Detach Shared Code Files

`Y` is `0⍴⊂''`, that is, a zero-length list of names.

Any existing attached shared code files are detached.

Detaching shared code files results in data being copied from the shared code files as needed. However, namespaces are always copied when a shared code file is first attached.

Before a shared code file is disconnected from the active workspace:

- if a name that was brought into the active workspace when the shared code file was attached has not had its associated code/data changed, then the name is expunged from the active workspace.
- if a name in the active workspace embeds references to objects residing in a shared code file, then the entire definitions of the referenced objects are copied (assimilated) into the active workspace. This includes (for example), tacit functions that are derived from functions in a shared code file and arrays that

contain references to data in a shared code file. These objects must still be functional following the disconnect.

As shared code files are read-only, they cannot be updated while they are in use. Instead, if a shared code file needs to be updated, it must be rebuilt. When a new version of a shared code file becomes available, anyone using the old version should detach it and attach the new one instead as soon as is practical.

Save Shared Code Files {R}←{X} (8667⌶) Y

Restriction: 64-bit Unicode only

This saves a shared code file. For more information on shared code files, see the *Shared Code Files User Guide*.

`Y` is 2-element vector of vectors in which:

- `Y[1]` is the slot identifier (an integer in the range 1-8) for the unique fixed virtual memory address at which to save the shared code file
- `Y[2]` is a character vector of the filename for the shared code file. If a filename extension is not provided, then **.dwx** is used. If a file of this name already exists, or the file cannot be created for any reason, then the operation will fail.

Optionally, `X` is a vector of character vectors or a matrix specifying the names of names of functions, operators and variables to save in the shared code file. If `X` is omitted, all functions, operators, and variables in the active workspace are saved in the shared code file.

EXAMPLE

```
      8667⌶ 1 'scf.dwx'
```

Sample Probability Distribution R←X(16808⌶)Y

This function generates an array of random numbers from a named probability distribution. Note that this is not currently implemented for AIX.

`Y` is a 2-item vector containing the name of the probability distribution from those listed in the table below, and the shape of the result.

`X` is a scalar or 1 or 2 element numeric vector that specifies parameters.

For example to get an array with shape (3 5 7) of uniform random numbers for the interval from ¯17.3 to 12.7, you'd enter

```
      ¯17.3 12.7 (16808 ⌶) 'Uniform' (3 5 7)
```

If you wanted a vector of 100,000 uniform random numbers for that interval, you'd enter

```
      ¯17.3 12.7 (16808 ⌶) 'Uniform' 100000
```

The domain rules for the distributions currently implemented are as follows:

Distribution	X[1]	X[2]	Domain Rules
`'Uniform'`	a	b	a < b ; A numeric interval. Example: 1.0 7.6
`'Beta'`	a	b	a > 0 AND b > 0
`'Bernoulli'`	probability		probability ≥ 0 AND probability ≤ 1
`'Binomial'`	trials	probability	trials is an integer ≥ 0; probability ≥ 0 AND probability ≤ 1
`'Cauchy'`	location	scale	location unrestricted; scale > 0
`'Chi Squared'`	degree of freedom		degree of freedom ≥ 0
`'Exponential'`	rate		rate ≥ 0
`'F'`	a	b	a ≥ eps AND b ≥ eps; where eps is smallest non-zero positive float number
`'Gamma'`	a	b	a ≥ 0 AND b ≥eps; where eps is smallest non-zero positive float number
`'Inverse Gamma'`	a	b	a ≥ 0 AND b ≥ 0
`'Laplace'`	location	scale	location unrestricted; scale ≥ 0
`'Logistic'`	location	scale	location unrestricted; scale ≥ 0
`'Log Normal'`	location	scale	location unrestricted; scale ≥ 0
`'Normal'`	location	scale	location unrestricted; scale ≥ 0
`'Poisson'`	rate		rate ≥ 0
`'Student T'`	degree of freedom		degree of freedom ≥eps where eps is smallest non-zero positive float number
`'Weibull'`	a	b	a ≥ eps AND b ≥ eps ; eps is smallest non-zero positive float number

Each of those distributions has a corresponding Wikipedia entry with a description of its theoretical foundation and usually graphs of the probability density functions and cumulative distribution functions for interesting sets of parameter values.

Example

The probability density function for the Beta distribution (see) with the parameter vector (2 5) has an interesting shape.

`BucketCount` counts random numbers that fall into a sequence of evenly distributed bucket intervals:

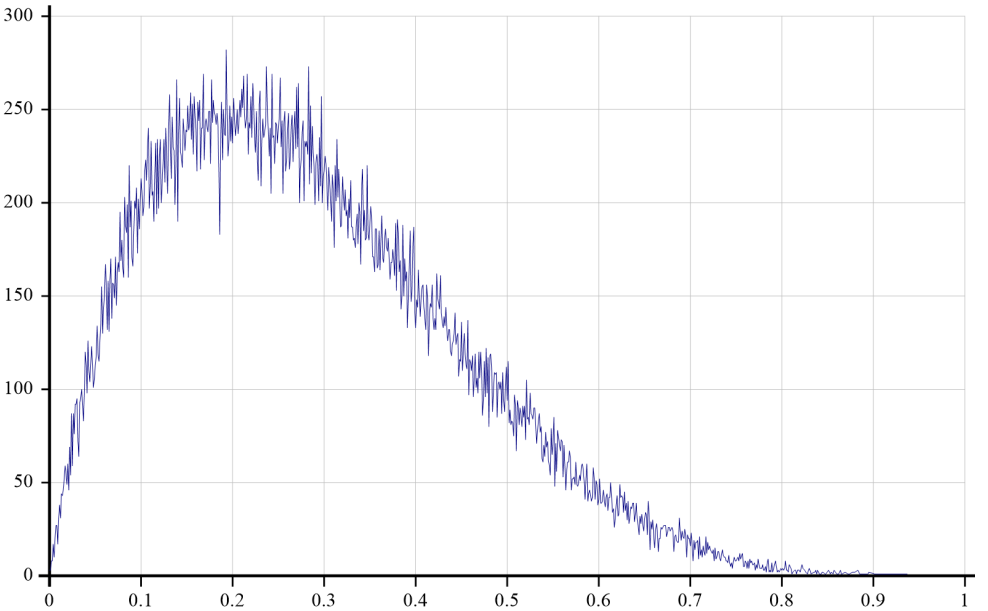
```
      BucketCounts←{
[1]         ir←⌊⍵÷÷⍺
[2]         kir←{⍺(≠⍵)}⌸ir
[3]         kir[;⎕IO]÷←⍺
[4]         kir[⍋⍋kir[;⎕IO];]
[5]     }
```

So then we can create 100,000 samples and calculate values for a density graph with 1,000 evenly spaced buckets by:

```
      rv←2 5 (16808⌶)'Beta' 100000
      bc←1000 BucketCounts rv
```

Using the Chart Wizard

we can plot `(⊂2)⌷⍉bc` against `(⊂1)⌷⍉bc` to get the graph:



Line Count R←50100⌶Y

This function is a compact version of the system function ⎕LC. If an expression requires only the most recent line(s) in the function calling stack, this is a more efficient alternative to using ⎕LC.

Y may be an integer specifying the depth of the function calling stack that is required in the result.

The result R is the same as ⎕LC, but truncated to the number of stack levels specified by Y.

EXAMPLE

```
          ∇ Foo
[1]     :If 4=⍴⎕LC
[2]         50100⌶0
[3]         50100⌶1
[4]         50100⌶2
[5]         50100⌶3
[6]         50100⌶4
[7]         50100⌶5
[8]         →
[9]     :Else
[10]        Foo
[11]    :EndIf
     ∇

      Foo

3
4 10
5 10 10
6 10 10 10
7 10 10 10
```

4.4.10 Inner Product R←Xf.gY

`f` and `g` are dyadic functions. The last axis of `X` must have the same length as the first axis of `Y`, or one of `X` and `Y` is single (`^/1=⍴X` or `^/1=⍴Y`).

The result of the derived function has shape `(¯1↓⍴X),1↓⍴Y`; each item is `f/x g¨y` where `x` and `y` are vectors taken from all the combinations of vectors along the last axis of `X` and the first axis of `Y`.

> **Note**
>
> - `g` must return a result.
> - `f` must return a result with the possible exception of the case when `1=⍴x g¨y`.
> - The expression `f/x g¨y` applies even when `R` or `x g¨y` or `X` or `Y` is empty. When `X` or `Y` is empty, the vector `x` is `X` reshaped to the appropriate length (`y` is `Y` reshaped to appropriate length).
> - `x` is just `X` itself if `X` is a scalar. Likewise `y` and `Y`.

Examples

```
      1 2 3+.×10 12 14
76
      +/1 2 3×10 12 14
76

      NAMES
HENRY
WILLIAM
JAMES
SEBASTIAN

      NAMES^.='WILLIAM  '
0 1 0 0
```

4.4.11 Key R←{X}f⌸Y

> **Classic Edition**
>
> The symbol ⌸ is not available in Classic Edition, and the Key operator is instead represented by ⎕U2338.

f may be any dyadic function that returns a result.

If X is specified, it is an array whose major cells specify keys for corresponding major cells of Y. The Key operator ⌸ applies the function f to each unique key in X and the major cells of Y having that key.

If X is omitted, Y is an array whose major cells represent keys. In this case, the Key operator applies the function f to each unique key in Y and the elements of ⍳≢Y having that key. f⌸Y is the same as Y f⌸⍳≢Y.

The elements of R appear in the order in which they first appear in Y.

Key is similar to the GROUP BY clause in SQL.

⎕CT and ⎕DCT are implicit arguments of the Key operator.

Example

In this example, both arrays are vectors so their major cells are their elements. The function {⍺':'⍵} is applied between the unique elements in suits ('Spades' 'Hearts' 'Clubs') and the elements in cards grouped according to their corresponding elements in suits, that is, ('2' 'Ace'), ('Queen' 'Jack') and ('4').

```
      cards←'2' 'Queen' 'Ace' '4' 'Jack'
      suits←'Spades' 'Hearts' 'Spades' 'Clubs' 'Hearts'

      suits,[1.5]cards
Spades  2
Hearts  Queen
Spades  Ace
Clubs   4
Hearts  Jack

       suits {α':'ω}⌸ cards
 Spades  : 2 Ace
 Hearts  :  Queen  Jack
 Clubs   : 4
```

Monadic Examples

```
      {α ω} ⌸ suits ⍝ indices of unique major cells
 Spades   1 3
 Hearts   2 5
 Clubs    4

      {α,≠ω} ⌸ suits ⍝ count of unique major cells
Spades  2
Hearts  2
Clubs   1
```

```
       letters←'zabayza'
       {α(≠ω)}⌸letters
z 2
a 3
b 1
y 1
```

Further Examples

`x` is a vector of stock codes, `y` is a corresponding matrix of values.

```
      ⍴x
10
      ⍴y
10 2
     x,y
 IBM   13 75
 AAPL  45 53
 GOOG  21  4
 GOOG  67 67
 AAPL  93 38
 MSFT  51 83
 IBM    3  5
 AAPL  52 67
 AAPL   0 38
 IBM    6 41
```

If we apply the function `{⍺ ⍵}` to `x` and `y` using the ⌸ operator, we can see how the rows (its major cells) of `y` are grouped according to the corresponding elements (its major cells) of `x`.

```
      x{⍺ ⍵}⌸y
 IBM   13 75
        3  5
        6 41
 AAPL  45 53
       93 38
       52 67
        0 38
 GOOG  21  4
       67 67
 MSFT  51 83
```

More usefully, we can apply the function `{⍺(+⌿⍵)}`, which delivers the stock codes and the corresponding totals in `y`:

```
      x{⍺(+⌿⍵)}⌸y
 IBM    22 121
 AAPL   190 196
 GOOG   88 71
 MSFT   51 83
```

There is no need for the function to use its left argument. So to obtain just the totals in `y` grouped by the stock codes in `x`:

```
      x{+⌿⍵}⌸y
 22 121
190 196
 88  71
 51  83
```

Defined Function Example

This example appends the data for a stock into a component file named by the symbol.

```
    ∇ r←stock foo data;fid;file
[1]   file←⊃stock
[2]   :Trap 0
[3]       fid←file ⎕FTIE 0
[4]       file ⎕FERASE fid
[5]   :EndTrap
[6]   fid←file ⎕FCREATE 0
[7]   r←data ⎕FAPPEND fid
[8]   ⎕FUNTIE fid
    ∇
```

```
    x foo⌸y
1 1 1 1
```

EXAMPLE

```
     {⍺ ⍵} ⌸ suits ⍝ indices of unique major cells
 Spades   1 3
 Hearts   2 5
 Clubs    4

     {⍺,≢⍵} ⌸ suits ⍝ count of unique major cells
Spades  2
Hearts  2
Clubs   1
```

Another Example

Given a list of names and scores., the problem is to sum the scores for each unique name. A solution is presented first without using the Key operator, and then with the Key operator.

```
      names ⍴ 12, some repeat
 Pete  Jay  Bob  Pete  Pete  Jay  Jim  Pete  Pete  Jim
 Pete  Pete

      (∪names)∘.≡names
1 0 0 1 1 0 0 1 1 0 1 1
0 1 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 1 0 0

      scores
66 75 71 100 22 10 67 77 55 42 1 78

      b←↓(∪names)∘.≡names
      ]disp b/¨⊂⍳12
┌→──────────────┬──┬┬──────┐
│1 4 5 8 9 11 12│2 6│3│7 10│
└~──────────────→┴~─→┴→┴~───→┘

      +/¨b/¨⊂scores
399 85 71 109

      ]disp {⊂⍵}⌸ names
┌→──────────────┬──┬┬──────┐
│1 4 5 8 9 11 12│2 6│3│7 10│
└~──────────────→┴~─→┴→┴~───→┘

      names {+/⍵}⌸ scores
399 85 71 109
```

4.4.12 Outer Product {R}←X∘.gY

g may be any dyadic function. The left operand of the operator is the symbol ∘. X and Y may be any arrays whose elements are appropriate to the function g.

Function g is applied to all combinations of the elements of X and Y. If function g returns a result, the shape of R is (⍴X),⍴Y. Each element of R is the item returned by function g when applied to the particular combination of elements of X and Y.

If X or Y is empty, the result R is a conformable empty array, and the operand function is applied *once* between the first items of X and Y to determine the prototype of R.

Examples

```
      1 2 3∘.×10 20 30 40
10 20 30  40
20 40 60  80
30 60 90 120

      1 2 3∘.⍴'AB'
 A    B
 AA   BB
 AAA  BBB

      1 2∘.,1 2 3
 1 1  1 2  1 3
 2 1  2 2  2 3

      (⍳3)∘.=⍳3
1 0 0
0 1 0
0 0 1
```

4.4.13 Over {R}←{X}f⍤gY

> **Classic Edition**
>
> The symbol `⍤` is not available in Classic Edition, and the Over operator is instead represented by `⌷U2365`.

`g` can be any monadic function which returns a result. `Y` can be any array that is suitable as the argument to function `g` with `gY` being suitable as the right argument to function `f`.

If `X` is omitted, `f` must be a monadic function. If `X` is specified, `f` must be a dyadic function and `X` can be any array that is suitable as argument to function `g` with `gX` being suitable as the left argument to function `f`.

The derived function is equivalent to `fgY` or `(gX)f(gY)` and need not return a result.

The Over operator allows functions to be *glued* together to build up more complex functions. For further information, see *Function Composition*.

EXAMPLES

```
      2 3 ,⍤0 'text'    ⍝ ,⍤0  ←→  {⍺⍵}
┌───┬────┐
│2 3│text│
└───┴────┘

     scores←82 90 76
     weights←20 35 45
     (weights×scores)÷⍤0(+/)weights ⍝ Weighted average
82.1
```

4.4.14 Power {R}←{X}(f⍣g)Y

If right operand `g` is a numeric integer scalar, power applies its left operand function `f` cumulatively `g` times to its argument. In particular, `g` may be Boolean 0 or 1 for conditional function application.

If right operand `g` is a scalar-returning dyadic *function*, then left operand function `f` is applied repeatedly **until** `((f Y) g Y)` or until a strong interrupt occurs. Notice that power calls its dyadic right operand `g` with the next `(f Y)` and current `(Y)` values of the iteration as left and right arguments. In particular, if `g` is `=` or `≡`, the result is sometimes termed a *fixpoint* of `f`.

If a left argument `X` is present, it is bound as left argument to left operand function `f`:

```
      X (f ⍣ g) Y → (X∘f ⍣ g) Y
```

A *negative* right operand `g` applies the *inverse* of the operand function `f`, `(⍣¯1⊢g)`times. In this case, `f` may be a primitive function or an expression of primitive functions combined with primitive operators:

∘	compose
⍤	each
∘.	outer product
⍨	commute
\	scan
[]	axis
⍣	power

If the function does not have an inverse, a negative argument `g` generates `DOMAIN ERROR`.

> **Hints and Recommendations**
>
> Dyalog Ltd recommends that the use of inverses in production code is limited to `⊥`, `⍊`, `+\`, `≠\`, `+⌿`, and `≠⌿`, and functions derived from these, for example, `2∘⊥` and `≠\[1]`.

Examples

```
(,∘⊂∘,⍤(1=≡,vec))vec     ⍝ ravel-enclose if simple.

a b c←1 0 1{(⊂⍤⍺)⍵}¨abc ⍝ enclose first and last.

cap←{(⍺⍺⍤⍺)⍵}            ⍝ conditional application.

a b c←1 0 1⊂cap¨abc      ⍝ enclose first and last.
```

```
    succ←1∘+                  ⍝ successor function.

    (succ⍣4)10                ⍝ fourth successor of 10.
14
    (succ⍣¯3)10               ⍝ third predecessor of 10.
7
    1+∘÷⍣=1                   ⍝ fixpoint: golden mean.
1.618033989

    f←(32∘+)∘(×∘1.8)          ⍝ Fahrenheit from Celsius.
    f 0 100
32 212

    c←f⍣¯1                    ⍝ c is Inverse of f.
    c 32 212                  ⍝ Celsius from Fahrenheit.
0 100

    invs←{(⍺⍺⍣¯1)⍵}           ⍝ inverse operator.

    +\invs 1 3 6 10           ⍝ scan inverse.
1 2 3 4

    2∘⊥invs 9                 ⍝ decode inverse.
1 0 0 1

    dual←{⍵⍵⍣¯1 ⍺⍺ ⍵⍵ ⍵}      ⍝ dual operator.

    mean←{(+/⍵)÷⍴⍵}           ⍝ mean function.

    mean dual× 1 2 3 4 5      ⍝ geometric mean.
2.605171085

    +/dual÷ 1 2 3 4 5         ⍝ parallel resistance.
0.4379562044

    mean dual(×⍨)1 2 3 4 5    ⍝ root-mean-square.
3.31662479

    ⍉dual↑ 'hello' 'world'    ⍝ vector transpose.
 hw  eo  lr  ll  od
```

> **Warning**
>
> Some expressions, such as the following, will cause an infinite internal loop and APL will appear to hang. In most cases this can be resolved by issuing a hard INTERRUPT.

```
!⍣-1
!⍣-2
```

One can ensure that weak interrupts and ⎕TKILL can interrupt by packaging the ⍣ within the dop `{⍺←⊢ ⋄ ⍺ (⍺⍺{⍺←⊢ ⋄ ⍺ ⍺⍺ ⍵}⍣⍵⍵) ⍵}`.

EXAMPLE

```
PowOp←{⍺←⊢ ⋄ ⍺ (⍺⍺{⍺←⊢ ⋄ ⍺ ⍺⍺ ⍵}⍣⍵⍵) ⍵}
tnum←!PowOp-&1 ⍝ using naked ⍣ will freeze APL

⎕TKILL tnum
```

4.4.15 Rank R←{X}(f⍤B)Y

> **Classic Edition**
>
> The symbol ⍤ is not available in Classic Edition, and the Rank operator is instead represented by ⎕U2364.

The Rank operator ⍤ applies monadic function `f` successively to sub-arrays of `Y`, or dyadic function `f` between sub-arrays of `X` and `Y`. Sub-arrays are selected by right operand `B`.

`B` is a numeric scalar or vector of up to three items, specifying the ranks of the cells to which `f` should be applied. The most general form is a three item vector `p q r`, where:

- `p` specifies the rank of the argument cells when `f` is applied monadically
- `q` specifies the rank of the left argument cells when `f` is applied dyadically
- `r` specifies the rank of the right argument cells when `f` is applied dyadically

If `B` is a two item vector `q r`, it is implicitly extended to `r q r`. If `B` has a single item `r`, it is implicitly extended to `r r r`.

If an item `k` of `B` is zero or positive it selects k-cells of the corresponding argument. If it is negative, it selects (r+k)-cells where `r` is the rank of the corresponding argument. A value of `¯1` selects major cells. For further information, see *Programming: Cells And Subarrays*.

If `X` is omitted, `f` may be any monadic function that returns a result. `Y` may be any array. The Rank operator `⍤` applies function `f` successively to the sub-arrays in `Y` specified by `p` (that is, the first item of `B`, as specified or implicitly extended).

If `X` is specified, it may be any array and `f` may be any dyadic function that returns a result. `Y` may be any array. In this case, the Rank operator applies function `f` successively between the sub-arrays in `X` specified by `q` and the sub-arrays in `Y` specified by `r`.

The sub-arrays of `R` are the results of the individual applications of `f`. If these results differ in rank or shape, they are extended to a common rank and shape in the manner of Mix. See *Section 3.4.54*.

Notice that it is necessary to prevent the right operand `k` binding to the right argument. This can be done using parentheses, for example, `(f⍤1)Y`. The same can be achieved using `⊢`, for example, `f⍤1⊢Y` because `⍤` binds tighter to its right operand than `⊢` does to its left argument, and `⊢` therefore resolves to Identity.

Monadic Examples

Using enclose (`⊂`) as the left operand elucidates the workings of the rank operator.

```
      Y
36 99 20  5
63 50 26 10
64 90 68 98

66 72 27 74
44  1 46 62
48  9 81 22
      ⍴Y
2 3 4
```

```
      ⊂⍤2 ⊢Y
┌──────────────┬──────────────┐
│36 99 20  5   │66 72 27 74   │
│63 50 26 10   │44  1 46 62   │
│64 90 68 98   │48  9 81 22   │
└──────────────┴──────────────┘
```

```
      ⊂⍤1 ⊢Y
┌──────────────┬──────────────┬──────────────┐
│36 99 20 5    │63 50 26 10   │64 90 68 98   │
├──────────────┼──────────────┼──────────────┤
│66 72 27 74   │44 1 46 62    │48 9 81 22    │
└──────────────┴──────────────┴──────────────┘
```

The function `{(⊂⍋⍵)⌷⍵}` sorts a vector.

```
      {(⊂⍋⍵)⌷⍵} 3 1 4 1 5 9 2 6 5
1 1 2 3 4 5 5 6 9
```

The rank operator can be used to apply the function to sub-arrays; in this case to sort the 1-cells (rows) of a 3-dimensional array.

```
      Y
36 99 20  5
63 50 26 10
64 90 68 98

66 72 27 74
44  1 46 62
48  9 81 22
```

```
      ({(⊂⍋⍵)⌷⍵}⍤1)Y
 5 20 36 99
10 26 50 63
64 68 90 98

27 66 72 74
 1 44 46 62
 9 22 48 81
```

Dyadic Examples

```
      10 20 30 (+⍤0 1)3 4⍴⍳12
10 11 12 13
24 25 26 27
38 39 40 41
```

Using the function `{⍺ ⍵}` as the left operand demonstrates how the dyadic case of the rank operator works.

```
      10 20 30 ({⍺ ⍵}⍤0 1)3 4⍴⍳12
┌──┬────────┐
│10│0 1 2 3 │
├──┼────────┤
│20│4 5 6 7 │
├──┼────────┤
│30│8 9 10 11│
└──┴────────┘
```

Note that a right operand of `¯1` applies the function between the major cells (in this case *elements*) of the left argument, and the major cells (in this case *rows*) of the right argument.

```
      10 20 30 ({⍺ ⍵}⍤¯1)3 4⍴⍳12
┌──┬────────┐
│10│0 1 2 3 │
├──┼────────┤
│20│4 5 6 7 │
├──┼────────┤
│30│8 9 10 11│
└──┴────────┘
```

4.4.16 Reduce

Reduce	R←f/[K]Y

`f` must be a dyadic function. `Y` may be any array whose items in the sub-arrays along the K^{th} axis are appropriate to function `f`.

The axis specification is optional. If present, `K` must identify an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←f⌿Y` implies the first axis of `Y`.

`R` is an array formed by applying function `f` between items of the vectors along the K^{th} (or implied) axis of `Y`. For a typical vector `Y`, the result `R` is:

```
      R  ↔ ⊂(1⊃Y)f(2⊃Y)f.....f(n⊃Y)
```

The shape `S` of `R` is the shape of `Y` excluding the K^{th} axis, i.e.

```
      S  ↔  ⍴R  ↔  (K≠⍳⍴⍴Y)/⍴Y
```

If `Y` is a scalar then for any function `f`, `R` is `Y`.

If the length of the K^{th} axis of `Y` is 1, or if the length of any other axis of `Y` is 0, then `f` is not applied and `R` is `S⍴Y`.

Otherwise, if the length of the K^{th} axis is 0 then the result depends on `f` and on `⊃Y` (the prototypical item of `Y`) as follows:

If `f` is one of the functions listed in *Table 4-1* then `R` is `S⍴⊂I`, where `I` is formed from `⊃Y` by replacing each depth-zero item of `⊃Y` with the *identity element* from the table.

Otherwise, if `f` is Catenate, `R` is `S⍴⊂0/⊃Y`. If `f` is Catenate First, `R` is `S⍴⊂0⌿⊃Y`. If `f` is Catenate along the J^{th} axis, `R` is `S⍴⊂0/[J]⊃Y`. See *Section 3.4.2*.

Otherwise, `DOMAIN ERROR` is reported.

Table 4-1: *Identity Elements*

Function		Identity
Add	+	0
Subtract	−	0
Multiply	×	1
Divide	÷	1
Residue	|	0
Minimum	⌊	$¯M^3$
Maximum	⌈	−M
Power	*	1
Binomial	!	1
And	∧	1
Or	∨	0
Less	<	0
Less or Equal	≤	1
Equal	=	1
Greater	>	0
Greater or Equal	≥	1
Not Equal	≠	0
Encode	⊤	0
Union	∪	⍬
Replicate	/⍀	1
Expand	\⍀	1
Rotate	⌽⍉	0

Examples

```
      ∨/0 0 1 0 0 1 0
1
      MAT
1 2 3
4 5 6

      +/MAT
6 15
```

```
      +⌿MAT
5 7 9

      +/[1]MAT
5 7 9

      +/(1 2 3)(4 5 6)(7 8 9)
 12 15 18

      ,/'ONE' 'NESS'
 ONENESS

      +/⍳0
0
```

```
      (⊂⍬)≡,/⍬
1
      (⊂'')≡,/0⍴'Hello' 'World'
1
      (⊂0 3 4⍴0)≡⍪/0⍴⊂2 3 4⍴0
1
```

Reduce N-Wise R←Xf/[K]Y

`f` must be a dyadic function. `X` must be a simple scalar or one-item integer array. `Y` may be any array whose sub-arrays along the `K`th axis are appropriate to function `f`.

The axis specification is optional. If present, `K` must identify an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←Xf⌿Y` implies the first axis of `Y`.

`R` is an array formed by applying function `f` between items of sub-vectors of length `X` taken from vectors along the `K`th (or implied) axis of `Y`.

---

3 `M` represents the largest representable value: typically this is 1.7E308, unless `⎕FR` is 1287, when the value is 1E6145.

`X` can be thought of as the width of a "window" which moves along vectors drawn from the `K`th axis of `Y`.

If `X` is zero, the result is a `(⍴Y)+(-⍴⍴Y)↑1` array of identity elements for the function `f`. See *Section 4.4.16.1*.

If `X` is negative, each sub-vector is reversed before being reduced.

Examples

```
      ⍳4
1 2 3 4

      3+/⍳4     ⍝ (1+2+3) (2+3+4)
6 9
      2+/⍳4     ⍝ (1+2) (2+3) (3+4)
3 5 7
      1+/⍳4     ⍝ (1) (2) (3) (4)
1 2 3 4

      0+/⍳4     ⍝ Identity element for +
0 0 0 0 0
      0×/⍳4     ⍝ Identity element for ×
1 1 1 1 1

      2,/⍳4     ⍝ (1,2) (2,3) (3,4)
 1 2  2 3  3 4
      ¯2,/⍳4    ⍝ (2,1) (3,2) (4,3)
 2 1  3 2  4 3
```

4.4.17 Reduce First

Reduce First	`R←f⌿Y`

The form `R←f⌿Y` implies reduction along the first axis of `Y`. See *Section 4.4.16.1* above.

Reduce First N-Wise	`R←Xf⌿[K]Y`

The form `R←Xf⌿Y` implies N-Wise reduction along the first axis of `Y`. See *Section 4.4.16.2*.

4.4.18 Scan R←f\[K]Y

`f` may be any dyadic function that returns a result. `Y` may be any array whose items in the sub-arrays along the `K`th axis are appropriate to the function `f`.

The axis specification is optional. If present, `K` must identify an axis of `Y`. If absent, the last axis of `Y` is implied. The form `R←f⍀Y` implies the first axis of `Y`.

`R` is an array formed by successive reductions along the `K`th axis of `Y`. If `V` is a typical vector taken from the `K`th axis of `Y`, then the `I`th element of the result is determined as `f/I↑V`.

The shape of `R` is the same as the shape of `Y`. If `Y` is an empty array, then `R` is the same empty array.

Examples

```
      ∨\0 0 1 0 0 1 0
0 0 1 1 1 1 1

      ^\1 1 1 0 1 1 1
1 1 1 0 0 0 0

      +\1 2 3 4 5
1 3 6 10 15

      +\(1 2 3)(4 5 6)(7 8 9)
 1 2 3  5 7 9  12 15 18
```

```
      M
1 2 3
4 5 6

      +⍀M
1 3  6
4 9 15

      +⌿M
1 2 3
5 7 9

      +⍀[1]M
1 2 3
5 7 9

      ,⍀'ABC'
A AB  ABC

      T←'ONE(TWO) BOOK(S)'

      ≠⍀T∈'()'
0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0

      ((T∈'()')⍲≠⍀T∈'()')/T
ONE BOOK
```

4.4.19 Scan First R←f⌿Y

The form `R←f⌿Y` implies scan along the first axis of `Y`. See *Section 4.4.18*.

4.4.20 Spawn {R}←{X}f&Y

`&` is a monadic operator with an ambivalent derived function. `&` spawns a new thread in which `f` is applied to its argument `Y` (monadic case) or between its arguments `X` and `Y` (dyadic case). The shy result of this application is the number of the newly created thread.

When function f terminates, its result (if any), the **thread result**, is returned. If the thread number is the subject of an active `⎕TSYNC`, the thread result appears as the result of `⎕TSYNC`. If no `⎕TSYNC` is in effect, the thread result is displayed in the session in the normal fashion.

Note that & can be used in conjunction with the **each** operator ¨ to launch many threads in parallel.

EXAMPLES

```
      ÷&4          ⍝ Reciprocal in background
0.25

      ⎕←÷&4        ⍝ Show thread number
1
0.25

      FOO&88       ⍝ Spawn monadic function.

      2 FOO&3      ⍝ dyadic

      {NIL}&0      ⍝ niladic

      ⍎&'NIL'      ⍝ ..

      X.GOO&99     ⍝ thread in remote space.

      ⍎&'⎕DL 2'    ⍝ Execute async expression.

      'NS'⍎&'FOO'  ⍝ .. remote .. .. ..

      PRT&¨↓⎕NL 9  ⍝ PRT spaces in parallel.
```

4.4.21 Stencil R←(f⌺g)Y

> **Classic Edition**
>
> The symbol ⌺ is not available in Classic Edition, and the Stencil operator is instead represented by ⎕U233A.

Stencil is used in image processing, artificial neural networks, computational fluid dynamics, cellular automata, and many other fields of application. The computation is sometimes referred to as tessellation, moving window, or stencil code. This operator applies the left operand function f to a series of (possibly overlapping) rectangles in the array Y.

In general, the right operand `⍵` is a 2- row matrix of positive non-zero integers with up to `⍴⍴⍺` columns. The first row contains the rectangle sizes, the second row the *movements* that is, how much to move the rectangle in each step. If `⍵` is a scalar or vector it specifies the rectangle size and the movement defaults to 1.

The predominant case uses a rectangle size which is odd and a movement of 1.

Rectangles are *centred* on successive elements of `⍺` and (unless the rectangle size is 1), padded with fill elements.

The first rectangle is centred on the first element of `⍺` preceded by the appropriate number of fill elements. Subsequent rectangles are centred on subsequent elements of `⍺` according to the size of the movement, and padded before or after as appropriate. When the movement is 1, each element of `⍺` in its turn is the middle of a rectangle.

`⍶` is invoked dyadically with a vector left argument indicating for each axis the number of fill elements and on what side; positive values mean that the padding precedes the array values, negative values mean that the padding follows the array values.

Example

```
      {⊂⍺ ⍵}⌺3 3⊢3 3⍴⍳12
┌─────────────┬─────────────┬─────────────┐
│┌─────┬─────┐│┌─────┬─────┐│┌─────┬─────┐│
│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│
│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│
│1 1 0 0 0│1 0 0 0 0│1 ¯1 0 0 0│
│  0 1 2  │  1 2 3  │  2 3 0  │
│  0 4 5  │  4 5 6  │  5 6 0  │
├─────────┼─────────┼─────────┤
│┌─────┬─────┐│┌─────┬─────┐│┌─────┬─────┐│
│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│
│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│
│0 1 0 1 2│0 0 1 2 3│0 ¯1 2 3 0│
│  0 4 5  │  4 5 6  │  5 6 0  │
│  0 7 8  │  7 8 9  │  8 9 0  │
├─────────┼─────────┼─────────┤
│┌─────┬─────┐│┌─────┬─────┐│┌─────┬─────┐│
│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│├─┬─┬─┼─┬─┬─┤│
│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│└─┴─┴─┼─┴─┴─┘│
│¯1 1 0 4 5│¯1 0 4 5 6│¯1 ¯1 5 6 0│
│  0 7 8  │  7 8 9  │  8 9 0  │
│  0 0 0  │  0 0 0  │  0 0 0  │
└─────────┴─────────┴─────────┘
```

```
      {+/,⍵}⌺3 3⊢3 3⍴⍳12
12 21 16
27 45 33
24 39 28
```

In the first expression above, the left operand function `{⊂⍺ ⍵}` simply displays its left and right arguments to illustrate the mechanics of the operation. The right operand `(3 3)` specifies that each rectangle contains 3 rows and 3 columns, and the movement is 1.

In order for the first element of `⍳` (1) to be centred, the first rectangle is padded with a row above and a column to the left, as indicated by the left argument `(1 1)` to the function.

Another way to think about the way Stencil operates is that it portions the array into sections or neighbourhoods in which elements can be analysed with respect to their immediate neighbours. Stencil has uses in image processing applications.

Examples

```
      {⊂⍺ ⍵}⌺(3 3,[.5]2)⊢3 3⍴⍳12
┌─────────────┬─────────────┐
│┌─────┬─────┐│┌─────┬─────┐│
│|1 1|0 0 0|||1 ¯1|0 0 0|||
│|   |0 1 2|||   |2 3 0|||
│|   |0 4 5|||   |5 6 0|||
│└─────┴─────┘│└─────┴─────┘│
├─────────────┼─────────────┤
│┌─────┬─────┐│┌─────┬─────┐│
│|¯1 1|0 4 5|||¯1 ¯1|5 6 0|||
│|    |0 7 8|||   |8 9 0|||
│|    |0 0 0|||   |0 0 0|||
│└─────┴─────┘│└─────┴─────┘│
└─────────────┴─────────────┘
      {⊂⍺ ⍵}⌺(3 3,[.5]3)⊢3 3⍴⍳12
┌─────────────┐
│┌─────┬─────┐│
│|1 1|0 0 0|||
│|   |0 1 2|||
│|   |0 4 5|||
│└─────┴─────┘│
└─────────────┘
```

```
      ⊢ A←5 5⍴0 0 1 0 0, 0 1 2 1 0, 1 2 3 2 1, 0 1 2 1 0
0 0 1 0 0
0 1 2 1 0
1 2 3 2 1
0 1 2 1 0
0 0 1 0 0
```

```
      ⊢ y←1=?10 10⍴4
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0
1 0 0 0 1 1 0 0 0 1
1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1 0 1 0 0 1 1 0 0 1
0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0
1 0 0 0 1 1 0 1 1 0
```

```
      {+/,A×⍵}⌺(⍴A) ⊢y
0 0 1 0 0 1 0 1 2 3
1 1 2 1 2 3 1 0 1 3
4 4 3 4 6 6 3 1 1 3
6 6 5 4 7 7 4 2 2 3
8 6 5 3 5 6 2 0 1 3
6 5 4 3 5 6 5 2 1 3
5 5 4 4 6 7 8 7 4 3
3 2 2 1 4 7 8 7 5 3
3 1 1 1 3 5 6 6 4 2
3 2 2 3 5 6 7 7 5 3
```

You can see that the result identifies where there are clusters in `y`.

Examples (odd rectangle, movement not 1)

If the movement is greater than one, corresponding portions are skipped as shown below.

```
      {⊂⍵}⌺(¯3 2) ⍳8
┌─────┬─────┬─────┬─────┐
│0 1 2│2 3 4│4 5 6│6 7 8│
└─────┴─────┴─────┴─────┘
      {(¯2↑⍕⍺),' f ',⍕⍵}⌺(¯3 2) ⍳8
 1 f 0 1 2
 0 f 2 3 4
 0 f 4 5 6
 0 f 6 7 8
⍝       ↑ middle
```

```
      {⊂⍵}⌺(¯5 2) ⍳9
┌─────────┬─────────┬─────────┬─────────┬─────────┐
│0 0 1 2 3│1 2 3 4 5│3 4 5 6 7│5 6 7 8 9│7 8 9 0 0│
└─────────┴─────────┴─────────┴─────────┴─────────┘
      {(¯2↑⍕⍺),' f ',⍕⍵}⌺(¯5 2) ⍳9
 2 f 0 0 1 2 3
 0 f 1 2 3 4 5
 0 f 3 4 5 6 7
 0 f 5 6 7 8 9
¯2 f 7 8 9 0 0
⍝         ↑ middle
```

Even Rectangle Size

For even rectangle sizes, the "middle" consists of two elements which are moved according to the movement parameter (equal to 1 in these examples).

Examples

```
      ⎕←s←{⊂⍵}⌺ 2 ⍳8
┌───┬───┬───┬───┬───┬───┬───┐
│1 2│2 3│3 4│4 5│5 6│6 7│7 8│
└───┴───┴───┴───┴───┴───┴───┘
      {(¯2↑⍕⍺),' f ',⍕⍵}⌺ 2⍳8
 0 f 1 2
 0 f 2 3
 0 f 3 4
 0 f 4 5
 0 f 5 6
 0 f 6 7
 0 f 7 8
⍝     ↑ ↑ middle
```

```
      □←s←{⊂⍵}⌺ 4⍳8
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│0 1 2 3│1 2 3 4│2 3 4 5│3 4 5 6│4 5 6 7│5 6 7 8│6 7 8 0│
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘

      ⍴s
7
      {(¯2↑⍵),' f ',⍵}⌺ 4⍳8
 1 f 0 1 2 3
 0 f 1 2 3 4
 0 f 2 3 4 5
 0 f 3 4 5 6
 0 f 4 5 6 7
 0 f 5 6 7 8
¯1 f 6 7 8 0
⍝       ↑ ↑ middle
```

Examples (even rectangle, movement not 1)

```
      {⊂⍵}⌺(¯4 2) ⍳8
┌─────────┬─────────┬─────────┬─────────┐
│0 1 2 3│2 3 4 5│4 5 6 7│6 7 8 0│
└─────────┴─────────┴─────────┴─────────┘

      {(¯2↑⍵),' f ',⍵}⌺(¯4 2) ⍳8
 1 f 0 1 2 3
 0 f 2 3 4 5
 0 f 4 5 6 7
¯1 f 6 7 8 0
⍝       ↑ ↑ middle
```

```
      {⊂⍵}⌺(¯6 2) ⍳8
┌─────────────┬─────────────┬─────────────┬─────────────┐
│0 0 1 2 3 4│1 2 3 4 5 6│3 4 5 6 7 8│5 6 7 8 0 0│
└─────────────┴─────────────┴─────────────┴─────────────┘

      {(⍵),' f ',⍵}⌺(¯6 2) ⍳8
2 f 0 0 1 2 3 4
0 f 1 2 3 4 5 6
0 f 3 4 5 6 7 8
¯2 f 5 6 7 8 0 0
⍝         ↑ ↑ middle
```

4.4.22 Variant {R}←{X}(f⍠B)Y

> **Classic Edition**
>
> The symbol ⍠ is not available in Classic Edition, and the Variant operator is instead represented by ⎕U2360. Note too that ⍠ and ⎕OPT are synonymous though only the latter is available in the Classic Edition.

The Variant operator ⍠ specifies the value of an *option* to be used by its left operand function `f`. An *option* is a named property of a function whose value in some way affects the operation of that function.

For example, the Search and Replace operators include options named `IC` and `Mode` which respectively determine whether or not *case* is ignored and in what manner the input document is processed.

One of the set of options may be designated as the *Principal option* whose value may be set using a short-cut form of syntax as described below. For example, the Principal option for the Search and Replace operators is `IC`.

For the operand function with right argument `Y` and optional left argument `X`, the right operand `B` specifies the values of one or more options that are applicable to that function. If `B` is empty, function `f` is called with default options. Otherwise, `B` may be a scalar, a 2-element vector, or a vector of 2-element vectors which specifies values for one or more options as follows:

- If `B` is a 2-element vector and the first element is a character vector, it specifies an option name in the first element and the option value (which may be any suitable array) in the second element.
- If `B` is a vector of 2-element vectors, each item of B is interpreted as above.
- If `B` is a scalar (a rank-0 array of any depth), it specifies the value of the Principal option,

Option names and their values must be appropriate for the left operand function, otherwise `DOMAIN ERROR` (error code 11) will be reported.

Example

```
      tn←'Dick'(⎕FCREATE⍠'Z' 1)0
```

The following illustrations and examples apply to functions derived from the Search and Replace operators.

Examples of operand `B`

The following expression sets the IC option to `1`, the Mode option to `'D'` and the EOL option to `'LF'`.

```
      ⍠('Mode' 'D')('IC' 1)('EOL' 'LF')
```

The following expression sets just the EOL option to 'CR'.

```
      ⍠'EOL' 'CR'
```

The following expression sets just the Principal option (which for the Search and Replace operators is IC) to 1.

```
      ⍠ 1
```

The order in which options are specified is typically irrelevant but if the same option is specified more than once, the rightmost one dominates. The following expression sets the option IC to 1:

```
      ⍠('IC' 0) ('IC' 1)
```

The Variant operator generates a derived function `f⍠B` and may be assigned to a name. The derived function is effectively function `f` bound with the option values specified by `B`.

The derived function may itself be used as a left operand to Variant to produce a second derived function whose options are further modified by the second application of the operator. The following sets the same options as the first example above:

```
      ⍠'Mode' 'D'⍠'IC' 1⍠'EOL' 'LF'
```

When the same option is specified more than once in this way, the outermost (rightmost) one dominates. The following expression also sets the option `IC` to 1:

```
      ⍠'IC' 0⍠'IC' 1
```

Further Examples

The following derived function returns the location of the word `'variant'` within its right argument using default values for all the options.

```
      f1 ← 'variant' ⍷S 0
      f1 'The variant Variant operator'
4
```

It may be modified to perform a case-insensitive search:

```
      (f1 ⍠ 1) 'The variant Variant operator'
4 12
```

This modified function may be named:

```
      f2 ← f1 ⍠ 1
      f2 'The variant Variant operator'
4 12
```

The modified function may itself be modified, in this case to revert to a case sensitive search:

```
      f3 ← f2 ⍠ 0
      f3 'The variant Variant operator'
4
```

This is equivalent to:

```
      (f1 ⍠ 1 ⍠ 0) 'The variant Variant operator'
4
```

Redundancy

A function that has been modified by Variant may be used in a context where the option or options that have been modified are not relevant. This is illustrated by the following example.

`JSON` is a character matrix that specifies 2 objects `a` and `b`.

```
      JSON
{
"a": "The answer is",
"b" : 42
}      ⍴JSON
4 21
```

It can be imported to a matrix `mat` ...

```
      ⊢mat←(⎕JSON⍠('Format' 'M'))JSON
┌─┬─┬──────────────┬─┐
│0│ │              │1│
├─┼─┼──────────────┼─┤
│1│a│The answer is │4│
├─┼─┼──────────────┼─┤
│1│b│42            │3│
└─┴─┴──────────────┴─┘
```

This result may be exported

```
      1(⎕JSON⍠('Format' 'M')) mat
{"a":"The answer is","b":42}
```

The ('Compact' 0) variant option delivers a more readable result:

```
      1(⎕JSON⍠('Format' 'M')('Compact' 0)) mat
{
  "a": "The answer is",
  "b": 42
}
```

A modified version of `⎕JSON` ...

```
      myJSON←(⎕JSON⍠('Format' 'M')('Compact' 0))
      myJSON
┌──────┬─┬──────────────────────────┐
│⎕JSON │⍠│┌──────────┬─────────────┐│
│      │ ││┌────────┬─┐│┌─────────┬─┐│
│      │ ||Format|M|||Compact|0|||
│      │ ||└────────┴─┘|└─────────┴─┘||
│      │ │└──────────┴─────────────┘│
└──────┴─┴──────────────────────────┘
```

... may be used both to import and export, despite the fact that the Compact option applies only to JSON export. In the context of JSON import it is redundant and ignored.

```
      ⊢mat← myJSON JSON
┌─┬─┬─────────────┬─┐
│0│ │             │1│
├─┼─┼─────────────┼─┤
│1│a│The answer is│4│
├─┼─┼─────────────┼─┤
│1│b│42           │3│
└─┴─┴─────────────┴─┘
      1 myJSON mat
{
  "a": "The answer is",
  "b": 42
}
```

Variant and .NET

The Variant operator may also be used in conjunction with .NET classes; it can used to cast an array into a specific .NET data type, and to specify which constructor should be used when creating a new instance of a .NET class which has overloaded constructors. For further information, see *.NET Framework Interface: Advanced Techniques*.

5 Other Syntax

5.1 Abort →

This is a special case of the Branch function used in the niladic sense. If it occurs in a statement it must be the only symbol in an expression or the only symbol forming an expression in a text string to be executed by ⍎. It clears the most recently suspended statement and all of its pendent statements from the state indicator.

The Abort function has no explicit result. The function is not in the function domain of operators.

EXAMPLES

```
      ∇ F
[1]    'F[1]'
[2]    G
[3]    'F[3]'
      ∇

      ∇ G
[1]    'G[1]'
[2]    →
[3]    'G[3]'
      ∇

      F
F[1]
G[1]

      ⎕VR'VALIDATE'
     ∇ VALIDATE
 [1]   →(12=1↑⎕AI)⍴0 ⋄ 'ACCOUNT NOT AUTHORISED' ⋄ →
     ∇

      VALIDATE
ACCOUNT NOT AUTHORISED

      1↑⎕AI
52
```

5.2 Assignment

5.2.1 Assignment X←Y

Assignment allocates the result of the expression Y to the *name* or *names* in X.

If Y is an array expression, X must contain one or more names which are variables, system variables, or are undefined. Following assignment, the name(s) in X become variable(s) with value(s) taken from the result of the expression Y.

If X contains a single name, the variable assumes the value of Y. If X contains multiple names then Y can be a single-item array of any rank (including a scalar) or a vector. If Y is a single-item array, the scalar value ⊃Y is assigned to all names in X. Otherwise, each element of Y is assigned to the corresponding name in X. Although not mandatory,

Dyalog recommends that the names in `X` are enclosed in parentheses to reduce potential ambiguity in assignment statements.

The assignment arrow (or specification arrow) is often read as 'Is' or 'Gets'.

Examples of single assignment

```
      A←2.3
      A
2.3

      A←⍳3
      A
1 2 3
```

Examples of multiple assignment using parentheses

```
      (A B)←2
      A
2
      B
2

      (P ⎕IO Q)←'TEXT' 1 (1 2 3)
      P
TEXT
      ⎕IO
1
      Q
1 2 3
```

Example of multiple assignment without parentheses

```
      year month day←2017 05 24
      day
24
      month
5
      year
2017
```

Implementation note: erroneous expressions such as `var 3←5` will result in `5` being assigned to `var`, even though a `SYNTAX ERROR` will be generated. In the case of `(var 3)←5` no assignment will be made.

Pass-through assignments are permitted. The value of `Y` is carried through each assignment:

```
      I←J←K←0
      I,J,K
0 0 0
```

Function Assignment

If `Y` is a function expression, `X` must be a single name which is either undefined, or is the name of an existing function or defined operator. `X` may not be the name of a system function, or a primitive symbol.

EXAMPLES

```
      PLUS←+
      PLUS
+

      SUM←+/
      SUM
+/

      MEAN←{(+/⍵)÷⍴⍵}
```

Namespace Reference Assignment

If an expression evaluates to a namespace reference, or *ref*, you may assign it to a name. A name assigned to a simple scalar *ref*, has name class 9, whereas one assigned to an *array* containing *refs* has name class 2.

```
      'f1'⎕WC'Form'
      'ns1' ⎕NS ''

      N←ns1
      ⎕NC'N'            ⍝ name class of a scalar ref
9
      F←f1
      ⎕NC'F'            ⍝ name class of a scalar ref
9
      refs←N F          ⍝ vector of refs.
      ⎕NC'refs'         ⍝ nameclass of vector.
2
      F2←2⊃refs
      ⎕NC 'F2'
9
```

Re-Assignment

A name that already exists may be assigned a new value if the assignment will not alter its name class, or will change it from 2 to 9 or vice versa. The table of permitted re-assignments is as follows:

	Ref	Variable	Function	Operator
Ref	Yes	Yes		
Variable	Yes	Yes		
Function			Yes	Yes
Operator			Yes	Yes

5.2.2 Assignment (Modified) {R}←X f←Y

`f` may be any dyadic function which returns an explicit result. `Y` may be any array appropriate to function `f`. `X` must be the *name* of an existing array appropriate to function `f`.

`R` is the “pass-through” value, that is, the value of `Y`. If the result of the derived function is not assigned or used, there is no explicit result.

The effect of the derived function is to reset the value of the array named by `X` to the result of `XfY`.

EXAMPLES

```
      A
1 2 3 4 5

      A+←10

      A
11 12 13 14 15

      ⎕←A×←2
2
      A
22 24 26 28 30

      vec←¯4+9?9 ⋄ vec
3 5 1 ¯1 ¯2 4 0 ¯3 2
      vec/⍨←vec>0 ⋄vec
3 5 1 4 2
```

5.2.3 Assignment (Indexed) {R}←X[I]←Y

Indexed Assignment is the Assignment function modified by the Indexing function. The phrase `[I]←` is treated as the function for descriptive purposes.

`Y` may be any array. `X` may be the *name* of any array or a selection from a named array `(EXP X)[I]←Y`, see *Section 5.2.5*. `I` must be a valid index specification. The shape of `Y` must conform with the shape (implied) of the indexed structure defined by `I`. If `Y` is a scalar or a 1-element vector it will be extended to conform. A side effect of Indexed Assignment is to change the value of the indexed elements of `X`.

`R` is the value of `Y`. If the result is not explicitly assigned or used it is suppressed.

`⎕IO` is an implicit argument of Indexed Assignment.

Three forms of indexing are permitted.

Simple Indexed Assignment

For vector `X`, `I` is a simple integer array whose items are from the set `⍳⍴R`. Elements of `X` identified by index positions `I` are replaced by corresponding elements of `Y`.

Examples

```
      +A←⍳5
1 2 3 4 5

      A[2 3]←10 ⋄ A
1 10 10 4 5
```

The last-most element of `Y` is assigned when an index is repeated in `I`:

```
      A[2 2]←100 101 ⋄ A
1 101 10 4 5
```

For matrix `X`, `I` is composed of two simple integer arrays separated by the semicolon character (`;`). The arrays select indices from the rows and columns of `X` respectively.

Examples

```
      +B←2 3⍴'REDSUN'
RED
SUN

      B[2;2]←'O' ⋄ B
RED
SON
```

For higher-rank array `X`, `I` is a series of simple integer arrays with adjacent arrays separated by a single semicolon character (`;`). Each array selects indices from an axis of `X` taken in row-major order.

Examples

```
      C
11 12 13
14 15 16

21 22 23
24 25 26

      C[1;1;3]←103 ⋄ C
11 12 103
14 15  16

21 22  23
24 25  26
```

An indexing array may be ELIDED. That is, if an indexing array is omitted from the `K`th axis, the indexing vector `⍳(⍴X)[K]` is implied:

```
      C[;1;2 3]←2 2⍴112 113 122 123 ⋄ C
11 112 113
14  15  16

21 122 123
24  25  26

      C[;;]←0 ⋄ C
0 0 0
0 0 0

0 0 0
0 0 0
```

Choose Indexed Assignment

The index specification `I` is a non-simple integer array. Each item identifies a single element of `X` by a set of indices with one element per axis of `X` in row-major order.

EXAMPLES

```
      C
11 12 13 14
21 22 23 24

      C[⊂1 1]←101 ⋄ C
101 12 13 14
 21 22 23 24

      C[(1 2) (2 3)]←102 203 ⋄ C
101 102  13 14
 21  22 203 24

      C[2 2⍴(1 3)(2 4)(2 1)(1 4)]←2 2⍴103 204 201 104 ⋄ C
101 102 103 104
201  22 203 204
```

A scalar may be indexed by the enclosed empty vector:

```
      S
10
      S[⊂⍳0]←⊂'VECTOR' ⋄ S
 VECTOR
      S[⊂⍳0]←5 ⋄ S
5
```

Choose Indexed Assignment may be used very effectively in conjunction with Index Generator (⍳) and Structural functions in order to assign into an array:

```
      C
11 12 13 14
21 22 23 24

      ⍳⍴C
1 1  1 2  1 3  1 4
2 1  2 2  2 3  2 4

      C[1 1⌽⍳⍴C]←1 2 ⋄ C
 1 12 13 14
21  2 23 24

      C[2 ¯1↑⍳⍴C]←99 ⋄ C
 1 12 13 99
21  2 23 99
```

Reach Indexed Assignment

The index specification `I` is a non-simple integer array, each of whose items reach down to a nested element of `X`. The items of an item of `I` are simple vectors (or scalars) forming sets of indices that index arrays at successive levels of `X` starting at the top-most level. A set of indices has one element per axis at the respective level of nesting of `X` in row-major order.

Examples

```
      D←(2 3⍴⍳6)(2 2⍴'SMITH' 'JONES' 'SAM' 'BILL')

      D
1 2 3   SMITH  JONES
4 5 6   SAM    BILL

      ≡J←⊂2 (1 2)
¯3

      D[J]←⊂'WILLIAMS' ⋄ D
1 2 3   SMITH  WILLIAMS
4 5 6   SAM    BILL

      D[(1 (1 1))(2 (2 2) 1)]←10 'W' ⋄ D
10 2 3   SMITH  WILLIAMS
 4 5 6   SAM    WILL

      E
GREEN  YELLOW  RED

      E[⊂2 1]←'M' ⋄ E
GREEN  MELLOW  RED
```

The context of indexing is important. In the last example, the indexing method is determined to be Reach rather than Choose since `E` is a vector, not a matrix as would be required for Choose. Observe that:

```
      ⊂2 1 ↔ ⊂(⊂2),(⊂1)
```

Note that for any array `A`, `A[⊂⍬]` represents a scalar quantity, which is the whole of `A`, so:

```
      A←5⍴0
      A
0 0 0 0 0
      A[⊂⍬]←1
      A
1
```

Combined Indexed and Selective Assignment

Instead of `X` being a name, it may be a selection from a named array, and the statement is of the form `(EXP X)[I]←Y`.

```
      MAT←4 3⍴'Hello' 'World'
      (2↑¨MAT)[1 2;]←'#'
      MAT
##llo  ##rld  ##llo
##rld  ##llo  ##rld
Hello  World  Hello
World  Hello  World
```

```
      MAT←4 3⍴'Hello' 'World'
      ⎕ML←1 ⍝ ∈ is Enlist
     (∈MAT)[2×⍳⌊0.5×⍴∈MAT]←'#'
      MAT
H#l#o  #o#l#  H#l#o
#o#l#  H#l#o  #o#l#
H#l#o  #o#l#  H#l#o
#o#l#  H#l#o  #o#l#
```

5.2.4 Assignment (Indexed Modified) {R}←X[I]f←Y

`f` may be any dyadic function which returns an explicit result. `Y` may be any array whose items are appropriate to function `f`. `X` must be the *name* of an existing array. `I` must be a valid index specification. The items of the indexed portion of `X` must be appropriate to function `f`.

`Y` is either an array of the same shape as the indices specified by `I` or a scalar that is notionally extended to be the same shape as those indices.

The operator loops through the indices specified by `I` in ravel order. For each successive index `i` in the set specified by `I`, it calculates the result of `X[i]fY[i]` and assigns it back to `X[i]`.

`R` is the "pass-through" value, that is, the value of `Y`. If the result of the derived function is not assigned or used, there is no explicit result.

Examples

```
      A
1 2 3 4 5

      +A[2 4]+←1
1

      A
1 3 3 5 5

      A[3]÷←2

      A
1 3 1.5 5 5
```

As the operator performs a loop, if an index in `I` is repeated, function `f` will be applied that number of times and successively to the same item of `X`.

Example

```
      B←3 5⍴0
      B[1 1 3;1 3 3 5]+←1
      B
2 0 4 0 2
0 0 0 0 0
1 0 2 0 1
```

5.2.5 Assignment (Selective) (EXP X)←Y

`X` is the *name* of a variable in the workspace, possibly modified by the indexing function `(EXP X[I])←Y`, see *Section 5.2.3*. `EXP` is an expression that **selects** elements of `X`. `Y` is an array expression. The result of the expression `Y` is allocated to the elements of `X` selected by `EXP`. Note that `X` may refer to a single name only.

The following functions may appear in the selection expression. Where appropriate these functions may be used with axis `[]` and with the Each operator ¨.

Table 5-1: *Functions for Selective Assignment*

↑	Take
↓	Drop
,	Ravel
⍪	Table
⌽⊖	Reverse, Rotate
⍴	Reshape
⊃	Disclose, Pick
⍉	Transpose (Monadic and Dyadic)
/⌿	Replicate
\⍀	Expand
[]	Index
∈	Enlist (⎕ML≥1)

Note: Mix and Split (monadic ↑ and ↓), Type (monadic ∈ when ⎕ML<1) and Membership (dyadic ∈) may not be used in the selection expression.

Examples

```
      A←'HELLO'
      ((A∈'AEIOU')/A)←'*'

      A
H*LL*

      Z←3 4⍴⍳12
      (5↑,Z)←0

      Z
0  0  0  0
0  6  7  8
9 10 11 12
```

```
      MAT←3 3⍴⍳9
      (1 1⍉MAT)←0

      MAT
0 2 3
4 0 6
7 8 0

      ⎕ML←1⍝ so ∈ is Enlist
      names←'Andy' 'Karen' 'Liam'
      (('a'=∈names)/∈names)←'*'
      names
 Andy  K*ren  Li*m
```

Each Operator

The functions listed in the table above may also be used with the Each Operator ¨.

EXAMPLES

```
      A←'HELLO' 'WORLD'
      (2↑¨A)←'*'
      A
 **LLO  **RLD

      A←'HELLO' 'WORLD'
      ((A='O')/¨A)←'*'
      A
 HELL*  W*RLD

      A←'HELLO' 'WORLD'
      ((A∈¨⊂'LO')/¨A)←'*'
      A
 HE***  W*R*D
```

Bracket Indexing

Bracket indexing may also be applied to the expression on the left of the assignment arrow.

EXAMPLES

```
      MAT←4 3⍴'Hello' 'World'
      (¯2↑¨MAT[;1 3])←'$'
      MAT
Hel$$  World  Hel$$
Wor$$  Hello  Wor$$
Hel$$  World  Hel$$
Wor$$  Hello  Wor$$
```

5.2.6 Assignment (Selective Modified) {R}←(EXP X)f←Y

`f` may be any dyadic function which returns an explicit result. `Y` may be any array whose items are appropriate to function `f`. `X` must be the *name* of an existing array. `EXP` is an expression that **selects** elements of `X`. (See *Section 5.2.5* for a list of allowed selection functions.) The selected elements of `X` must be appropriate to function `f`.

`Y` is either an array of the same shape as the selected elements of `X` or a scalar that is notionally extended to be the same shape as the selection.

The operator loops through the selected elements of `X` in ravel order. For each selected element `X[i]`, it calculates the result of `X[i]fY[i]` and assigns it back to the same element `X[i]`.

`R` is the "pass-through" value, that is, the value of `Y`. If the result of the derived function is not assigned or used, there is no explicit result.

EXAMPLE

```
      A
12 36 23 78 30

      ((A>30)/A) ×← 100
      A
12 3600 23 7800 30
```

As the operator performs a loop, if an element of `X` is selected more than once, function `f` will be applied the corresponding number of times and successively to the same element of `X`.

```
      a←3⍴0
      (5⍴a)+←1
      a
2 2 1
```

5.3 Axis

5.3.1 Axis (with Monadic Operand) R←f[B]Y

`f` must be a monadic primitive mixed function taken from those shown in *Table 5-2* below, or a function derived from the operators Reduction (`/`) or Scan (`\`). `B` must be a numeric scalar or vector. `Y` may be any array whose items are appropriate to function `f`.

For an alternative method of applying any function to a subset of axes, see *Section 4.4.15*.

Table 5-2: *Primitive monadic mixed functions with optional axis.*

Function	Name	Range of B
`⌽ or ⊖`	Reverse	`B∊⍳⍴⍴Y`
`↑`	Mix	`(0≠1|B)^(B>⎕IO-1)^(B<⎕IO+⍴⍴Y)`
`↓`	Split	`B∊⍳⍴⍴Y`
`,`	Ravel	fraction, or zero or more axes of `Y`
`⊂`	Enclose	`(B≡⍳0)∨(^/B∊⍳⍴⍴Y)`

In most cases, `B` must be an integer which identifies a specific axis of `Y`. However, when `f` is the Mix function (`↑`), `B` can be a fractional value whose lower and upper integer bounds select an adjacent pair of axes of `Y` or an extreme axis of `Y`.

For Ravel (`,`) and Enclose (`⊂`), `B` can be a **vector** of two or more axes.

`⎕IO` is an implicit argument of the derived function which determines the meaning of `B`.

EXAMPLES

```
      ⌽[1]2 3⍴⍳6
4 5 6
1 2 3

      ↑[0.1]'ONE' 'TWO'
OT
NW
EO
```

5.3.2 Axis (with Dyadic Operand) `R←XfBY`

`f` must be a dyadic primitive scalar function, or a dyadic primitive mixed function taken from *Table 5-3* below. `B` must be a numeric scalar or vector. `X` and `Y` may be any arrays whose items are appropriate to function `f`.

For an alternative method of applying any function to a subset of axes, see *Section 4.4.15*.

Table 5-3: *Primitive dyadic mixed functions with optional axis.*

Function	Name	Range of B
`/ or ⌿`	Replicate	`B∊⍳⍴⍴Y`
`\ or ⍀`	Expand	`B∊⍳⍴⍴Y`
`⊂`	Partitioned Enclose	`B∊⍳⍴⍴Y`
`⌽ or ⊖`	Rotate	`B∊⍳⍴⍴Y`
`, or ⍪`	Catenate	`B∊⍳⍴⍴Y`
`, or ⍪`	Laminate	`(0≠1|B)^(B>⎕IO-1)^(B<⎕IO+(⍴⍴X)⌈⍴⍴Y)`
`↑`	Take	one or more axes of `Y`
`↓`	Drop	one or more axes of `Y`
`⌷`	Index	one or more axes of `Y`

In most cases, `B` must be an integer value identifying the axis of `X` and `Y` along which function `f` is to be applied.

Exceptionally, `B` must be a fractional value for the Laminate function (`,`) whose upper and lower integer bounds identify a pair of axes or an extreme axis of `X` and `Y`. For Take (`↑`) and Drop (`↓`), `B` can be a **vector** of two or more axes.

`⎕IO` is an implicit argument of the derived function which determines the meaning of `B`.

Examples

```
      1 4 5 =[1] 3 2⍴⍳6
1 0
0 1
1 0

      2 ¯2 1/[2]2 3⍴'ABCDEF'
AA  C
DD  F

      'ABC',[1.1]'='
A=
B=
C=

      'ABC',[0.1]'='
ABC
===

      ⎕IO←0

      'ABC',[¯0.5]'='
ABC
===
```

Axis with Scalar Dyadic Functions

Axis `[X]` can take a scalar dyadic function as operand. This has the effect of "stretching" a lower rank array to fit a higher rank one. The arguments must be conformable along the specified axis (or axes) with elements of the lower rank array being replicated along the other axes.

For example, if `H` is the higher rank array, `L` the lower rank one, `X` is an axis specification, and `f` a scalar dyadic function, then the expressions `Hf[X]L` and `Lf[X]H` are conformable if `(⍴L)≡⍴H)[X]`. Each element of L is replicated along the remaining `(⍴H)~X` axes of `H`.

In the special case where both arguments have the same rank, the right one will play the role of the higher rank array. If `R` is the right argument, `L` the left argument, `X` is an axis specification and `f` a scalar dyadic function, then the expression `Lf[X]R` is conformable if `(⍴L)≡⍴R)[X]`.

Examples

```
      mat
10 20 30
40 50 60

      mat+[1]1 2       ⍝ add along first axis
11 21 31
42 52 62

      mat+[2]1 2 3     ⍝ add along last axis
11 22 33
41 52 63

      cube
 100  200  300
 400  500  600

 700  800  900
1000 1100 1200

      cube+[1]1 2
 101  201  301
 401  501  601

 702  802  902
1002 1102 1202

      cube+[3]1 2 3
 101  202  303
 401  502  603

 701  802  903
1001 1102 1203

      cube+[2 3]mat
 110  220  330
 440  550  660

 710  820  930
1040 1150 1260
```

```
      cube+[1 3]mat
 110  220  330
 410  520  630

 740  850  960
1040 1150 1260
```

5.4 Branch →Y

Y may be a scalar or vector which, if not empty, has a simple numeric scalar as its first element. The function has no explicit result. It is used to modify the normal sequence of execution of expressions or to resume execution after a statement has been interrupted. Branch is not in the function domain of operators.

The following distinct usages of the branch function occur:

	Entered in a Statement in a Defined Function	Entered in Immediate Execution Mode
→LINE	Continue with the specific line	Restart execution at the specific line of the most recently suspended function
→⍳0	Continue with the next expression	No effect

In a defined function, if Y is non-empty then the first element in Y specifies a statement line in the defined function to be executed next. If the line does not exist, then execution of the function is terminated. For this purpose, line 0 does not exist. (Note that statement line numbers are independent of the index origin ⎕IO).

If Y is empty, the branch function has no effect. The next expression is executed on the same line, if any, or on the next line if not. If there is no following line, the function is terminated.

The :GoTo statement may be used in place of Branch in a defined function.

EXAMPLE

```
     ∇ TEST
[1]    1
[2]    →4
[3]    3
[4]    4
     ∇

      TEST
1
4
```

In general it is better to branch to a LABEL than to a line number. A label occurs in a statement followed by a colon and is assigned the value of the statement line number when the function is defined.

EXAMPLE

```
     ∇ TEST
[1]    1
[2]    →FOUR
[3]    3
[4]   FOUR:4
     ∇
```

The previous examples illustrate unconditional branching. There are numerous APL idioms which result in conditional branching. Some popular idioms are identified in the following table:

Branch Expression	Comment
`→TEST/L1`	Branches to label `L1` if `TEST` results in 1 but not if `TEST` results in 0.
`→TEST⍴L1`	Similar to above.
`→TEST↑L1`	Similar to above.
`→L1⍴⍨TEST`	Similar to above.
`→L1⌈⍳TEST`	Similar to above but only if `⎕IO←→1` .
`→L1×⍳TEST`	Similar to above but only if `⎕IO←→1` .
`→(L1,L2,L3)[N]`	Unconditional branch to a selected label.
`→(T1,T2,T3)/ L1,L2,L3`	Branches to the first selected label dependent on tests `T1` , `T2` , `T3` . If all tests result in 0, there is no branch.
`→N⌽L1,L2,L3`	Unconditional branch to the first label after rotation.

A branch expression may occur within a statement including ⋄ separators:

```
[5]    →NEXT⍴⍨TEST ⋄ A←A+1 ⋄ →END
[6]   NEXT:
```

In this example, the expressions `'A←A+1'` and `'→END'` are executed only if `TEST` returns the value 1. Otherwise control branches to label `NEXT`.

In immediate execution mode, the branch function permits execution to be continued within the most recently suspended function, if any, in the state indicator. If the state indicator is empty, or if the argument `Y` is the empty vector, the branch expression has no effect. If a statement line is specified which does not exist, the function is terminated. Otherwise, execution is restarted from the beginning of the specified statement line in the most recently suspended function.

EXAMPLE

```
     ∇ F
[1]   1
[2]   2
[3]   3
     ∇

      2 ⎕STOP'F'
      F
1

F[2]
      )SI
#.F[2]*
      →2
2
3
```

The system constant `⎕LC` returns a vector of the line numbers of statement lines in the state indicator, starting with that in the most recently suspended function. It is convenient to restart execution in a suspended state by the expression:

```
      →⎕LC
```

5.5 Indexing R←X[Y]

`X` may be any array. `Y` must be a valid index specification. `R` is an array composed of elements indexed from `X` and the shape of `R` is determined by the index specification.

For an alternative method of indexing, see *Section 3.4.35.1*.

`⎕IO` is an implicit argument of Indexing.

Three forms of indexing are permitted. The form used is determined by context.

Simple Indexing

For vector `X`, `Y` is a simple integer array composed of items from the set `⍳⍴X`.

`R` consists of elements selected according to index positions in `Y`. `R` has the same shape as `Y`.

Examples

```
      A←10 20 30 40 50

      A[2 3⍴1 1 1 2 2 2]
10 10 10
20 20 20

      A[3]
30

      'ONE' 'TWO' 'THREE'[2]
 TWO
```

For matrix `X`, `Y` is composed of two simple integer arrays separated by the semicolon character (`;`). The arrays select indices from the rows and columns of `X` respectively.

Examples

```
      +M←2 4⍴10×⍳8
10 20 30 40
50 60 70 80

      M[2;3]
70
```

For higher-rank array `X`, `Y` is composed of a simple integer array for each axis of `X` with adjacent arrays separated by a single semicolon character (`;`). The arrays select indices from the respective axes of `X`, taken in row-major order.

Examples

```
      ⊢A←2 3 4⍴10×⍳24
 10  20  30  40
 50  60  70  80
 90 100 110 120

130 140 150 160
170 180 190 200
210 220 230 240

      A[1;1;1]
10

      A[2;3 2;4 1]
240 210
200 170
```

If an indexing array is omitted for the `K`th axis, the index vector `⍳(⍴X)[K]` is assumed for that axis.

Examples

```
      A[;2;]
 50  60  70  80
170 180 190 200

      M
10 20 30 40
50 60 70 80

      M[;]
10 20 30 40
50 60 70 80

      M[1;]
10 20 30 40

      M[;1]
10 50
```

Choose Indexing

The index specification `Y` is a non-simple array. Each item identifies a single element of `X` by a set of indices with one element per axis of `X` in row-major order.

EXAMPLES

```
      M
10 20 30 40
50 60 70 80

      M[⊂1 2]
20

      M[2 2⍴⊂2 4]
80 80
80 80

      M[(2 1)(1 2)]
50 20
```

A scalar may be indexed by the enclosed empty vector:

```
      S←'Z'
      S[3⍴⊂⍳0]
ZZZ
```

Simple and Choose indexing are indistinguishable for vector `X`:

```
      V←10 20 30 40

      V[⊂2]
20
      ⊂2
2
      V[2]
20
```

Reach Indexing

The index specification `Y` is a non-simple integer array, each of whose items reach down to a nested element of `X`. The items of an item of `Y` are simple vectors (or scalars) forming sets of indices that index arrays at successive levels of `X` starting at the top-

most level. A set of indices has one element per axis at the respective level of nesting of X in row-major order.

EXAMPLES

```
      G←('ABC' 1)('DEF' 2)('GHI' 3)('JKL' 4)
      G←2 3⍴G,('MNO' 5)('PQR' 6)
      G
  ABC  1   DEF  2   GHI  3
  JKL  4   MNO  5   PQR  6

      G[((1 2)1)((2 3)2)]
 DEF  6

      G[2 2⍴⊂(2 2)2]
5 5
5 5
      G[⊂⊂1 1]
  ABC  1

      G[⊂1 1]
  ABC  1

      V←,G

      V[⊂⊂1]
  ABC  1

      V[⊂1]
  ABC  1

      V[1]
  ABC  1
```

Indexing Classes

If Y is a ref to an instance of a Class with a Default property, indexing is applied to the Default property. Similarly, indexing applied to a .NET collection returns the appropriate item(s) of the collection.

EXAMPLE

```
      ↑⎕SRC c
:Class c
    :Property Default p
    :Access Public Shared
        ∇ r←get
          r←2 3 ⍴⍳4⎕A
        ∇
    :EndProperty
:EndClass
      c[2;3;]
UVWX
```

5.6 Zilde R←⍬

The empty vector (⍳0) may be represented by the numeric constant ⍬ called ZILDE.

6 System Functions and Variables

6.1 Introduction

Dyalog includes a collection of built-in facilities that provide various services related to both the APL and the external environment. They have distinguished case-insensitive names beginning with the ⎕ symbol, and are implicitly available in a clear workspace. Collectively, these facilities are referred to as **System Functions** but they are variously implemented as constants, variables, functions, operators, and in one case, as a namespace.

Note that Dyalog can extend any of these facilities by, for example, adding extra elements, rows, or columns to a result, so code should take this possibility into account.

6.2 System Functions and Variables (A-Z)

□A	□AI	□AN	□ARBIN	□ARBOUT
□AT	□ATX	□AV	□AVU	□BASE
□C	□CLASS	□CLEAR	□CMD	□CR
□CS	□CSV	□CT	□CY	□D
□DCT	□DF	□DIV	□DL	□DM
□DMX	□DQ	□DR	□DT	□ED
□EM	□EN	□EX	□EXCEPTION	□EXPORT
□FAPPEND	□FAVAIL	□FCHK	□FCOPY	□FCREATE
□FDROP	□FERASE	□FHIST	□FHOLD	□FIX
□FLIB	□FMT	□FNAMES	□FNUMS	□FPROPS
□FR	□FRDAC	□FRDCI	□FREAD	□FRENAME
□FREPLACE	□FRESIZE	□FSIZE	□FSTAC	□FSTIE
□FTIE	□FUNTIE	□FX	□INSTANCES	□IO
□JSON	□KL	□LC	□LOAD	□LOCK
□LX	□MAP	□MKDIR	□ML	□MONITOR
□NA	□NAPPEND	□NC	□NCOPY	□NCREATE
□NDELETE	□NERASE	□NEW	□NEXISTS	□NGET
□NINFO	□NL	□NLOCK	□NMOVE	□NNAMES
□NNUMS	□NPARTS	□NPUT	□NQ	□NR
□NREAD	□NRENAME	□NREPLACE	□NRESIZE	□NS
□NSI	□NSIZE	□NTIE	□NULL	□NUNTIE
□NXLATE	□OFF	□OPT	□OR	□PATH
□PFKEY	□PP	□PROFILE	□PW	□R
□REFS	□RL	□RSI	□RTL	□S
□SAVE	□SD	□SE	□SH	□SHADOW
□SHELL	□SI	□SIGNAL	□SIZE	□SM
□SR	□SRC	□STACK	□STATE	□STOP
□SVC	□SVO	□SVQ	□SVR	□SVS

⎕TALLOC	⎕TC	⎕TCNUMS	⎕TGET	⎕THIS
⎕TID	⎕TKILL	⎕TNAME	⎕TNUMS	⎕TPOOL
⎕TPUT	⎕TRACE	⎕TRAP	⎕TREQ	⎕TS
⎕TSYNC	⎕UCS	⎕USING	⎕VFI	⎕VGET
⎕VR	⎕VSET	⎕WA	⎕WC	⎕WG
⎕WN	⎕WS	⎕WSID	⎕WX	⎕XML
⎕XSI	⎕XT	⎕Á		

6.3 System Functions (by Category)

The following tables list the system functions divided into appropriate categories.

Settings Affecting Behaviour of Primitive Functions

Name	Description
⎕CT	Comparison Tolerance
⎕DCT	Decimal Comp Tolerance
⎕DIV	Division Method
⎕FR	Floating-Point Representation
⎕IO	Index Origin
⎕ML	Migration Level
⎕PP	Print Precision
⎕RL	Random Link

The following table describes the dependencies that exist between functions, operators and these system variables.

Table 6-1: *Implicit Arguments*

System Variable	Monadic Functions	Dyadic Functions	Operators
⎕CT, ⎕DCT	⌈ ⌊ ∪	~ < ≤ = ≥ > ≠ ≡ ≢ ⍳ ∈ ∪ ∩ ⍷ | ∨ ∧	⌸
⎕DIV	÷	÷	
⎕FR ⍝ 1	÷ * ⍟ ! ○ ⌹	+ − × ÷ * ⍟ | ! ○ ∨ ∧ ⊥ ⊤ ⌹	
⎕FR ⍝ 2	⌈ ⌊ ∪	~ < ≤ = ≥ > ≠ ≡ ≢ ⍳ ∈ ∪ ∩ ⍷	⌸
⎕FR ⍝ 3	⍒ ⍋	⌈ ⌊ ⍒ ⍋ ⍳	
⎕IO	⍳ ? ⍒ ⍋ ⍳	⍳ ? ⍒ ⍋ ⍉ ⊃ ⌷ ⍳	⌸ @ []
⎕ML	∈ ↑ ⊃ ≡		
⎕PP	⍕		
⎕RL	?	?	

where, for ⎕FR, 1 indicates functions that compute real numbers and whose precision depends on ⎕FR, 2 indicates functions that perform tolerant comparisons and 3 indicates functions that perform tolerant comparisons.

> **Note**
>
> Tolerant comparisons depend on `⎕FR` to select which of `⎕CT` and `⎕DCT` is used; `⎕FR` also determines the precision of the comparison computation that can affect results. However, even primitives involving intolerant comparison (including the tolerant ones with all comparison tolerances set to 0) can depend on `⎕FR` if the argument contains DECFs. This is because DECFs must be converted to doubles for comparison. If two DECFs are different but correspond to the same double, then they will be treated as intolerantly unequal when `⎕FR` is `1287` but equal when it is `645`.

Session Information/Management

Name	Description
⎕AI	Account Information
⎕AN	Account Name
⎕CLEAR	Clear workspace (WS)
⎕CY	Copy objects into active WS
⎕DL	Delay execution
⎕LOAD	Load a saved WS
⎕OFF	End the session
⎕PATH	Search Path
⎕SAVE	Save the active WS
⎕TS	Timestamp

System Constants

Name	Description
⎕A	Alphabetic upper case characters
⎕D	Digits
⎕NULL	Null Item

Tools and Access to External Utilities

Name	Description
⎕C	Case Convert
⎕CMD	Execute the Windows Command Processor or another program
⎕CMD	Start a Windows AP
⎕CSV	Comma Separated Values
⎕DR	Data Representation (Monadic)
⎕DR	Data Representation (Dyadic)
⎕DT	Datetime
⎕FMT	Resolve display
⎕FMT	Format array
⎕JSON	JSON Convert
⎕MAP	Map a file
⎕NA	Declare a DLL function
⎕R	Replace
⎕S	Search
⎕SH	Execute a UNIX command or another program
⎕SH	Start a UNIX AP
⎕SHELL	Execute a shell command or another program
⎕UCS	Unicode Convert
⎕USING	Microsoft .NET Search Path
⎕VFI	Verify and Fix numeric
⎕XML	XML Convert

Manipulating Functions and Operators

Name	Description
⎕AT	Object Attributes
⎕CR	Canonical Representation
⎕ED	Edit one or more objects
⎕EX	Expunge objects
⎕FX	Fix definition
⎕LOCK	Lock a function
⎕MONITOR	Monitor set
⎕MONITOR	Monitor query
⎕OR	Object Representation
⎕NR	Nested Representation
⎕PROFILE	Profile Application
⎕REFS	Local References
⎕STOP	Set Stop vector
⎕STOP	Query Stop vector
⎕TRACE	Set Trace vector
⎕TRACE	Query Trace vector
⎕VR	Vector Representation

Namespaces and Objects

Name	Description
□BASE	Base Class
□CLASS	Class
□CS	Change Space
□DF	Display Format
□FIX	Fix
□INSTANCES	Instances
□NEW	New Instance
□NS	Namespace
□SRC	Source
□THIS	This

Input and Output

Name	Description
□	Evaluated Input/Output
⍞	Character Input/Output
□ARBIN	Arbitrary Input
□ARBOUT	Arbitrary Output
□RTL	Response Time Limit

Component Files

Name	Description
□FAPPEND	Append a component to File
□FAVAIL	File system Availability
□FCHK	File Check and Repair
□FCOPY	Copy a File
□FCREATE	Create a File
□FDROP	Drop a block of components
□FERASE	Erase a File
□FHIST	File History
□FHOLD	File Hold
□FLIB	List File Library
□FNAMES	Names of tied Files
□FNUMS	Tie Numbers of tied Files
□FPROPS	File Properties
□FRDAC	Read File Access matrix
□FRDCI	Read Component Information
□FREAD	Read a component from File
□FRENAME	Rename a File
□FREPLACE	Replace a component on File
□FRESIZE	File Resize
□FSIZE	File Size
□FSTAC	Set File Access matrix
□FSTIE	Share-Tie a File
□FTIE	Tie a File exclusively
□FUNTIE	Untie Files

Native Files

Name	Description
□MKDIR	Create a directory
□NAPPEND	Append to File
□NCOPY	Copy files and directories
□NCREATE	Create a File
□NDELETE	Delete a File or Directory
□NERASE	Erase a File
□NEXISTS	Discover whether or not a file or directory exists
□NGET	Read Text File
□NINFO	Query or set information about one or more files and/or directories
□NLOCK	Lock a region of a file
□NMOVE	Move files and directories
□NNAMES	Names of tied Files
□NNUMS	Tie Numbers of tied Files
□NPARTS	Split a file name into its constituent parts.
□NPUT	Write Text File
□NREAD	Read from File
□NRENAME	Rename a File
□NREPLACE	Replace data on File
□NRESIZE	File Resize
□NSIZE	File Size
□NTIE	Tie a File exclusively
□NUNTIE	Untie Files
□NXLATE	Specify Translation Table

Threads

Name	Description
□TALLOC	Allocate Token Range
□TCNUMS	Thread Child Numbers
□TID	Current Thread Identity
□TKILL	Kill Threads
□TNAME	Current Thread Name
□TNUMS	Thread Numbers
□TSYNC	Wait for Threads to Terminate

Synchronisation

Name	Description
□TALLOC	Allocate Token Range
□TGET	Get Tokens
□TKILL	Kill Threads
□TPOOL	Token Pool
□TPUT	Put Tokens
□TREQ	Token Requests

Error Handling

Name	Description
□DM	Diagnostic Message
□DMX	Extended Diagnostic Message
□EM	Event Messages
□EN	Event Number
□EXCEPTION	Reports the most recent Microsoft .NET Exception
□SIGNAL	Signal event
□TRAP	Event Trap

Stack and Workspace Information

Name	Description
⎕LC	Line Count
⎕LX	Latent Expression
⎕NC	Name Classification
⎕NL	Name List
⎕NSI	Namespace Indicator
⎕RSI	Space Indicator
⎕SI	State Indicator
⎕SHADOW	Shadow names
⎕SIZE	Size of objects
⎕STACK	Report Stack
⎕STATE	Return State of an object
⎕VGET	Value Get
⎕VSET	Value Set
⎕WA	Workspace Available
⎕WSID	Workspace Identification
⎕XSI	Extended State Indicator

Shared Variables

Name	Description
□SVC	Set access Control
□SVC	Query access Control
□SVO	Shared Variable Offer
□SVO	Query degree of coupling
□SVQ	Shared Variable Query
□SVR	Retract offer
□SVS	Query Shared Variable State

GUI and COM Support

Name	Description
□DQ	Await and process events
□EXPORT	Export objects
□NQ	Place an event on the Queue
□WC	Create GUI object
□WG	Get GUI object properties
□WN	Query GUI object Names
□WS	Set GUI object properties
□WX	Expose GUI property names

Miscellaneous

Name	Description
⎕A	Underscored Alphabetic Characters
⎕AV	Atomic Vector
⎕AVU	Atomic Vector - Unicode
⎕KL	Key Labels
⎕PFKEY	Programmable Function Keys
⎕SD	Screen Dimensions
⎕SM	Screen Map
⎕SR	Screen Read
⎕OPT	Variant Operator
⎕TC	Terminal Control
⎕XT	Associate External variable
⎕XT	Query External variable

System Variables

System variables retain information used by the system in some way. Many system variables affect the behaviour of primitive functions and operators to which they act as *implicit arguments*.

System variables can be localised by inclusion in the header line of a defined function or in the argument list of the system function ⎕SHADOW. When a system variable is localised, it retains its previous value until it is assigned a new one. This feature is known as "pass-through localisation". The exception to this rule is ⎕TRAP.

A system variable can never be undefined. Default values are assigned to all system variables in a clear workspace.

Name	Description	Scope
⎕AVU	Atomic Vector – Unicode	Namespace
⎕CT	Comparison Tolerance	Namespace
⎕DCT	Decimal Comp Tolerance	Namespace
⎕DIV	Division Method	Namespace
⎕FR	Floating-Point Representation	Namespace
⎕IO	Index Origin	Namespace
⎕LX	Latent Expression	Workspace
⎕ML	Migration Level	Namespace
⎕PATH	Search Path	Session
⎕PP	Print Precision	Namespace
⎕PW	Print Width	Session
⎕RL	Random Link	Namespace
⎕RTL	Response Time Limit	Namespace
⎕SM	Screen Map	Workspace
⎕TNAME	Thread Name	Workspace
⎕TRAP	Event Trap	Workspace
⎕USING	Microsoft .NET Search Path	Namespace
⎕WSID	Workspace ID	Workspace
⎕WX	Window Expose	Namespace

In other words, *Section 6.4.117* and *Section 6.4.121* relate to the session. *Section 6.4.77*, *Section 6.4.138*, *Section 6.4.165*, and *Section 6.4.180* relate to the active workspace. All the other system variables relate to the current namespace.

Session	Workspace	Namespace
⎕PATH	⎕LX	⎕AVU
⎕PW	⎕SM	⎕CT
	⎕TRAP	⎕DCT
	⎕WSID	⎕DIV
		⎕FR
		⎕IO
		⎕ML
		⎕PP
		⎕RL
		⎕RTL
		⎕USING
		⎕WX

Note that the value assigned to a system variable must be appropriate, otherwise an error will be reported immediately.

EXAMPLE

```
      ⎕IO←3
DOMAIN ERROR
      ⎕IO←3
      ^
```

6.4 System Functions (A-Z)

6.4.1 Character Input/Output ⍞

⍞ is a variable which communicates between the user's terminal and APL. Its behaviour depends on whether it is being assigned or referenced.

When ⍞ is assigned with a vector or a scalar, the array is displayed without the normal ending new-line character. Successive assignments of vectors or scalars to ⍞ without any intervening input or output cause the arrays to be displayed on the same output line.

Example

```
      ⎕←'2+2' ⋄ ⎕←'=' ⋄ ⎕←4
2+2=4
```

Output through ⎕ is independent of the print width in ⎕PW. The way in which lines exceeding the print width of the terminal are treated is dependent on the characteristics of the terminal. Numeric output is formatted in the same manner as direct output (see *Programming: Display Of Arrays*).

When ⎕ is assigned with a higher-rank array, the output is displayed in the same manner as for direct output except that the print width ⎕PW is ignored.

When ⎕ is referenced, terminal input is expected without any specific prompt, and the response is returned as a character vector.

If the ⎕ request was preceded by one or more assignments to ⎕ without any intervening input or output, the last (or only) line of the output characters are returned as part of the response.

Example

```
      mat←↑⍬⍬⍬⍬⍬
```

Examples

```
      ⎕←'OPTION : ' ⋄ R←⎕
OPTION : INPUT

      R
OPTION : INPUT

      ⍴R
14
```

The output of simple arrays of rank greater than 1 through ⎕ includes a new-line character at the end of each line. Input through ⎕ includes the preceding output through ⎕ since the last new-line character.

A soft interrupt causes an `INPUT INTERRUPT` error if entered while ⎕ is awaiting input, and execution is then suspended (unless the interrupt is trapped):

```
      R←⎕
```

(Interrupt)

```
INPUT INTERRUPT
```

A time limit is imposed on input through ⍞ if ⎕RTL is set to a non-zero value:

```
      ⎕RTL←5 ⋄ ⍞←'PASSWORD ? ' ⋄ R←⍞
PASSWORD ?
TIMEOUT
      ⎕RTL←5 ⋄ ⍞←'PASSWORD : ' ⋄ R←⍞
                                 ^
```

The TIMEOUT interrupt is a trappable event.

6.4.2 Evaluated Input/Output ⎕

⎕ is a variable which communicates between the user's terminal and APL. Its behaviour depends on whether it is being assigned or referenced.

When ⎕ is assigned an array, the array is displayed at the terminal in exactly the same form as is direct output (see *Programming: Display Of Arrays*).

EXAMPLE

```
      ⎕←2+⍳5
3 4 5 6 7

      ⎕←2 4⍴'WINEMART'
WINE
MART
```

When ⎕ is referenced, a prompt (⎕:) is displayed at the terminal, and input is requested. The response is evaluated and an array is returned if the result is valid. If an error occurs in the evaluation, the error is reported as normal (unless trapped by a ⎕TRAP definition) and the prompt (⎕:) is again displayed for input. An EOF interrupt reports INPUT INTERRUPT and the prompt (⎕:) is again displayed for input. A soft interrupt is ignored and a hard interrupt reports INTERRUPT and the prompt (⎕:) is redisplayed for input.

Examples

```
      10×⍞+2
⍞:
      ⍳3
30 40 50

      2+⍞
⍞:
      X
VALUE ERROR
      X
     ^
⍞:
      2+⍳3
5 6 7
```

A system command may be entered. The system command is effected and the prompt is displayed again (unless the system command changes the environment):

```
      ⍴3,⍞
⍞:
      )WSID
WS/MYWORK
⍞:
      )SI
⍞
⍞:
      )CLEAR
CLEAR WS
```

If the response to a ⍞: prompt is an abort statement (→), the execution will be aborted:

```
      1 2 3 = ⍞
⍞:
      →
```

A trap definition on interrupt events set for the system variable ⎕TRAP in the range 1000-1008 has no effect whilst awaiting input in response to a ⍞: prompt.

EXAMPLE

```
      ⎕TRAP←(11 'C' '''ERROR''')(1000 'C' '''STOP''')

      2+⎕
⎕:
      (Interrupt Signal)
INTERRUPT
⎕:
      'C'+2
ERROR
```

A time limit set in system variable ⎕RTL has no effect whilst awaiting input in response to a ⎕: prompt.

6.4.3 Underscored Alphabetic Characters R←⎕A

⎕A is a deprecated feature. Dyalog **strongly** recommends that you move away from the use of ⎕A and of the underscored alphabet itself, as these symbols now constitute the sole remaining non-standard use of characters in Dyalog applications.

In Versions of Dyalog APL prior to Version 11.0, ⎕A was a simple character vector, composed of the letters of the alphabet with underscores. If the Dyalog Alt font was in use, these symbols displayed as additional National Language characters.

Version 10.1 and Earlier

```
      ⎕A
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

For compatibility with previous versions of Dyalog APL, functions that contain references to ⎕A will continue to return characters with the same *index* in ⎕AV as before. However, the display of ⎕A is now ⎕Á, and the old underscored symbols appear as they did in previous Versions when the Dyalog Alt font was in use.

Current Version

```
      ⎕Á
ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛÝþãìðòõ
```

6.4.4 Alphabetic Characters R←⎕A

This is a simple character vector, composed of the letters of the alphabet.

EXAMPLE

```
      ⎕A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

6.4.5 Account Information R←⎕AI

This is a simple integer vector, whose four elements are:

⎕AI[1]	user identification. [4]
⎕AI[2]	compute time for the APL session in milliseconds.
⎕AI[3]	connect time for the APL session in milliseconds.
⎕AI[4]	keying time for the APL session in milliseconds.

Elements beyond 4 are not defined but reserved.

EXAMPLE

```
     ⎕AI
52 7396 2924216 2814831
```

6.4.6 Account Name R←⎕AN

This is a simple character vector containing the user (login) name. Under UNIX and Linux this is the real user name, whereas ⎕AI returns the effective user id.

EXAMPLE

```
      ⎕AN
Pete

      ⍴⎕AN
4
```

[4] Under Windows, this is the aplnid (network ID from configuration dialog box). Under UNIX and Linux this is the effective UID of the account whereas ⎕AN returns the real name.

6.4.7 Arbitrary Input R←X ⎕ARBIN Y

This transmits a stream of 8-bit codes in `Y` to an output device specified by `X` prior to reading from an input device specified by `X`.

`Y` may be a scalar or a simple vector of integer numbers in the range 0-255.

`X` may take several forms:

`terminate (input output)`	`⎕ARBIN`	`codes`
`terminate input`	`⎕ARBIN`	`codes`

terminate

This is a numeric scalar or vector that specifies how the read operation should be terminated.

- If it is a numeric scalar, it defines the number of bytes to be read.
- If it is a numeric vector, it defines a set of terminating bytes.
- If it is the null vector, the read terminates on Newline (10).

input

This is a simple numeric scalar that specifies the input device.

- If it is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If it is negative, it represents the tie number of a file opened by `⎕NTIE` or `⎕NCREATE`.

output

If specified, this is a simple numeric integer that identifies the output device.

- If it is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If it is negative, it represents the tie number of a file opened by `⎕NTIE` or `⎕NCREATE`.

The result `R` is a simple numeric vector. Each item of `R` is the numeric representation of an 8-bit code in the range 0 to 255 received from the input device. The meaning of the code is dependent on the characteristics of the input device. If a set of delimiters was defined by `terminate`, the last code returned will belong to that set.

`⎕RTL` (Response Time Limit) is an implicit argument of `⎕ARBIN`. This allows a time limit to be imposed on input. If the time limit is reached, `⎕ARBIN` returns with the codes read up to that point. This does not apply under Windows.

The operation will fail with a `DOMAIN ERROR` if `Y` contains anything other than numbers in the range 0-255, or if the current process does not have permission to read from or write to the specified device(s).

Examples (UNIX)

```
      )SH mkfifo ./fifo

      in←'./fifo'⎕NTIE 0
      out←'./fifo'⎕NTIE 0

      (10 (in out))⎕ARBIN ⎕UCS ⎕D
48 49 50 51 52 53 54 55 56 57

      (0 (in out))⎕ARBIN 10
10

⍝ cope with parity on line ending 10
      ((10+0 128) (in out))⎕ARBIN 10
10
```

6.4.8 Arbitrary Output {R}←X ⎕ARBOUT Y

`Y` may be a scalar or a simple vector of integer numbers in the range 0-255.

`X` is a simple numeric integer that specifies the output device.

- (Non-Windows platforms only) If `X` is positive or zero, it represents a file descriptor that must have been associated by the command that started Dyalog APL.
- If `X` is negative, it represents the tie number of a file opened by `⎕NTIE` or `⎕NCREATE`.

If `Y` is an empty vector, no codes are sent to the output device.

The shy result R is 0.

The operation will fail with a DOMAIN ERROR if Y contains anything other than numbers in the range 0-255 or tie numbers associated with currently open native files, or if the current process does not have permission to write to the specified device.

EXAMPLES

Write ASCII digits '123' to stream 9:

```
      9 ⎕ARBOUT 49 50 51
```

Write ASCII characters 'ABC' to MYFILE:

```
     'MYFILE' ⎕NCREATE ¯1
      ¯1 ⎕ARBOUT 65 66 67
```

Append the string 'Κάλο Πάσχα' to the same file, and close it:

```
      ¯1 ⎕ARBOUT 'UTF-8' ⎕UCS'Κάλο Πάσχα'
     ⎕NUNTIE ¯1
```

6.4.9 Attributes R←{X} ⎕AT Y

Y can be a simple character scalar, vector or matrix, or a vector of character vectors representing the names of 0 or more defined functions or operators. Used dyadically, this function closely emulates the APL2 implementation. Used monadically, it returns information that is more appropriate for Dyalog APL.

Y specifies one or more names. If Y specifies a single name as a character scalar, a character vector, or as a scalar enclosed character vector, the result R is a vector. If Y specifies one or more names as a character matrix or as a vector of character vectors R is a matrix with one row per name in Y.

Monadic Use

If X is omitted, R is a 4-element vector or a 4 column matrix with the same number of rows as names in Y containing the following attribute information:

R[1] or R[;1]: Each item is a 3-element integer vector representing the function header syntax:

1	Function result	0 if the function has no result 1 if the function has an explicit result ¯1 if the function has a shy result
2	Function valence	0 if the object is a niladic function or not a function 1 if the object is a monadic function 2 if the object is a dyadic function ¯2 if the object is an ambivalent function
3	Operator valence	0 if the object is not an operator 1 if the object is a monadic operator 2 if the object is a dyadic operator

The following values correspond to the syntax shown alongside:

```
 0  0  0     ∇ FOO
 1  0  0     ∇ Z←FOO
¯1  0  0     ∇ {Z}←FOO
 0 ¯2  0     ∇ {A} FOO B
¯1  1  2     ∇ {Z}←(F OP G)B
```

`R[2]` or `R[;2]`: Each item is the (`⎕TS` form) timestamp of the time the function was last fixed.

`R[3]` or `R[;3]`: Each item is an integer reporting the current `⎕LOCK` state of the function:

0	Not locked
1	Cannot display function
2	Cannot suspend function
3	Cannot display or suspend

`R[4]` or `R[;4]`: Each item is a character vector - the network ID of the user who last fixed (edited) the function.

Example

```
    ∇ {z}←{l}(fn myop)r
[1]   ...

    ∇ z←foo
[1]   ...

    ∇ z←{larg}util rarg
[1]   ...

      ⎕LOCK'foo'

      util2←util
```

```
      ]display ⎕AT 'myop' 'foo' 'util' 'util2'
.→-------------------------------------------.
↓ .→------. .→-----------------.     .→---.   |
| |¯1 ¯2 1| |1996 8 2 2 13 56 0|   0 |john|   |
| '~------' '~-----------------'     '----'   |
| .→----.   .→------------.          .⊖.      |
| |1 0 0|   |0 0 0 0 0 0 0|        3 | |      |
| '~----'   '~------------'          '-'      |
| .→-----.  .→-----------------.     .→---.   |
| |1 ¯2 0|  |1996 3 1 14 12 10 0|  0 |pete|   |
| '~-----'  '~-----------------'     '----'   |
| .→-----.  .→-------------------.   .→-----. |
| |1 ¯2 0|  |1998 8 26 16 16 42 0| 0 |graeme| |
| '~-----'  '~-------------------'   '------' |
'⊂-------------------------------------------'
```

Dyadic Use

The dyadic form of ⎕AT emulates APL2. It returns the same rank and shape result containing information that matches the APL2 implementation as closely as possible.

The number of elements or columns in R and their meaning depends upon the value of X which may be 1, 2, 3 or 4.

If X is 1, R specifies *valences* and contains 3 elements (or columns) whose meaning is as follows:

1	Explicit result	1 if the object has an explicit result or is a variable 0 otherwise
2	Function valence	0 if the object is a niladic function or not a function 1 if the object is a monadic function 2 if the object is an ambivalent function
3	Operator valence	0 if the object is not an operator 1 if the object is a monadic operator 2 if the object is a dyadic operator

If `X` is 2, `R` specifies *fix times* (the time the object was last updated) for functions and operators named in `Y`. The time is reported as 7 integer elements (or columns) whose meaning is as follows. The fix time reported for names in `Y` which are not defined functions or operators is 0.

1	Year
2	Month
3	Day
4	Hour
5	Minute
6	Second
7	Milliseconds (this is always reported as 0)

If `X` is 3, `R` specifies *execution properties* and contains 4 elements (or columns) whose meaning is as follows:

1	Displayable	0 if the object is displayable 1 if the object is not displayable
2	Suspendable	0 if execution will suspend in the object 1 if execution will not suspend in the object
3	Weak Interrupt behaviour	0 if the object responds to interrupt 1 if the object ignores interrupt
4		(always 0)

If `X` is 4, `R` specifies *object size* and contains 2 elements (or columns) that both report the `⎕SIZE` of the object.

6.4.10 Extended Attributes `R←X ⎕ATX Y`

This function provides information about a name in a workspace, including its usage, history, restrictions, role and origin.

> **Note**
>
> To retrieve this information for an unnamed value, wrap `⎕ATX` in a dfn and use the name `⍵`, for example `{60⎕ATX'⍵'}`

`Y` can be a simple character scalar, a simple or enclosed character vector, or a vector of character scalars and vectors (as least one must be a character vector) of the name(s) for which information is required.

`X` can be a scalar or a vector indicating the information required:

Group	⍺	Meaning	Default
Identity	0	Name	''
Syntax	10	Function result (0 : none or not a function, 1 : explicit, ¯1 : shy)	0
	11	Function valence (0 : niladic, 1 : monadic, 2 : dyadic, ¯2 : ambivalent)	0
	12	Operator valence: (0 : not an operator, 1 : monadic, 2 : dyadic)	0
Last edit	20	Author of last edit	''
	21	Number of days passed between 1899-12-31 at 00:00 UTC and last edit (includes fractional days)	0
	22	Local timestamp at last edit (format is the 7-item vector described by ⎕TS . See *Timestamp*)	⍬
	23	Number of bytes required for storage without sharing	0
Restrictions	30	Source can be displayed	¯1
	31	Execution can be suspended mid-execution	¯1
	32	Responds to weak interrupt	¯1
Class[5]	40	Syntactic supra-class (¯1 : invalid name, 0 : undefined, 1 : label, 2 : variable, 3 : function, 4 : operator, 8 : event, 9 : object)	¯1
	41	Syntactic sub-class (0 : none, 1 : traditional/plain, 2 : field/dynamic/instance, 3 : property/derived/primitive, 4 : class, 5 : interface, 6 : external, 7 : external interface)	0
	42	Full syntactic class (sum of supra- and sub-class)	¯1
Source	50	File name	''
	51	File encoding	''
	52	File checksum	''
	53	File line separators (13 : Carriage Return, 10 : Line Feed, 13 10 : Carriage Return followed by Line Feed,	⍬

Group	X	Meaning	Default
		133 : New Line, 11 : Vertical Tab, 12 : Form Feed, 8232 : Line Separator, 8233 : Paragraph Separator)	
	54	Definition's offset from top	0
	55	Number of lines in definition	0
Definition	60	Verbatim source (as typed)	0⍴⊂''
	61	Normalised source (with AUTOFORMAT=1 and TABSTOPS=4)	0⍴⊂''
	62	Most precise available source (verbatim with fallback to normalised)	0⍴⊂''

R depends on the combination of X and Y:

		X	
		Scalar	Vector
Y	Simple character scalar/vector	Requested value (not enclosed)	Vector of requested values
	Enclosed character vector	Requested value (enclosed)	Scalar containing vector of requested values
	Vector of character scalars/vectors	Vector of requested values	Outer shape from ⍴⍺ , inner shape from ⍴⍵

Examples

```
      Att
10 11 12 20 23 30 31 32 40 41 42 50 51 52 53 54 55
```

```
      foo←{⍵ ⍵}
      Att ⎕ATX 'foo'
┌─┬──┬─┬──┬────┬──┬──┬──┬─┬─┬───┬─────┬─┬─┐
│1│¯2│0│0 1 6 1 6 │¯1│¯1│¯1│3│2│3.2│     │0│0│
└─┴──┴─┴──┴────┴──┴──┴──┴─┴─┴───┴─────┴─┴─┘
```

```
      x←42
      Att ⎕ATX 'x'
┌─┬─┬─┬──┬──┬──┬──┬─┬───┬─┬─┐
│0│0│0│32│¯1│¯1│¯1│2│1│2.1│0│0│
└─┴─┴─┴──┴──┴──┴──┴─┴───┴─┴─┘
```

```
      10 11 12 30 31 32 40 41 42 ⎕ATX 'x' 'foo'
┌─┬─┬─┬──┬─┬─┬──┬──┬──┬──┬─┬─┬─┬─┬───┬───┐
│0│1│0│¯2│0│0│¯1│¯1│¯1│¯1│¯1│¯1│2│3│1 2│2.1 3.2│
└─┴─┴─┴──┴─┴─┴──┴──┴──┴──┴──┴──┴─┴─┴───┴───┘
```

```
      2 ⎕FIX'foo ← {' 'ω ω }'
      60 61 ⎕ATX 'foo'
┌──────────┬──────────┐
│┌────────┐│┌────────┐│
│├─┬──┬───┤│├─┬──┬───┤│
│└─┴──┴───┘│└─┴──┴───┘│
└──────────┴──────────┘
```

```
      src←':namespace c' ':endnamespace' '' 'range←{α↓⍳ω}'
      2 ⎕FIX src
      55 54 ⎕ATX'c' 'range'
┌─┬─┐
│2 1│0 3│
└─┴─┘
```

```
      2 1↑¨0 3↓¨⊂src
┌──────────┬──────────┐
│┌────────┐│┌────────┐│
│├─┬──┬───┤│├─┬──┬───┤│
│└─┴──┴───┘│└─┴──┴───┘│
└──────────┴──────────┘
```

6.4.11 Atomic Vector R←⎕AV

⎕AV is a deprecated feature and is replaced by ⎕UCS.

This is a simple character vector of all 256 characters in the Classic Dyalog APL character.

5 Names in the Class group that can return ¯1 (meaning "invalid name") might return a different value in future versions of Dyalog, including values that are not currently possible and ones that deviate from the current ⎕NC values.

In the Classic Edition the contents of ⎕AV are defined by the Output Translate Table.

In the Unicode Edition, the contents of ⎕AV are defined by the system variable ⎕AVU.

EXAMPLES

```
      ⎕AV[48+⍳10]
0123456789

      5 52⍴12↓⎕AV
%'αω_abcdefghijklmnopqrstuvwxyz¯.θ0123456789⊢¥$£¢
ΔABCDEFGHIJKLMNOPQRSTUVWXYZý·⌂⍙ÁÂÃÇÈÊËÌÍÎÏÐÒÓÔÕÙÚÛ
Ýþãìðòõ{€}¬[]¨ÀÄÅÆ˜ÉÑÖØÜßàáâäåæçèéêëíîïñ[/≠\⍀<≤=≥>≠∨∧
-+÷×?∈⍴~↑↓⍳○*⌈⌊∇⍥(⊂⊃∩∪⊥⊤|;,▽⍫⍒⍋⌽⍉⍟⍱⍲⍕⍎⍷!⍤±⍣¨⍝≡≠⍬⍵⍺"#&´
┘┐┌┴┼┤├┬└|@ùúû^ü`|¶:⍷¿¡⋄←→⍪)] §⎕⍞⍁%'αω_abcdefghijk
```

6.4.12 Atomic Vector - Unicode ⎕AVU

⎕AVU specifies the contents of the atomic vector, ⎕AV, and is used to translate data between Unicode and non-Unicode character formats when required, for example when:

- Unicode Edition loads or copies a Classic Edition workspace or a workspace saved by a Version prior to Version 12.0.
- Unicode Edition reads character data from a non-Unicode component file, or receives data type 82 from a TCP socket.
- Unicode Edition writes data to a non-Unicode component file
- Unicode Edition reads or writes data from or to a Native File using conversion code 82.
- Classic Edition loads or copies a Unicode Edition workspace
- Classic Edition reads character data from a Unicode component file, or receives data type 80, 160, or 320 from a TCP socket.
- Classic Edition writes data to a Unicode component file.

⎕AVU is an integer vector with 256 elements, containing the Unicode code points which define the characters in ⎕AV. ⎕AVU has Namespace scope.

Note

In Versions of Dyalog prior to Version 12.0 and in the Classic Edition, a character is stored internally as an index into the atomic vector, ⎕AV. When a character is displayed or printed, the index in ⎕AV is translated to a number in the range 0-255 which

represents the index of the character in an Extended ASCII font. This mapping is done by the Output Translate Table which is user-configurable. Note that although ASCII fonts typically all contain the same symbols in the range 0-127, there are a number of different Extended ASCII font layouts, including proprietary APL fonts, which provide different symbols in positions 128-255. The actual symbol that appears on the screen or on the printed page is therefore a function of the Output Translate Table and the font in use. Classic Edition provides two different fonts (and thus two different ⎕AV layouts) for use with the Development Environment, named *Dyalog Std* (with APL underscores) and *Dyalog Alt* (without APL underscores).

The default value of ⎕AVU corresponds to the use of the **Dyalog Alt** Output Translate Table and font in the Classic Edition or in earlier versions of Dyalog APL.

```
      2 13⍴⎕AVU[97+⍳26]
193 194 195 199 200 202 203 204 205 206 207 208 210
211 212 213 217 218 219 221 254 227 236 240 242 245
      ⎕UCS 2 13⍴⎕AVU[97+⍳26]
ÁÂÃÇÈÊËÌÍÎÏÐÒ
ÓÔÕÙÚÛÝþãìðòõ
```

⎕AVU can be localised, in order to make it straightforward to write access functions which receive or read data from systems with varying atomic vectors. If you have been using Dyalog Alt for most things but have some older code which uses underscores, you can bring this code together in the same workspace and have it all look "as it should" by using the Alt and Std definitions for ⎕AVU as you copy each part of the code into the same Unicode Edition workspace.

```
      )COPY avu.dws Std.⎕AVU
C:\Program Files\Dyalog\Dyalog APL 12.0 Unicode\ws\avu saved Thu Dec 06
11:24:32 2007

      2 13⍴⎕AVU[97+⍳26]
9398 9399 9400 9401 9402 9403 9404 9405 9406 9407 9408 9409 9410
9411 9412 9413 9414 9415 9416 9417 9418 9419 9420 9421 9422 9423
       ⎕UCS 2 13⍴⎕AVU[97+⍳26]
ABCDEFGHIJKLM
NOPQRSTUVWXYZ
```

> **Hints and Recommendations**
>
> The majority of Unicode fonts display APL's underscored alphabet as a circled alphabet instead. Unicode does not contain the underscored letters as distinct glyphs; Dyalog APL maps these characters to the Unicode positions for the circled alphabet, and provides APL fonts (such as APL385 Unicode) that display them as underscores. To avoid confusion, Dyalog Ltd recommends that you remove underscored characters from your APL application at your earliest convenience.

Rules for Conversion on Import

When the Unicode Edition imports APL objects from a non-Unicode source, function comments and character data of type 82 are converted to Unicode. When the Classic Edition imports APL objects from a Unicode source, this translation is performed in reverse.

If the objects are imported from a Version 12.0 (or later) workspace (that is, from a workspace that contains its own value of ⎕AVU) the value of `#.⎕AVU` (the value of ⎕AVU in the root) in the *source* workspace is used. Otherwise, such as when APL objects are imported from a pre-Version 12 workspace, from a component file, or from a TCP socket, the local value of ⎕AVU in the *target* workspace is used.

Rules for Conversion on Export

When the Unicode Edition exports APL objects to a non-Unicode destination, such as a non-Unicode Component File or non-Unicode TCPSocket Object, function comments (in ⎕ORS) and character data of type 82 are converted to ⎕AV indices using the local value of ⎕AVU.

When the Classic Edition exports APL objects to a Unicode destination, such as a Unicode Component File or Unicode TCPSocket Object, function comments (in ⎕ORS) and character data of type 82 are converted to Unicode using the local value of ⎕AVU.

In all cases, if a character to be translated is not defined in ⎕AVU, a `TRANSLATION ERROR` (event number 92) will be signalled.

6.4.13 Base Class R←⎕BASE.Y

⎕BASE is used to access the base class implementation of the name specified by Y.

Y must be the name of a Public member (Method, Field or Property) that is provided by the Base Class of the current Class or Instance.

⎕BASE is typically used to call a method in the Base Class which has been *superseded* by a Method in the current Class.

Note that ⎕BASE.Y is *special syntax* and any direct reference to ⎕BASE on its own or in any other context, is meaningless and causes SYNTAX ERROR.

In the following example, Class DomesticParrot derives from Class Parrot and supersedes its Speak method. DomesticParrot.Speak calls the Speak method in its Base Class Parrot, via ⎕BASE.

```
:Class Parrot: Bird
    ∇ R←Speak
      :Access Public
      R←'Squark!'
    ∇
:EndClass ⍝ Parrot

:Class DomesticParrot: Parrot
    ∇ R←Speak
      :Access Public
      R←⎕BASE.Speak,' Who''s a pretty boy, then!'
    ∇
:EndClass ⍝ DomesticParrot

      Maccaw←⎕NEW Parrot
      Maccaw.Speak
Squark!

      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy, then!
```

6.4.14 Case Convert R←{X}⎕C Y

Y is any array. R is an identical array except that character arrays within it are either folded for case-less comparison, or mapped to upper or lower case.

For a discussion of case folding and case conversion (mapping), see Character Properties, Case Mappings & Names FAQ.

If the optional left-argument `X` is omitted, `R` is a copy of `Y` with character arrays folded, for case-less comparison.

If `X` is specified, the following cases are supported:

X	Description
`1`	`R` is a copy of `Y` with character arrays mapped to upper case.
`¯1`	`R` is a copy of `Y` with character arrays mapped to lower case.
`¯3`	`R` is a copy of `Y` with character arrays folded, for case-less comparison (this is equivalent to monadic use).

Examples

```
      ⎕C 42 'Pete' 'Πέτρος'
42  pete  πέτροσ
      1 ⎕C 42 'Pete' 'Πέτρος'
42  PETE  ΠΈΤΡΟΣ
      ¯1 ⎕C 42 'Pete' 'Πέτρος'
42  pete  πέτρος

      (⊂'pete'){α≡⎕C ω}¨'PETE' 'Pete' 'pEte'
1 1 1
```

Example

Greek has two forms of lower-case Sigma, namely "σ" and "ς" but a single upper-case Sigma "Σ". Each lower-case form remains unchanged when mapped to lower-case, but both fold to "σ", while "Σ" is mapped to lower-case "σ" .

```
      ⎕C 'ίσως'
ίσωσ
      1 ⎕C 'ίσως'
ΊΣΩΣ
      ¯1⎕C 1 ⎕C 'ίσως'
ίσωσ
```

> **Note**
>
> Refs in `Y` are not followed but just returned unchanged.

6.4.15 Class R←{X}⎕CLASS Y

Monadic Case

Monadic ⎕CLASS returns a list of references to Classes and Interfaces that specifies the class hierarchy for the Class or Instance specified by Y.

Y must be a reference to a Class or to an Instance of a Class.

R is a vector of vectors whose items represent nodes in the Class hierarchy of Y. Each item of R is a vector whose first item is a Class reference and whose subsequent items (if any) are references to the Interfaces supported by that Class.

Example 1

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
    Bird (derived from Animal)
        Parrot (derived from Bird)

:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot
```

```
      ⎕CLASS Eeyore←⎕NEW Animal
#.Animal
      ⎕CLASS Robin←⎕NEW Bird
#.Bird    #.Animal
      ⎕CLASS Polly←⎕NEW Parrot
#.Parrot    #.Bird    #.Animal

     ⎕CLASS⍨ Parrot Animal
 #.Parrot    #.Bird    #.Animal      #.Animal
```

Example 2

The Penguin Class example (see*Programming: Interface Example*) illustrates the use of Interfaces.

In this case, the `Penguin` Class derives from `Animal` (as above) but additionally supports the `BirdBehaviour` and `FishBehaviour` Interfaces, thereby inheriting members from both.

```
     Pingo←⎕NEW Penguin
     ⎕CLASS Pingo
#.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal
```

Dyadic Case

If `X` is specified, `Y` must be a reference to an Instance of a Class and `X` is a reference to an Interface that is supported by Instance `Y` or to a Class upon which Instance `Y` is based.

In this case, `R` is a reference to the implementation of Interface `X` by Instance `Y`, or to the implementation of (Base) Class `X` by Instance `Y`, and is used as a *cast* in order to access members of `Y` that correspond to members of Interface of (Base) Class `X`.

Example 1

Once again, the Penguin Class example (see*Programming: Interface Example*) is used to illustrate the use of Interfaces.

```
      Pingo←⎕NEW Penguin
      ⎕CLASS Pingo
  #.Penguin  #.FishBehaviour  #.BirdBehaviour    #.Animal

      (FishBehaviour ⎕CLASS Pingo).Swim
I can dive and swim like a fish
      (BirdBehaviour ⎕CLASS Pingo).Fly
Although I am a bird, I cannot fly
      (BirdBehaviour ⎕CLASS Pingo).Lay
I lay one egg every year
      (BirdBehaviour ⎕CLASS Pingo).Sing
Croak, Croak!
```

Example 2

This example illustrates the use of dyadic ⎕CLASS to cast an Instance to a lower Class and thereby access a member in the lower Class that has been superseded by another Class higher in the tree.

```
      Polly←⎕NEW DomesticParrot
      Polly.Speak
Squark! Who's a pretty boy, then!
```

Note that the `Speak` method invoked above is the `Speak` method defined by Class `DomesticParrot`, which supersedes the `Speak` methods of sub-classes `Parrot` and `Bird`.

You may use a cast to access the (superseded) `Speak` method in the sub-classes `Parrot` and `Bird`.

```
      (Parrot ⎕CLASS Polly).Speak
Squark!
      (Bird ⎕CLASS Polly).Speak
Tweet, tweet!
```

6.4.16 Clear Workspace ⎕CLEAR

A clear workspace is activated, having the name `CLEAR WS`. The active workspace is lost. All system variables assume their default values. The maximum size of workspace is available.

Apart from .NET objects, the contents of the session namespace ⎕SE are not affected. .NET objects in ⎕SE are disconnected from .NET because ⎕CLEAR closes the current .NET AppDomain.

EXAMPLE

```
      ⎕CLEAR
      ⎕WSID
CLEAR WS
```

6.4.17 Execute Windows Command {R}←⎕CMD Y

⎕CMD executes the Windows Command Processor or UNIX shell or starts another Windows application program. ⎕CMD is a synonym of ⎕SH. Either system function may be used in either environment (Windows or UNIX) with exactly the same effect. ⎕CMD is probably more natural for the Windows user. This section describes the behaviour of ⎕CMD and ⎕SH under Windows. See *Section 6.4.131* for a discussion of the behaviour of these system functions under UNIX.

The system commands *Section 7.2.25* and *Section 7.2.3* provide similar facilities. For further information, see *Section 7.2.25* and *Section 7.2.3*.

See also *Section 6.4.134*.

Executing the Windows Command Processor

If Y is a simple character vector, ⎕CMD invokes the Windows Command Processor (normally `cmd.exe`) and passes the command specified by character vector Y to it for execution. The term command means here an instruction recognised by the Command Processor, or the pathname of a program (with optional parameters) to be executed by it. In either case, APL waits for the command to finish and then returns the result R, a vector of character vectors containing its result. Each element in R corresponds to a line of output produced by the command.

Example

```
      Z←⎕CMD'dir'
      ⍴Z
12
      ↑Z
 Volume in drive C is OS
 Volume Serial Number is B438-9B76

 Directory of C:\Users\Pete\Documents\Dyalog APL-64 17.0 Unicode Files

23/06/2018  15:59    <DIR>          .
23/06/2018  15:59    <DIR>          ..
23/06/2018  14:53           181,488 default.dlf
13/06/2018  20:13         1,262,296 def_uk.dse
14/06/2018  14:36           108,976 UserCommand20.cache
               3 File(s)      1,552,760 bytes
               2 Dir(s)  101,371,097,088 bytes free
```

If the command specified in `Y` already contains the redirection symbol (>) the capture of output through a pipe is avoided and the result `R` is empty. If the command specified by `Y` issues prompts and expects user input, it is **ESSENTIAL** to explicitly redirect input and output to the console.

If this is done, APL detects the presence of a ">" in the command line, runs the command processor in a **visible** window, and does not direct output to the pipe. If you fail to do this your system will appear to hang because there is no mechanism for you to receive or respond to the prompt.

Example

```
      ⎕CMD 'DATE <CON >CON'
```

(Command Prompt window appears)

```
Current date is Wed 19-07-1995

Enter new date (dd-mm-yy): 20-07-95
```

(COMMAND PROMPT window disappears)

Spaces in pathnames

If `Y` specifies a program (with or without parameters) and the pathname to the program contains spaces, you must enclose the string in double-quotes.

For example, to start a version of Excel to which the pathname is:

```
C:\Program Files\Microsoft Office\OFFICE11\excel.exe
```

the argument to `⎕CMD` should be:

```
⎕CMD '"c:\program files\microsoft office\office11\excel.exe"'
```

Double-Quote Restriction

The Windows Command Processor does not permit more than one set of double-quotes in a command string.

The following statements are all valid:

```
⎕CMD 'c:\windows\system32\notepad.exe c:\myfile.txt'
⎕CMD 'c:\windows\system32\notepad.exe "c:\myfile.txt"'
⎕CMD '"c:\windows\system32\notepad.exe" c:\myfile.txt'
```

Whereas the next statement, which contains two sets of double-quotes, will fail:

```
⎕CMD '"c:\windows\system32\notepad.exe" "c:\myfile.txt"'
```

Such a statement can however be executed using the second form of `⎕CMD`(where the argument is a 2-element vector of character vectors) which does not use the Windows Command Processor and is not subject to this restriction. However, the call to `⎕CMD` will return immediately, and no output from the command will be returned.

```
⎕CMD'"c:\windows\system32\notepad.exe" "c:\myfile.txt"' ''
```

Implementation Notes

The right argument of `⎕CMD` is simply passed to the appropriate command processor for execution and its output is received using an *unnamed pipe*.

By default, ⎕CMD will execute the string `('cmd.exe /c',Y)`; where `Y` is the argument given to ⎕CMD. However, the implementation permits the use of alternative command processors as follows:

Before execution, the argument is prefixed and postfixed with strings defined by the APL parameters CMD_PREFIX and CMD_POSTFIX. The former specifies the name of your command processor and any parameters that it requires. The latter specifies a string which may be required. If CMD_PREFIX is not defined, it defaults to the name defined by the environment variable COMSPEC followed by "/c". If COMSPEC is not defined, it defaults to `cmd.exe`. If CMD_POSTFIX is not defined, it defaults to an empty vector.

⎕CMD treats certain characters as having special meaning as follows:

#	marks the start of a trailing comment,
;	divides the command into sub-commands,
>	if found within the last sub-command, causes ⎕CMD to use a visible window.

If you simply wish to open a Command Prompt window, you may execute the command as a Windows Program (see below). For example:

```
      ⎕CMD 'cmd.exe' ''
```

Starting a Windows Program

If `Y` is a 2-element vector of character vectors, ⎕CMD starts the executable program named by `Y[1]` with the initial window parameter specified by `Y[2]`. The shy result is an integer scalar containing the window handle allocated by the window manager. Note that in this case APL does not wait for the program specified by `Y` to finish, but returns immediately. The shy result `R` is the process identifier (PID).

`Y[1]` must specify the name or complete pathname of an executable program. If the name alone is specified, Windows will search the following directories:

1. the current directory,
2. the Windows directory,
3. the Windows system directory,
4. the directories specified by the PATH variable,
5. the list of directories mapped in a network.

Note that `Y[1]` may contain the complete command line, including any suitable parameters for starting the program. If Windows fails to find the executable program, `⎕CMD` will fail and report `FILE ERROR 2`.

`Y[2]` specifies the window parameter and may be one of the following. If not, a `DOMAIN ERROR` is reported.

`'Normal'''`	Application is started in a normal window, which is given the input focus
`'Unfocused'`	Application is started in a normal window, which is NOT given the input focus
`'Hidden'`	Application is run in an invisible window
`'Minimized''Minimised'`	Application is started as an icon which is NOT given the input focus
`'Maximized''Maximised'`	Application is started maximized (full screen) and is given the input focus

There is no way to terminate an application started by `⎕CMD` from APL; it will run until it completes or is terminated by an external mechanism. Furthermore, if the window parameter is HIDDEN, the user is unaware of the application (unless it makes itself visible) and has no means to close it.

Examples

```
      Path←'c:\Program Files\Microsoft Office\Office\'
      ⎕←⎕CMD (Path,'excel.exe') ''
33
      ⎕CMD (Path,'winword /mMyMacro') 'Minimized'
```

Executing Programs

Either form of `⎕CMD` may be used to execute a program. The difference is that when the program is executed via the Command Processor, APL waits for it to complete and returns any result that the program would have displayed in the Command Window had it been executed from a Command Window. In the second case, APL starts the program (in parallel).

Note

This function is disabled and instead generates a `DOMAIN ERROR` if the RIDE_SPAWNED parameter is non-zero. This is designed to prevent it being invoked from a Ride session which does not support this type of user interface. For further details, see the Ride User Guide.

6.4.18 Start Windows Auxiliary Processor `{R}←X ⎕CMD Y`

Used dyadically, `⎕CMD` starts an Auxiliary Processor. The effect, as far as the APL workspace is concerned, is identical under both Windows and UNIX, although the method of implementation differs. `⎕CMD` is a synonym of `⎕SH`. Either function may be used in either environment (Windows or UNIX) with exactly the same effect. `⎕CMD` is probably more natural for the Windows user. This section describes the behaviour of `⎕CMD` and `⎕SH` under Windows. See *Section 6.4.132* for a discussion of the behaviour of these system functions under UNIX.

`X` must be a simple character vector containing the name (or pathname) of a Dyalog APL Auxiliary Processor (AP). Although it is possible for users to create their own APs, Dyalog recommends that user write their own DLLs/shared libraries instead.

`Y` may be a simple character scalar or vector, or a vector of character vectors. Under Windows the contents of `Y` are ignored.

`⎕CMD` loads the Auxiliary Processor into memory. If no other APs are currently running, `⎕CMD` also allocates an area of memory for communication between APL and its APs.

The shy result `R` is the process id of the Auxiliary Processor task.

The effect of starting an AP is that one or more **external functions** are defined in the workspace. These appear as locked functions and may be used in exactly the same way as regular defined functions.

When an external function is used in an expression, the argument(s) (if any) are passed to the AP for processing via the communications area described above. APL halts whilst the AP is processing, and waits for a result. Under Windows, unlike under UNIX, it is not possible for external functions to run in parallel with APL.

Although it is still possible for users to create their own APs, Dyalog strongly recommends creating shared libraries/DLLs instead.

6.4.19 Canonical Representation R←⎕CR Y

Y must be a simple character scalar or vector which represents the name of a defined function or operator.

If Y is a name of a defined function or operator, R is a simple character matrix. The first row of R is the function or operator header. Subsequent rows are lines of the function or operator. R contains no unnecessary blanks, except for leading indentation of control structures, trailing blanks that pad each row, and the blanks in comments. If Y is the name of a variable, a locked function or operator, an external function, or is undefined, R is an empty matrix whose shape is `0 0`.

Example

```
      ∇R←MEAN X    ⍝ Arithmetic mean
[1]   R←(+/X)÷⍴X
[2]   ∇
      +F←⎕CR'MEAN'
R←MEAN X    ⍝ Arithmetic mean
R←(+/X)÷⍴X

      ⍴F
2 30
```

The definition of ⎕CR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators.

If Y is a name assigned to a primitive function, R is a one-element vector containing the corresponding function symbol. If Y is a name assigned to a system function, R is a one element nested array containing the name of the system function.

Examples

```
      PLUS←+
      +F←⎕CR'PLUS'
+
      ⍴F
1
      C←⎕CR
      C'C'
 ⎕CR
      ⍴C'C'
1
```

```
      ∇R←CONDITION (FN1 ELSE FN2) X
[1]   →CONDITION/L1
[2]   R←FN2 X ⋄ →0
[3]  L1:R←FN1 X
[4]   ∇

      2 ⎕STOP 'ELSE'
      (X≥0) ⌊ ELSE ⌈ X←¯2.5

ELSE[2]
       X
¯2.5
       ⎕CR'FN2'
⌈
       →⎕LC
¯2
```

If Y is a name assigned to a derived function, R is a vector whose elements represent the arrays, functions, and operators from which Y was constructed. Constituent functions are represented by their own ⎕CRs, so in this respect the definition of ⎕CR is recursive. Primitive operators are treated like primitive functions, and are represented by their corresponding symbols. Arrays are represented by themselves.

Example

```
      BOX←2 2∘⍴
      +F←⎕CR'BOX'
 2 2 ∘⍴
      ⍴F
3
      ]display F
.→----------.
| .→--.     |
| |2 2| ∘ ⍴ |
| '~--' - - |
'∈----------'
```

If Y is a name assigned to a defined function, R is the ⎕CR of the defined function. In particular, the name that appears in the function header is the name of the original defined function, not the assigned name Y.

EXAMPLE

```
      AVERAGE←MEAN
      ⎕CR'AVERAGE'
R←MEAN X    ⍝ Arithmetic mean
R←(+/X)÷⍴X
```

6.4.20 Change Space {R}←{X}⎕CS Y

Y must be namespace reference (ref) or a simple character scalar or vector identifying the name of a namespace.

If specified, X is a simple character scalar, vector, matrix or a nested vector of character vectors identifying zero or more workspace objects to be *exported* into the namespace Y.

The identifiers in X and Y may be simple names or compound names separated by '.' and including the names of the special namespaces '⎕SE', '#', and '##'.

The result R is the full name (starting #.) of the space in which the function or operator was executing prior to the ⎕CS.

⎕CS changes the space in which the current function or operator is running to the namespace Y and returns the original space, in which the function was previously running, as a shy result. **After the** ⎕CS, references to *global* names (with the exception of those specified in X) are taken to be references to *global* names in Y. References to *local* names (that is, those local to the current function or operator) are, with the exception of those with name class 9, unaffected. Therefore any names in the new space that match local names in the containing defined function will be invisible. Local names with name class 9 are also not visible.

When the function or operator terminates, the calling function resumes execution in its original space.

The names listed in X are temporarily *exported* to the namespace Y. If objects with the same name exist in Y, these objects are effectively *shadowed* and are inaccessible. Note that Dyadic ⎕CS may be used only if there is a traditional function in the state indicator (stack). Otherwise there would be no way to retract the export. In this case (for example in a clear workspace) DOMAIN ERROR is reported.

Note that calling ⎕CS with an empty argument Y obtains the namespace in which a function is currently executing.

Example

This simple example illustrates how ⎕CS may be used to avoid typing long pathnames when building a tree of GUI objects. Note that the objects NEW and OPEN are created as children of the FILE menu as a result of using ⎕CS to change into the F.MB.FILE namespace.

```
     ∇ MAKE_FORM;F;OLD
[1]    'F'⎕WC'Form'
[2]    'F.MB'⎕WC'MenuBar'
[3]    'F.MB.FILE'⎕WC'Menu' '&File'
[4]
[5]    OLD←⎕CS'F.MB.FILE'
[6]    'NEW'⎕WC'MenuItem' '&New'
[7]    'OPEN'⎕WC'MenuItem' '&Open'
[8]    ⎕CS OLD
[9]
[10]   'F.MB.EDIT'⎕WC'Menu' '&Edit'
[11]
[12]   OLD←⎕CS'F.MB.EDIT'
[13]   'UNDO'⎕WC'MenuItem' '&Undo'
[14]   'REDO'⎕WC'MenuItem' '&Redo'
[15]   ⎕CS OLD
[16]   ...
     ∇
```

Example

Suppose a form F1 contains buttons B1 and B2. Each button maintains a count of the number of times it has been pressed, and the form maintains a count of the total number of button presses. The single callback function PRESS and its subfunction FMT can reside in the form itself

```
      )CS F1
#.F1
      ⍝ Note that both instances reference
      ⍝ the same callback function
      'B1'⎕WS'Event' 'Select' 'PRESS'
      'B2'⎕WS'Event' 'Select' 'PRESS'

      ⍝ Initialise total and instance counts.
      TOTAL ← B1.COUNT ← B2.COUNT ← 0
```

```
    ∇ PRESS MSG
[1]   'FMT' 'TOTAL'⎕CS⊃MSG ⍝     Switch to instance space
[2]   (TOTAL COUNT)+←1    ⍝    Incr total & instance count
[3]   ⎕WS'Caption'(COUNT FMT TOTAL)⍝ Set instance caption
    ∇
```

```
    ∇ CAPT←INST FMT TOTL      ⍝ Format button caption.
[1]   CAPT←(⍕INST),'/',⍕TOTL  ⍝ For example, 40/100.
    ∇
```

Example

This example uses ⎕CS to explore a namespace tree and display the structure. Note that it must export its own name (tree) each time it changes space, because the name tree is global.

```
      ∇ tabs tree space;subs     ⍝ Display namespace tree
[1]    tabs,space
[2]    'tree'⎕CS space
[3]    →(⍴subs←↓⎕NL 9)↓0
[4]    (tabs,'.   ')∘tree¨subs
     ∇

      )NS x.y
#.x.y
      )NS z
#.z
      ''tree '#'
#
.   x
.   .   y
.   z
```

> **Note**
>
> `⎕CS` is not permitted in a dfn or dop. If used therein it will cause a `NONCE ERROR`.

6.4.21 Comma Separated Values {R}←{X} ⎕CSV Y

This function imports and exports Comma Separated Value (CSV) data.

Monadic ⎕CSV imports data from a CSV file or converts data from CSV format to an internal format. Dyadic ⎕CSV exports data to a CSV file or converts data from internal format to a CSV format.

Internal Format

Arrays that result from importing CSV data or arrays that are suitable for exporting as CSV data are represented by 3 possible structures:

- A table (a matrix whose elements are character vectors or scalars, or numbers).
- A vector, each of whose items contain field (column) values. Character field values are character matrices; numeric field values are numeric vectors.
- A vector, each of whose items contain field (column) values. Character field values are vectors of character vectors; numeric field values are numeric vectors.

Note that when importing CSV data, all fields are assumed to be character fields unless otherwise specified (see *Column Types* below). A field that contains only "numbers" will not be converted to numeric data unless specified as being numeric.

MetaCharacters

Some characters in a CSV file are metacharacters which define the structure of the data; for example, the field separator character between fields. Characters which are not metacharacters are literal characters. The variant options QuoteChar, EscapeChar and DoubleQuote make it possible to interpret metacharacters as literal characters and thus permit fields to contain field separator characters, leading and trailing spaces, and line-endings.

Fixed-width fields do not require these options and they are ignored if fixed-width fields are being processed.

Monadic ⎕CSV

```
R←⎕CSV Y
```

`Y` is an array that specifies just the source of the CSV data (see below) or a 1,2,3 or 4-element vector containing:

`[1]`	Source of CSV Data
`[2]`	Description of the CSV data
`[3]`	Column Types
`[4]`	Header Row Indicator

Source may be one of:

- a character vector or scalar containing a file name
- a native tie number
- a character vector or scalar containing CSV data with embedded newline characters. To avoid this source being interpreted as a file name, `Y[2]` must be specified as `'S'`.
- a vector of character vectors and/or scalars containing CSV data with implicit newlines after each character vector or scalar

Description

If `Y[1]` is a file name or tie number *Description* may be one of:

- a character vector specifying the file encoding such as `'UTF-8'` (see *Section 6.4.92*).
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

If omitted or empty, the file encoding is deduced (see below).

If `Y[1]` is a character array containing CSV data *Description* is a character scalar `'S'` (simple) or `'N'` (nested). The default is `'N'`

Column Types

This is a scalar numeric code or vector of numeric codes that specifies the field types from the list below. If *Column Types* is zilde or omitted, the default is 1 (all fields are character).

0	The field is ignored.
1	The field contains character data.
2	The field is to be interpreted as being numeric. Empty cells and cells which cannot be converted to numeric values are not tolerated and cause an error to be signalled.
3	The field is to be interpreted as being numeric but invalid numeric values are tolerated. Empty fields and fields which cannot be converted to numeric values are replaced with the Fill variant option (default 0).
4	The field is to be interpreted numeric data but invalid numeric data is tolerated. Empty fields and fields which cannot be converted to numeric values are returned instead as character data; this type is disallowed when variant option Invert is set to 1.
5	The field is to be interpreted as being numeric but empty fields are tolerated and are replaced with the Fill variant option (default 0). Non-empty cells which cannot be converted to numeric values are not tolerated and cause an error to be signalled.

Note that if *Column Types* is specified by a scalar 4, all numeric data in all fields will be converted to numbers.

Header Row Indicator

This is a Boolean value (default 0) to specify whether or not the first record in a CSV file is a list of column labels. If *Header Row Indicator* is 1, the first record (the *header row*) is treated differently from other records. It is assumed to contain character data (labels) regardless of `Y[3]` and is returned separately in the result.

Variant options

Monadic `⍠CSV` may be applied using the Variant operator with the following options. The Principal option is Invert.

Name	Meaning	Default
Invert	0, 1 or 2 (see below)	0
Separator	The field separator, any single character. If Widths is other than 0 , Separator is ignored.	','
Widths	A vector of numeric values describing the width (in characters) of the corresponding columns in the CSV source, or 0 for variable width delimited fields	0
Decimal	The decimal mark in numeric fields - one of '.' or ','	'.'
Thousands	The thousands separator in numeric fields, which may be specified as an empty character vector (meaning no separator is defined) or a character scalar	''
Trim	A Boolean specifying whether undelimited/unescaped whitespace is trimmed at the beginning and end of fields	1
Ragged	A Boolean specifying whether records with varying numbers of fields are allowed; see notes below	0
Fill	The numeric value substituted for invalid numeric data in columns of type 3	0
Records	The maximum number of records to process or 0 for no limit.	0
QuoteChar	The field quote character (delimiter), which may be specified as an empty character vector (meaning none is defined) or a character scalar	"
EscapeChar	The escape character, which may be specified as an empty character vector (meaning none is defined) or a character scalar	''
DoubleQuote	A Boolean which indicates whether (1) or not (0) a quote character within a quoted field is represented by two consecutive quote characters	1

The Separator, QuoteChar and EscapeChar characters, when defined, must be different.

Other options defined for export are also accepted but ignored.

Invert Option

This option specifies how the CSV data should be returned as follows:

0	A table (a matrix whose elements are character vectors or scalars or numbers).
1	A vector, each of whose items contain field (column) values. Character field values are character matrices; numeric field values are numeric vectors.
2	A vector, each of whose items contain field (column) values. Character field values are vectors of character vectors; numeric field values are numeric vectors.

QuoteChar, EscapeChar and DoubleQuote Options

If EscapeChar is set then any character may be prefixed by the escape character. The escape character is typically defined as `'\'`. The escape character immediately followed by the character `c` is the literal character `c` even if `c` alone would have been a metacharacter.

If QuoteChar is set then fields may be delimited by the specified quote character. Within quoted fields all characters except the quote character, and the escape character if defined, are literal characters.

If DoubleQuote is set to 1 then two consecutive quote characters within a quoted field are interpreted as the single literal quote character.

Result

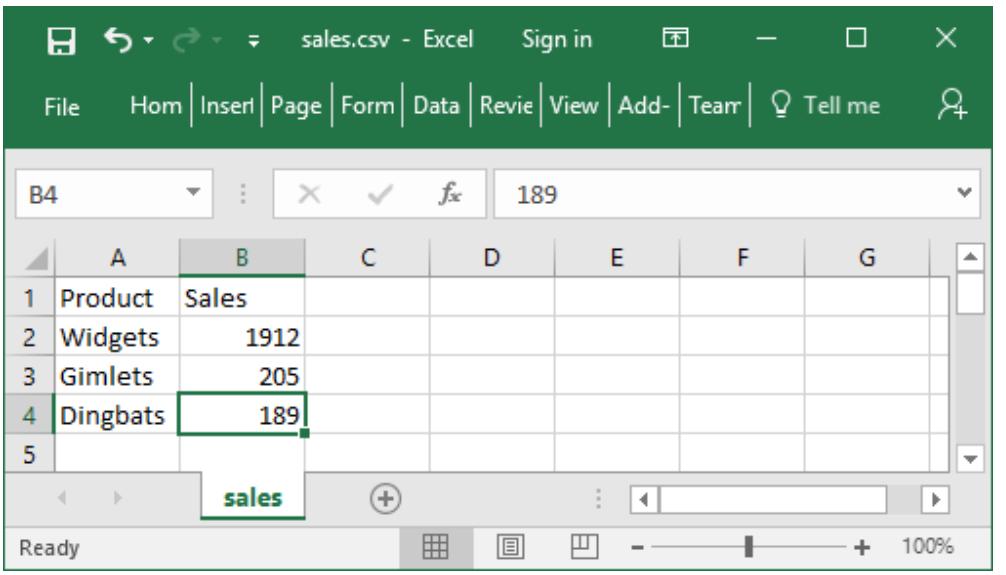
The result `R` contains the imported data.

If `Y[4]` does not specify that the data contains a header then `R` contains the entire data in the form specified by the Invert variant option.

If `Y[4]` does specify that the data contains a header then `R` is a 2-element vector where:

- `R[1]` is the imported data excluding the header.
- `R[2]` is a vector of character vectors containing the header record.

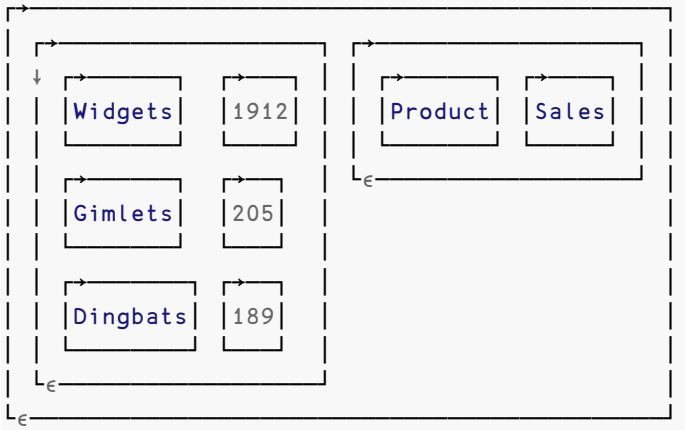
EXAMPLES



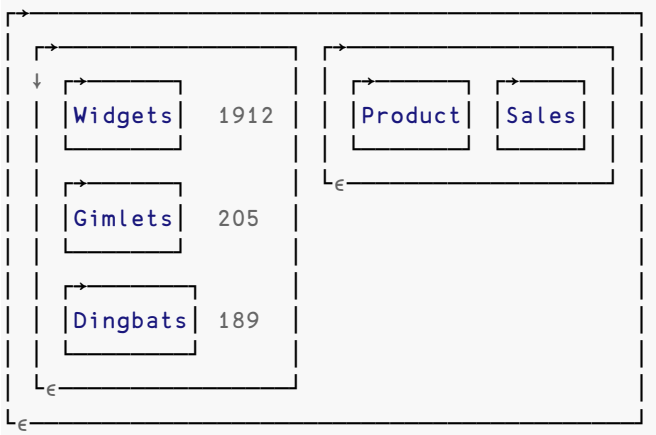

```
      ⍇⎕NGET CSVFile←'c:\Dyalog16.0\sales.csv'
┌→──────────────────────────────────────────┐
│Product,Sales                              │
│             Widgets,1912                  │
│                        Gimlets,205        │
│                                   Dingbats,189│
│                                           │
└──────────────────────────────────────────┘
```

```
      ⎕CSV CSVFile
┌→──────────────────┐
↓ ┌→──────┐ ┌→──┐   │
  │Product│ │Sales│  │
  └───────┘ └─────┘  │
  ┌→──────┐ ┌→──┐   │
  │Widgets│ │1912│  │
  └───────┘ └────┘   │
  ┌→──────┐ ┌→──┐   │
  │Gimlets│ │205│   │
  └───────┘ └───┘   │
  ┌→──────┐ ┌→──┐   │
  │Dingbats│ │189│  │
  └───────┘ └───┘   │
└∊──────────────────┘
```

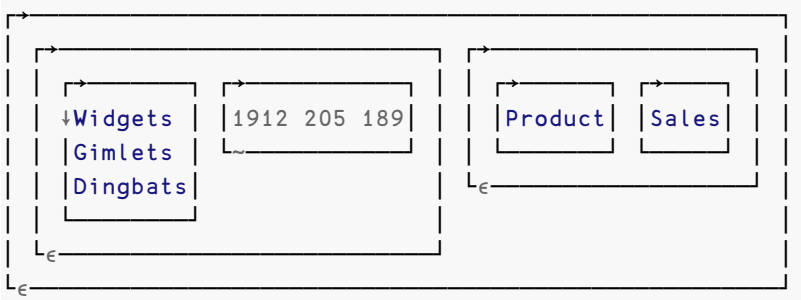
```
      ⎕CSV CSVFile'' ⍬ 1 ⍝ Header row
```




```
      ⎕CSV CSVFile''(1 2)1 ⍝ Fields are Char, Num
```




```
      (⎕CSV⍠'Invert' 1)CSVFile'' (1 2) 1    ⍝ Invert 1
```



```
      ⊃(⎕CSV⍠'Invert' 2)CSVFile'' (1 2) 1     ⍴ Invert 2
┌→──────────────────────────────────────────────────────┐
│ ┌→──────────────────────────────────┐ ┌→────────────┐ │
│ │ ┌→─────┐ ┌→─────┐ ┌→─────┐       │ │1912 205 189│ │
│ │ │Widgets│ │Gimlets│ │Dingbats│    │ └~────────────┘ │
│ │ └──────┘ └──────┘ └──────┘       │                 │
│ └∈─────────────────────────────────┘                 │
└∈──────────────────────────────────────────────────────┘
```

Notes

- When `Y` specifies just the source of the CSV data, it does not need to be enclosed or ravelled to create a 1-element vector.
- `Y[2]`, the description of the source, distinguishes an otherwise ambiguous character vector source (which could contain either CSV data or a file name). The other source forms are unambiguous but the description, when given, must still match the given source type.
- Tab-separated fields may be imported by specifying `'Separator' (⎕UCS 9)`.
- Fields containing embedded new lines are supported (they must, of course, appear in quotes or be prefixed by the escape character). On import, line endings are always converted to a single line feed character.
- If Ragged is not set then all records must have the same number of fields (character delimited format) or same number of characters (fixed width field format).
- If Ragged is set:
- The expected number of columns must be specified using the Widths variant option and/or the column types in `Y[3]`.
- In character delimited format, all processed records are implicitly extended or truncated as required so that they contain the expected number of fields; implicitly added fields will be empty.
- In fixed width format, all processed records are implicitly extended with spaces or truncated as required so that they contain as many characters as are specified in the Widths option declaration.

File handling

Data may be read from a named file or a tied native file. A tied native file may be read in sections by repeatedly invoking `⎕CSV` for a specified maximum number of records (specified by the Records variant) until no more data is read.

In all cases the files must contain text using one of the supported encodings (see *Section 6.4.92*). The method used to determine the file encoding is as follows:

- If a Byte Order Mark (BOM) is encountered at the start of the file, it is used regardless of `Y[2]` (if specified). Note, however, that the BOM can only be encountered if the file is read from the start - specifically, if a native file is read in sections, any BOM present will only be encountered when the first section is read.
- Otherwise, the file will be read and decoded according to the file encoding in `Y[2]` if specified.
- Otherwise: Native files will be decoded as if `'UTF-8'` had been specified. Files specified by name will be examined and the likely file encoding will be deduced using the same heuristics performed by `⎕NGET`.
- Native files will be decoded as if `'UTF-8'` had been specified.
- Files specified by name will be examined and the likely file encoding will be deduced using the same heuristics performed by `⎕NGET`.

Note that:

- native files are read from the current file position. On successful completion, the file position will be at the first unprocessed character (end of file if the Records variant option is not specified). If an error is signalled the file position is undefined.
- the result does not report the file encoding or line ending type as it does with `⎕NGET`. If this information is required then it must be obtained by other means.

Dyadic `⎕CSV`

```
{R}←X ⎕CSV Y
```

The left argument `X` is either:

- a matrix or a vector of vectors/matrices containing the data to be converted to CSV format.
- or a 2-element vector containing a matrix or vector of vectors/matrices containing the data to be converted to CSV format, and a vector of character vectors containing the header record.

`Y` is a 1 or 2-element vector containing:

`[1]`	Destination of CSV Data (see below)
`[2]`	Description of the CSV data (see below)

Destination - may be one of:

- a character vector or scalar containing a file name
- a native tie number
- an empty character vector, indicating that the CSV data is to be returned in the result `R`

Description

If `Y[1]` is a file name or tie number, *Description* may be:

- a character vector specifying the file encoding such as `'UTF-8'` (see *Section 6.4.92*).
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

If `Y[1]` is empty, *Description* may be a character scalar `'S'` (simple) or `'N'` (nested). If omitted, the default is `'S'`

Variant options

Dyadic `⎕CSV` may be applied using the Variant operator with the following options.

Name	Meaning	Default
IfExists	a character vector `'Error'` or `'Replace'` which specifies, when creating a named file which already exists, whether to overwrite it (`'Replace'`) or signal an error (`'Error'`)	`'Error'`
Separator	the field separator, any single character. If Widths is other than `0` , Separator is ignored.	`','`
Widths	a vector of numeric values describing the width (in characters) of the corresponding columns in the CSV source, or `0` for variable width delimited fields	`0`
Decimal	the decimal mark in numeric fields - one of `'.'` or `','`	`'.'`
Thousands	the thousands separator in numeric fields, which may be specified as an empty character vector (meaning no separator is defined) or a character scalar	`''`
Trim	a Boolean specifying whether whitespace is trimmed at the beginning and end of character fields	`1`
LineEnding	the line ending sequence - see *Line separators:*	(13 10) on Windows; 10 on other platforms
QuoteChar	The field quote character (delimiter), which may be specified as an empty character vector (meaning none is defined) or a character scalar	`"`
EscapeChar	The escape character, which may be specified as an empty character vector (meaning none is defined) or a character scalar	`''`
DoubleQuote	A Boolean which indicates whether (1) or not (0) a quote character within a quoted field is represented by two consecutive quote characters	`1`

The Separator, QuoteChar and EscapeChar characters, when defined, must be different. Other options defined for import are also accepted but ignored.

The Overwrite variant option (Boolean) from Version 16.0 remains supported but is deprecated in favour of IfExists.

QuoteChar, EscapeChar and DoubleQuote options

- The CSV text will be generated such that it can be read back according to the corresponding rules for import.
- If these options do not permit this (for example, a field contains the quote character and neither DoubleQuote or EscapeChar are set) an error is signalled.
- Quoting and Escaping is used as conservatively as possible.
- If both QuoteChar and EscapeChar are set, quoting is favoured.

If `Y` specifies that the CSV data is written to a file then `R` is the number of bytes (not characters) written, and is shy.

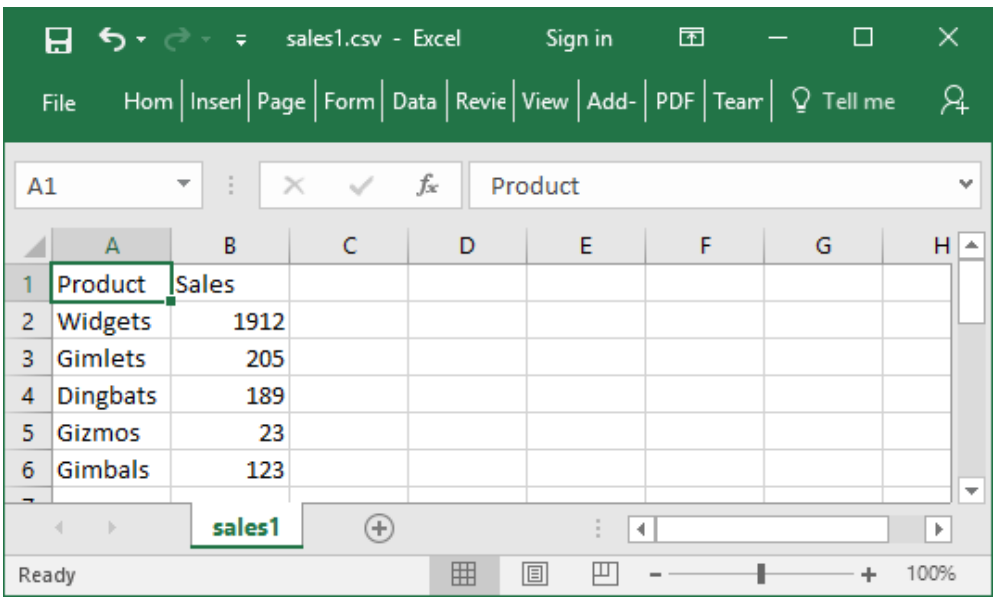
Otherwise, `R` is the CSV data in the format specified in Y, and is not shy.

Examples

```
      CSVFile←'c:\Dyalog16.0\sales.csv'
      □←DATA HDR←□CSV CSVFile''(1 2)1
┌→──────────────────────────────────────────────┐
│ ┌→──────────────────┐ ┌→──────────────────┐   │
│ ↓ ┌→────────┐       │ │ ┌→───────┐ ┌→───┐ │   │
│ │ │Widgets│  1912 │ │ │Product│ │Sales│ │   │
│ │ └─────────┘       │ │ └────────┘ └────┘ │   │
│ │ ┌→────────┐       │ └∊──────────────────┘   │
│ │ │Gimlets│  205  │                         │
│ │ └─────────┘       │                         │
│ │ ┌→─────────┐      │                         │
│ │ │Dingbats│ 189   │                         │
│ │ └──────────┘      │                         │
│ └∊──────────────────┘                         │
└∊──────────────────────────────────────────────┘
```

```
     DATA⍪←'Gizmos' 23
      DATA HDR ⎕CSV''
┌→──────────────┐
│Product,Sales  │
│               │
│Widgets,1912   │
│               │
│Gimlets,205    │
│               │
│Dingbats,189   │
│               │
│Gizmos,23      │
│               │
│               │
└───────────────┘
```

```
       CSVFile1←'c:\Dyalog16.0\sales1.csv'
       ⎕←DATA HDR ⎕CSV CSVFile1

67
       DATA⍪←'Gimbals' 123
       ⎕←DATA HDR ⎕CSV CSVFile1
FILE NAME ERROR: Unable to create file ("The file exists.")
       ⎕←DATA HDR ⎕CSV CSVFile1
      ^
       ⎕←DATA HDR(⎕CSV⍠'IfExists' 'Replace')CSVFile1

80
```



Notes

- When `Y` contains only the destination of the CSV data (that is, omits the description in its second element) it does not have to be enclosed to form a single element vector.
- Native files are written from the current file position. On successful completion, the file position will be at the end of the written data. If an error is signalled the amount of data written is undefined.
- If the file encoding specifies that a BOM is required and output is to a native file, it will only be written if the file position is initially at 0 - that is, the start of the file is being written.
- When fixed width fields are written, character data shorter than the specified width is padded with spaces to the right and character data longer than the specified width signals an error. Numeric data is converted to character data as far as possible so that it fits into the specified width. If this is not possible, an error is signalled.
- Tab-separated fields may be exported by specifying `'Separator' (⎕UCS 9)`.
- Fields containing a single embedded new line are supported. On export, line feed characters are mapped back to the defined line ending sequence.

6.4.22 Comparison Tolerance ⎕CT

The value of ⎕CT determines the precision with which two numbers are judged to be equal. Two numbers, X and Y, are judged to be equal if (|X-Y)≤⎕CT×(|X)⌈|Ywhere ≤ is applied without tolerance.

Thus ⎕CT is not used as an absolute value in comparisons, but rather specifies a relative value that is dependent on the magnitude of the number with the greater magnitude. It then follows that ⎕CT has no effect when either of the numbers is zero.

⎕CT may be assigned any value in the range from 0 to 2*¯32 (about 2.3E¯10). A value of 0 ensures exact comparison. The value in a clear workspace is 1E¯14. ⎕CT has Namespace scope.

If *Section 6.4.57* is 1287, the system uses ⎕DCT. See *Section 6.4.25*.

⎕CT and ⎕DCT are implicit arguments of the monadic primitive functions Ceiling (⌈), Floor (⌊) and Unique (∪), and of the dyadic functions Equal (=), Excluding (~), Find (⍷), Greater (>), Greater or Equal (≥), Greatest Common Divisor (∨), Index of (⍳), Intersection (∩), Less (<), Less or Equal (≤), Lowest Common Multiple (∧), Match (≡), Membership (∈), Not Match (≢), Not Equal (≠), Residue (|) and Union (∪), as well as ⎕FMT O-format.

EXAMPLES

```
      ⎕CT←1E¯10
      1.00000000001 1.0000001 = 1
1 0
```

6.4.23 Copy Workspace {R}←{X}⎕CY Y

Y must be a simple character scalar or vector identifying a saved workspace (or Session file). X is optional. If present, it must be a simple character scalar, vector or matrix or a vector of character vectors that specifies one or more APL names.

Each name in X is taken to be the name of an active object in the workspace identified by Y. If X is omitted, the names of all defined active objects in that workspace are implied (defined functions and operators, variables, labels and namespaces).

Each object named in X (or implied) is copied from the workspace identified by Y to become the active object referenced by that name in the active workspace if the object can be copied. A copied label is re-defined to be a variable of numeric type. If the name

of the copied object has an active referent in the active workspace, the name is disassociated from its value and the copied object becomes the active referent to that name. In particular, a function in the state indicator which is disassociated may be executed whilst it remains in the state indicator, but it ceases to exist for other purposes, such as editing.

The shy result `R` is `0⍴⊂''`.

You may copy an object from a namespace by specifying its full pathname. The object will be copied to the current namespace in the active workspace, losing its original parent and gaining a new one in the process. You may only copy a GUI object into a namespace that is a suitable parent for that object. For example, you could only copy a Group object from a saved workspace if the current namespace in the active workspace is itself a Form, SubForm or Group.

See *Section 7.2.5* for further information and, in particular, the manner in which dependent and referenced objects are copied, and copying objects from Session (.dse) files.

A `DOMAIN ERROR` is reported in any of the following cases:

- `Y` is ill-formed, or is not the name of a workspace with access authorised for the active user account.
- Any name in `X` is ill-formed.
- An object named in `X` does not exist as an active object in workspace named in `Y`.

An object being copied has the same name as an active label.

When copying data between Classic and Unicode Editions, `⎕CY` will fail and a `TRANSLATION ERROR` will be reported if *any* object in workspace `Y` fails conversion between Unicode and `⎕AV` indices, whether or not that object is specified by `X`. See *Section 6.4.12* for further details.

A `WS FULL` is reported if the active workspace becomes full during the copying process.

EXAMPLE

```
      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇
      'FOO' ⎕CY 'BACKUP'
      ⎕VR'FOO'
     ∇ R←FOO X
[1]    R←10×X
     ∇
```

System variables are copied if explicitly included in the left argument, but not if the left argument is omitted.

EXAMPLE

```
      ⎕LX

      ('⎕LX' 'X')⎕CY'WS/CRASH'
      ⎕LX
→RESTART
```

A copied object may have the same name as an object being executed. If so, the name is disassociated from the existing object, but the existing object remains defined in the workspace until its execution is completed.

EXAMPLE

```
      )SI
#.FOO[1]*

      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      'FOO'⎕CY'WS/MYWORK'

      FOO
1 2 3
      )SI
#.FOO[1]*
      →⎕LC
10
```

6.4.24 Digits R←⎕D

This is a simple character vector of the digits from 0 to 9.

EXAMPLE

```
      ⎕D
0123456789
```

6.4.25 Decimal Comparison Tolerance ⎕DCT

The value of ⎕DCT determines the precision with which two numbers are judged to be equal when the value of ⎕FR is 1287. If ⎕FR is 645, the system uses ⎕CT.

⎕DCT may be assigned any value in the range from 0 to 2*¯32 (about 2.3283064365386962890625E¯10). A value of 0 ensures exact comparison. The value in a clear workspace is 1E¯28. ⎕DCT has Namespace scope.

⎕CT and ⎕DCT are implicit arguments of the monadic primitive functions Ceiling (⌈), Floor (⌊) and Unique (∪), and of the dyadic functions Equal (=), Excluding (~), Find (⍷), Greater (>), Greater or Equal (≥), Greatest Common Divisor (∨), Index of (⍳), Intersection (∩), Less (<), Less or Equal (≤), Lowest Common Multiple (∧), Match (≡), Membership (∈), Not Match (≢), Not Equal (≠), Residue (|) and Union (∪), as well as ⎕FMT O-format.

For further information, see *Section 6.4.22*.

EXAMPLES

```
      ⎕DCT←1E¯10
      1.00000000001 1.0000001 = 1
1 0
```

6.4.26 Display Form {R}←⎕DF Y

⎕DF sets the *Display Form* of a namespace, a GUI object, a Class, or an Instance of a Class.

Y must be ⎕NULL or a simple character array that specifies the display form of a namespace. If defined, this array will be returned by the *format* functions and ⎕FMT instead of the default for the object in question. This also applies to the string that is

displayed when the name is referenced but not assigned (the *default display*). If `Y` is `⎕NULL`, `⎕DF` resets the Display Form to the default.

The result `R` is the previous value of the Display Form which initially is `⎕NULL`.

Example

```
      'F'⎕WC'Form'
      ⍹F
#.F
      ⍴⍹F
3
      ⎕FMT F
#.F
      ⍴⎕FMT F
1 3
      F ⍝ default display uses ⍹
#.F

      F.⎕DF 'Pete''s Form'
      ⍹F
Pete's Form
      ⍴⍹F
11
      ⎕FMT F
Pete's Form
      ⍴⎕FMT F
1 11
```

Notice that `⎕DF` will accept any character array, but `⎕FMT` always returns a matrix.

```
      F.⎕DF 2 2 5⍴⎕A
      F
ABCDE
FGHIJ

KLMNO
PQRST
      ⍴⍹F
2 2 5
```

```
      ⍴⍞←⎕FMT F
ABCDE
FGHIJ

KLMNO
PQRST
5 5
```

Note that ⎕DF defines the Display Form statically, rather than dynamically.

```
      'F'⎕WC'Form' 'This is the Caption'
      F
#.F

      F.(⎕DF Caption)⍝ set display form to current caption
      F
This is the Caption

      F.Caption←'New Caption' ⍝ changing caption does not
                              ⍝ change the display form
      F
This is the Caption
```

You may use the Constructor function to assign the Display Form to an Instance of a Class. For example:

```
:Class MyClass
    ∇ Make arg
      :Access Public
      :Implements Constructor
      ⎕DF arg
    ∇
:EndClass ⍝ MyClass

      PD←⎕NEW MyClass 'Pete'
      PD
Pete
```

It is possible to set the Display Form for the Root and for ⎕SE

```
      )CLEAR
clear ws
      #
#
      ⎕DF ⎕WSID
      #
CLEAR WS

      ⎕SE
⎕SE
      ⎕SE.⎕DF 'Session'
      ⎕SE
Session
```

Note that ⎕DF applies directly to the object in question and is not automatically applied in a hierarchical fashion.

```
      'X'⎕NS ''
      X
#.X

      'Y'X.⎕NS ''
      X.Y
#.X.Y
      X.⎕DF 'This is X'
      X
This is X

      X.Y
#.X.Y
```

6.4.27 Division Method ⎕DIV

The value of ⎕DIV determines how division by zero is to be treated. If ⎕DIV=0, division by 0 produces a DOMAIN ERROR except that the special case of 0÷0 returns 1.

If ⎕DIV=1, division by 0 returns 0.

⎕DIV may be assigned the value 0 or 1. The value in a clear workspace is 0.

⎕DIV is an implicit argument of the monadic function Reciprocal (÷) and the dyadic function Divide (÷). ⎕DIV has Namespace scope.

Examples

```
      ⎕DIV←0

      1 0 2 ÷ 2 0 1
0.5 1 2

      ÷0 1
DOMAIN ERROR
      ÷0 1
      ^

      ⎕DIV←1

      ÷0 2
0 0.5

      1 0 2 ÷ 0 0 4
0 0 0.5
```

6.4.28 Delay {R}←⎕DL Y

Y must be a simple non-negative single numeric value (of any rank). A pause of approximately Y seconds is caused.

The shy result R is a scalar numeric value indicating the length of the pause in seconds.

The pause may be interrupted by a weak or strong interrupt.

6.4.29 Diagnostic Message R←⎕DM

This niladic function returns the last reported APL error as a three-element vector, giving error message, line in error and position of caret pointer.

Example

```
      2÷0
DOMAIN ERROR
      2÷0
     ^

      ⎕DM
 DOMAIN ERROR         2÷0         ^
```

Note: ⎕SIGNAL can be used to reset the value of this system constant.

6.4.30 Extended Diagnostic Message R←⎕DMX

⎕DMX is a system object that provides information about the last reported APL error. ⎕DMX has *thread scope*, that is, its value differs according to the thread from which it is referenced. In a multi-threaded application therefore, each thread has its own value of ⎕DMX.

⎕DMX contains the following Properties (name class 2.6). Note that this list is likely to change. Your code should not assume that this list will remain unchanged. You should also not assume that the display form of ⎕DMX will remain unchanged.

`Category`	character vector	The category of the error
`DM`	nested vector	Diagnostic message. This is the same as ⎕DM , but thread safe
`EM`	character vector	Event message; this is the same as ⎕EM ⎕EN
`EN`	integer	Error number. This is the same as ⎕EN , but thread safe
`ENX`	integer	Sub-error number
`HelpURL`	character vector	URL of a web page that will provide help for this error. APL identifies and has a handler for URLs starting with *http:* , *https:* , *mailto:* and *www.* This list may be extended in future
`InternalLocation`	nested vector	Identifies the line of interpreter source code (file name and line number) which raised the error. This information may be useful to Dyalog support when investigating an issue
`Message`	character vector	Further information about the error
`OSError`	see below	If applicable, identifies the error generated by the Operating System or by a child process
`Vendor`	character vector	For system generated errors, `Vendor` will always contain the character vector `'Dyalog'` . This value can be set using ⎕SIGNAL

OSError is a 3-element vector whose items are as follows:

1	integer	This indicates how an error from the operating system or a child process was retrieved. 0 = by the C-library `errno()` function 1 = by the Windows `GetLastError()` function 2 = as the exit status of a child process
2	integer	Error code. The error number returned by the operating system using `errno()` or `GetLastError()` as above
3	character vector	The description of the error returned by the operating system

EXAMPLE

```
      1÷0
DOMAIN ERROR
      1÷0
     ^
      ⎕DMX
 EM       DOMAIN ERROR
 Message  Divide by zero

      ⎕DMX.InternalLocation
 arith_su.c  554
```

Isolation of Handled Errors

⎕DMX cannot be explicitly localised in the header of a function. However, for all trapped errors, the interpreter creates an environment which effectively makes the current instance of ⎕DMX local to, and available only for the duration of, the trap-handling code.

With the exception of ⎕TRAP with Cutback, ⎕DMX is implicitly localised within:

- Any function which explicitly localises ⎕TRAP
- The `:Case[List]` or `:Else` clause of a `:Trap` control structure.
- The right hand side of a D-function Error-Guard.

and is implicitly un-localised when:

- A function which has explicitly localised ⎕TRAP terminates (even if the trap definition has been inherited from a function further up the stack).
- The `:EndTrap` of the current `:Trap` control structure is reached.
- A D-function Error-Guard exists.

During this time, if an error occurs then the localised ⎕DMX is updated to reflect the values generated by the error.

The same is true for ⎕TRAP with Cutback, with the exception that if the cutback trap event is triggered, the updated values for ⎕DMX are preserved until the function that set the cutback trap terminates.

The benefit of the localisation strategy is that code which uses error trapping as a standard operating procedure (such as a file utility which traps `FILE NAME ERROR` and creates missing files when required) will not pollute the environment with irrelevant error information.

Example

```
     ∇ tie←NewFile name
[1]    :Trap 22
[2]        tie←name ⎕FCREATE 0
[3]    :Else
[4]        ⎕DMX
[5]        tie←name ⎕FTIE 0
[6]        name ⎕FERASE tie
[7]        tie←name ⎕FCREATE 0
[8]    :EndTrap
[9]    ⎕FUNTIE tie
     ∇
```

⎕DMX is cleared by `)RESET`:

```
      )RESET
      ⍴⎕FMT ⎕DMX
0 0
```

Note: ⎕SIGNAL can be used to reset the value of this system constant.

The first time we run `NewFile 'pete'`, the file doesn't exist and the ⎕FCREATE in `NewFile[2]` succeeds.

```
      NewFile 'pete'
1
```

If we run the function again, the ⎕FCREATE in `NewFile[2]`generates an error which triggers the `:Else` clause of the `:Trap`. On entry to the `:Else` clause, the values in ⎕DMX reflect the error generated by ⎕FCREATE. The file is then tied, erased and recreated.

```
EM       FILE NAME ERROR
Message  File exists
```

After exiting the `:Trap` control structure, the shadowed value of ⎕DMX is discarded, revealing the original value that it shadowed.

```
      ⍴⎕FMT ⎕DMX
0 0
```

Example

The `EraseFile` function also uses a `:Trap` in order to ignore the situation when the file doesn't exist.

```
    ∇ EraseFile name;tie
[1]   :Trap 22
[2]       tie←name ⎕FTIE 0
[3]       name ⎕FERASE tie
[4]   :Else
[5]       ⎕DMX
[6]   :EndTrap
    ∇
```

The first time we run the function, it succeeds in tieing and then erasing the file.

```
      EraseFile 'pete'
```

The second time, the ⎕FTIE fails. On entry to the `:Else` clause, the values in ⎕DMX reflect this error.

```
      EraseFile 'pete'
EM       FILE NAME ERROR
Message  Unable to open file
OSError  1 2  The system cannot find the file specified.
```

Once again, the local value of ⎕DMX is discarded on exit from the `:Trap`, revealing the shadowed value as before.

```
      ⍴⎕FMT ⎕DMX
0 0
```

Example

In this example only the error number (`EN`) property of ⎕DMX is displayed in order to simplify the output:

```
     ∇ foo n;⎕TRAP
[1]    'Start foo'⎕DMX.EN
[2]    ⎕TRAP←(2 'E' '→err')(11 'C' '→err')
[3]    goo n
[4]   err:'End foo:'⎕DMX.EN
     ∇

     ∇ goo n;⎕TRAP
[1]    ⎕TRAP←5 'E' '→err'
[2]    ⍎n⊃'÷0' '1 2+1 2 3' '∘'
[3]   err:'goo:'⎕DMX.EN
     ∇
```

In the first case a `DOMAIN ERROR` (11) is generated on `goo[2]`. This error is not included in the definition of `⎕TRAP` in `goo`, but rather the Cutback `⎕TRAP` definition in `foo`. The error causes the stack to be cut back to `foo`, and then execution branches to `foo[4]`. Thus `⎕DMX.EN` in `foo` retains the value set when the error occurred in `goo`.

```
      foo 1
Start foo  0
End foo:  11
```

In the second case a `LENGTH ERROR` (5) is raised on `goo[2]`. This error is included in the definition of `⎕TRAP` in `goo` so the value `⎕DMX.EN` while in `goo` is 5, but when `goo` terminates and `foo` resumes execution the value of `⎕DMX.EN` localised in `goo` is lost.

```
      foo 2
Start foo  0
goo:  5
End foo:  0
```

In the third case a `SYNTAX ERROR` (2) is raised on `goo[2]`. Since the `⎕TRAP` statement is handled within `goo` (although the applicable `⎕TRAP` is defined in `foo`), the value `⎕DMX.EN` while in `goo` is 2, but when `goo` terminates and `foo` resumes execution the value of `⎕DMX.EN` localised in `goo` is lost.

```
      foo 3
Start foo  0
goo:  2
End foo:  0
```

6.4.31 Dequeue Events {R}←⎕DQ Y

Windows only.

⎕DQ awaits and processes events. Y specifies the GUI objects(s) for which events are to be processed. Objects are identified by their names, as character scalars/vectors, or by namespace references. These may be objects of type Root, Form, Locator, FileBox, MsgBox, PropertySheet, TCPSocket, Timer, Clipboard and pop-up Menu. Sub-objects (children) of those named in Y are also included. However, any objects which exist, but are not named in Y, are effectively disabled (do not respond to the user).

If Y is #, '#', or '.', all objects currently owned and subsequently created by the current thread are included in the ⎕DQ. Note that because the Root object is owned by thread 0, events on Root are reported only to thread 0.

If Y is empty it specifies the object associated with the current namespace and is only valid if the current space is one of the objects listed above.

Otherwise, Y contains the name(s) of or reference(s) to the objects for which events are to be processed. Effectively, this is the list of objects with which the user may interact. A DOMAIN ERROR is reported if an element of Y refers to anything other than an existing "top-level" object.

Associated with every object is a set of events. For every event there is defined an "action" which specifies how that event is to be processed by ⎕DQ. The "action" may be a number with the value 0, 1 or ¯1, a character vector containing the name of a "callback function", or a character vector containing the name of a callback function coupled with an arbitrary array. Actions can be defined in a number of ways, but the following examples will illustrate the different cases.

```
      OBJ ⎕WS 'Event' 'Select' 0

      OBJ ⎕WS 'Event' 'Select' 1

      OBJ ⎕WS 'Event' 'Select' 'FOO'

      OBJ ⎕WS 'Event' 'Select' 'FOO' 10

      OBJ ⎕WS 'Event' 'Select' 'FOO&'
```

These are treated as follows:

Action = 0 (the default)

⎕DQ performs "standard" processing appropriate to the object and type of event. For example, the standard processing for a KeyPress event in an Edit object is to action the key press, that is, to echo the character on the screen.

Action = ¯1

This disables the event. The "standard" processing appropriate to the object and type of event is **not** performed, or in some cases is reversed. For example, if the "action code" for a KeyPress event (22) is set to `¯1`, `⎕DQ` simply ignores all keystrokes for the object in question.

Action = 1

`⎕DQ` terminates and returns information pertaining to the event (the **event message**) in `R` as a nested vector whose first two elements are the name of the object (that generated the event) and the event code. `R` may contain additional elements depending upon the type of event that occurred.

Action = fn

`fn` is a character vector containing the name of a *callback* function. This function is automatically invoked by `⎕DQ` whenever the event occurs, and **prior** to the standard processing for the event. The callback is supplied the **event message** (see above) as its right argument, and, if specified, the array `larg` as its left argument. If the callback function fails to return a result, or returns the scalar value `1`, `⎕DQ` then performs the standard processing appropriate to the object and type of event. If the callback function returns a scalar 0, the standard processing is not performed or in some cases is reversed.

If the callback function returns its event message with some of the parameters changed, these changes are incorporated into the standard processing. An example would be the processing of a keystroke message where the callback function substitutes upper case for lower case characters. The exact nature of this processing is described in the reference section on each event type.

Action = ⍎expr

If `Action` is set to a character vector whose first element is the execute symbol (`⍎`) the remaining string will be executed automatically whenever the event occurs. The default processing for the event is performed first and may not be changed or inhibited in any way.

Action = fn&

`fn` is a character vector containing the name of a *callback* function. The function is executed in a new thread. The default processing for the event is performed first and may not be changed or inhibited in any way.

The Result of ⎕DQ

`⎕DQ` terminates, returning the shy result `R`, in one of four instances.

Firstly, `⎕DQ` terminates when an event occurs whose "action code" is 1. In this case, its result is a nested vector containing the **event message** associated with the event. The structure of an event message varies according to the event type (see *Object Reference*). However, an event message has at least two elements of which the first is a ref to the object or a character vector containing the name of the object, and the second is a character vector or numeric code which identifies the event type.

`⎕DQ` also terminates if all of the objects named in `Y` have been deleted. In this case, the result is an empty character vector. Objects are deleted either using `⎕EX`, or on exit from a defined function or operator if the names are localised in the header, or on closing a form using the system menu.

Thirdly, `⎕DQ` terminates if the object named in its right argument is a special *modal* object, such as a `MsgBox`, `FileBox` or `Locator`, and the user has finished interacting with the object (for example, by pressing an "OK" button). The return value of `⎕DQ` in this case depends on the action code of the event.

Finally, `⎕DQ` terminates with a `VALUE ERROR` if it attempts to execute a callback function that is undefined.

6.4.32 Data Representation (Dyadic) `R←X ⎕DR Y`

Dyadic `⎕DR` converts the data type of its argument `Y` according to the type specification `X`. See *Section 6.4.33* for a list of data types but note that 1287 is not a permitted value in `X`.

Case 1

`X` is a single integer value. The bits in the right argument are interpreted as elements of an array of type `X`. The shape of the resulting new array will typically be changed along

the last axis. For example, a character array interpreted as Boolean will have 8 times as many elements along the last axis.

Case 2

`X` is a 2-element integer value. `Y` is any array.

`X[1]` is either 0 or a data type. If `X[1]` is 0, `Y` is converted to data type `X[2]`. If `X[1]` is non-zero, the bits in `Y` are first interpreted as being of type `X[1]` before being converted to type `X[2]`. If `Y` is a scalar it is ravelled. Conversion of `Y` from one internal data type to another is performed so as to preserve its values without loss of precision.

The result `R` is a two element nested array comprised of:

1. The converted elements or a fill element (0 or blank) where the conversion failed
2. A Boolean array of the same shape indicating which elements were successfully converted.

Examples

```
      bits← 0 1 0 0 1 0 0 0 , 0 1 0 0 1 0 1 1
      80 ⎕DR bits
HK
      83 ⎕DR bits
72 75
      163 ⎕DR bits
19272

      0 645 ⎕DR 72 75
┌─────┬───┐
│72 75│1 1│
└─────┴───┘
      163 645 ⎕DR 72 75
┌─────┬─┐
│19272│1│
└─────┴─┘
```

Case 3: Classic Edition Only

`X` is a 3-element integer value and `X[2 3]` is `163 82`. The bits in the right argument are interpreted as elements of an array of type `X[1]`. The system then converts them to the

character representation of the corresponding 16 bit integers. This case is provided primarily for compatibility with APL*PLUS. For new applications, the use of the [conv] field with ⎕NAPPEND and ⎕NREPLACE is recommended.

Conversion to and from character (data type 82) uses the translate vector given by ⎕NXLATE 0. By default this is the mapping defined by the current output translate table (usually WIN.DOT).

Notes

- The internal representation of data may be modified during workspace compaction. For example, numeric arrays and (in the Unicode Edition) character arrays will, if possible, be squeezed to occupy the least possible amount of memory. However, the internal representation of the result R is guaranteed to remain unmodified until it is re-assigned (or partially re-assigned) with the result of any function.
- The precise operation of dyadic ⎕DR depends upon the byte-ordering scheme of the computer system.

6.4.33 Data Representation (Monadic) R←⎕DR Y

Monadic ⎕DR returns the type of its argument Y. The result R is an integer scalar containing one of the following values. Note that the internal representation and data types for character data differ between the Unicode and Classic Editions.

Table: Unicode Edition

Value	Data Type
11	1 bit Boolean
80	8 bits character
83	8 bits signed integer
160	16 bits character
163	16 bits signed integer
320	32 bits character
323	32 bits signed integer
326	Pointer (32-bit or 64-bit as appropriate)
645	64 bits Floating
1287	128 bits Decimal
1289	128 bits Complex

Table: Classic Edition

Value	Data Type
11	1 bit Boolean
82	8 bits character
83	8 bits signed integer
163	16 bits signed integer
323	32 bits signed integer
326	Pointer (32-bit or 64-bit as appropriate)
645	64 bits Floating
1287	128 bits Decimal
1289	128 bits Complex

6.4.34 Date-time R←X □DT Y

This function validates date-times or converts date-times between one format and another.

A date-time is a date and time of day represented by a *timestamp*, a *time number* or a *military time-zone character*.

- A *timestamp* is a date-time expressed as a multiple element numeric vector, of which there are several different sorts (principally `⎕TS` format).
- A *time number* is a date-time expressed as a scalar numeric value, of which there are several different sorts.
- A *military time zone character* is a scalar character that represents the current date-time ("now") in a particular time zone. For example, `'A'` represents the current date-time (UTC) + 1 hour.

`Y` is an array of any shape whose elements contain a timestamp, time number or military time zone character, in any combination.

`X` may be a single integer value or a 2-element integer vector.

When `X` is a single integer value, it must be either 0 or a date-time code listed in the tables below. 0 specifies that the elements of `Y` are to be validated. A non-zero value specifies the date-time representation to which the elements of `Y` are to be converted. In this case, the numeric elements of `Y` are interpreted as follows:

- scalars are assumed to be time numbers of type Dyalog Date Number (code 1)
- vectors are assumed to be `⎕TS` timestamps (code ¯1)

When `X` is a 2-element integer vector, `X[1]` is a date-time code that explicitly specifies the date-time representation of the numeric elements in `Y`. `X[2]` is either 0 or a date-time code listed in the tables below. 0 specifies that the elements of `Y` are to be validated. A non-zero value specifies the date-time representation to which the elements of `Y` are to be converted.

Character scalars in `Y` are always interpreted as meaning "now".

`R` is an array of the same shape as `Y`, where each element is either a timestamp, time number or Boolean value as determined by the second or only element in `X`.

Time numbers in `R` may be of type DECF even if `⎕FR` is 645 if their magnitude can be too great to store precisely in a double. See the table below for the type numbers where this is so.

Time Numbers

If a value in `X` is positive it indicates that a time number type is expected in `Y` or generated in `R`, as follows. Note that the last column indicated whether (Yes) or not (No) negative numbers are allowed.

Group	Code	Description	Category	Date and time[6] represented by 0 (Epoch)	Negative values allowed? [7]
Dyalog APL	1	Dyalog Date Number	Day count with fractional part	1899-12-31 00:00	Yes
	2	Dyalog component file time	Tick count 1÷60s ticks[8]	1970-01-01 00:00	Yes
Other languages	10	J (J nanosecond time)	Tick count[9] 1ns ticks [2]	2000-01-01 00:00	Yes
	11	Shakti K7	Tick count 1ms ticks[2]	2024-01-01 00:00	Yes
	12	JavaScript / D / Q / Go UnixMilli	Tick count 1ms ticks[2]	1970-01-01 00:00	Yes
	13	R (R chron format)	Day count with fractional part	1970-01-01 00:00	Yes
	14	Shakti K9	Tick count 1ms ticks[2]	2001-01-01 00:00	Yes
	15	Go UnixMicro	Tick count 1µs ticks[2]	1970-01-01 00:00	Yes
	16	Go UnixNano	Tick count 1ns ticks[2]	1970-01-01 00:00	Yes
	17	APL+Win and APL64 workspace timestamp	Tick count 1µs ticks[2]	1900-01-01 00:00	No
UNIX	20	Unix time	Tick count 1s ticks[2]	1970-01-01 00:00	Yes

6 **In the Proleptic Gregorian Calendar.**

7 **No date-time may represent a date earlier than ¯4713-01-01 00:00.**

8 There are the same number of ticks per day regardless of leap seconds.

9 Generated as DECF values regardless of the setting of □FR due to their magnitude.

Group	Code	Description	Category	Date and time[6] represented by 0 (Epoch)	Negative values allowed? [7]
	21	Apollo NCS UUID	Tick count 4µs ticks[2]	1980-01-01 00:00	No
	22	OSF DCE UUID	Tick count 1ns ticks[2]	1582-10-15 00:00	No
Microsoft Windows	30	Microsoft DOS date/ time	Encoded broken-down time 2s resolution	N/A	No
	31	Microsoft Win32 FILETIME	Tick count[3] 100ns ticks	1601-01-01 00:00	No
	32	Microsoft CLR DateTime (.NET)(Ticks property thereof)	Tick count [3] 100ns ticks	0001-01-01 00:00	No
	33	Microsoft OLE Automation Date(also known as Variant Time)	Day count with fractional part	1899-12-30 00:00	Yes [10]
Application	40	Excel (1900 Date System)[11] / Lotus 1-2-3	Day count with fractional part[12]	1899-12-31 00:00[13]	No
	41	Excel (1904 Date System)[4]	Day count with	1904-01-01 00:00	No

10 For negative numbers, the integral part counts backward from 1899-12-30 and the fractional part counts forward from the date so reached.

11 Excel supports two time number conventions. On Windows the 1900 Date System is the default and on macOS the 1904 Date System is the default. Both systems can use either convention and the convention in use is stored in the worksheet so that the platforms interoperate.

12 Count includes the invalid date 1900-02-29.

13 Microsoft Excel converts day 0 to the invalid date 1900-01-00.

Group	Code	Description	Category	Date and time[6] represented by 0 (Epoch)	Negative values allowed? [7]
			fractional part		
	42	Stata statistics package	Tick count 1ms ticks[2]	1960-01-01 00:00	Yes
	43	SPSS statistics package	Tick count 1s ticks[2]	1582-10-14 00:00	No
	44	SAS	Tick count 1s ticks[2]	1960-01-01 00:00	Yes
Julian Date and variants	50	Julian Date	Day count with fractional part	¯4717-11-24 12:00	No
	51	J (J dayno)	Day count with fractional part	1800-01-01 00:00	No
	52	Reduced Julian Date	Day count with fractional part	1858-11-16 12:00	Yes
	53	Modified Julian Date	Day count with fractional part	1858-11-17 00:00	Yes
	54	Dublin Julian Date	Day count with fractional part	1899-12-31 12:00	Yes
	55	CNES Julian Date	Day count with fractional part	1950-01-01 00:00	Yes

Group	Code	Description	Category	Date and time[6] represented by 0 (Epoch)	Negative values allowed? [7]
	56	CCSDS Julian Date	Day count with fractional part	1958-01-01 00:00	Yes
Decimal encoded[14]	60	Floating-point decimal encoded format Digits take the form yyyymmdd.hhmmss	Encoded broken-down time 1s resolution	N/A	No
	61	Integer decimal encoded format Digits take the form yyyymmddhhmmss(J digit time)	Encoded broken-down time 1s resolution	N/A	No
Misc. Operating Systems	70	AmigaOS	Tick count 1ms ticks[2]	1978-01-01 00:00	No

Time Stamps

If a value in `X` is negative it indicates that a timestamp type is expected in `Y` or generated in `R`, as follows:

[14] Decimal encoded formats encode human-readable dates and times into a single number with the most significant part in the most significant decimal digit, for example 2020/01/23 (year/month/day) is encoded as 20200123, and 13:17:56 (hour:minute:second) is encoded as 131756. The date must be between 1 January 0001 and 31 December 9999 in the Proleptic Gregorian Calendar.

Group	Code	Description	Max elements	Element contents[15]	Elided element implicit values (in Y)[16]
APL 7-element vector	¯1	Millisecond precision (⎕TS)	7	Year, month, day-of-month, hour, minute, second, millisecond	1 1 1 0 0 0 0
	¯2	Microsecond precision	7	Year, month, day-of-month, hour, minute, second, microsecond	1 1 1 0 0 0 0
	¯3	Nanosecond precision (J expanded digit time)	7	Year, month, day-of-month, hour, minute, second, nanosecond	1 1 1 0 0 0 0
ISO components	¯10	ISO day-of-year components	6	Year, day-of-year, hour, minute, second, microsecond	1 1 0 0 0 0 0
	¯11	ISO day-of-week components	7	Year, week, day-of-week, hour, minute, second, microsecond	1 1 0 0 0 0
Decimal encoded[2]	¯20	Decimal encoded date and time	2	Decimal encoded date, decimal encoded time	1 0 1 0 1 0
DateTimePicker	¯30		7	International Day Number,	

15 **All dates must be between 1 January 0001 and 28 February 4000 in the Proleptic Gregorian Calendar.**

16 **If a timestamp has fewer than the maximum number of elements, the remaining elements take the default values shown.**

Group	Code	Description	Max elements	Element contents[15]	Elided element implicit values (in Y)[16]
		DateTime format		hour, minute, second	1 1 1 0 0 0

Military time zone characters

Any element in `Y` may be specified as a military time zone character and is implicitly replaced by the current time in the time zone they represent. The time zones are as follows:

Character	Time zone name	Time zone
A	Alpha	UTC +1
B	Bravo	UTC +2
C	Charlie	UTC +3
D	Delta	UTC +4
E	Echo	UTC +5
F	Foxtrot	UTC +6
G	Golf	UTC +7
H	Hotel	UTC +8
I	India	UTC +9
J	Juliet	Local time
K	Kilo	UTC +10
L	Lima	UTC +11
M	Mike	UTC +12
N	November	UTC -1
O	Oscar	UTC -2
P	Papa	UTC -3
Q	Quebec	UTC -4
R	Romeo	UTC -5
S	Sierra	UTC -6
T	Tango	UTC -7
U	Uniform	UTC -8
V	Victor	UTC -9
W	Whisky	UTC -10
X	X-ray	UTC -11
Y	Yankee	UTC -12
Z	Zulu	UTC +0

The resolutions of system clocks vary by platform.

Examples

Timestamp to time number conversion

```
      ¯1 1 ⎕DT ⊂⎕TS
43886.48039
      1 ⎕DT ⊂⎕TS
43886.48039
      1 ⎕DT ⎕TS 'J'
43886.48039 43886.48039

      1 ⎕DT ⊂⍬ ⍝ cf Elided element implicit values
¯693594
      1 ⎕DT ⊂1 1 1 0 0 0 0
¯693594
       ¯30 1 ⎕DT⊂(44217 15 13 54)
44217.63465
```

Time number to timestamp conversion

```
      1 ¯1 ⎕DT 0 43508.42843
┌──────────────────┬──────────────────────────┐
│1899 12 31 0 0 0 0│2019 2 13 10 16 56 352    │
└──────────────────┴──────────────────────────┘
      ¯1 ⎕DT 0 43508.42843
┌──────────────────┬──────────────────────────┐
│1899 12 31 0 0 0 0│2019 2 13 10 16 56 352    │
└──────────────────┴──────────────────────────┘
      2 ¯1 ⎕DT 3⊃⎕FRDCI 1 1
┌──────────────────────┐
│2020 2 26 11 33 54 466│
└──────────────────────┘
      1 ¯30 ⎕DT 44217.63465
┌─────────────────┐
│44217 15 13 53   │
└─────────────────┘
```

Time number to time number conversion

```
      2 1 ⎕DT 3⊃⎕FRDCI 1 1
43886.48188
      1 ⎕DT 'J'
43886.48371
      ⍝ Local time is UTC-05:00
      3600÷⍨¯/20 ⎕DT 'JZ'
¯5
```

Timestamp to timestamp conversions

```
      ¯30 ⎕DT ⊂⎕TS
┌──────────────┐
│44216 16 5 46│
└──────────────┘

      ¯30 ¯1 ⎕DT⊂32000 15 10 0
┌────────────────────┐
│1987 8 12 15 10 0 0│
└────────────────────┘
```

Validation

```
      0 ⎕DT ⎕TS (2020 13 1) 'J' 'DT' #
1 0 1 0 0

      ¯30 0 ⎕DT⊂32000 15 10 0
1
```

6.4.35 Edit Object {R}←{X}⎕ED Y

⎕ED invokes the Editor. Y is a simple character vector, a simple character matrix, or a vector of character vectors, containing the name(s) of objects to be edited.

The optional left argument X is a character scalar or character vector (where =/≠X Y) which specifies the type(s) of the corresponding (new) object(s) named in Y as:

∇	function/operator
→	simple character vector
∈	vector of character vectors
–	character matrix
⊛	Namespace script
○	Class script
∘	Interface
◇	array: use array notation

If `Y` names an existing object, the type specification for that name in `X` is ignored, unless `X` is `◇`.

If `X` is `◇`, `Y` must be undefined or an array. The Editor opens in array-notation mode; the resulting array can be of any type or structure.

If `⎕ED` is called from the Session, it opens Edit windows for the object(s) named in `Y` and returns a null result. The cursor is positioned in the first of the Edit windows opened by `⎕ED`, but may be moved to the Session or to any other window which is currently open. The effect is almost identical to using `)ED`.

If `⎕ED` is called from a defined function or operator, its behaviour is different. On asynchronous terminals, the Edit windows are automatically displayed in "full-screen" mode (ZOOMED). In all implementations, the user is restricted to those windows named in `Y`. The user may not skip to the Session even though the Session may be visible.

`⎕ED` terminates and returns a result ONLY when the user explicitly closes all the windows for the named objects. In this case the result contains the names of any objects which have been newly (re)fixed in the workspace as a result of the `⎕ED`, and has the same structure as `Y`.

Objects named in `Y` that cannot be edited are silently ignored. Objects qualified with a namespace path are (for example, `a.b.c.foo`) are silently ignored if the namespace does not exist.

Variants of Edit Object

The behaviour of ⎕ED may be modified using the variant operator ⍠ with the following options:

- `'ReadOnly'` - 0 or 1
- `'EditName'` - `'Default'`, `'Allow'` or `'Disallow'`.

If `ReadOnly` is set to 1, the edit window and all edit windows opened from it will be read-only. Note that setting `ReadOnly` to 0 will have no effect if the edit window is inherently read-only due to the nature of its content.

The `'EditName'` option determines whether or not the user may open another edit window by clicking a name, and its values are interpreted as follows:

EditName	⎕ED called from session	⎕ED called from function
`'Default'`	Allow	Disallow
`'Allow'`	Allow	Allow
`'Disallow'`	Disallow	Disallow

There is no Principal Option.

Examples

```
      A←3 11⍴'Hello World'
```

In the first example, ⎕ED will display the contents of `A` as an editable character array which the user may change. The user can double-click on *Hello* to open an edit window on an object named `Hello` (which will be a new function if `Hello` is currently undefined). Furthermore, the user can enter any arbitrary name and double-click to edit it. This may be undesirable in an application.

```
      ⎕ED A
```

In the second example, the Edit window will display the contents of `A` as a ReadOnly Character array. The user can still open a new edit by double-clicking *Hello* or *World* but nothing else.

```
      (⎕ED ⍠ 'ReadOnly' 1) 'A'
```

In the final example, the Edit window will display the contents of `A` as a ReadOnly Character array and the user cannot open a new edit window.

```
      (⎕ED ⍠('ReadOnly' 1)('EditName' 'Disallow'))'A'
```

6.4.36 Event Message R←⎕EM Y

`Y` must be a simple non-negative integer scalar or vector of event codes. If `Y` is a scalar, `R` is a simple character vector containing the associated event message. If `Y` is a vector, `R` is a vector of character vectors containing the corresponding event messages.

If `Y` refers to an undefined error code "`n`", the event message returned is "`ERROR NUMBER n`".

See *Programming: Apl Errors*.

EXAMPLE

```
      ⎕EM 11
DOMAIN ERROR
```

6.4.37 Event Number R←⎕EN

This simple integer scalar reports the identification number for the most recent event which occurred, caused by an APL action or by an interrupt or by the `⎕SIGNAL` system function. Its value in a clear workspace is `0`.

EXAMPLE

```
      ÷0
DOMAIN ERROR: Divide by zero
      ÷0
     ^
      ⎕EN
11
```

See *Programming: Apl Errors*.

> **Note**
>
> `⎕SIGNAL` can be used to reset the value of this system constant.

6.4.38 Exception R←⎕EXCEPTION

This is a system object that identifies the most recent *Exception* thrown by a Microsoft .NET object.

⎕EXCEPTION derives from the Microsoft .NET class System.Exception. Among its properties are the following, all of which are strings:

`Source`	The name of the .NET namespace in which the exception was generated
`StackTrace`	The calling stack
`Message`	The error message

```
      ⎕USING←'System'
      DT←⎕NEW DateTime (100000 0 0)
EXCEPTION: Year, Month, and Day parameters describe an un-representable
DateTime.
      DT←⎕NEW DateTime(100000 0 0)
     ^
      ⎕EN
90
```

```
      ⎕EXCEPTION.Message
Year, Month, and Day parameters describe an un-representable DateTime.

      ⎕EXCEPTION.Source
mscorlib

      ⎕EXCEPTION.StackTrace
   at System.DateTime.DateToTicks(Int32 year,
                                  Int32 month, Int32 day)

   at System.DateTime..ctor(Int32 year,
                            Int32 month, Int32 day)
```

Similarly to GUI objects, ⎕EXCEPTION is not preserved across a save/load cycle if the workspace is loaded in a different interpreter.

> **Note**
>
> `⎕SIGNAL` can be used to reset the value of this system constant.

6.4.39 Expunge Object `{R}←⎕EX Y`

`Y` must be a simple character scalar, vector or matrix, or a vector of character vectors containing a list of names. `R` is a simple Boolean vector with one element per name in `Y`.

Each name in `Y` is disassociated from its value if the active referent for the name is a defined function, operator, variable or namespace.

The value of an element of `R` is 1 if the corresponding name in `Y` is now available for use. This does not necessarily mean that the existing value was erased for that name. A value of 0 is returned for an ill-formed name or for a distinguished name in `Y`. The result is suppressed if not used or assigned.

Examples

```
      ⎕EX'VAR'
      +⎕EX'FOO' '⎕IO' 'X' '123'
1 0 1 0
```

If a named object is being executed the existing value will continue to be used until its execution is completed. However, the name becomes available immediately for other use.

Examples

```
      )SI
#.FOO[1]*

      ⎕VR'FOO'
     ∇ R←FOO
[1]    R←10
     ∇
      +⎕EX'FOO'
1
      )SI
#.FOO[1]*

     ∇FOO[⎕]
defn error

      FOO←1 2 3
      →⎕LC
10
      FOO
1 2 3
```

If a named object is an external variable, the external array is disassociated from the name:

```
      ⎕XT'F'
FILES/COSTS
      ⎕EX'F' ⋄ ⎕XT'F'
```

If the named object is a GUI object, the object and all its children are deleted and removed from the screen. The expression `⎕EX'.'` deletes all GUI objects owned by the current thread in the Root namespace **but not** those in sub-namespaces. In addition, if this expression is executed by thread 0, it resets all the properties of `'.'` to their default values. Furthermore, any unprocessed events in the event queue are discarded.

If the named object is a shared variable, the variable is retracted.

If the named object is the last remaining external function of an auxiliary process, the AP is terminated.

If the named object is the last reference into a dynamic link library, the DLL is freed.

If the named object is a *Programming: Triggers*, the name is disconnected from the Trigger Function and the Trigger Function will not be invoked when the Trigger is reassigned. The connection can be re-established by re-fixing the Trigger Function.

6.4.40 Export Object {R}←{X}⎕EXPORT Y

⎕EXPORT is used to set or query the export type of a defined function (or operator) referenced by the ⎕PATH mechanism.

Y is a character matrix or vector-of-vectors representing the names of functions and operators whose export type is to be set or queried.

X is an integer scalar or vector (one per name in the namelist) indicating the export type. X can currently be one of the values:

- 0 - not exported.
- 1 - exported (default).

A scalar or 1-element-vector type is replicated to conform with a multi-name list.

The result R is a vector that reports the export type of the functions and operators named in Y. When used dyadically to set export type, the result is shy.

When the path mechanism locates a referenced function (or operator) in the list of namespaces in the ⎕PATH system variable, it examines the function's export type:

0	This instance of the function is ignored and the search is resumed at the next namespace in the ⎕PATH list. Type-0 is typically used for functions residing in a utility namespace which are not themselves utilities, for example the private sub-function of a utility function.
1	This instance of the function is executed in the namespace in which it was found and the search terminated. The effect is exactly as if the function had been referenced by its full path name.

Warning: The left domain of ⎕EXPORT may be extended in future to include extra types 2, 3,... (for example, to change the behaviour of the function). This means that, while ⎕EXPORT returns a Boolean result in the first version, this may not be the case in the future. If you need a Boolean result, use 0≠ or an equivalent.

```
(0≠⎕EXPORT ⎕NL 3 4)⌿⎕NL 3 4  ⍝ a list of exported
                             ⍝ functions and ops.
```

⎕EXPORT does not support derived functions and will not be extended to support them; nor will it be extended to support other types of functions that may be developed in the future. ⎕EXPORT may therefore be considered an archaic feature.

6.4.41 File Append Component {R}←X ⎕FAPPEND Y

Access code 8

Y must be a simple integer scalar or a 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. Subject to a few restrictions, X may be any array.

The shy result R is the number of the component to which X is written, and is 1 greater than the previously highest component number in the file, or 1 if the file is new.

EXAMPLES

```
      (1000?1000) ⎕FAPPEND 1

      ⎕←(2 3⍴⍳6) 'Geoff' (⎕OR'FOO') ⎕FAPPEND 1
12

      ⎕←A B C ⎕FAPPEND¨1
13 14 15

Dump←{
    tie←⍺ ⎕FCREATE 0              ⍝ create file.
    (⎕FUNTIE tie){}⍵ ⎕FAPPEND tie ⍝ append and untie.
}
```

Information

Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.42 File System Available `R←⎕FAVAIL`

`⎕FAVAIL` returns the Boolean scalar value 1. This function is provided only for compatibility with other implementations of APL.

6.4.43 File Check and Repair `R←{X} ⎕FCHK Y`

`⎕FCHK` validates and repairs component files, and validates files associated with external variables, following an abnormal termination of the APL process or operating system.

`Y` must be a simple character scalar or vector which specifies the name of the file to be exclusively checked or repaired. For component files, the file must be named in accordance with the operating system's conventions, and may be a relative or absolute pathname. The file must exist and must not be tied. If no file extension is supplied, the set of extensions specified by the **CFEXT** parameter are tried one after another until the file is found or the set of extensions is exhausted. See *Installation/Configuration: Configuration Parameters*.

For files associated with external variables, any filename extension must be specified even if `⎕XT` would not require it. The file must exist and must not currently be associated with an external variable.

Options for `⎕FCHK` are specified using the Variant operator `⍠` or by the optional left argument `X`. The former is recommended but the older mechanism using the left argument is still supported.

In either case, the default behaviour is as follows:

1. If the file appears to have been cleanly untied previously, return `0`, that is, report that the file is good.
2. Otherwise, validate the file and return the appropriate result. If the file is corrupt, no attempt is made to repair it.

The result `R` is a vector of the numbers of missing or damaged components. `R` may include non-positive numbers of "pseudo components" that indicate damage to parts of the file other than in specific components:

`0`	ACCESS MATRIX.
`-1`	Free-block tree.
`-2`	Component index tree.

Other negative numbers represent damage to the file metadata; this set may be extended in the future.

Specifying options using Variant

Using Variant, the options are as follows:

- Task
- Repair
- Force

Rebuild causes the *file indices* to be discarded and rebuilt. *Repair* only takes place on files which have been checked and found to be damaged. It involves a rebuild, but that only takes place if it is needed. Note that Repair and Force only apply if Task is `'Scan'`.

Task

Scan	causes the file to be checked and optionally repaired (see `'Repair'` below)
`Rebuild`	causes the file to be unconditionally rebuilt

Repair (principle option)

0	do not repair
1	causes the file to be repaired if damage is found

Force

0	do not validate the file if it appears to have been properly closed
1	validate the file even if it appears to have been properly closed

Default values are highlighted thus{ .shaded } in the above tables.

EXAMPLES

To check a file and attempt to fix it if damage is found:

```
      (⎕FCHK ⍠ 1)'suspect.dcf'
```

To forcibly check a file and attempt to fix it if damage is found:

```
      (⎕FCHK ⍠ ('Repair' 1)('Force'1))'suspect.dcf'
```

Specifying options using a left argument

Using the optional left-argument, `X` must be a vector of zero or more character vectors from among `'force'`, `'repair'` and `'rebuild'`, which determine the detailed operation of the function. Note that these options are case-insensitive.

- If `X` contains `'force'`, `⎕FCHK` will validate the file even if it appears to have been cleanly untied.
- If `X` contains `'repair'`, `⎕FCHK` will repair the file, following validation, if it appears to be damaged. This option may be used in conjunction with `'force'`.
- If `X` contains `'rebuild'`, `⎕FCHK` will repair the file unconditionally.
- If `X` contains `'force'`, `⎕FCHK` will validate the file even if it appears to have been cleanly untied.

- If X contains 'repair', ⎕FCHK will repair the file, following validation, if it appears to be damaged. This option may be used in conjunction with 'force'.
- If X contains 'rebuild', ⎕FCHK will repair the file unconditionally.

Following a *check* of the file, a non-null result indicates that the file is damaged.

Following a *repair* of the file, the result indicates those components that could not be recovered. Un-recovered components will give a FILE COMPONENT DAMAGED error if read but may be replaced without error.

Repair can recover only check-summed components from the file, that is, only those components that were written with the checksum option enabled (see *Section 6.4.56*).

Following an operating system crash, repair may result in one or more individual components being rolled back to a previous version or not recovered at all, unless Journaling levels 2 or 3 were also set when these components were written.

6.4.44 File Copy R←X ⎕FCOPY Y

Access Code: 4609

Y must be a simple integer scalar or 1 or 2-element vector containing the file tie number and optional passnumber. The file need not be tied exclusively.

X is a character vector containing the name of a new file to be copied to. If no file extension is supplied, the first extension specified by the **CFEXT** parameter will be added. See *Installation/Configuration: Configuration Parameters*.

⎕FCOPY creates a copy of the tied file specified by Y, named X.

The new file X will have the same component level information, including the user number and update time as the original. The operating system file creation, modification and access times will be set to the time at which the copy occurred.

Unless otherwise specified (see File Properties below) the new file X will have the same file properties as the original, except that it will be a large-span file regardless of the span of the original.

The result R is the file tie number associated with the new file X.

Note that the Access Code is 4609, which is the sum of the Access Codes for ⎕FREAD (1), ⎕FRDCI (512) and ⎕FRDAC (4096).

Note also that although the file need not be tied exclusively, the ⎕FCOPY function will not yield the file to other APL processes while it is running, and it may take some considerable time to run in the case of a large component file.

EXAMPLE

```
      told←'oldfile32'⎕FTIE 0
      'S' ⎕FPROPS told
32
      tnew←'newfile64' ⎕FCOPY told

      'S' ⎕FPROPS tnew
64
```

If `X` specifies the name of an existing file, the operation fails with a `FILE NAME ERROR`.

> **Note**
>
> This operation is atomic. If an error occurs during the copy operation (such as disk full) or if a strong interrupt is issued, the copy will be aborted and the new file `X` will not be created.

File Properties

⎕FCOPY allows you to specify properties for the new file via the variant operator ⌸ used with the following options:

- `'J'` - journaling level; a numeric value.
- `'C'` - checksum level; 0 or 1.
- `'Z'` - compression; 0 or 1.
- `'U'` - Unicode; 0 or 1
- `'S'` - File Size (span); 64

The Principal Option is as follows:

- 0 - sets `('J' 0) ('C' 0)`
- 1 - sets `('J' 1) ('C' 1)`
- 2 - sets `('J' 2) ('C' 1)`
- 3 - sets `('J' 3) ('C' 1)`

Examples

```
      newfid←'newfile' (⎕FCOPY ⍠3) 1

      'SEUJCZ' ⎕FPROPS newfid
64 0 1 3 1 0
```

Alternatively:

```
      JFCOPY←⎕FCOPY ⍠ 3
```

will name a variant of `⎕FCREATE` which will create component file with level 3 journaling, and checksum enabled. Then:

```
      newfid←'newfile' JFCOPY 1
```

> **Note**
>
> Setting `('U' 0)` (no Unicode support) is discouraged as it may cause the copy to fail with a `TRANSLATION ERROR`. Similarly using a Classic interpreter to `⎕FCOPY` files may result in `TRANSLATION ERROR`s.

> **Information**
>
> Small-span (32-bit) component files are currently read-only; Dyalog Ltd recommends using `⎕FCOPY` to convert any such files to large-span (64-bit). This ability is scheduled for removal in a future release. For information on how to identify calls to small-span component files in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.45 File Create `{R}←X ⎕FCREATE Y`

`Y` must be a simple integer scalar or a 1 or 2 element vector. The first element is the *file tie number*. The second element, if specified, must be 64[17].

The *file tie number* must not be the tie number associated with another tied file.

`X` must be either:

1. a simple character scalar or vector which specifies the name of the file to be created. If no file extension is supplied, the first extension specified by the **CFEXT** parameter will be added. See *Installation/Configuration: Configuration Parameters*.
2. a vector of length 1 or 2 whose items are:- a simple character scalar or vector as above.
3. an integer scalar specifying the file size limit in bytes.

The newly created file is tied for exclusive use.

The shy result of `⎕FCREATE` is the tie number of the new file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUM     ⍝ With next available number,
      file ⎕FCREATE tie   ⍝ ... create file.
```

to:

```
      tie←file ⎕FCREATE 0 ⍝ Create with first available..
```

Examples

```
      '..\BUDGET\SALES'    ⎕FCREATE 2    ⍝ Windows
      '../budget/SALES.85' ⎕FCREATE 2    ⍝ UNIX

      'COSTS' 200000 ⎕FCREATE 4         ⍝ max size 200000
```

17 This element sets the span of the file which in earlier Versions of Dyalog APL could be 32 or 64. Small-span (32-bit) component files may no longer be created and this element is retained only for backwards compatibility of code.

File Properties

⎕FCREATE allows you to specify properties for the newly created file via the variant operator ⍠ used with the following options:

- 'J' - journaling level; a numeric value
- 'C' - checksum level; 0 or 1
- 'Z' - compression; 0 or 1
- 'U' - Unicode; 0 or 1
- 'S' - File Size (span); 64

The Principal Option is as follows:

- 0 - sets ('J' 0) ('C' 0)
- 1 - sets ('J' 1) ('C' 1)
- 2 - sets ('J' 2) ('C' 1)
- 3 - sets ('J' 3) ('C' 1)

See also: *Section 6.4.56*.

EXAMPLES

```
      'newfile' (⎕FCREATE⍠3) 0
1
      'SEUJCZ' ⎕FPROPS 1
64 0 1 3 1 0
```

Alternatively:

```
      JFCREATE←⎕FCREATE ⍠ 3
```

will name a variant of ⎕FCREATE which will create component file with level 3 journaling, and checksum enabled. Then:

```
      'newfile'JFCREATE 0
1
```

> **Information**
>
> Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0 it will not be possible to create files with this combination of properties. For information on how to identify code that creates component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.46 File Drop Component {R}←⎕FDROP Y

Access code 32

`Y` must be a simple integer vector of length 2 or 3 whose elements are:

`[1]`	a file tie number
`[2]`	a number specifying the position and number of components to be dropped. A positive value indicates that components are to be removed from the beginning of the file; a negative value indicates that components are to be removed from the end of the file
`[3]`	an optional passnumber which if omitted is assumed to be zero

The shy result of a `⎕FDROP` is a vector of the numbers of the dropped components. This is analogous to `⎕FAPPEND` in that the result is potentially useful for updating some sort of dictionary:

```
      cnos,←vec ⎕FAPPEND¨tie ⍝ Append index to dictionary

      cnos~←⎕FDROP tie,-⍴vec ⍝ Remove index from dict.
```

Note that the result vector, though potentially large, is generated only on request.

Examples

```
      ⎕FSIZE 1
1 21 5436 4294967295

      ⎕FDROP 1 3 ⋄ ⎕FSIZE 1
4 21 5436 4294967295

      ⎕FDROP 1 ¯2 ⋄ ⎕FSIZE 1
4 19 5436 4294967295
```

6.4.47 File Erase {R}←X ⎕FERASE Y

Access code 4

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. X must be a character scalar or vector containing the name of the file associated with the tie number Y. This name must be identical with the name used to tie the file (except that the directory delimiters / and \ are treated as being the same) and the file must be exclusively tied. The file named in X is erased and untied.

The shy result of ⎕FERASE is the tie number of the erased file.

Examples

```
      'SALES'⎕FERASE 'SALES' ⎕FTIE 0

      './temp' ⎕FCREATE 1
      'temp' ⎕FERASE 1
FILE NAME ERROR
      'temp'⎕FERASE 1

      ⎕←'.\temp'⎕FERASE 1 ⍝ Works with / or \
1
```

6.4.48 File History `R←⎕FHIST Y`

Access code 16384

`Y` must be a simple integer vector of length 1 or 2 containing the file tie number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

The result is a numeric matrix with shape (5 2) whose rows represent the most recent occurrence of the following events.

1. File creation (see note)
2. Undefined, currently `(0 0)`
3. Last update of the access matrix
4. Undefined, currently `(0 0)`
5. Last update performed by `⎕FAPPEND`, `⎕FCREATE`, `⎕FDROP` or `⎕FREPLACE`

For each event, the first column contain the user number and the second a timestamp. Like the timestamp reported by `⎕FRDCI` this is measured in 60ths of a second since 1st January 1970 (UTC).

Note: `⎕FHIST` collects information only if journaling and/or checksum is in operation. If neither is in use, the collection of data for `⎕FHIST` is disabled and its result is entirely 0. If a file has both journaling and checksum disabled, and then either is enabled, the collection of data for `⎕FHIST` is enabled too. In this case, the information in row 1 of `⎕FHIST` relates to the most recent enabling `⎕FPROPS` operation rather than the original `⎕FCREATE`.

In the examples that follow, the `FHist` function is used below to format the result of `⎕FHIST`.

```
    ∇ r←FHist tn;cols;rows;fhist;fmt;ToTS;I2D
[1]   rows←'Created' 'Undefined' 'Last ⎕FSTAC'
[2]   rows,←'Undefined' 'Last Updated'
[3]   cols←'User' 'TimeStamp'
[4]   fmt←'ZI4,2(⊂-⊃,ZI2),⊂ ⊃,ZI2,2(⊂:⊃,ZI2)'
[5]   I2D←{⍵+2 ⎕NQ'.' 'IDNToDate'⍵}
[6]   ToTS←{d t←1 1 0 0 0⊂⍉⌊0 24 60 60 60⊤⍵
[7]       ↓fmt ⎕FMT(0 ¯1↓↑I2D¨25568+,d),0 ¯1↓t}
[8]   fhist←⎕FHIST tn
[9]   fhist[;2]←ToTS fhist[;2]
[10]  fhist[;1]←⍕¨fhist[;1]
[11]  r←((⊂''),rows),cols⍪fhist
    ∇
```

Examples

```
    'c:\temp'⎕FCREATE 1 ⋄ FHist 1
              User  TimeStamp
Created       0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last ⎕FSTAC   0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last Updated  0     2012-01-14 12:29:53

     (⍳10)⎕FAPPEND 1  ⋄ FHist 1
              User  TimeStamp
Created       0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last ⎕FSTAC   0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last Updated  0     2012-01-14 12:29:55

     ⎕FUNTIE 1

     'c:\temp'⎕FCREATE 1 ⋄ FHist 1
              User  TimeStamp
Created       0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last ⎕FSTAC   0     2012-01-14 12:29:53
Undefined     0     1970-01-01 00:00:00
Last Updated  0     2012-01-14 12:29:55
```

6.4.49 File Hold {R}←{X} ⎕FHOLD Y

Access code 2048

This function holds component file(s) and/or external variable(s). It is used to synchronise access to resources shared between multiple cooperating Dyalog processes. It is not intended to synchronise access between Dyalog threads; for this purpose you should use `:Hold`.

For a multi-threaded and multi-process application, a single `⎕FHOLD` is used to synchronise inter-process access, while `:Hold` is used in multiple threads to synchronise access between threads in the same process. See also *Programming: Hold*.

If applied to component files, then `Y` is an integer scalar, vector, or one-row matrix of file tie numbers, or a two-row matrix whose first row contains file tie numbers and whose second row contains passnumbers.

If applied to external variables, then `Y` is a simple scalar character, a character vector, a non-simple scalar character vector, or a vector of character vectors that specifies one or more names of external variable(s) (NOT the file names associated with those variables). Note that when `Y` is simple, each character in `Y` is interpreted as a variable name. If applied to component files **and** external variables, `Y` is a vector whose elements are either integer scalars representing tie numbers, or character scalars or vectors containing names of external variables.

The effect is as follows:

1. **All** of the user's preceding holds (if any) are released, whether referenced in `Y` or not.
2. Execution is suspended until the designated files are free of holds by any other task.
3. When all the designated files are free, execution proceeds. Until the hold is released, other tasks using `⎕FHOLD` on any of the designated files will wait.

The optional left argument `X` is a non-negative integer that specifies a time-out in milliseconds. If step 2 (see above) does not complete before the time-out value specified by `X`, `⎕FHOLD` times out and signals a `TIMEOUT` error (1006) after releasing any holds that have succeeded.

A time-out value of 0 indicates that the `⎕FHOLD` should time out at once without waiting if it cannot immediately acquire all holds. If `X` is `¯1`, `⎕FHOLD` behaves as the monadic case, and does not time out.

If `Y` is empty, all of the user's preceding holds (if any) are released, and execution continues.

A hold is released by any of the following:

- Another `⎕FHOLD`
- Untying or retying all the designated files. If some but not all are untied or retied, they become free for another task but the hold persists for those that remain tied.
- Termination of APL.
- Any untrapped error or interrupt.
- A return to immediate execution mode.

Note that a hold is not released by a request for input through `⎕` or `⍞`.

`⎕FHOLD` is generally useful only when called from a defined function, as holds set in immediate execution (desk calculator) mode are released immediately.

If `Y` is a matrix, the shy result `R` is `Y[1;]`. Otherwise, the shy result `R` is `Y`.

Examples

```
      ⎕FHOLD 1

      ⎕FHOLD ⍬

      ⎕FHOLD ⊂'XTVAR'

      ⎕FHOLD 1 2,[0.5]0 16385

      ⎕FHOLD 1 'XTVAR'

      3000 ⎕FHOLD 1
TIMEOUT
      3000 ⎕FHOLD 1
           ^
```

6.4.50 Fix Script `{R}←{X}⎕FIX Y`

`⎕FIX` establishes Namespaces, Classes, Interfaces and functions from the script specified by `Y` in the workspace.

In this section, the term *namespace* covers scripted Namespaces, Classes and Interfaces.

`Y` may be a simple character vector, or a vector of character vectors or character scalars. The value of `X` determines what `Y` may contain.

If `Y` is a simple character vector, it must start with `file://`, followed by the name of a file which must exist. The contents of the file must follow the same rules that apply to `Y` when `Y` is a vector of character vectors or scalars. The file name can be relative or absolute; when considering cross-platform portability, using "/" as the directory delimiter is recommended, although "\" is also valid under Windows.

If specified, `X` must be a numeric scalar. It may currently take the value `0`, `1` or `2`. If not specified, the value is assumed to be `1`.

If `X` is `0`, `Y` must specify a single valid *namespace* which may or may not be named, or a file containing such a definition. If so, the shy result `R` contains a reference to the *namespace*. Even if the *namespace* is named, it is not established *per se*, although it will exist for as long as at least one reference to it exists.

If `X` is `1`, `Y` must specify a single valid *namespace* which may or may not be named, or a file containing such a definition. If so, the shy result `R` contains a reference to the *namespace*. If `Y` contains the definition of a named *namespace*, the *namespace* is established in the workspace.

If `X` is `2`, `Y` is either a character vector containing the name of a script file, or a vector of character vectors that represents a script.

`Y` may specify a series of **named** *namespaces* or function definitions, or a combination of functions and namespaces.

- If the script contains more than one item, tradfn definitions must be delimited by ∇symbols.
- Derived and assigned functions may be specified only within namespaces.

In this case, the shy result `R` is a vector of character vectors, containing the names of all of the objects that have been established in the workspace; the order of the names in `R` is not defined. Currently `2 ⎕FIX` is not certain to be an atomic operation, although this might change in future versions.

Example 1

In the first example, the Class specified by `Y` is *named* (`MyClass`) but the result of `⎕FIX` is discarded. The end-result is that `MyClass` is established in the workspace as a Class.

```
      ⍞←⎕FIX ':Class MyClass' ':EndClass'
#.MyClass
```

Example 2

In the second example, the Class specified by `Y` is *named* (`MyClass`) and the result of `⎕FIX` is assigned to a different name (`MYREF`). The end-result is that a Class named `MyClass` is established in the workspace, and `MYREF` is a reference to it.

```
      MYREF←⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MyClass MYREF
      ⎕NC'MyClass' 'MYREF'
9.4 9.4
      MYREF
#.MyClass
      MYREF≡MyClass
1
```

Example 3

In the third example, the left-argument of `0` causes the named Class `MyClass` to be visible only via the reference to it (`MYREF`). It is there, but hidden.

```
      MYREF←0 ⎕FIX ':Class MyClass' ':EndClass'
      )CLASSES
MYREF
      MYREF
#.MyClass
```

Example 4

The fourth example illustrates the use of un-named Classes.

```
      src←':Class' '∇Make n'
      src,←'Access Public' 'Implements Constructor'
      src,←'⎕DF n' '∇' ':EndClass'
      MYREF←⎕FIX src
      )CLASSES
MYREF
      MYINST←⎕NEW MYREF'Pete'
      MYINST
Pete
```

Example 5

In the final example, the left argument of `2` allows a script containing multiple objects to be fixed:

```
      src←':Namespace andys' '∇foo' '2' '∇'
      src,←':EndNamespace' 'dfn←{α ω}' '∇r←tfn'
      src,←'r←33' '∇' ':Class c1' '∇goo' '1'
      src,←'∇' ':EndClass'
      ≠⎕←2⎕FIX src
 c1  tfn  dfn  andys
4
```

Restrictions

`⎕FIX` is unable to fix a namespace from `Y` when `Y` specifies a multi-line dfn which is preceded by a `⋄` (diamond separator).

```
      ⎕FIX':Namespace iaK' 'a←1 ⋄ adfn←{' 'ω' ' }' ':EndNamespace'
DOMAIN ERROR: There were errors processing the script
      ⎕FIX':Namespace iaK' 'a←1 ⋄ adfn←{' 'ω' ' }' ':EndNamespace'
      ^
```

> **Legacy**
>
> Before Dyalog v20.0, it was possible to define dfns with unmatched parentheses and brackets. These are now rejected. TradFns will continue to fix as before, but subtle differences in how the code behaves might not be backwards-compatible and could have unexpected results.

Variant Options

`⎕FIX` may be applied using the Variant operator with the options Quiet, FixWithErrors, AllowLateBinding and InjectReferences. These options apply only to namespaces and classes specified by the script. There is no principal option.

Quiet Option

0	If the script contains errors, these are displayed in the Status Window.
1	If the script contains errors, the errors are not shown in the Status Window.

FixWithErrors Option

0	If the script contains errors, `⎕FIX` fails with `DOMAIN ERROR`.
1	`⎕FIX` fixes all the namespaces and classes in the script regardless of any errors they may contain.
2	If the script contains errors, `⎕FIX` displays a message box prompting the user to choose whether or not to fix all the offending namespaces and classes in the script.

AllowLateBinding Option

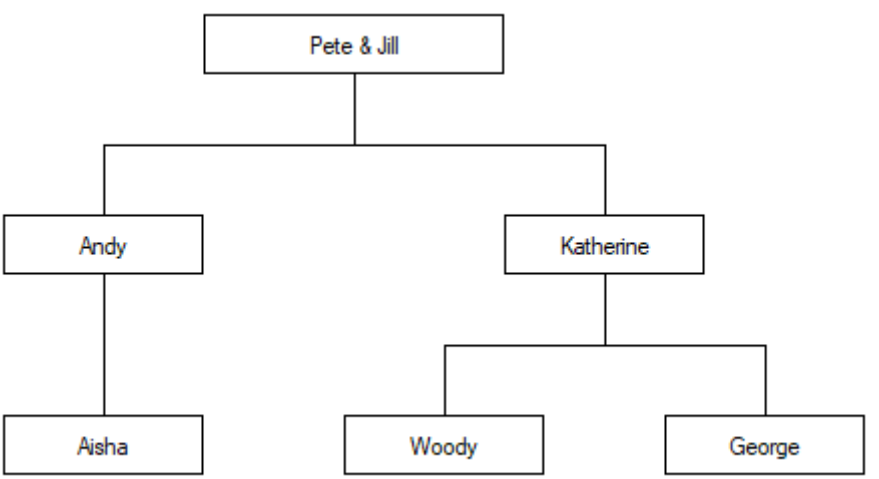
0	FIX will only fix a Class whose Base class (if specified) is defined in the script or is present in the workspace.
1	FIX will fixes a Class whose Base class is neither defined in the script nor present in the workspace.

InjectReferences Option

'All'	In order to implement lexical scope, FIX will insert internal references into all objects in the script.
'InClasses'	In order to implement lexical scope, FIX will insert internal references ONLY into Classes and sub-classes in the script, but not into namespaces.
'None'	No internal references are inserted and lexical scope does not apply.

See *Earlier Release Notes: Lexical Scope In Scripts*.

The following examples illustrate how different values of the InjectReferences option affect the scope of objects in scripts. The examples are based on the following family tree:



Two scripts are defined to map this tree onto a structure of Classes and Namespaces. In this scheme, female family members are represented by Classes and male family members by Namespaces.

So the scripted tree for `Pete` has a parent Namespace:

```
:Namespace Pete
    :Namespace Andy
        :Class Aisha
        :Access Public
        :Endclass
    :EndNamespace

    :Class Katherine
    :Access Public
        :Namespace Woody
        :EndNamespace
        :Namespace George
        :EndNamespace
    :EndClass
:EndNamespace
```

While the scripted tree for `Jill` has a parent Class:

```
:Class Jill
:Access Public
    :Namespace Andy
        :Class Aisha
        :Access Public
        :Endclass
    :EndNamespace

    :Class Katherine
    :Access Public
        :Namespace Woody
        :EndNamespace
        :Namespace George
        :EndNamespace
    :EndClass
:EndClass
```

Using the `Pete` Namespace, after executing the expression:

```
      2(]FIX:'InjectReferences' 'All')]SRC Pete
```

- Code in `Pete` may refer to `Aisha` , `Andy` , `George` , `Katherine`, and `Woody`
- Code in `Andy` may refer to `Aisha` and `Katherine`
- … and so forth.

But after executing:

```
      2(⎕FIX⍠'InjectReferences' 'InClasses')⎕SRC Pete
```

- Code in `Pete` may refer only to `Andy` and `Katherine`
- Code in `Andy` may refer only to `Aisha`
- … and so forth.

The following tables show which objects in Namespace `Pete` can *see* (that is, refer to) which other objects representing members of the family, in each case; `All`, `InClasses` and `None`.

'All'	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✔	✔	✔	✔	✔
Andy			✔	✔		
Aisha	✔	✔	✔			
Katherine	✔	✔		✔	✔	✔
Woody						✔
George					✔	

'InClasses'	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✔		✔		
Andy			✔			
Aisha	✔	✔	✔			
Katherine	✔	✔		✔	✔	✔
Woody						
George						

'None'	Pete	Andy	Aisha	Katherine	Woody	George
Pete		✔		✔		
Andy			✔			
Aisha						
Katherine						
Woody						
George						

Whilst the next set of tables show the same for Class `Jill`.

'All'	Jill	Andy	Aisha	Katherine	Woody	George
Jill	✔	✔	✔	✔	✔	✔
Andy			✔	✔		
Aisha	✔	✔	✔			
Katherine	✔	✔		✔	✔	✔
Woody						✔
George					✔	

'InClasses'	Jill	Andy	Aisha	Katherine	Woody	George
Jill	✔	✔	✔	✔	✔	✔
Andy			✔			
Aisha	✔	✔	✔			
Katherine	✔	✔		✔	✔	✔
Woody						
George						

'None'	Jill	Andy	Aisha	Katherine	Woody	George
Jill						
Andy			✔			
Aisha						
Katherine						
Woody						
George						

6.4.51 Component File Library R←⎕FLIB Y

Y must be a simple character scalar or vector which specifies the name of the directory whose APL component files are to be listed. If Y is empty, the current working directory is assumed.

The result R is a character matrix containing the names of the component files in the directory with one row per file. The number of columns is given by the longest file name. Each file name is prefixed by Y followed by a directory delimiter character. The ordering of the rows is not defined.

If there are no APL component files accessible to the user in the directory in question, the result is an empty character matrix with 0 rows and 0 columns.

EXAMPLES

```
      ⎕FLIB ''
SALESFILE
COSTS

      ⎕FLIB '.'
./SALESFILE
./COSTS

      ⎕FLIB '../budget'
../budget/SALES.85
../budget/COSTS.85
```

6.4.52 Format (Monadic) R←⎕FMT Y

Y may be any array. R is a simple character matrix which appears the same as the default display of Y. If Y contains control characters from ⎕TC, they will be resolved.

EXAMPLES

```
      A←⎕FMT 'n' ,⎕TC[1],'o'

      ⍴A
1 1
      A
n

      A←⎕VR 'FOO'

      A
     ∇ R←FOO
[1]    R←10
     ∇

      ⍴A
31
      B←⎕FMT A

      B
     ∇ R←FOO
[1]    R←10
     ∇

      ⍴B
3 12
```

6.4.53 Format (Dyadic) R←X ⎕FMT Y

Y must be a simple array of rank not exceeding two, or a non-simple scalar or vector whose items are simple arrays of rank not exceeding two. The simple arrays in Y must be homogeneous, either character or numeric. All numeric values in Y must be simple; if Y contains any complex numbers, dyadic ⎕FMT will generate a DOMAIN ERROR. X must be a simple character vector. R is a simple character matrix.

X is a format specification that defines how columns of the simple arrays in Y are to appear. A simple scalar in Y is treated as a one-element matrix. A simple vector in Y is

treated as a one-column matrix. Each column of the simple arrays in `Y` is formatted in left-to-right order according to the format specification in `X` taken in left-to-right order and used cyclically if necessary.

`R` has the same number of rows as the longest column (or implied column) in `Y`, and the number of columns is determined from the format specification.

The **format specification** consists of a series of control phrases, with adjacent phrases separated by a single comma, selected from the following:

`rAw`	Alphanumeric format
`rEw.s`	Scaled format
`rqFw.d`	Decimal format
`rqG⍞pattern⍞`	Pattern
`rqIw`	Integer format
`Tn`	Absolute tabulation
`Xn`	Relative tabulation
`⍞t⍞`	Text insertion

(Alternative surrounding pairs for Pattern or Text insertion are `< >`, `⊂ ⊃`, `[] []` or `¨ ¨`.)

where

`r`	is an optional repetition factor indicating that the format phrase is to be applied to r columns of `Y`
`q`	is an optional usage of qualifiers or affixtures from those described below.
`w`	is an integer value specifying the total field width per column of `Y` , including any affixtures.
`s`	is an integer value specifying the number of significant digits in Scaled format; `s` must be less than `w-1`
`d`	is an integer value specifying the number of places of decimal in Decimal format; `d` must be less than `w` .
`n`	is an integer value specifying a tab position relative to the notional left margin (for `T` -format) or relative to the last formatted position (for `X` -format) at which to begin the next format.
`t`	is any arbitrary text excluding the surrounding character pair. Double quotes imply a single quote in the result.
`pattern`	see following section **G format**

Qualifiers q are as follows

B	leaves the field blank if the result would otherwise be zero.
C	inserts commas between triads of digits starting from the rightmost digit of the integer part of the result.
Km	scales numeric values by 1Em where m is an integer; negation may be indicated by ¯ or - preceding the number.
L	left justifies the result in the field width.
Ov▯t▯	replaces specific numeric value v with the text t .
S▯p▯	substitutes standard characters. p is a string of pairs of symbols enclosed between any of the Text Insertion delimiters; the first of each pair is the standard symbol, and the second is the symbol to be substituted. Standard symbols are: * overflow fill character . decimal point , triad separator for C qualifier 0 fill character for Z qualifier _ loss of precision character ¯ high minus symbol
Z	fills unused leading positions in the result with zeros (and commas if C is also specified).
9	digit selector

Affixtures are as follows

M▯t▯	prefixes negative results with the text t instead of the negative sign.
N▯t▯	post-fixes negative results with the text t
P▯t▯	prefixes positive or zero results with the text t .
Q▯t▯	post-fixes positive or zero results with the text t .
R▯t▯	presets the field with the text t which is repeated as necessary to fill the field. The text will be replaced in parts of the field filled by the result, including the effects of other qualifiers and affixtures except the B qualifier

The surrounding affixture delimiters may be replaced by the alternative pairs described for Text Insertion.

Examples

A vector is treated as a column:

```
      'I5' ⎕FMT 10 20 30
   10
   20
   30
```

The format specification is used cyclically to format the columns of the right argument:

```
      'I3,F5.2' ⎕FMT 2 4⍴⍳8
  1 2.00  3 4.00
  5 6.00  7 8.00
```

The columns of the separate arrays in the items of a non-simple right argument are formatted in order. Rows in a formatted column beyond the length of the column are left blank:

```
      '2I4,F7.1' ⎕FMT (⍳4)(2 2⍴ 0.1×⍳4)
   1   0    0.2
   1   0    0.4
   3
   4
```

Characters are right justified within the specified field width, unless the `L` qualifier is specified:

```
      'A2' ⎕FMT 1 6⍴'SPACED'
 S P A C E D
```

If the result is too wide to fit within the specified width, the field is filled with asterisks:

```
      'F5.2' ⎕FMT 0.1×5 1000 ¯100
  0.50
*****
*****
```

Relative tabulation (`X`-format) identifies the starting position for the next format phrase relative to the finishing position for the previous format, or the notional left margin if none. Negative values are permitted providing that the starting position is not brought back beyond the left margin. Blanks are inserted in the result, if necessary:

```
      'I2,X3,3A1' ⎕FMT (⍳3)(2 3⍴'TOPCAT')
1   TOP
2   CAT
3
```

Absolute tabulation (T-format) specifies the starting position for the next format relative to the notional left margin. If position 0 is specified, the next format starts at the next free position as viewed so far. Blanks are inserted into the result as required. Over-written columns in the result contain the most recently formatted array columns taken in left-to-right order:

```
      X←'6I1,T5,A1,T1,3A1,T7,F5.1'

      X ⎕FMT (1 6⍴⍳6)('*')(1 3⍴'ABC')(22.2)
ABC4*6 22.2
```

If the number of specified significant digits exceeds the internal precision, low order digits are replaced by the symbol _:

```
      'F20.1' ⎕FMT 1E18÷3
3333333333333333__._
```

The Text Insertion format phrase inserts the given text repeatedly in all rows of the result:

```
      MEN←3 5⍴'FRED BILL JAMES'
      WOMEN←2 5⍴'MARY JUNE '

      '5A1,<|>' ⎕FMT MEN WOMEN
FRED |MARY |
BILL |JUNE |
JAMES|     |
```

The last example also illustrates that a Text Insertion phrase is used even though the data is exhausted.

The following example illustrates effects of the various qualifiers:

```
      X←'F5.1,BF6.1,X1,ZF5.1,X1,LF5.1,K3CS<.,,.>F10.1'

      X ⎕FMT ⍉5 3⍴¯1.5 0 25
 ¯1.5  ¯1.5 ¯01.5 ¯1.5   ¯1.500,0
  0.0       000.0 0.0         0,0
 25.0  25.0 025.0 25.0   25.000,0
```

Affixtures allow text to be included within a field. The field width is not extended by the inclusion of affixtures. `N` and `Q` affixtures shift the result to the left by the number of characters in the text specification. Affixtures may be used to enclose negative results in parentheses in accordance with common accounting practice:

```
      'M<(>N<)>Q< >F9.2' ⎕FMT 150.3 ¯50.25 0 1114.9
 150.30
 (50.25)
   0.00
1114.90
```

One or more format phrases may be surrounded by parentheses and preceded by an optional repetition factor. The format phrases within parentheses will be re-used the given number of times before the next format phrase is used. A Text Insertion phrase will not be re-used if the last data format phrase is preceded by a closing parenthesis:

```
      'I2,2(</>,ZI2)' ⎕FMT 1 3⍴⌽100⊥3↑⎕TS
20/07/89
```

G Format

Only the `B`, `K`, `S` and `O` qualifiers are valid with the `G` option

`⍞pattern⍞` is an arbitrary string of characters, excluding the delimiter characters. Characters '9' and '`Z`' (unless altered with the `S` qualifier) are special and are known as **digit selectors**.

The result of a `G` format will have length equal to the length of the pattern.

The data is rounded to the nearest integer (after possible scaling). Each digit of the rounded data replaces one digit selector in the result. If there are fewer data digits than digit selectors, the data digits are padded with leading zeros. If there are more data digits than digit selectors, the result will be filled with asterisks.

A '9' digit selector causes a data digit to be copied to the result.

A 'Z' digit selector causes a non-zero data digit to be copied to the result. A zero data digit is copied if and only if digits appear on each side of it. Otherwise a blank appears. Similarly text between digit selectors appears only if digits appear on each side of the text. Text appearing before the first digit selector or after the last will always appear in the result.

Examples

```
      'G⊂99/99/99⊃'⎕FMT 0 100 100 ⍳8 7 89
08/07/89

      'G⊂ZZ/ZZ/ZZ⊃'⎕FMT 80789 + 0 1
 8/07/89
 8/07/9

      'G⊂Andy ZZ Pauline ZZ⊃' ⎕FMT 2721.499 2699.5
Andy 27 Pauline 21
Andy 27

      ⍴⎕←'K2G⊂DM Z.ZZZ.ZZ9,99⊃' ⎕FMT 1234567.89 1234.56
DM 1.234.567,89
DM     1.234,56
2 15
```

An error will be reported if:

- Numeric data is matched against an `A` control phrase.
- Character data is matched against other than an `A` control phrase.
- The format specification is ill-formed.
- For an F control phrase, `d>w-2`
- For an E control phrase, `s>w-2`

O Format Qualifier

The O format qualifier replaces a specific numeric value with a text string and may be used in conjunction with the E, F, I and G format phrases.

An O-qualifier consists of the letter "O" followed by the optional numeric value which is to be substituted (if omitted, the default is 0) and then the text string within pairs of symbols such as "<>". For example:

`O - qualifier`	Description
`O<nil>`	Replaces the value 0 with the text "nil"
`O42<N/A>`	Replaces the value 42 with the text "N/A"
`O0.001<1/1000>`	Replaces the value 0.001 with the text "1/1000"

The replacement text is inserted into the field in place of the numeric value. The text is normally right-aligned in the field, but will be left-aligned if the L qualifier is also specified.

It is permitted to specify more than one O-qualifier within a single phrase.

The O-qualifier is ⎕CT sensitive.

EXAMPLES

```
      'O<NIL>F7.2'⎕FMT 12.3 0 42.5
 12.30
   NIL
 42.50

      'O<NIL>LF7.2'⎕FMT 12.3 0 42.5
12.30
NIL
42.50

      'O<NIL>O42<N/A>I6'⎕FMT 12 0 42 13
   12
  NIL
  N/A
   13

      'O99<replace>F20.2'⎕FMT 99 100 101
            replace
             100.00
             101.00
```

⎕CT and ⎕DCT are implicit arguments of ⎕FMT with the O format qualifier.

6.4.54 File Names R←⎕FNAMES

The result is a character matrix containing the names of all tied files, with one file name per row. The number of columns is that required by the longest file name.

A file name is returned precisely as it was specified when the file was tied, except that the directory delimiter \ is replaced by /. If no files are tied, the result is a character matrix with 0 rows and 0 columns. The rows of the result are in the order in which the files were tied.

EXAMPLES

```
      '/usr/pete/SALESFILE' ⎕FSTIE 16

      '..\budget\COSTFILE' ⎕FSTIE 2

      'PROFIT' ⎕FCREATE 5

       ⎕FNAMES
/usr/pete/SALESFILE
../budget/COSTFILE
PROFIT

      ⍴⎕FNAMES
3 19
      ⎕FNUMS,⎕FNAMES
16 /usr/pete/SALESFILE
 2 ../budget/COSTFILE
 5 PROFIT
```

6.4.55 File Numbers R←⎕FNUMS

The result is an integer vector of the *file tie number* of all tied files. If no files are tied, the result is empty. The elements of the result are in the order in which the files were tied.

EXAMPLES

```
      '/home/pete/SALESFILE' ⎕FSTIE 16

      '../budget/COSTFILE' ⎕FSTIE 2

      'PROFIT' ⎕FCREATE 5

      ⎕FNUMS
16 2 5

      ⎕FNUMS,⎕FNAMES
16 /home/pete/SALESFILE
 2 ../budget/COSTFILE
 5 PROFIT

      ⎕FUNTIE ⎕FNUMS
      ⍴⎕FNUMS
0
```

6.4.56 File Properties R←X ⎕FPROPS Y

Access Code 1 (to read) or 8192 (to change properties)

⎕FPROPS reports and sets the properties of a component file.

Y must be a simple integer scalar or 1 or 2-element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted, it is assumed to be 0.

X must be a simple character scalar or vector containing one or more valid Identifiers listed in the table below, or a 2-element nested vector which specifies an Identifier and a (new) value for that property. To set new values for more than one property, X must be is a vector of 2-element vectors, each of which contains an Identifier and a (new) value for that property.

If the left argument is a simple character array, the result R contains the current values for the properties identified by X. If the left argument is nested, the result R contains the previous values for the properties identified by X.

Identifier	Property	Description / Legal Values
S	File Size (read only)	32 = Small-span Component Files (<4GB) 64 = Large-span Component Files
E	Endian-ness (read only)	0 = Little-endian 1 = Big-endian
U	Unicode	0 = Characters will be written as type 82 arrays 1 = Characters will be written as Unicode arrays
J	Journaling	0 = Disable Journaling 1 = Enable *APL crash proof* Journaling 2 = Enable *System crash proof* Journaling; repair needed on recovery 3 = Enable full *System crash proof* Journaling
C	Checksum	0 = Disable checksum 1 = Enable checksum
Z	Compression	0 = Disable compression 1 = Enable compression

The default properties for a newly created file are as follows:

- S = 64
- U = 1 (in Unicode Edition) or 0 (in Classic Edition)
- J = 1
- C = 1
- Z = 0
- E depends upon the computer architecture.

Note that the defaults for C and J can be overridden by calling ⎕FCREATE via the Variant operator ⍠. For further information, see *Section 6.4.45*.

Journaling Levels

Level 1 journaling (APL crash-proof) automatically protects a component file from damage in the event of abnormal termination of the APL process. The file state will be implicitly committed between updates and an incomplete update will automatically be rolled forward or back when the file is re-tied. In the event of an operating system crash the file may be more seriously damaged. If checksum was also enabled it may be repaired using ⎕FCHK but some components may be restored to a previous state or not restored at all.

Level 2 journaling (system crash-proof – repair needed on recovery) extends level 1 by ensuring that a component file is fully repairable using ⎕FCHK with no component loss in the event of an operating system failure. If an update was in progress when the system crashed the affected component will be rolled back to the previous state. Tying and modifying such a file without first running ⎕FCHK may however render it un-repairable.

Level 3 journaling (system crash-proof) extends level 2 by protecting a component file from damage in the event of abnormal termination of the APL process and also the operating system. Rollback of an incomplete update will be automatic and no explicit repair will be needed.

Enabling journaling on a component file will reduce performance of file updates; higher journaling levels have a greater impact.

Journaling levels 2 and 3 cannot be set unless the checksum option is also enabled.

The default level of journaling may be changed using the **APL_FCREATE_PROPS_J** parameter (see *Dyalog for Microsoft Windows Installation and Configuration Guide: Configuration Parameters* for more information).

Checksum Option

The checksum option is enabled by default. This enables a damaged file to be repaired using ⎕FCHK. It will however reduce the performance of file updates slightly and result in larger component files. The default may be changed using the **APL_FCREATE_PROPS_C** parameter (See User Guide).

Enabling the checksum option on an existing non-empty component file will result in all previously written components without a checksum being check-summed and converted. This operation which will take place when ⎕FPROPS is changed, may not therefore be instantaneous.

Journaling and checksum settings may be changed at any time a file is exclusively tied.

EXAMPLE

```
      tn←'myfile64' ⎕FCREATE 0
      'SEUJ' ⎕FPROPS tn
64 0 1 0
```

The following expression disables Unicode and switches Journaling on. The function returns the previous settings:

```
      ('U' 0)('J' 1) ⎕FPROPS tn
1 0
```

Note that to set the value of just a single property, the following two statements are equivalent:

```
      'J' 1 ⎕FPROPS tn
      (,⊂'J' 1) ⎕FPROPS tn
```

Properties may be read by a task with ⎕FREAD permission (access code 1), and set by a task with ⎕FSTAC access (8192). To set the value of the Journaling property, the file must be exclusively tied.

Recommendation

It is recommended that all component files are protected by a minimum of Level 1 Journalling and have Checksum enabled.

Unprotected files should only be used:

- for temporary work files where speed is paramount and integrity a secondary issue
- or where compatibility with Versions of Dyalog prior to Version 12.0 is required

This recommendation is given for the following reasons:

- Unprotected files are easily damaged by abnormal termination of the interpreter
- They cannot be repaired using ⎕FCHK
- They do not support ⎕FHIST
- They are not well supported by the Dyalog File Server (DFS)
- They do not support compression of components
- Additional features added in future may not be supported

Compression Option

Components are compressed using the *LZ4* compressor which delivers a medium level of compression, but is considered to be very fast compared to other algorithms.

Compression is intended to deliver a performance gain reading and writing large components on fast computers with slow (for example, network) file access. Conversely, on a slow computer with fast file access compression may actually reduce read/write performance. For this reason it is optional at the component level.

The default for the `'Z'` property is 0 which means no compression; 1 means compression. When written, components are compressed or not according to the current value of the `'Z'` property. Changing this property does not change any components already in the file.

A component file may therefore contain a mixture of normal and compressed components. Note that only the data in file components are compressed, the file access matrix and other header information is not compressed.

When read, compressed components are decompressed regardless of the value of the `'Z'` property.

An exclusive tie is not needed to change the file property.

Compression is not supported for files in which both Journalling and Checksum are disabled.

> **Information**
>
> Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*. Be aware that converting from non-checksummed components to checksummed components can take a significant amount of time.

6.4.57 Floating-Point Representation ⎕FR

The value of ⎕FR determines the way that floating-point operations are performed.

If ⎕FR is 645, all floating-point calculations are performed using IEEE 754 64-bit floating-point operations and the results of these operations are represented internally using *binary64*[18] floating-point format.

18 https://en.wikipedia.org/wiki/Double_precision_floating-point_format

If ⎕FR is 1287, all floating-point calculations are performed using IEEE 754-2008 128-bit decimal floating-point operations and the results of these operations are represented internally using *decimal128*[19] format.

Note that when you change ⎕FR, its new value only affects subsequent floating-point operations and results. Existing floating-point values stored in the workspace remain unchanged.

The default value of ⎕FR (its value in a `clear ws`) is configurable.

⎕FR has namespace scope but may be localised. If so, like most other system variables, it inherits its initial value from the global environment.

However: Although ⎕FR *can* vary, the system is *not designed* to allow "seamless" modification during the running of an application and the dynamic alteration of is not recommended. Strange effects may occur. For example, the type of a constant contained in a line of code (in a function or class), will depend on the value of ⎕FR *when the function is fixed*.

Also note:

```
      ⎕FR←1287
      x←1÷3

      ⎕FR←645
      x=1÷3
1
```

The decimal number has 17 more 3's. Using the tolerance which applies to binary floats (type 645), the numbers are equal. However, the "reverse" experiment yields 0, as tolerance is much narrower in the decimal universe:

```
      ⎕FR←645
      x←1÷3
      ⎕FR←1287
      x=1÷3
0
```

Since ⎕FR can vary, it will be possible for a single workspace to contain floating-point values of both types (existing variables are not converted when ⎕FR is changed). For example, an array that has just been brought into the workspace from external storage may have a different type from ⎕FR in the current namespace. Conversion (if necessary)

19 https://en.wikipedia.org/wiki/Decimal128_floating-point_format

will only take place when a *new* floating-point array is generated as the result of "a calculation". The result of a computation returning a floating-point result will *not* depend on the type of the arrays involved in the expression: ⎕FR at the time when a computation is performed decides the result type, alone.

Structural functions generally do NOT change the type, for example:

```
      ⎕FR←1287
      x←1.1 2.2 3.3

      ⎕FR←645
      ⎕DR x
1287
      ⎕DR 2↑x
1287
```

128-bit decimal numbers not only have greater precision (roughly 34 decimal digits); they also have significantly larger range – from `¯1E6145` to `1E6145`. Loss of precision is accepted on conversion from 645 to 1287, but the magnitude of a number may make the conversion impossible, in which case a `DOMAIN ERROR` is issued:

```
      ⎕FR←1287
      x←1E1000
      ⎕FR←645 ⋄ x+0
DOMAIN ERROR
```

When experimenting with ⎕FR it is important to note that numeric constants entered into the Session are evaluated (and assigned a data type) before the line is actually executed. This means that constants are evaluated according to the value of ⎕FR that pertained before the line was entered. For example:

```
      ⎕FR←645
      ⎕FR
645

      ⎕FR←1287 ⋄ ⎕DR 0.1
645
      ⎕DR 0.1
1287
```

WARNING: The use of COMPLEX numbers when ⎕FR is 1287 is not recommended, because:

- any 128-bit decimal array into which a complex number is inserted or appended will be forced in its entirety into complex representation, potentially losing precision.
- All comparisons are done using ⎕DCT when ⎕FR is 1287, and the default value of `1E¯28` is equivalent to 0 for complex numbers.

6.4.58 File Read Access R←⎕FRDAC Y

Access code 4096

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. The result is the access matrix for the designated file.

For details see *Programming: Component Files*.

EXAMPLES

```
      ⎕FRDAC 1
28 2105 16385
 0 2073 16385
31   ¯1     0
```

6.4.59 File Read Component Information R←⎕FRDCI Y

Access code 512

Y must be a simple integer vector of length 2 or 3 containing the file tie number, component number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

The result is a 3 element numeric vector containing the following information:

1. the size of the component in bytes (that is, how much disk space it occupies).
2. the user number of the user who last updated the component.
3. the time of the last update in 60ths of a second since 1st January 1970 (UTC).

EXAMPLE

```
      ⎕FRDCI 1 13
2200 207 3.702094494E10
```

6.4.60 File Read Components R←⎕FREAD Y

Access code 1

Y is a 2 or 3 item vector containing the file tie number, the component number(s), and an optional passnumber. If the passnumber is omitted it is assumed to be zero. All elements of Y must be integers.

The second item in Y may be scalar which specifies a single component number or a vector of component numbers. If it is a scalar, the result is the value of the array that is stored in the specified component on the tied file. If it is a vector, the result is a vector of such arrays.

Note that any invocation of ⎕FREAD is an atomic operation. Thus if `compnos` is a vector, the statement:

```
      ⎕FREAD tie compnos passno
```

will return the same result as:

```
      {⎕FREAD tie ⍵ passno}¨compnos
```

However, the first statement will, in the case of a share-tied file, prevent any potential intervening file access from another user (without the need for a ⎕FHOLD). It will also perform slightly faster, especially when reading from a share-tied file.

EXAMPLES

```
      ⍴SALES←⎕FREAD 1 241
3 2 12

GetFile←{⎕IO←0                  ⍝ Extract contents.
    tie←⍵ ⎕FSTIE 0              ⍝ new tie number.
    fm to←2↑⎕FSIZE tie          ⍝ first and next component.
    cnos←fm+⍳to-fm              ⍝ vector of component nos.
    cvec←⎕FREAD tie cnos        ⍝ vector of components.
    cvec⊣⎕FUNTIE tie          ⍝ ... untie and return.
}
```

6.4.61 File Rename {R}←X ⎕FRENAME Y

Access code 128

Y must be a simple 1 or 2 element integer vector containing a file tie number and an optional passnumber. If the passnumber is omitted it is assumed to be zero.

X must be a simple character scalar or vector containing the new name of the file. This name must be in accordance with the operating system's conventions, and may be specified with a relative or absolute pathname. If no file extension is supplied, the first extension specified by the **CFEXT** parameter will be added. See *Installation/ Configuration: Configuration Parameters*.

The file being renamed must be tied exclusively.

The shy result of ⎕FRENAME is the tie number of the file.

EXAMPLES

```
      'SALES' ⎕FTIE 1
      'PROFIT' ⎕FTIE 2

      ⎕FNAMES
SALES
PROFIT

      'SALES.85' ⎕FRENAME 1
      '../profits/PROFITS.85' ⎕FRENAME 2
```

```
      ⎕FNAMES
SALES.85
../profits/PROFITS.85

Rename←{
    fm to←⍵
    ⎕FUNTIE to ⎕FRENAME fm ⎕FTIE 0
}
```

6.4.62 File Replace Component {R}←X ⎕FREPLACE Y

Access code 16

Y must be a simple 2 or 3 element integer vector containing the file tie number, the component number, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The component number specified must lie within the file's component number limits.

X is any array (including, for example, the ⎕OR of a namespace), and overwrites the value of the specified component. The component information (see *Section 6.4.59*) is also updated.

The shy result of ⎕FREPLACE is the file index (component number of replaced record).

EXAMPLE

```
      SALES←⎕FREAD 1 241

      (SALES×1.1) ⎕FREPLACE 1 241
```

Define a function to replace (index, value) pairs in a component file JMS.DCF:

```
Frep←{
    tie←⍺ ⎕FTIE 0
    _←{⍵ ⎕FREPLACE tie ⍺}/¨⍵
    ⎕FUNTIE tie
}

      'jms'Frep(3 'abc')(29 'xxx')(7 'yyy')
```

> **Information**
>
> Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.63 File Resize `{R}←{X}⎕FRESIZE Y`

Access code 1024

`Y` must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero.

`X` is an integer that specifies the maximum permitted size of the file in bytes. The value 0 means the maximum possible size of file.

An attempt to update a component file that would cause it to exceed its maximum size will fail with a `FILE FULL` error (21). A side effect of `⎕FRESIZE` is to cause the file to be compacted. This process removes any gaps in the file caused by replacing a component with a shorter array. Any interrupt entered at the keyboard during the compaction is ignored. Note that if the left argument is omitted, the file is simply compacted and the maximum file size remains unchanged.

During compaction, the file is restructured by reordering the components and by amalgamating the free areas at the end of the file. The file is then truncated and excess disk space is released back to the operating system. For a large file with many components, this process may take a significant time.

The shy result of `⎕FRESIZE` is the tie number of the file.

Example

```
      'test'⎕FCREATE 1 ⋄ ⎕FSIZE 1
1 1 120 1.84467440 7E19
      (10 1000⍴1.1)⎕FAPPEND 1 ⋄ ⎕FSIZE 1
1 2 80288 1.84467440 7E19

      100000 ⎕FRESIZE 1 ⍝ Limit size to 100000 bytes

      (10 1000⍴1.1)⎕FAPPEND 1
FILE FULL
      (10 1000⍴1.1)⎕FAPPEND 1
     ^

      ⎕FRESIZE 1      ⍝ Force file compaction.
```

> **Information**
>
> Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.64 File Size `R←⎕FSIZE Y`

`Y` must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero. The result is a 4 element numeric vector containing the following:

Element	Description
1	the number of first component
2	1 + the number of the last component, (that is, the result of the next ⎕FAPPEND)
3	the current size of the file in bytes
4	the file size limit in bytes

EXAMPLE

```
      ⎕FSIZE 1
1 21 65271 4294967295
```

6.4.65 File Set Access {R}←X ⎕FSTAC Y

Access code 8192

Y must be a simple integer scalar or 1 or 2 element vector containing the file tie number followed by an optional passnumber. If the passnumber is omitted it is assumed to be zero.

X must be a valid access matrix, that is, a 3-column integer matrix with any number of rows. The function sets access control for a set of specific users (1st column) and file operations (2nd column) with specified passnumbers (3rd column). Note that a 0 in the 1st column specifies **all** users, a ¯1 in the 2nd column specifies **all** file operations, and a 0 in the 3rd column specifies that **no** passnumber is required. For further details, see *Programming: Component Files*.

The shy result of ⎕FSTAC is the tie number of the file.

EXAMPLES

```
      'SALES' ⎕FCREATE 1
      (3 3⍴28 2105 16385 0 2073 16385 31 ¯1 0) ⎕FSTAC 1
      ((⎕FRDAC 1),21 2105 16385) ⎕FSTAC 1

       (1 3⍴0 ¯1 0)⎕FSTAC 2 ⍝ Let everyone do anything
```

> **Information**
>
> Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.66 File Share Tie `{R}←X ⎕FSTIE Y`

`Y` must be 0 or a simple 1 or 2 element integer vector containing an available file tie number to be associated with the file for further file operations, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The tie number must not already be associated with a tied file.

`X` must be a simple character scalar or vector which specifies the name of the file to be tied. The file must be named in accordance with the operating system's conventions, and may be specified with a relative or absolute pathname. If no file extension is supplied, the set of extensions specified by the **CFEXT** parameter are tried one after another until the file is found or the set of extensions is exhausted. See *Installation/ Configuration: Configuration Parameters*.

The file must exist and be accessible by the user. If it is already tied by another task, it must not be tied exclusively.

The shy result of `⎕FSTIE` is the tie number of the file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create, share tie or exclusive tie operation, allocates the first (closest to zero) available tie number and returns it as an explicit result. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUMS  ⍝ With next available number,
      file ⎕FSTIE tie   ⍝ ... share tie file.
```

to:

```
      tie←file ⎕FSTIE 0 ⍝ Tie with 1st available number.
```

EXAMPLE

```
      'SALES' ⎕FSTIE 1

      '../budget/COSTS' ⎕FSTIE 2
```

Variant Options

Mode

Writing to a component file is not always permitted. For example, restrictions on writing to a component file might be imposed by operating system permissions, the host filesystem, or individual component file property settings.

The **Mode** variant option specifies whether the file that is being tied will only be read or must be writeable. Possible values are:

- `P` (tied as **p**ermitted) – the file will be tied for write access if possible, otherwise for read access only. If the file permissions do not allow the file to be written to, any subsequent attempt to write to it will fail. This is the default.
- `R` (**r**ead mode) – the file will be tied for read access only; any subsequent attempt to write to it will fail.
- `W` (**w**rite mode) – if the file permissions do not allow the file to be written to, the attempt to tie it will fail.

The **Mode** variant option is independent of any *Programming: Component Files*.

EXAMPLE

```
      'cf' (⎕FSTIE⍠'Mode' 'W') 1
FILE ACCESS ERROR: cf.dcf: File is not writable
      'cf'(⎕FSTIE⍠'Mode' 'W')1
                ^
```

> **Information**
>
> Small-span (32-bit) component files are currently read-only; this support is scheduled for removal in a future release, after which it will not be possible to tie small-span component files. Dyalog Ltd recommends using `⎕FCOPY` to convert any such files to large-span (64-bit). For information on how to identify calls to small-span component files in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.67 Exclusive File Tie `{R}←X ⎕FTIE Y`

Access code 2

`Y` must be 0 or a simple 1 or 2 element integer vector containing an available file tie number to be associated with the file for further file operations, and an optional passnumber. If the passnumber is omitted it is assumed to be zero. The tie number must not already be associated with a share tied or exclusively tied file.

`X` must be a simple character scalar or vector which specifies the name of the file to be exclusively tied. The file must be named in accordance with the operating system's conventions, and may be a relative or absolute pathname. If no file extension is supplied, the set of extensions specified by the **CFEXT** parameter are tried one after another until the file is found or the set of extensions is exhausted. See *Installation/ Configuration: Configuration Parameters*.

The file must exist and the user must have write access to it. It may not already be tied by another user.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create, share tie or exclusive tie operation, allocates the first (closest to zero) available tie number, and returns it as an explicit result. This allows you to simplify code. For example:

from:

```
      tie←1+⌈/0,⎕FNUMS ⍝ With next available number,
      file ⎕FTIE tie   ⍝ ... tie file.
```

to:

```
      tie←file ⎕FTIE 0 ⍝ Tie with first available number.
```

The shy result of ⎕FTIE is the tie number of the file.

EXAMPLES

```
      'SALES' ⎕FTIE 1

      '../budget/COSTS' ⎕FTIE  2

      '../budget/expenses' ⎕FTIE 0
3
```

Variant Options

Mode

Writing to a component file is not always permitted. For example, restrictions on writing to a component file might be imposed by operating system permissions, the host filesystem, or individual component file property settings.

The **Mode** variant option specifies whether the file that is being tied will only be read or must be writeable. Possible values are:

- `P` (tied as **p**ermitted) – the file will be tied for write access if possible, otherwise for read access only. If the file permissions do not allow the file to be written to, any subsequent attempt to write to it will fail. This is the default.
- `R` (**r**ead mode) – the file will be tied for read access only; any subsequent attempt to write to it will fail.
- `W` (**w**rite mode) – if the file permissions do not allow the file to be written to, the attempt to tie it will fail.

The **Mode** variant option is independent of any *Programming: Component Files*.

EXAMPLE

```
      'cf' (⎕FTIE⍠'Mode' 'W') 1
FILE ACCESS ERROR: cf.dcf: File is not writable
      'cf'(⎕FTIE⍠'Mode' 'W')1
                ^
```

> **Information**
>
> Small-span (32-bit) component files are currently read-only; this support is scheduled for removal in a future release, after which it will not be possible to tie small-span component files. Dyalog Ltd recommends using `⎕FCOPY` to convert any such files to large-span (64-bit). For information on how to identify calls to small-span component files in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.68 File Untie `{R}←⎕FUNTIE Y`

`Y` must be a simple integer scalar or vector (including Zilde). Files whose tie numbers occur in `Y` are untied. Other elements of `Y` have no effect.

If `Y` is empty, no files are untied, but all the interpreter's internal file buffers are flushed and the operating system is asked to flush all file updates to disk. This special facility allows the programmer to add extra security (at the expense of performance) for application data files.

The shy result of `⎕FUNTIE` is a vector of tie numbers of the files **actually untied**.

EXAMPLE

```
      ⎕FUNTIE ⎕FNUMS ⍝ Unties all tied files

      ⎕FUNTIE ⍬      ⍝ Flushes all buffers to disk
```

6.4.69 Fix Definition {R}←⎕FX Y

Y is the representation form of a function or operator which may be:

- its canonical representation form similar to that produced by ⎕CR except that redundant blanks are permitted other than within names and constants, and the first and last rows may start with a del symbol (∇).
- its nested representation form similar to that produced by ⎕NR except that redundant blanks are permitted other than within names and constants, and the first and last items may be del (∇) symbols.
- its object representation form produced by ⎕OR.
- its vector representation form similar to that produced by ⎕VR except that additional blanks are permitted other than within names and constants.

⎕FX attempts to create (fix) a function or operator in the workspace or current namespace from the definition given by Y. ⎕IO is an implicit argument of ⎕FX. Note that ⎕FX does not update the source of a scripted namespace, or of class or instance; the only two methods of updating the source of scripted objects is via the Editor, or by calling ⎕FIX.

If the function or operator is successfully fixed, R is a simple character vector containing its name and the result is shy. Otherwise R is an integer scalar containing the (⎕IO dependent) index of the row of the canonical representation form in which the first error preventing its definition is detected. In this case the result R is **not shy**.

> **Legacy**
>
> Before Dyalog v20.0, it was possible to define dfns with unmatched parentheses and brackets. These are now rejected. TradFns will continue to fix as before, but subtle differences in how the code behaves might not be backwards-compatible and could have unexpected results.

Functions and operators which are pendent, that is, in the state indicator without a suspension mark (*), retain their original definition until they complete, or are cleared from the state indicator. All other occurrences of the function or operator assume the new definition. The function or operator will fail to fix if it has the same name as an existing variable, or a visible label.

6.4.70 Instances R←⎕INSTANCES Y

⎕INSTANCES returns a list all the current instances of the Class specified by Y.

Y must be a reference.

If Y is a reference to a Class, R is a vector of references to all existing Class Instances of Y. Otherwise, R is empty.

Examples

This example illustrates a simple inheritance tree or Class hierarchy. There are 3 Classes, namely:

```
Animal
    Bird (derived from Animal)
        Parrot (derived from Bird)
```

```
:Class Animal
...
:EndClass ⍝ Animal

:Class Bird: Animal
...
:EndClass ⍝ Bird

:Class Parrot: Bird
...
:EndClass ⍝ Parrot
```

```
      Eeyore←⎕NEW Animal
      Robin←⎕NEW Bird
      Polly←⎕NEW Parrot

      ⎕INSTANCES Parrot
 #.[Parrot]
      ⎕INSTANCES Bird
 #.[Bird]  #.[Parrot]
      ⎕INSTANCES Animal
 #.[Animal]  #.[Bird]  #.[Parrot]
```

```
      Eeyore.⎕DF 'eeyore'
      Robin.⎕DF 'robin'
      Polly.⎕DF 'polly'

      ⎕INSTANCES Parrot
polly
      ⎕INSTANCES Bird
robin  polly
      ⎕INSTANCES Animal
eeyore  robin  polly
```

6.4.71 Index Origin ⎕IO

⎕IO determines the index of the first element of a non-empty vector.

⎕IO may be assigned the value 0 or 1. The value in a clear workspace is 1. ⎕IO has Namespace scope.

⎕IO is an implicit argument of any function derived from the Axis operator ([K]), of the monadic functions Fix (⎕FX), Grade Down (⍒), Grade Up (⍋), Index Generator (⍳), Roll (?), and of the dyadic functions Deal (?), Grade Down (⍒), Grade Up (⍋), Index (⌷), Index Of (⍳), Indexed Assignment, Indexing, Pick (⊃) and Transpose (⍉).

Examples

```
        ⎕IO←1
        ⍳5
1 2 3 4 5

        ⎕IO←0
        ⍳5
0 1 2 3 4

        +/[0]2 3⍴⍳6
3 5 7

        'ABC',[¯.5]'='
ABC
===
```

6.4.72 JSON Convert `R←{X}⎕JSON Y`

This function imports and exports data in JavaScript Object Notation (JSON) Data Interchange Format.

JSON supports a limited number of data types and there is not a direct correspondence between JSON and APL data structures. In particular:

- JSON does not support arrays with rank >1.
- JSON standard includes Boolean values true and false which are distinct from numeric values 1 and 0, and have no direct APL equivalent.
- The JSON5 standard includes numeric constants `Infinity`, `-Infinity`, `NaN` and `-NaN` which have no direct APL equivalent.
- JSON object members are named and these names might not be valid names in APL.

These differences are catered for in various ways as discussed below.

If specified, `X` must be a numeric scalar with the value `0` (import JSON) or `1` (export JSON). If `X` is not specified and `Y` is a character array, `X` is assumed to be `0` (import); otherwise it is assumed to be 1 (export).

Although this system function was designed with an optional left argument, it is strongly recommended that the argument should always be used.

Other options for `⎕JSON` are `Format`, `Compact`, `Null`, `HighRank`, `Charset` and `Dialect` which are specified using `⎕OPT` (see *Section 4.4.22*). The Principal Option is `Format`.

The `Dialect` Variant option is either `'JSON'` (the default) or `'JSON5'`. The latter enables JSON5 extensions on import and export.

JSON Import (`X` is `0`)

`Y` is a character vector or matrix in JSON format. There is an implied newline character between each row of a matrix.

The content of the result `R` depends upon the `Format` variant, which can be `'D'` (the default) or `'M'`.

If `Format` is `'D'` (which stands for "data") the JSON described by `Y` is converted to APL object(s) and `R` is an array or a namespace containing arrays and sub-namespaces.

- JSON objects are created as APL namespaces.

- JSON null is converted to the value specified by the `Null` variant, which may be either `⊂'null'` (the default) or `⎕NULL`.
- JSON true and false and the JSON5 numeric constants `Infinity`, `-Infinity`, `NaN` and `-NaN` are converted to enclosed character vectors `⊂'true'`,`⊂'false'` and so forth.
- If the JSON source contains object names which are not valid APL names they are converted to APL objects with mangled names. See JSON Name Mangling. `7162⌶` can be used to obtain the original name. See *Section 4.4.9.67*

If `Format` is `'M'` (which stands for "matrix") the result `R` is a matrix whose columns contain the following:

[;1]	depth
[;2]	name (for JSON object members)
[;3]	value
[;4]	JSON type (integer: see below)

- The representation of null, true and false are the same as for `Format 'D'`.
- Object names are reported as specified in the JSON text; they are not mangled as they are for `Format 'D'`.

Type	Description
1	Object
2	Array
3	Numeric
4	String
5	Null
6	No APL equivalent (represented by character string)
7	JavaScript Object (export only)

Duplicate Names

The JSON standard says that members of a JSON object should have unique names and that different implementations behave differently when there are duplicates. Dyalog handles duplicate names as follows:

- No error is generated.

- For `Format 'D'`, the last member encountered is used and all previous members with the same name are discarded.
- For `Format 'M'` all duplicate members are recorded in the result matrix.

Examples

```
      ρJSON
18 19
      JSON
{
 "a": {
  "b": [
   "string 1",
   "string 2"
  ],
  "c": true,
  "d": {
   "e": false,
   "fα": [
    "string 3",
    123,
    1000.2,
    null
   ]
  }
 }
}
```

Import as Data (`Format 'D'`)

```
      j←0 ⎕JSON JSON
      j
#.[JSON object]
      j.⎕NL 9
a
      j.a.⎕NL 2
b
c
      j.a.b
┌─────────┬─────────┐
│string 1 │string 2 │
└─────────┴─────────┘
      j.a.c
┌────┐
│true│
└────┘
      j.a.⎕NL 9
d
      j.a.d.⎕NL 2 ⍝ Note that f⍺ is an invalid APL name
e
Δ f Δ 9082 Δ
      j.a.d.e
┌─────┐
│false│
└─────┘
      j.a.d.Δ f Δ 9082 Δ
┌─────────┬───┬──────┬──────┐
│string 3 │123│1000.2│┌────┐│
│         │   │      ││null│
│         │   │      │└────┘│
└─────────┴───┴──────┴──────┘
```

Import as Matrix (`Format 'M'`)

```
      (⎕JSON⍠'M')JSON
┌─┬──┬─────────┬─┐
│0│  │         │1│
├─┼──┼─────────┼─┤
│1│a │         │1│
├─┼──┼─────────┼─┤
│2│b │         │2│
├─┼──┼─────────┼─┤
│3│  │string 1 │4│
├─┼──┼─────────┼─┤
│3│  │string 2 │4│
├─┼──┼─────────┼─┤
│2│c │┌─────┐  │6│
│ │  ││true │  │ │
│ │  │└─────┘  │ │
├─┼──┼─────────┼─┤
│2│d │         │1│
├─┼──┼─────────┼─┤
│3│e │┌──────┐ │6│
│ │  ││false │ │ │
│ │  │└──────┘ │ │
├─┼──┼─────────┼─┤
│3│fα│         │2│
├─┼──┼─────────┼─┤
│4│  │string 3 │4│
├─┼──┼─────────┼─┤
│4│  │123      │3│
├─┼──┼─────────┼─┤
│4│  │1000.2   │3│
├─┼──┼─────────┼─┤
│4│  │┌────┐   │5│
│ │  ││null│   │ │
│ │  │└────┘   │ │
└─┴──┴─────────┴─┘
```

JSON Export (`X` is `1`)

`Y` is the data to be exported as JSON and may be an array, a namespace or a matrix representation of JSON such as would have been produced by JSON Import with

`Format 'M'.Y` is interpreted according to the `Format` variant which may be `'D'` (the default) or `'M'`.

`JSON` will signal `DOMAIN ERROR` if `Y` is incompatible with the specified (or implied) value of `Format`.

If `Format` is `M`, the data values in `Y[;3]` must correspond precisely with the JSON types specified in `Y[;4]` as specified in the following table.

`Y[;4]` **(Type)**	`Y[;3]` **(Value)**
1	Empty array
2	Empty array
3	Numeric scalar
4	Character vector
5	Null
6	Enclosed character vector
7	Enclose character vector

`R` is a character vector whose content depends upon the value of the `Compact` variant.

Compact	**Description**
0	The JSON text is padded with spaces and new lines for readability.
1	The JSON text is compacted into its minimal form.

The `Charset` variant option may be used to restrict the output to ASCII characters.

Charset	**Description**
'Unicode'	All Unicode characters in `Y` are passed unchanged in the result `R`.
'ASCII'	Non-ASCII characters are converted to an encoded string of the form "\uNNNN" where "NNNN" is the upper-case hexadecimal value of the character in the Unicode system. For example, é (e-acute) is converted to "\u00E9". Furthermore, if the Dialect is JSON5, values less than hex 100 are converted to the form \xNN.

The `HighRank` variant option may be used to instruct ⎕JSON to pre-process higher rank arrays into a form that can be represented by JSON. Note that if necessary, the transformation is applied recursively throughout the high-rank array(s) specified by `Y`.

HighRank	Description
'Split'	High rank data is split into nested vectors.
'Error'	Higher rank data is rejected (`DOMAIN ERROR`)

The name of any namespace member that begins with ⍙ and otherwise conforms to the conversion format used for JSON object names will be demangled.

Example

```
      j              ⍝ See above
#.[JSON object]
      ⍴JS←1 ⎕JSON j
94
      JS
{"a":{"b":["string 1","string 2"],"c":true,"d":{"e":false,"f⍙":["string
3",123,1000.2,null]}}}

      1(⎕JSON⍠'Compact' 0) j

{
      "a": {
            "b": [
                  "string 1",
                  "string 2"
            ],
            "c": true,
            "d": {
                  "e": false,
                  "f⍙": [
                        "string 3",
                        123,
                        1000.2,
                        null
                  ]
            }
      }
}
```

If there are any mis-matches between the values in `Y[;3]` and the types in `Y[;4]`, `⎕JSON` will signal `DOMAIN ERROR` and report the first row where there is a mis-match (`⎕IO` sensitive) as illustrated in the following example.

Example

```
      M←(⎕JSON⍠'Format' 'M')'{"values": [ 75, 300 ]}'
      M
┌─┬──────┬───┬─┐
│0│      │   │1│
├─┼──────┼───┼─┤
│1│values│   │2│
├─┼──────┼───┼─┤
│2│      │75 │3│
├─┼──────┼───┼─┤
│2│      │300│3│
└─┴──────┴───┴─┘

      M[3;3]←⊂'75'  ⍝ character not numeric

      M             ⍝ but looks the same as before

┌─┬──────┬───┬─┐
│0│      │   │1│
├─┼──────┼───┼─┤
│1│values│   │2│
├─┼──────┼───┼─┤
│2│      │75 │3│
├─┼──────┼───┼─┤
│2│      │300│3│
└─┴──────┴───┴─┘

      1 (⎕JSON⍠'Format' 'M')M
DOMAIN ERROR: JSON export: value does not match the specified type in ro
w 3 (⎕IO=1)
      1(⎕JSON⍠'Format' 'M')M
      ^
```

Charset Example

```
      ns←⎕NS ''
      ns.dé←'DÉ'
      ns.dé
DÉ
      ⎕JSON ns
{"dé":"DÉ"}
      (⎕JSON⍠'Charset' 'ASCII')ns
{"d\u00E9":"D\u00C9"}
```

High Rank Example

```
      d
┌─────┬──────────────────────────┐
│1 2  │ABC                       │
│A B  │DEF                       │
├─────┼──────────────────────────┤
│1 2 3│1            0.5          │
│4 5 6│0.3333333333 0.25         │
│     │                          │
│     │0.2          0.1666666667 │
│     │0.1428571429 0.125        │
└─────┴──────────────────────────┘
      1 ⎕JSON d
DOMAIN ERROR: JSON export: the right argument cannot be converted
(⎕IO=1)
      1 ⎕JSON d
      ^
      1 (⎕JSON⍠'HighRank' 'Split') d
[[[[1,2],"AB"],["ABC","DEF"]],[[[1,2,3],[4,5,6]],...
```

Raw Text

An enclosed character vector is inserted into the result of JSON export as raw text. This feature may be used to export special JSON values such as `null`, `true` and `false`. Without the extra enclosure, the character vectors are exported as strings:

Example

```
      ⎕JSON 'null' 'true' 'false'
["null","true","false"]
      ⎕JSON ⊂⍤1⊢'null' 'true' 'false'
[null,true,false]
```

The same mechanism may be used to inject any raw text, although unless this is valid JSON it cannot then be re-imported.

The following example illustrates how JavaScript objects may be exported. In the example, the object is a JavaScript function which is specified by the contents of an enclosed character vector.

Example

```
      'Slider' ⎕NS ''
      Slider.range←⊂'true'          ⍝ Note the ⊂
      Slider.min←0
      Slider.max←500
      Slider.values←75 300

      fn1←' function( event, ui ) {'
      fn2←'$( "#amount" ).val( "$" + ui.values[ 0 ] +'
      fn2,←' " - $" + ui.values[ 1 ] );}'

      Slider.slide←,/fn1 fn2 ⍝ Enclosed character vec
      ⍴JS←1 ⎕JSON Slider
159
      JS
{"max":500,"min":0,"range":true,"slide": function( event, ui ) {$( \"#am
ount\" ).val( \"$\" + ui.values[ 0 ] + \" - $\" + ui.values[ 1 ]);},"val
ues":[75,300]}
```

Wrappers

A wrapper is an enclosed vector of the form:

```
      ⊂code special
```

The nature of the `special` data structure is identified within the wrapper by a leading numeric code. Code 1 is used to identify JSON values such as `null`, `true` and `false`. Codes 2, 3 and 4 are used to identify different forms of datasets.

This wrapper mechanism has been chosen to identify special treatment because a scalar enclosure cannot be represented in JSON/JavaScript.

A wrapper may be specified directly in the right argument to `⎕JSON` and/or as part of the array structure specified by the right argument, as a sub-array or in a namespace.

This allows a special array to be processed appropriately as part of a general data structure that is to be rendered in JSON notation.

Wrappers for special JSON values

Wrappers may be used to export JSON special values such as `null`, `true` and `false` using code 1. This mechanism is supplementary to the use of enclosed character vectors. See **RawText** above.

Example

```
      ⎕JSON⍠'Dense'(1 'null')(1 'true')(1 'false')
[null,true,false]
```

Datasets

The term dataset is used here to mean a collection of data, usually presented in tabular form. Each named column represents a particular variable. Each row corresponds to a given member of the dataset in question. It lists values for each of the variables, such as height and weight of an object.

Datasets are often represented in APL as a collection of variables.

```
      Fields←'Item' 'Price' 'Qty'
      Items←'Knife' 'Fork'
      Price←3 4
      Qty←23 45
```

As an aside, note that using this scheme each variable represents an inverted index into the dataset and enables rapid searches.

```
      (Price<4)/Items
┌─────┐
│Knife│
└─────┘
```

A conventional way to represent this dataset is as a matrix:

```
      Fields⍪⍉↑ Items Price Qty

┌────┬─────┬──┐
│Item│Price│Qty│
├────┼─────┼──┤
│Knife│    3│23 │
├────┼─────┼──┤
│Fork │    4│45 │
└────┴─────┴──┘
```

Another is as a 2-item vector containing the names of the fields and a matrix of their values:

```
      (Fields (⍉↑Items Price Qty))

┌──────────────────┬──────────────┐
│ ┌────┬─────┬──┐ │ ┌─────┬─┬──┐ │
│ │Item│Price│Qty│ │ │Knife│3│23│ │
│ └────┴─────┴──┘ │ ├─────┼─┼──┤ │
│                 │ │Fork │4│45│ │
│                 │ └─────┴─┴──┘ │
└──────────────────┴──────────────┘
```

A third way retains the inverted nature of the data structure, storing the values as a vector. The advantage of this structure is that it consumes significantly less memory compared to the matrix forms, because numeric columns are stored as simple numeric vectors.

```
      (Fields (Items Price Qty))

┌──────────────────┬──────────────────────────┐
│ ┌────┬─────┬──┐ │ ┌─────────┬─────┬──────┐ │
│ │Item│Price│Qty│ │ │ ┌─────┬─┐ │3 4  │23 45│ │
│ └────┴─────┴──┘ │ │ │Knife│Fork│ │     │     │ │
│                 │ │ └─────┴─┘ │     │     │ │
│                 │ └─────────┴─────┴──────┘ │
└──────────────────┴──────────────────────────┘
```

In JSON, these three data structures are all expressed as follows:

```
[
      {
            "Item": "Knife",
            "Price": 3,
            "Qty": 23
      },
      {
            "Item": "Fork",
            "Price": 4,
            "Qty": 45
      }
]
```

Examples

```
      Fields,[1]↑[1]Items Price Qty

┌─────┬─────┬──┐
│Item │Price│Qty│
├─────┼─────┼──┤
│Knife│3    │23 │
├─────┼─────┼──┤
│Fork │4    │45 │
└─────┴─────┴──┘
      ⎕JSON ⊂ 2 (Fields,[1]↑[1]Items Price Qty)
[{"Item":"Knife","Price":3,"Qty":23},{"Item":"Fork","Price":4,"Qty":45}]
```

Note that if you omit the wrapper the operation fails:

```
      ⎕JSON Fields,[1]↑[1]Items Price Qty)

DOMAIN ERROR: JSON export: the right argument cannot be converted
(⎕IO=1)
⎕JSON Fields,[1]↑[1]Items Price Qty)
^
```

Further Examples

```
      ⎕JSON ⊂ 3 ((↑[1]Items Price Qty)Fields)
[{"Item":"Knife","Price":3,"Qty":23},{"Item":"Fork","Price":4,"Qty":45}]

      ⎕JSON ⊂ 4 ((Items Price Qty)Fields)
[{"Item":"Knife","Price":3,"Qty":23},{"Item":"Fork","Price":4,"Qty":45}]
```

Note that if you omit the wrapper, the operation generates a different result.

```
      ⎕JSON ((Items Price Qty)Fields)
[[["Knife","Fork"],[3,4],[23,45]],["Item","Price","Qty"]]
```

Selection

For codes 2, 3 and 4 the extension also provides the facility to optionally select elements of the dataset, so the array may contain 2, 3 or 4 items:

```
      ⊂(code dataset {records} {fields})
```

where records and fields are integer indices that select which fields and which records are to be exported. The following example selects the first record and the first and third fields (Items and Qty)

```
      ⎕JSON⊂4 ((Items Price Qty)Fields)1(1 3)
[{"Item":"Knife","Qty":23}]
```

Namespaces and Sub-Arrays

Wrappers in namespaces and sub-arrays are recognised for special treatment.

Example

```
      ns.Items←'Fork' 'Knife'
      ns.Price←3 4
      ns.Qty←23 45
      ns.(ds←⊂4(⍉('Item' 'Price' 'Qty')(Items Price Qty)))
      ⎕JSON ns
{"Items":["Knife","Fork"],"Price":[3,4],"Qty":[23,45],"ds":[{"Item":"Kni
fe","Price":3,"Qty":23},{"Item":"Fork","Price":4,"Qty":45}]}

      a←'the' 'answer' 'is' 42
      a[3]←⊂ns.ds
      ⎕JSON a
["the","answer",[{"Item":"Knife","Price":3,"Qty":23},{"Item":"Fork","Pri
ce":4,"Qty":45}],42]
```

JSON Name Mangling

When Dyalog converts from JSON to APL data, and a member of a JSON object has a name which is not a valid APL name, it is renamed.

Example

In this example, the JSON describes an object containing two numeric items, one named *a* (which is a valid APL name) and the other named *2a* (which is not):

```
{"a": 1, "2a": 2}
```

When this JSON is imported as an APL namespace using ⎕JSON, Dyalog converts the name *2a* to a valid APL name. The *name mangling* algorithm creates a name beginning with ⍙.

```
      (⎕JSON'{"a": 1, "2a": 2}').⎕NL 2
a
⍙2a
```

When Dyalog exports JSON it performs the reverse *name mangling*, so:

```
      1 ⎕JSON ⎕JSON'{"a": 1, "2a": 2}'
{"a":1,"2a":2}
```

Should you need to create and decode these names directly,`7162⌶` provides the same name mangling and un-mangling operations. See *Section 4.4.9.67*.

```
      0(7162⌶)'2a'
⍙2a
      1(7162⌶)'⍙2a'
2a
```

6.4.73 Key Label R←⎕KL Y

`Y` is a simple character vector or a vector of character vectors containing Input Codes for Keyboard Shortcuts. In the Classic Edition, keystrokes are associated with Keyboard Shortcuts by the Input Translate Table.

`R` is a simple character vector or a vector of character vectors containing the labels associated with the codes. If `Y` specifies codes that are not defined, the corresponding elements of `R` are the codes in `Y`.

`⎕KL` provides the information required to build device-independent help messages into applications, particularly full-screen applications using `⎕SM` and `⎕SR`.

EXAMPLES

```
      ⎕KL 'RC'
Right

      ⎕KL 'ER' 'EP' 'QT' 'F1' 'F13'
  Enter  Esc  Shift+Esc  F1  Shift+F1
```

> **Dyalog on Microsoft Windows**
>
> In the Unicode edition, `⎕KL` can also get the keystrokes to type APL glyphs:
>
> ```
>       ⎕KL '⌸'
> Control+Shift+K
> ```

6.4.74 Line Count R←⎕LC

This is a simple vector of line numbers drawn from the state indicator. The most recently activated line is shown first. If a value corresponds to a defined function in the state indicator, it represents the current line number where the function is either suspended or pendent.

The value of ⎕LC changes immediately upon completion of the most recently activated line, or upon completion of execution within ⍎ or ⎕. If a ⎕STOP control is set, ⎕LC identifies the line on which the stop control is effected. In the case where a stop control is set on line 0 of a defined function, the first entry in ⎕LC is 0 when the control is effected.

The value of ⎕LC in a clear workspace is the null vector.

EXAMPLES

```
      )SI
#.TASK1[5]*
⍟
#.BEGIN[3]

      ⎕LC
5 3
```

```
      →⎕LC
      ⎕LC

      ⍴⎕LC
0
```

6.4.75 Load Workspace ⎕LOAD Y

Y must be a simple character scalar or vector containing the name of a file that contains a saved workspace. . If no file extension is given it is implied. See *Installation/ Configuration: Wsext*.

If Y is ill-formed or does not identify a saved workspace or the user account does not have access permission to the workspace, a DOMAIN ERROR is reported.

Otherwise, the active workspace is replaced by the workspace identified in Y. The active workspace is lost. If the loaded workspace was saved by the)SAVE system command, the latent expression (⎕LX) is immediately executed, unless APL was invoked with the -x option. If the loaded workspace was saved by the ⎕SAVE system function, execution resumes from the point of exit from the ⎕SAVE function, with the result of the ⎕SAVE function being 0, running in the same namespace in which the ⎕SAVE was executed.

The workspace identification and time-stamp when saved is not displayed.

If the workspace contains any GUI objects whose Visible property is 1, these objects will be displayed. If the workspace contains a non-empty ⎕SM but does not contain an SM GUI object, the form defined by ⎕SM will be displayed in a window on the screen.

The system switches to the namespace that was the current namespace when the workspace was saved.

6.4.76 Lock Definition {R}←{X}⎕LOCK Y

Y must be a simple character scalar, or vector which is taken to be the name of a defined function or operator in the active workspace. ⎕LOCK does not apply to dfns or derived functions.

The active referent to the name in the workspace is locked. Stop, trace and monitor settings, established by the ⎕STOP, ⎕TRACE and ⎕MONITOR functions, are cancelled.

The optional left argument X specifies to what extent the function code is hidden. X can be 0, 1, 2, or 3 (the default) with the following meaning:

- 0: The argument is well-formed, but could not be locked (for example, it is a dfn).
- 1: The object may not be displayed and you may not obtain its character form using ⎕ATX, ⎕CR, ⎕NR, or ⎕VR.
- 2: If an error or exception occurs that would normally cause a suspension of execution within the locked function or operator, the state indicator is cut back to the statement that called it and the suspension is triggered there instead.
- 3: Both 1 and 2 apply. You can neither display the locked object nor suspend execution within it.

Locks are additive, so that

```
      1 ⎕LOCK'FOO' ⋄ 2 ⎕LOCK'FOO'
```

is equivalent to:

```
      3 ⎕LOCK'FOO'
```

The shy result R is the lock state (1, 2, or 3) of Y.

A DOMAIN ERROR is reported if Y is ill-formed.

Examples

```
      ⎕FX'r←foo' 'r←10'
      62 ⎕ATX'foo'
  r←foo   r←10
      ≠62 ⎕ATX'foo'
2
      ⎕LOCK'foo'
      ≠62 ⎕ATX'foo'
0
```

6.4.77 Latent Expression ⎕LX

This may be a character vector or scalar representing an APL expression. The expression is executed automatically when the workspace is loaded. If APL is invoked using the `-x` flag, this execution is suppressed.

The value of ⎕LX in a clear workspace is `''`. ⎕LX has workspace scope.

EXAMPLE

```
      ⎕LX←'''GOOD MORNING PETE'''

      )SAVE GREETING
GREETING saved Tue Sep 8 10:49:29 1998

      )LOAD GREETING
./GREETING saved Tue Sep 8 10:49:29 1998
GOOD MORNING PETE
```

6.4.78 Map File R←{X}⎕MAP Y

⎕MAP function associates a mapped file with an APL array in the workspace.

Two types of mapped files are supported; *APL* and *raw*. An *APL* mapped file contains the binary representation of a Dyalog APL array, including its header. A file of this type must be created using the utility function ⍙MPUT (supplied in the `util` workspace). When you map an APL file, the rank, shape and data type of the array is obtained from the information on the file.

A *raw* mapped file is an arbitrary collection of bytes. When you map a raw file, you must specify the characteristics of the APL array to be associated with this data. In particular, the data type and its shape.

The type of mapping is determined by the presence (raw) or absence (APL) of the left argument to ⎕MAP.

The right argument Y specifies the name of the file to be mapped and, optionally, the access type and a start byte in the file. Y may be a simple character vector, or a 2 or 3-element nested vector containing:

1. file name (character scalar/vector)
2. access code (character scalar/vector) : one of : `'R'` or `'r'` (read-only access), `'W'` or `'w'` (read-write access). If not specified, the file is mapped read-only.

3. start byte offset (integer scalar/vector). This is only applicable for read-only access and is not supported for read-write access. It must be a multiple of the word size (4 on 32-bit systems, 8 on 64-bit systems). The default is 0.

If you map a file with read-only access you may modify the corresponding array in the workspace, however your changes are not written back to the file.

If `X` is specified, it defines the type and shape to be associated with *raw* data on file. `X` must be an integer scalar or vector. The first item of `X` specifies the data type and must be one of the following values:

Classic Edition	11, 82, 83, 163, 323 or 645
Unicode Edition	11, 80, 83, 160, 163, 320, 323 or 645

The values are more fully explained in *Section 6.4.33*.

Following items determine the shape of the mapped array. A value of `¯1` on any (but normally the first) axis in the shape is replaced by the system to mean: read as many complete records from the file as possible. Only one axis may be specified in this way. Note that if `X` is a singleton, the data on the file is mapped as a scalar and only the first value on the file is accessible.

If no left argument is given, file is assumed to contain a simple APL array, complete with header information (type, rank, shape, etc.). Such mapped files may only be updated by changing the associated array using indexed/pick assignment: `var[a]←b`, the new values must be of the same type as the originals.

Note that a *raw* mapped file may be updated *only* if its *file offset* is 0. Note also that Windows does not support mapped files of zero length.

Examples

Map raw file as a read-only *vector* of doubles:

```
      vec←645 ¯1 ⎕MAP'c:\myfile'
```

Map raw file as a 20-column read-write *matrix* of 1-byte integers:

```
      mat←83 ¯1 20 ⎕MAP'c:\myfile' 'W'
```

Replace some items in mapped file:

```
      mat[2 3;4 5]←2 2⍴⍳4
```

Map bytes 100-160 in raw file as a 5×2 read-only matrix of doubles:

```
      dat←645 5 2 ⎕MAP'c:\myfile' 'R' 80
```

Put simple 4-byte integer array on disk ready for mapping:

```
      (⊃83 323 ⎕DR 2 3 4⍴⍳24)⍙MPUT'c:\myvar'
```

Then, map a read-write variable:

```
      var←⎕MAP'c:\myvar' 'w'
```

Note that a mapped array need not be *named*. In the following example, a 'raw' file is mapped, summed and released, all in a single expression:

```
      +/163 ¯1 ⎕MAP'c:\shorts.dat'
42
```

If you fail to specify the shape of the data, the data on file will be mapped as a scalar and only the first value in the file will be accessible:

```
      83 ⎕MAP 'myfile'    ⍝ map FIRST BYTE of file.
¯86
```

Compatibility between Editions

In the Unicode Edition ⎕MAP will fail with a TRANSLATION ERROR (event number 92) if you attempt to map an APL file which contains character data type 82.

In order for the Unicode Edition to correctly interpret data in a raw file that was written using data type 82, the file may be mapped with data type 83 and the characters extracted by indexing into ⎕AVU.

6.4.79 Make Directory {R}←{X}⎕MKDIR Y

This function creates new directories.

Y is a character vector or scalar containing a single directory name, or a vector of character vectors containing zero or more directory names. Names must conform to the naming rules of the host Operating System.

By default, for each name in Y the path must exist and the base name must not exist (see *Section 6.4.99*), otherwise an error is signalled. The optional left argument X and the variant option **Unique** can be used to amend this behaviour.

The result R depends on the value of the variant option **Unique**. If **Unique** is not present, it is assumed to have a value of 0.

Unique	**Result** R
0	If Y specifies a single name, the shy result R is a scalar 1 if a directory was created or 0 if not. If Y is a vector of character vectors, R is a vector of 1s and 0s with the same length as Y.
1	If Y specifies a single name, the shy result R is a character vector containing the name of the directory that was created. If Y is a vector of character vectors, R is a vector of character vectors with the same length as Y.

The optional left argument X is a numeric scalar that modifies the default behaviour when the base name in Y already exists and/or the path in Y does not already exist. If omitted, it is assumed to be 0. Possible values and the effect that they have on the default behaviour are:

0	The base name in Y must not exist and the path in Y must exist, otherwise an error is signalled.
1	No action is taken if a directory specified by Y already exists (the return value indicates whether a new directory was created). Has no effect when the variant option **Unique** is set.
2	Any part of the *paths* specified in Y which does not already exist will be created in preparation of creating the corresponding directory.
3	Combination of 1 and 2.

Variant Options

⍁MKDIR may be applied using the variant operator with the option **Unique**. There is no primary option.

Unique Option (Boolean)

The **Unique** option specifies whether the base name (see *Section 6.4.99*) in `Y` is modified so that the name is unique (does not already exist).

Unique	Effect on Behaviour
`0`	The directory named in `Y` will be created.
`1`	The name in `Y` is modified by extending the base name with random characters and the directory is created. The name of the directory is returned in the result `R`.

If a directory cannot be created (for example, if a directory with that name already exists, or write access is denied) then an error is signalled.

Examples

```
      ⎕NEXISTS '/Users/Pete/Documents/temp'
0
      ⎕←⎕MKDIR '/Users/Pete/Documents/temp'
1
      ⎕←⎕MKDIR '/Users/Pete/Documents/temp'
FILE NAME ERROR: Directory exists
      ⎕←⎕MKDIR'/Users/Pete/Documents/temp'
     ^

      ⎕←⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
FILE NAME ERROR: Unable to create directory
("The system cannot find the path specified.")
      ⎕←⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
     ^

      ⎕←2 ⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
1

      ⎕←⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
FILE NAME ERROR: /Users/Pete/Documents/temp/t1/t2: Already exists
      ⎕←⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
        ^

      ⎕←(⎕MKDIR⍠'Unique'1)'/Users/Pete/Documents/temp/t1/t2'
/Users/Pete/Documents/temp/t1/t2djM0X8

      ⊢⎕MKDIR'temp1' 'temp2'
1 1
```

> **Note**
>
> When multiple names are specified they are processed in the order given. If an error occurs at any point whilst creating directories, processing will immediately stop and an error will be signalled. The operation is not atomic; some directories may be created before this happens. In the event of an error there will be no result and therefore no indication of how many directories were created before the error occurred.

6.4.80 Migration Level ⎕ML

⎕ML determines the degree of migration of the Dyalog APL language towards IBM's APL2. Setting this variable to other than its default value of 1 changes the interpretation of certain symbols and language constructs. ⎕ML has Namespace scope.

⎕ML←0		Original Native Dyalog
⎕ML←1	Z←∈R	Monadic '∈' is interpreted as 'enlist' rather than 'type'.
⎕ML←2	Z←↑R	Monadic '↑' is interpreted as 'first' rather than 'mix'.
	Z←⊃R	Monadic '⊃' is interpreted as 'mix' rather than 'first'.
	Z←≡R	Monadic '≡' returns a positive rather than a negative value, if its argument has non-uniform depth.
⎕ML←3	R←X⊂[K]Y	Dyadic '⊂' follows the APL2 (rather than the original Dyalog APL) convention.
	⎕TC	The order of the elements of ⎕TC is the same as in APL2.

Subsequent versions of Dyalog APL may provide further migration levels.

Examples

```
      X←2(3 4)

      ⎕ML←0
      ∈X
0  0 0
      ↑X
2 0
3 4
      ⊃X
2
      ≡X
¯2
```

```
      ⎕ML←1
      ∈X
2 3 4
      ↑X
2 0
3 4
      ⊃X
2
      ≡X
¯2
```

```
      ⎕ML←2
      ∈X
2 3 4
      ↑X
2
      ⊃X
2 0
3 4
      ≡X
2
```

6.4.81 Set Monitor {R}←X ⎕MONITOR Y

`Y` must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator.

Note that `⎕MONITOR` does not apply to dfns or dops.

`X` must be a simple non-negative integer scalar or vector. `R` is a simple integer vector of non-negative elements.

`X` identifies the numbers of lines in the function or operator named by `Y` on which a monitor is to be placed. Numbers outside the range of line numbers in the function or operator (other than `0`) are ignored. The number `0` indicates that a monitor is to be placed on the function or operator as a whole. The value of `X` is independent of `⎕IO`.

`R` is a vector of numbers on which a monitor has been placed in ascending order. The result is suppressed unless it is explicitly used or assigned. `R` will be empty for dfns and dops.

The effect of `⎕MONITOR` is to accumulate timing statistics for the lines for which the monitor has been set. See *Section 6.4.82* for details.

Examples

```
      +(0,⍳10) ⎕MONITOR 'FOO'
0 1 2 3 4 5
```

Existing monitors are cancelled before new ones are set:

```
      +1 ⎕MONITOR 'FOO'
1
```

All monitors may be cancelled by supplying an empty vector:

```
      ⍬ ⎕MONITOR 'FOO'
```

Monitors may be set on a locked function or operator, but no information will be reported. Monitors are saved with the workspace.

6.4.82 Query Monitor R←⎕MONITOR Y

`Y` must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator.

Note that `⎕MONITOR` does not apply to dfns or dops.

`R` is a simple non-negative integer matrix of 5 columns with one row for each line in the function or operator `Y` which has the monitor set, giving:

Column 1	Line number
Column 2	Number of times the line was executed
Column 3	CPU time in milliseconds
Column 4	Elapsed time in milliseconds
Column 5	Reserved

The value of `0` in column one indicates that the monitor is set on the function or operator as a whole. `R` will be empty for dfns and dops.

EXAMPLE

```
      ∇ FOO
[1]   A←?25 25⍴100
[2]   B←⍋A
[3]   C←⍋B
[4]   R1←⌊0.5+A+.×B
[5]   R2←A=C
      ∇

      (0,⍳5) ⎕MONITOR 'FOO' ⍝ Set monitor

      FOO                   ⍝ Run function

      ⎕MONITOR 'FOO'        ⍝ Monitor query
0 1 1418 1000 0
1 1   83    0 0
2 1  400    0 0
3 1  397    0 0
4 1  467 1000 0
5 1  100    0 0
```

6.4.83 Name Association {R}←{X}⎕NA Y

⎕NA provides access from APL to compiled functions within a library. A library is implemented according to the Operating System as follows:

```
- a Dynamic Link Library(DLL) under Windows
- a Shared Library (.so or .dylib) under Linux or macOS
- a static library (.a) under AIX
```

A DLL[20] is a collection of functions typically written in C (or C++) each of which may take arguments and return a result.

Instructional examples using ⎕NA can be found in the supplied workspace `quadna`.

The DLL may be part of the standard operating system software, a library purchased from a third party supplier, or one that you have written yourself.

The right argument Y is a character vector that identifies the name and syntax of the function to be associated. The left argument X is a character vector that contains the name to be associated with the external function. If the ⎕NA is successful, a function

20 Classic Edition: - translated to/from ANSI

(name class 3) is established in the active workspace with name `X`. If `X` is omitted, the name of the external function itself is used for the association.

The shy result `R` is a character vector containing the name of the external function that was fixed.

For example, `math.dll` might be a library of mathematical functions containing a function `divide`.

In a compiled language such as C, the types of arguments and results of functions must be declared explicitly. Typically, these types will be published with the documentation that accompanies the DLL. For example, function `divide` might be declared:

```
double divide(int32_t, int32_t);
```

which means that it expects two long (4-byte) integer arguments and returns a double (8-byte) floating point result. Notice the correspondence between the C declaration and the right argument of ⎕NA:

```
    C:              double     divide        (int32_t,  int32_t);
    APL:  'div' ⎕NA 'F8     math|divide        I4         I4 '
```

So to associate the APL name `div` with this external function:

```
      'div' ⎕NA 'F8 math|divide I4 I4'
```

where `F8` and `I4`, specify the types of the result and arguments expected by `divide`. The association has the effect of establishing a new function: `div` in the workspace, which when called, passes its arguments to `divide` and returns the result.

```
      )FNS
div
      div 10 4
2.5
```

It is imperative that care be taken when coding type declarations. A DLL *cannot* check types of data passed from APL. A wrong type declaration will lead to erroneous results or may even cause the workspace to become corrupted and crash. During development, you may wish to prevent this happening. See *Installation/Configuration: Configuration Parameters*.

The full syntax for the right argument of ⎕NA is:

```
    [result] library|function [arg1] [arg2] ...
```

Note that functions associated with DLLs are never dyadic. All arguments are passed as items of a (possibly nested) vector on the right of the function.

Locating the DLL

The DLL may be specified using a full pathname, file extension, and function type.

Be aware

A 32-bit interpreter can only load 32-bit DLLs/shared libraries; a 64-bit interpreter can only load 64-bit DLLs/shared libraries.

If a DLL/shared library has a missing dependency, the error generated by the operating system, and therefore reported by Dyalog will suggest that the DLL/shared library that was explicitly called in the `⎕NA` call is missing.

Pathname

APL uses the `LoadLibrary()` system function under Windows or `dlopen()` under UNIX, Linux and macOS to load the DLL. If a full or relative pathname is omitted, these functions search a list of directories determined by the operating system. This list always includes the directory which contains the Dyalog program, and on all non-Windows platforms, $DYALOG/lib. For further details, see the operating system documentation about these functions.

Alternatively, a full or relative pathname may be supplied in the usual way:

```
      ⎕NA'... c:\mydir\mydll|foo ...'
```

Errors

If the specified DLL (or a dependent DLL) fails to load it will generate:

```
FILE ERROR 2 No such file or directory
```

It is frequently the case that this error is a result of a missing dependency; operating systems do not return error codes which allow the interpreter to generate a more specific error.

If the DLL loads successfully, but the specified library function is not accessible, it will generate:

```
VALUE ERROR
```

File Extension

Under Windows, if the file extension is omitted, **.dll** is assumed. Note that some DLLs are in fact **.exe** files, and in this case the extension must be specified explicitly:

```
      NA'... mydll.exe|foo ...'
```

libc.a on Non-Windows Platforms

On non-Windows platforms many of the most useful system library functions appear in `libc.a`. The `quadna` workspace includes the function `NonWindows.Setup` which has code which will locate `libc.a` on each platform.

Data Type Coding Scheme

The type coding scheme introduced above is of the form:

```
[direction] [special] type [width] [array][[count]]
```

The options are summarised in the following table and their functions detailed below.

Description	Symbol	Meaning
Direction	`<`	Pointer to array *input* to DLL function.
	`>`	Pointer to array *output* from DLL
	`=`	Pointer to input/output
Special	`0`	Null-terminated string.
	`#`	Byte-counted string
Type	`I`	int
	`U`	unsigned int
	`C`	char
	`T`	char [1]
	`F`	float
	`D`	decimal
	`J`	complex
	`P`	uintptr-t [21]
	`A`	APL array
	`Z`	APL array with header (as passed to a TCP/IP socket)
Width	`1`	1-byte
	`2`	2-byte
	`4`	4-byte
	`8`	8-byte
	`16`	16-byte (128-bit)
Array	`[n]`	Array of length *n* elements
	`[]`	Array, length determined at call-time
Structure	`{...}`	Structure.
Count	`[int]`	Rather than explicitly declaring multiple adjacent occurrences of the same the count option may be used

21 equivalent to U4 on 32-bit versions and U8 on 64-bit versions

In the Classic Edition, `C` specifies untranslated character, whereas `T` specifies that the character data will be translated to/from `⎕AV`.

In the Unicode Edition, C and T are identical (no translation of character data is performed) except that for C the default width is 1 and for T the default width is "wide" (2 bytes under Windows, 4 bytes under UNIX, Linux or macOS).

The use of T with default width is recommended to ensure portability between Editions.

Direction

C functions accept data arguments either by *value* or by *address*. This distinction is indicated by the presence of a '`*`' or '`[]`' in the argument declaration:

```
int  num1;       // value of num1 passed.
int *num2;       // Address of num2 passed.
int num3[];      // Address of num3 passed.
```

An argument (or result) of an external function of type pointer, must be matched in the `⎕NA` call by a declaration starting with one of the characters: `<`, `>`, or `=`.

In C, when an address is passed, the corresponding value can be used as either an *input* or an *output* variable. An output variable means that the C function overwrites values at the supplied address. Because APL is a call-by-value language, and doesn't have pointer types, we accommodate this mechanism by distinguishing output variables, and having them returned explicitly as part of the result of the call.

This means that where the C function indicates a *pointer type*, we must code this as starting with one of the characters: `<`, `>` or `=`.

`<`	indicates that the address of the argument will be used by C as an input variable and values at the address will *not* be over-written.
`>`	indicates that C will use the address as an output variable. In this case, APL must allocate an output array over which C can write values. After the call, this array will be included in the nested result of the call to the external function.
`=`	indicates that C will use the address for both input and output. In this case, APL duplicates the argument array into an output buffer whose address is passed to the external function. As in the case of an output only array, the newly modified copy will be included in the nested result of the call to the external function.

EXAMPLES

`<I2`	Pointer to 2-byte integer - *input* to external function
`>C`	Pointer to character *output* from external function.
`=T`	Pointer to character *input* to and *output* from function.
`=A`	Pointer to APL array *modified* by function.

Special

In C it is common to represent character strings as *null-terminated* or *byte counted* arrays. These special data types are indicated by inserting the symbol `0` (null-terminated) or `#` (byte counted) between the direction indicator (`<`, `>`, `=`) and the type (`T` or `C`) specification. For example, a pointer to a null-terminated input character string is coded as `<0T[]`, and an output one coded as `>0T[]`.

Note that while appending the array specifier '`[]`' is formally correct, because the presence of the special qualifier (`0` or `#`) *implies* an array, the '`[]`' may be omitted: `<0T`, `>0T`, `=#C`, etc.

Note also that the 0 and # specifiers may be used with data of all types (excluding `A` and `Z`) and widths. For example, in the Classic Edition, `<0U2` may be useful for dealing with Unicode.

Type

The data type of the argument may be one of the following characters and may be specified in lower or upper case:

Code	Type	Description
`I`	Integer	The value is interpreted as a 2s complement signed integer
`U`	Unsigned integer	The value is interpreted as an unsigned integer
`C`	Character	The value is interpreted as a character. In the Unicode Edition, the value maps directly onto a Unicode code point. In the Classic Edition, the value is interpreted as an index into `⎕AV` . This means that `⎕AV` *positions* map onto corresponding ANSI *positions* . For example, with `⎕IO=0` : `⎕AV[35] = 's'` , maps to `ANSI[35] = '`
`T`	Translated character	The value is interpreted as a character. In the Unicode Edition, the value maps directly onto a Unicode code point. In the Classic Edition, the value is *translated* using standard Dyalog `⎕AV` to ANSI translation. This means that `⎕AV` *characters* map onto corresponding ANSI *characters* . For example, with `⎕IO=0` : `⎕AV[35] = 's'` maps to `ANSI[115] = 's'`
`UTF`	Unicode encoded	`>UTF8[]` will translate to a UTF-8 encoded string `<UTF16[]` will translate from a UTF-16LE encoded string
`F`	Float	The value is interpreted as an IEEE 754-2008 binary64 floating point number
`D`	Decimal	The value is interpreted as an IEEE 754-2008 decimal128 floating point number (DPD format on AIX, BID format on other platforms)
`J`	Complex	
`P`	uintptr-t	This is equivalent to U4 on 32-bit versions and U8 on 64-bit versions
`∇`	Function pointer	This allows the passing of an APL function for the function to call
`A`	APL array	This is the same format as is used to transmit APL arrays to an Auxiliary Processor (AP)
`Z`	APL array with header	This is the same format as is used to transmit APL arrays over TCP/IP Sockets

Width

The type specifier may be followed by the width of the value in bytes. For example:

`I4`	4-byte signed integer.
`U2`	2-byte unsigned integer.
`F8`	8-byte floating point number.
`F4`	4-byte floating point number.
`D16`	16-byte decimal floating-point number

Type	Possible values for Width	Default value for Width
`I`	1, 2, 4, 8	4
`U`	1, 2, 4, 8	4
`C`	1,2,4	1
`T`	1,2,4	wide character(see below)
`UTF`	8,16	none
`F`	4, 8	8
`D`	16	16
`J`	16	16
`P`	Not applicable	
`▽`	Not applicable	
`A`	Not applicable	
`Z`	Not applicable	

In the Unicode Edition, the default width is the width of a *wide character* according to the convention of the host operating system. This translates to T2 under Windows and T4 under UNIX, Linux or macOS.

Note that 32-bit versions can support 64-bit integer *arguments*, but not 64-bit integer *results*.

EXAMPLES

`I2`	16-bit integer
`<I4`	Pointer to input 4-byte integer
`U`	Default width unsigned integer
`=F4`	Pointer to input/output 4-byte floating point number.

Arrays

Arrays are specified by following the basic data type with `[n]` or `[]`, where `n` indicates the number of elements in the array. In the C declaration, the number of elements in an array may be specified explicitly at compile time, or determined dynamically at runtime. In the latter case, the size of the array is often passed along with the array, in a separate argument. In this case, `n`, the number of elements is omitted from the specification. Note that C deals only in scalars and rank 1 (vector) arrays.

```
int vec[10];                // explicit vector length.
unsigned size, list[];      // undetermined length.
```

could be coded as:

`I[10]`	vector of 10 ints.
`U U[]`	unsigned integer followed by an array of unsigned integers.

Confusion sometimes arises over a difference in the declaration syntax between C and ⎕NA. In C, an argument declaration may be given to receive a pointer to either a single scalar item, or to the first element of an array. This is because in C, the address of an array is deemed to be the address of its first element.

```
   void foo (char *string);
   char ch = 'a', ptr = "abc";
   foo(&ch);// call with address of scalar.
   foo(ptr);// call with address of array.
```

However, from APL's point of view, these two cases are distinct and if the function is to be called with the address of (pointer to) a *scalar*, it must be declared: `'<T'`. Otherwise, to be called with the address of an *array*, it must be declared: `'<T[]'`. Note that it is perfectly acceptable in such circumstances to define more than one name association to the same DLL function specifying different argument types:

```
      'FooScalar'⎕NA'mydll|foo <T'    ⋄ FooScalar'a'
      'FooVector'⎕NA'mydll|foo <T[]' ⋄ FooVector'abc'
```

Structures

Arbitrary data structures, which are akin to nested arrays, are specified using the symbols `{}`. For example, the code `{F8 I2}` indicates a structure comprised of an 8-byte *float* followed by a 2-byte *int*. Furthermore, the code `<{F8 I2}[3]` means an input pointer to an array of 3 such structures.

For example, this structure might be defined in C thus:

```
typedef struct
{
    double  f;
    short   i;
} mystruct;
```

A function defined to receive a count followed by an *input* pointer to an array of such structures:

```
void foo(unsigned count, mystruct *str);
```

An appropriate `⎕NA` declaration would be:

```
      ⎕NA'mydll.foo U <{F8 I2}[]'
```

A call on the function with two arguments - a count followed by a vector of structures:

```
      foo 4,⊂(1.4 3)(5.9 1)(6.5 2)(0 0)
```

Notice that for the above call, APL converts the two Boolean `(0 0)` elements to an 8-byte float and a 2-byte int, respectively.

Note that if the C compiler would add extra space for alignment within a structure the `⎕NA` syntax will need to code that explicitly. For example:

```
typedef struct
{
    short i;
    /* most C compilers would add 6 bytes of alignment here */
    double d;
} mystruct;
```

An appropriate ⎕NA declaration would be:

```
      ⎕NA'mydll.foo U <{I2  {I1[6]} F8}[]'
```

A call on the function with two arguments - a count followed by a vector of structures:

```
      pad←⊂6⍴0
      foo 4,⊂(3 pad 1.4)(1 pad 5.9 )(2 pad 6.5 )(0 pad 0)
```

A library designer tries to avoid defining structures that induce padding.

Count

If a definition includes multiple adjacent occurrences of the same item, the count syntax may be used rather than explicitly repeating the same definition.

For example:

`>I8[3]` rather than `>I8 >I8 >I8`

`{I8 U8 I8 P}[2]` rather than `{I8 U8 I8 P} {I8 U8 I8 P}`

Using a Function

A DLL function may or may not return a result, and may take zero or more arguments. This syntax is reflected in the coding of the right argument of ⎕NA. However, notice that the corresponding associated APL function is a result-returning niladic (if it takes no arguments) or monadic function. It cannot be dyadic and it must *always* return a vector result - a null one if there is no output from the DLL function. See Result Vector section below. Examples of the various combinations are:

DLL function Non-result-returning

```
⎕NA    'mydll|fn1'          ⍝ Niladic
⎕NA    'mydll|fn2 <0T'      ⍝ Monadic - 1-element arg
⎕NA    'mydll|fn3 =0T <0T'  ⍝ Monadic - 2-element arg
```

DLL function Result-returning

```
⎕NA 'I4 mydll|fn4'           ⍝ Niladic
⎕NA 'I4 mydll|fn5 F8'        ⍝ Monadic - 1-element arg
⎕NA 'I4 mydll|fn6 >I4[] <0T' ⍝ Monadic - 2-element arg
```

When the external function is called, the number of elements in the argument must match the number defined in the ⎕NA definition. Using the examples above:

```
      fn1                    ⍝ Niladic Function.
      fn2, ⊂'Single String'  ⍝ 1-element arg
      fn3 'This' 'That'      ⍝ 2-element arg
```

Note in the second example, that you must enclose the argument string to produce a single item (nested) array in order to match the declaration. Dyalog converts the type of a numeric argument if necessary, so for example in `fn5` defined above, a Boolean value would be converted to double floating point (F8) prior to being passed to the DLL function.

Multi-Threading

Appending the '&' character to the function name causes the external function to be run in its own system thread. For example:

```
      ⎕NA'... mydll|foo& ...'
```

This means that other APL threads can run concurrently with the one that is calling the ⎕NA function.

Name Mangling

C++ and some other languages will by default mangle (or decorate) function names which are exported from a DLL file. The given external function name must exactly match the exported name, either by matching the name mangling or by ensuring the names exported from the library are not mangled.

Call by Ordinal Number

Under Windows, a DLL may associate an *ordinal number* with any of its functions. This number may then be used to call the function as an alternative to calling it by name. Using ⎕NA to call by ordinal number uses the same syntax but with the function name replaced with its ordinal number. For example:

```
      ⎕NA'... mydll|57 ...'
```

Pointer Arguments

When passing pointer arguments there are three cases to consider.

< Input pointer

In this case you must supply the data array itself as argument to the function. A pointer to its first element is then passed to the DLL function.

```
      fn2 ⊂'hello'
```

> Output pointer

Here, you must supply the **number of elements** that the output will need in order for APL to allocate memory to accommodate the resulting array.

```
      fn6 10 'world'  ⍝ 1st arg needs space for 10 ints.
```

Note that if you were to reserve fewer elements than the DLL function actually used, the DLL function would write beyond the end of the reserved array and may cause the interpreter to crash with a System Error (syserror 999 on Windows or SIGSEGV on UNIX, Linux or macOS).

= **Input/Output**

As with the input-only case, a pointer to the first element of the argument is passed to the DLL function. The DLL function then overwrites some or all of the elements of the array, and the new value is passed back as part of the result of the call. As with the output pointer case, if the input array were too short, so that the DLL wrote beyond the end of the array, the interpreter would almost certainly crash.

```
      fn3 '.....' 'hello'
```

Specifying Pointers Explicitly

⎕NA syntax enables APL to pass arguments to DLL functions by *value* or *address* as appropriate. For example if a function requires an integer followed by a *pointer* to an integer:

```
void fun(int value, int *addr);
```

You might declare and call it:

```
      ⎕NA'mydll|fun I <I' ⋄ fun 42 42
```

The interpreter passes the *value* of the first argument and the *address* of the second one.

Two common cases occur where it is necessary to pass a pointer explicitly. The first is if the DLL function requires a *null pointer*, and the second is where you want to pass on a pointer which itself is a result from a DLL function.

In both cases, the pointer argument should be coded as `P`. This causes APL to pass the pointer unchanged, *by value*, to the DLL function.

In the previous example, to pass a null pointer, (or one returned from another DLL function), you must code a separate ⎕NA definition.

```
      'fun_null'⎕NA'mydll|fun I P' ⋄ fun_null 42 0
```

Now APL passes the *value* of the second argument (in this case 0 - the null pointer), rather than its address.

Note that by using P, which is 4-byte for 32-bit processes and 8-byte for 64-bit processes, you will ensure that the code will run unchanged under both 32-bit and 64-bit versions of Dyalog APL.

Result Vector

In APL, a function cannot overwrite its arguments. This means that any output from a DLL function must be returned as part of the explicit result, and this includes output via 'output' or 'input/output' pointer arguments.

The general form of the result from calling a DLL function is a nested vector. The first item of the result is the defined explicit result of the external function, and subsequent items are implicit results from output, or input/output pointer arguments.

The length of the result vector is therefore: 1 (if the function was declared to return an explicit result) + the number of output or input/output arguments.

⎕NA Declaration	Result	Output Arguments	Result Length
`mydll|fn1`	0		0
`mydll|fn2 <0T`	0	0	0
`mydll|fn3 =0T <0T`	0	1 0	1
`I4 mydll|fn4`	1		1
`I4 mydll|fn5 F8`	1	0	1
`I4 mydll|fn6 >I4[] <0T`	1	1 0	2

Note that the result vector from a function that is declared `void()` and has no output parameters is `0` (zilde).

As a convenience, if the result would otherwise be a 1-item vector, it is disclosed. Using the third example above:

```
      ⍴fn3 '.....' 'abc'
5
```

`fn3` has no explicit result; its first argument is input/output pointer; and its second argument is input pointer. Therefore as the length of the result would be 1, it has been disclosed.

64 bit integer results

When a 64 bit integer result is returned it is converted into 128 bit decimal floating point, because this is the only APL data type that can fully preserve all 64 bits of the result. If you wish to perform arithmetic with this value, you must set ⎕FR to 1287 in order to preserve the same precision. If this is not done then the precision will be 53 bits which might not be enough.

Callbacks (∇)

Currently, support for a ⎕NA function to call an APL function is limited to the use of the NAG (National Algorithms Group) library of functions. This library is a FORTRAN library and FORTRAN passes arguments by reference (address) rather than by value. The expression:

```
∇f8←(P P P P)
```

declares a callback function that returns a double and takes 4 pointer arguments. The result can be any of the normal results. It is not possible to return a pointer. The arguments can be from 0 to 16 P values.

The argument when passed can be the name of an APL function or the ⎕OR of a function.

The function when called can then decode the pointer arguments appropriately using a ⎕NA of `MEMCPY()`.

ANSI /Unicode Versions of Library Calls

Under Windows, most library functions that take character arguments, or return character results have two forms: one Unicode (Wide) and one ANSI. For example, a function such as `MessageBox()`, has two forms `MessageBoxA()` and `MessageBoxW()`. The `A` stands for ANSI (1-byte) characters, and the `W` for wide (2-byte Unicode) characters.

It is essential that you associate the form of the library function that is appropriate for the Dyalog Edition you are using, that is, `MessageBoxA()` for the Classic Edition, but `MessageBoxW()` for the Unicode Edition.

Whilst this is convenient it is not complete. It is adequate for character arrays that consist of characters from UCS-2 (that is, those that will fit in an array with a □DR of 80 or 160). If a more complete support is required then the W form of the function would be required and explicit use of UTF16 specified.

To simplify writing portable code for both Editions, you may specify the character * instead of A or W at the end of a function name. This will be replaced by A in the Classic Edition and W in the Unicode Edition.

The default name of the associated function (if no left argument is given to □NA), will be without the trailing letter (MessageBox).

Type Definitions

The C language encourages the assignment of defined names to primitive and complex data types using its #define and typedef mechanisms. Using such abstractions enables the C programmer to write code that will be portable across many operating systems and hardware platforms.

Windows software uses many such names and Microsoft documentation will normally refer to the type of function arguments using defined names such as HANDLE or LPSTR rather than their equivalent C primitive types: int or char*.

It is beyond the scope of this manual to list *all* the Microsoft definitions and their C primitive equivalents, and indeed, DLLs from sources other than Microsoft may well employ their own distinct naming conventions.

In general, you should consult the documentation that accompanies the DLL in order to convert typedefs to primitive C types and thence to □NA declarations. The documentation may well refer you to the 'include' files which are part of the Software Development Kit, and in which the types are defined.

The following table of some commonly encountered Windows typedefs and their □NA equivalents might prove useful.

Windows typedef	NA equivalent
HWND	P
HANDLE	P
GLOBALHANDLE	P
LOCALHANDLE	P
DWORD	U4
WORD	U2
BYTE	U1
LPSTR	=0T[] (note 1)
LPCSTR	<0T[] (note 2)
WPARAM	U (note 3)
LPARAM	U4 (note 3)
LRESULT	I4
BOOL	I
UINT	U
ULONG	U4
ATOM	U2
HDC	P
HBITMAP	P
HBRUSH	P
HFONT	P
HICON	P
HMENU	P
HPALETTE	P
HMETAFILE	P
HMODULE	P
HINSTANCE	P
COLORREF	{U1[4]}

Windows typedef	⎕NA equivalent
POINT	`{I I}`
POINTS	`{I2 I2}`
RECT	`{I I I I}`
CHAR	`T or C`

Notes

1. `LPSTR` is a pointer to a null-terminated string. The definition does not indicate whether this is input or output, so the safest coding would be `=0T[]` (providing the vector you supply for input is long enough to accommodate the result). You may be able to improve simplicity or performance if the documentation indicates that the pointer is 'input only' (`<0T[]`) or 'output only' (`>0T[]`). See Direction above.
2. `LPCSTR` is a pointer to a *constant* null-terminated string and therefore coding `<0T[]` is safe.

3. `WPARAM` is an unsigned value, `LPARAM` is signed. They are 32 bit values in a 32-bit APL, and 64-bit in a 64 bit APL. You should consult the documentation for the specific function that you intend to call to determine what type they represent

4. The use of type T with default width ensures portability of code between Classic and Unicode Editions. In the Classic Edition, T (with no width specifier) implies 1-byte characters which are translated between ⎕AV and ASCII, while in the Unicode Edition, T (with no width specifier) implies 2-byte (Unicode) characters.

The Dyalog DLL

The Dyalog DLL (see *Installation/Configuration: Runtime Applications And Components*) contains three functions: `MEMCPY`, `STRNCPY` and `STRLEN`.

MEMCPY

`MEMCPY` is an extremely versatile function used for moving arbitrary data between memory buffers.

Its C definition is:

```
void *MEMCPY(      // copy memory
    void *to,      // target address
    void *fm,      // source address
    size_t size    // number of bytes to copy
);
```

MEMCPY copies size bytes starting from source address fm, to destination address to. The source and destination areas should not overlap; if they do the behaviour is undefined and the result is the first argument.

MEMCPY's versatility stems from being able to associate to it using many different type declarations.

Example

Suppose a global buffer (at address: addr) contains (numb) double floating point numbers. To copy these to an APL array, we could define the association:

```
      'doubles' ⎕NA 'dyalog32|MEMCPY >F8[] I4 U4'
      doubles numb addr (numb×8)
```

Notice that:

- As the first argument to doubles is an output argument, we must supply the number of elements to reserve for the output data.
- MEMCPY is defined to take the number of *bytes* to copy, so we must multiply the number of elements by the element size in bytes.

Example

Suppose that a database application requires that we construct a record in global memory prior to writing it to file. The record structure might look like this:

```
typedef struct {
    int empno;     // employee number.
    float salary;  // salary.
    char name[20]; // name.
} person;
```

Then, having previously allocated memory (addr) to receive the record, we can define:

```
      'prec' ⎕NA 'dyalog64|MEMCPY P <{P F4 T[20]} P'
      prec addr(99 12345.60 'Charlie Brown')(4+4+20)
```

STRNCPY

STRNCPY is used to copy null-terminated strings between memory buffers.

Its C definition is:

```
void *STRNCPY(  // copy null-terminated string
    char *to,   // target address
    char *fm,   // source address
    size_t size // MAX number of chars to copy
);
```

STRNCPY copies a maximum of size characters from the null-terminated source string at address fm, to the destination address to. If the source and destination strings overlap, the result is the first argument.

If the source string is shorter than size, a null character is appended to the destination string.

If the source string (including its terminating null) is longer than size, only size characters are copied and the resulting destination string is not null-terminated

EXAMPLE

Suppose that a database application returns a pointer (addr) to a structure that contains two (max 20-char) null-terminated strings.

```
typedef struct {     // null-terminated strings:
    char first[20];  // first name (max 19 chars + 1 null).
    char last[20];   // last name. (max 19 chars + 1 null).
} name;
```

To copy the names *from* the structure:

```
      'get'⎕NA'dyalog64|STRNCPY >0T1[] P U4'
      get 20 addr 20
Charlie
      get 20 (addr+20) 20
Brown
```

Note that (as this is a 64-bit example), ⎕FR must be 1287 for the addition to be reliable.

To copy data *from* the workspace *into* an already allocated (new) structure:

```
      'put'⎕NA'dyalog32|STRNCPY I4 <0T[] U4'
      put new 'Bo' 20
      put (new+4) 'Peep' 20
```

Notice in this example that you must ensure that names no longer than 19 characters are passed to `put`. More than 19 characters would not leave `STRNCPY` enough space to include the trailing null, which would probably cause the application to fail.

STRNCPYA

This is a synonym for `STRNCPY`. It is there so that `STRNCPY*` (on Microsoft Windows) selects between `STRNCPYA` and `STRNCPYW`.

STRNCPYW

This is a cover for the C standard function `wcsncpy()`. It is named this way so that (on Microsoft Windows) `STRNCPY*` will behave helpfully.

STRLEN

`STRLEN` calculates the length of a C string (a 0-terminated string of bytes in memory). Its C declaration is:

```
size_t STRLEN(       // calculate length of string
    const char *s    // address of string
);
```

Example

Suppose that a database application returns a pointer (`addr`) to a null-terminated string and you do not know the upper bound on the length of the string.

To copy the string into the workspace:

```
      'len'⎕NA'P dyalog32|STRLEN P'
      'cpy'⎕NA'dyalog32|MEMCPY >T[] P P'
      cpy l addr (l←len addr)
Bartholemew
```

Examples

The following examples all use functions from the Microsoft Windows `user32.dll`.

This DLL should be located in a standard Windows directory, so you should not normally need to give the full path name of the library. However if trying these examples results in the error message `FILE ERROR 1 No such file or directory`, you must locate the DLL and supply the full path name (and possibly extension).

GetCaretBlinkTime()

The Windows function `GetCaretBlinkTime` retrieves the caret blink rate. It takes no arguments and returns an unsigned *int* and is declared as follows:

```
UINT GetCaretBlinkTime(void);
```

The following statements would provide access to this routine through an APL function of the same name.

```
      ⎕NA 'U user32|GetCaretBlinkTime'
      GetCaretBlinkTime
530
```

The following statement would achieve the same thing, but using an APL function called `BLINK`.

```
      'BLINK' ⎕NA 'U user32|GetCaretBlinkTime'
      BLINK
530
```

SetCaretBlinkTime()

The Microsoft Windows function `SetCaretBlinkTime` sets the caret blink rate. It takes a single unsigned `int` argument, does not return a result and is declared as follows:

```
void SetCaretBlinkTime(UINT);
```

The following statements would provide access to this routine through an APL function of the same name:

```
      ⎕NA 'user32|SetCaretBlinkTime U'
      SetCaretBlinkTime 1000
```

MessageBox()

The Microsoft Windows function `MessageBox` displays a standard dialog box on the screen and awaits a response from the user. It takes 4 arguments. The first is the window handle for the window that owns the message box. This is declared as an unsigned *int*. The second and third arguments are both pointers to null-terminated strings containing the message to be displayed in the Message Box and the caption to be used in the window title bar. The 4th argument is an unsigned *int* that specifies the Message Box type. The result is an *int* which indicates which of the buttons in the message box the user has pressed. The function is declared as follows:

```
int MessageBox(HWND, LPCSTR, LPCSTR, UINT);
```

The following statements provide access to this routine through an APL function of the same name. Note that the 2nd and 3rd arguments are both coded as input pointers to type T null-terminated character arrays which ensures portability between Editions.

```
      ⎕NA 'I user32|MessageBox* P <0T <0T U'
```

The following statement displays a Message Box with a stop sign icon together with 2 push buttons labelled OK and Cancel (this is specified by the value 19).

```
      MessageBox 0 'Message' 'Title' 19
```

The function works equally well in the Unicode Edition because the `<0T` specification is portable.

```
      MessageBox 0 'Το Μήνυμα' 'Ο Τίτλος' 19
```

Note that a simpler, portable (and safer) method for displaying a Message Box is to use Dyalog APL's primitive `MsgBox` object.

FindWindow()

The Microsoft Windows function `FindWindow` obtains the window handle of a window which has a given character string in its title bar. The function takes two arguments. The first is a pointer to a null-terminated character string that specifies the window's

class name. However, if you are not interested in the class name, this argument should be a NULL pointer. The second is a pointer to a character string that specifies the title that identifies the window in question. This is an example of a case described above where two instances of the function must be defined to cater for the two different types of argument. However, in practice this function is most often used without specifying the class name. The function is declared as follows:

```
HWND FindWindow(LPCSTR, LPCSTR);
```

The following statement associates the APL function `FW` with the second variant of the FindWindow call, where the class name is specified as a NULL pointer. To indicate that APL is to pass the *value* of the NULL pointer, rather than its address, we need to code this argument as `I4`.

```
      'FW' ⎕NA 'P user32|FindWindow* P <0T'
```

To obtain the handle of the window entitled "CLEAR WS - Dyalog APL/W":

```
      ⎕←HNDL←FW 0 'CLEAR WS - Dyalog APL/W'
59245156
```

GetWindowText()

The Microsoft Windows function `GetWindowText` retrieves the caption displayed in a window's title bar. It takes 3 arguments. The first is an unsigned *int* containing the window handle. The second is a pointer to a buffer to receive the caption as a null-terminated character string. This is an example of an output array. The third argument is an *int* which specifies the maximum number of characters to be copied into the output buffer. The function returns an *int* containing the actual number of characters copied into the buffer and is declared as follows:

```
int GetWindowText(HWND, LPSTR, int);
```

The following associates the "`GetWindowText`" DLL function with an APL function of the same name. Note that the second argument is coded as "`>0T`" indicating that it is a pointer to a character output array.

```
      ⎕NA 'I user32|GetWindowText* P >0T I'
```

Now change the Session caption using `)WSID` :

```
      )WSID MYWS
was CLEAR WS
```

Then retrieve the new caption (max length 255) using window handle `HNDL` from the previous example:

```
      ]display GetWindowText HNDL 255 255
.→-------------------------.
|    .→-----------------.  |
| 19 |MYWS - Dyalog APL/W|  |
|    '-------------------'  |
'∈-------------------------'
```

There are three points to note.

1. Firstly, the number 255 is supplied as the second argument. This instructs APL to allocate a buffer large enough for a 255-element character vector into which the DLL routine will write.
2. Secondly, the result of the APL function is a nested vector of 2 elements. The first element is the result of the DLL function. The second element is the output character array.
3. Finally, notice that although we reserved space for 255 elements, the result reflects the length of the actual text (19).

An alternative way of coding and using this function is to treat the second argument as an input/output array.

For example:

```
      ⎕NA 'I User32|GetWindowText* P =0T I'

      ]display GetWindowText HNDL (255⍴' ') 255
.→-------------------------.
|    .→-----------------.  |
| 19 |MYWS - Dyalog APL/W|  |
|    '-------------------'  |
'∈-------------------------'
```

In this case, the second argument is coded as `=0T`, so when the function is called an array of the appropriate size must be supplied. This method uses more space in the workspace, although for small arrays (as in this case) the real impact of doing so is negligible.

GetCharWidth()

The function `GetCharWidth` returns the width of each character in a given range. Its first argument is a device context (handle). Its second and third arguments specify font positions (start and end). The third argument is the resulting integer vector that contains the character widths (this is an example of an output array). The function returns a Boolean value to indicate success or failure. The function is defined as follows. Note that this function is provided in the library: `gdi32.dll`.

```
BOOL GetCharWidth(HDC, UINT, UINT, LPINT);
```

The following statements provide access to this routine through an APL function of the same name:

```
      ⎕NA 'U4 gdi32|GetCharWidth* P U U >I[]'

     'Prin'⎕WC'Printer'

      ]display GetCharWidth ('Prin' ⎕WG 'Handle') 65 67 3
┌→──────────────┐
│   ┌→───────┐  │
│ 1 │50 50 50│  │
│   └~───────┘  │
└∊──────────────┘
```

Note: `'Prin'⎕WG'Handle'` returns a handle which is represented as a number. The number will be in the range (0 - *232] on a 32-bit version and (0 - 264*] on a 64-bit version. These can be passed to a P type parameter. Older versions used a 32-bit signed integer.

The `quadna` workspace

The following example from the supplied workspace: quadna.dws. `quadna` illustrates several techniques which are important in advanced ⎕NA programming. Function `DllVersion` returns the major and minor version number for a given DLL. Note that this example assumes that the computer is running the 64-bit version of Dyalog.

In advanced DLL programming, it is often necessary to administer memory outside APL's workspace. In general, the procedure for such use is:

1. Allocate global memory.

2. Lock the memory.
3. Copy any DLL input information from workspace into memory.
4. Call the DLL function.
5. Copy any DLL output information from memory to workspace.
6. Unlock the memory.
7. Free the memory.

Notice that steps 1 and 7 and steps 2 and 6 complement each other. That is, if you allocate global system memory, you must free it after you have finished using it. If you continue to use global memory without freeing it, your system will gradually run out of resources. Similarly, if you lock memory (which you must do before using it), then you should unlock it before freeing it. Although on some versions of Windows, freeing the memory will include unlocking it, in the interests of good style, maintaining the symmetry is probably a good thing.

```
    ∇ version←DllVersion file;Alloc;Free;Lock;Unlock;Size
                  ;Info;Value;Copy;size;hndl;addr;buff;ok
[1]
[2]  'Alloc'⎕NA'P kernel32|GlobalAlloc U4 P'
[3]   'Free'⎕NA'P kernel32|GlobalFree P'
[4]   'Lock'⎕NA'P kernel32|GlobalLock P'
[5] 'Unlock'⎕NA'U4 kernel32|GlobalUnlock P'
[6]
[7]  'Size'⎕NA'U4 version|GetFileVersionInfoSize* <0T >U4'
[8]  'Info'⎕NA'U4 version|GetFileVersionInfo*<0T U4 U4 P'
[9] 'Value'⎕NA'U4 version|VerQueryValue* P <0T >P >U4'
[10]
[11] 'Copy'⎕NA'dyalog64|MEMCPY >U4[] P P'
[12]
[13]  :If ×size←⊃Size file 0              ⍝ Size of info
[14]  :AndIf ×hndl←Alloc 0 size           ⍝ Alloc memory
[15]    :If ×addr←Lock hndl               ⍝ Lock memory
[16]      :If ×Info file 0 size addr      ⍝ Version info
[17]        ok buff size←Value addr'\' 0 0 ⍝ Version value
[18]        :If ok
[19]          buff←Copy(size÷4)buff size  ⍝ Copy info
[20]          version←(2/2*16)⊤⊃2↓buff    ⍝ Split version
[21]        :EndIf
[22]      :EndIf
[23]      ok←Unlock hndl                  ⍝ Unlock memory
[24]    :EndIf
[25]    ok←Free hndl                      ⍝ Free memory
[26]  :EndIf
    ∇
```

Lines [2-11] associate APL function names with the DLL functions that will be used.

Lines [2-5] associate functions to administer global memory.

Lines [7-9] associate functions to extract version information from a DLL.

Line[11] associates `Copy` with `MEMCPY` function from **dyalog64.dll**.

Lines [13-26] call the DLL functions.

Line [13] requests the size of buffer required to receive version information for the DLL. A size of 0 will be returned if the DLL does not contain version information.

Notice that care is taken to balance memory allocation and release:

On line [14], the :If clause is taken only if the global memory allocation is successful, in which case (and only then) a corresponding Free is called on line [25].

Unlock on line[23] is called if and only if the call to Lock on line [15] succeeds.

A result is returned from the function *only* if all the calls are successful Otherwise, the calling environment will sustain a `VALUE ERROR`.

More Examples

```
⎕NA'I4 advapi32 |RegCloseKey           P'
⎕NA'I4 advapi32 |RegCreateKeyEx*       P <0T U4 <0T U4 U4 P >P >U4'
⎕NA'I4 advapi32 |RegEnumValue*         P U4 >0T =U4 =U4 >U4 >0T =U4'
⎕NA'I4 advapi32 |RegOpenKey*           P <0T >P'
⎕NA'I4 advapi32 |RegOpenKeyEx*         P <0T U4 U4 >P'
⎕NA'I4 advapi32 |RegQueryValueEx*      P <0T =U4 >U4 >0T =U4'
⎕NA'I4 advapi32 |RegSetValueEx*        P <0T =U4 U4 <0T U4'
⎕NA'P  dyalog32 |STRNCPY               P P P'
⎕NA'P  dyalog32 |STRNCPYA              P P P'
⎕NA'P  dyalog32 |STRNCPYW              P P P'
⎕NA'P  dyalog32 |MEMCPY                P P P'
⎕NA'I4 gdi32    |AddFontResource*      <0T'
⎕NA'I4 gdi32    |BitBlt                P I4 I4 I4 I4 P I4 I4 U4'
⎕NA'U4 gdi32    |GetPixel              P I4 I4'
⎕NA'P  gdi32    |GetStockObject        I4'
⎕NA'I4 gdi32    |RemoveFontResource*   <0T'
⎕NA'U4 gdi32    |SetPixel              P I4 I4 U4'
⎕NA'   glu32    |gluPerspective        F8 F8 F8 F8'
⎕NA'I4 kernel32 |CopyFile*             <0T <0T I4'
⎕NA'P  kernel32 |GetEnvironmentStrings'
⎕NA'U4 kernel32 |GetLastError'
⎕NA'U4 kernel32 |GetTempPath*          U4 >0T'
⎕NA'P  kernel32 |GetProcessHeap'
⎕NA'I4 kernel32 |GlobalMemoryStatusEx ={U4 U4 U8 U8 U8 U8 U8 U8}'
⎕NA'P  kernel32 |HeapAlloc             P U4 P'
⎕NA'I4 kernel32 |HeapFree              P U4 P'
⎕NA'   opengl32 |glClearColor          F4 F4 F4 F4'
⎕NA'   opengl32 |glClearDepth          F8'
⎕NA'   opengl32 |glEnable              U4'
⎕NA'   opengl32 |glMatrixMode          U4'
⎕NA'I4 user32   |ClientToScreen        P ={I4 I4}'
⎕NA'P  user32   |FindWindow*           <0T <0T'
⎕NA'I4 user32   |ShowWindow            P I4'
⎕NA'I2 user32   |GetAsyncKeyState      I4'
⎕NA'P  user32   |GetDC                 P'
⎕NA'I4 User32   |GetDialogBaseUnits'
⎕NA'P  user32   |GetFocus'
⎕NA'U4 user32   |GetSysColor           I4'
⎕NA'I4 user32   |GetSystemMetrics      I4'
⎕NA'I4 user32    |InvalidateRgn         P P I4'
⎕NA'I4 user32   |MessageBox*           P <0T <0T U4'
⎕NA'I4 user32   |ReleaseDC             P P'
⎕NA'P  user32   |SendMessage*          P U4 P P'
⎕NA'P  user32   |SetFocus              P'
```

```
⎕NA'I4 user32    |WinHelp*              P <0T U4 P'
⎕NA'I4 winnm     |sndPlaySound         <0T U4'
```

6.4.84 Native File Append {R}←X ⎕NAPPEND Y

This function appends the ravel of its left argument `X` to the end of the designated native file. `X` must be a simple homogeneous APL array. `Y` is a 1- or 2-element integer vector. `Y[1]` is a negative integer that specifies the tie number of a native file. The optional second element `Y[2]` specifies the data type to which the array `X` is to be converted before it is written to the file.

The shy result is the location of the internal file pointer which will be pointing to the end of the newly written data.

Note that `8 ⎕NINFO ⍎Y` can be used to report the current position of the file pointer.

Unicode Edition

Unless you specify the data type in `Y[2]`, a character array will by default be written using type 80.

If the data will not fit into the specified character width (bytes) `⎕NAPPEND` will fail with a `DOMAIN ERROR`.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point are in the range 256-65535 and >65535 respectively.

EXAMPLE

```
      n←'test'⎕NCREATE 0
      'abc' ⎕NAPPEND n

      'ταβέρνα'⎕NAPPEND n
DOMAIN ERROR
      'ταβέρνα'⎕NAPPEND n
     ^

      'ταβέρνα'⎕NAPPEND n 160

      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 7
ταβέρνα
```

To write 2 or more lines, you must insert appropriate end-of-line codes.

```
      ('hello',(⎕UCS 13 10),'world')⎕NAPPEND ¯1 ⍝ Windows
      ('hello',(⎕UCS 10),'world')⎕NAPPEND ¯1    ⍝ Other
```

6.4.85 Name Classification R←⎕NC Y

Y must be a simple character scalar, vector, matrix, or vector of vectors that specifies a list of names. R is a simple numeric vector containing one element per name in Y. Each element of R is the name class of the active referent to the object named in Y.

If Y is **simple**, a name class may be:

Name Class	Description
¯1	invalid name
0	undefined name
1	Label
2	Variable
3	Function
4	Operator
8	Event
9	Object (GUI, namespace, COM, .NET)

If `Y` is **nested** a more precise analysis of name class is obtained whereby different types are identified by a decimal extension. For example, defined functions have name class 3.1, dfns have name class 3.2, and so forth. The complete set of name classification is as follows:

	Array (2)	**Function (3)**	**Operator (4)**	**Namespace (9)**
n.1	Variable	Traditional	Traditional	Created by `⎕NS` , `)NS` or `:Namespace`
n.2	Field	dfn	dop	Instance
n.3	Property	Derived or Primitive	Derived or Primitive	
n.4				Class
n.5				Interface
n.6	External Shared	External		External Class
n.7				External Interface

In addition, values in `R` are negative to identify names of methods, properties and events that are inherited through the *class hierarchy* of the current class or instance.

Variable (Name-Class 2.1)

Conventional APL arrays have name-class 2.1.

```
      NUM←88
      CHAR←'Hello World'

      ⎕NC ↑'NUM' 'CHAR'
2 2

      ⎕NC 'NUM' 'CHAR'
2.1 2.1

      'MYSPACE'⎕NS ''
      MYSPACE.VAR←10
      MYSPACE.⎕NC'VAR'
2
      MYSPACE.⎕NC⊂'VAR'
2.1
```

Field (Name-Class 2.2)

Fields defined by APL Classes have name-class 2.2.

```
:Class nctest
    :Field Public pubFld
    :Field pvtFld

    ∇ r←NameClass x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Field, whether Public or Private, viewed from a Method that is executing within the Instance Space, is 2.2.

```
      ncinst.NameClass'pubFld' 'pvtFld'
2.2 2.2
```

Note that an internal Method sees both Public and Private Fields in the Class Instance. However, when viewed from *outside* the instance, only public fields are visible

```
      ⎕NC 'ncinst.pubFld' 'ncinst.pvtFld'
¯2.2 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if ⎕NC is executed inside this space:

```
      ncinst.⎕NC'pubFld' 'pvtFld'
¯2.2 0
```

Note that the names of Fields are reported as being *unused* if the argument to ⎕NC is simple.

```
      ncinst.⎕NC 2 6⍴'pubFldpvtFld'
0 0
```

Property (Name-Class 2.3)

Properties defined by APL Classes have name-class 2.3.

```
:Class nctest
    :Field pvtFld←99

    :Property pubProp
    :Access Public
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    :Property pvtProp
        ∇ r←get
          r←pvtFld
        ∇
    :EndProperty

    ∇ r←NameClass x
      :Access Public
      r←⎕NC x
    ∇
...
:EndClass ⍝ nctest

      ncinst←⎕NEW nctest
```

The name-class of a Property, whether Public or Private, *viewed from a Method that is executing within the Instance Space*, is 2.3.

```
      ncinst.NameClass'pubProp' 'pvtProp'
2.3 2.3
```

Note that an internal Method sees both Public and Private Properties in the Class Instance. However, when viewed from *outside* the instance, only Public Properties are visible

```
      ⎕NC 'ncinst.pubProp' 'ncinst.pvtProp'
¯2.3 0
```

In this case, the name-class is negative to indicate that the name has been exposed by the class hierarchy, rather than existing in the associated namespace which APL has created to contain the instance. The same result is returned if `⎕NC` is executed inside this space:

```
      ncinst.⎕NC 'pubProp' 'pvtProp'
¯2.3 0
```

Note that the names of Properties are reported as being *unused* if the argument to `⎕NC` is simple.

```
      ncinst.⎕NC 2 6⍴'pubProppvtProp'
0 0
```

External Property (Name-Class 2.6)

Properties exposed by external objects (.NET and COM and the APL GUI) have name-class `¯2.6`.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'Day' 'Month' 'Year'
¯2.6 ¯2.6 ¯2.6

      'ex' ⎕WC 'OLEClient' 'Excel.Application'
      ex.⎕NC 'Caption' 'Version' 'Visible'
¯2.6 ¯2.6 ¯2.6

     'f'⎕WC'Form'
      f.⎕NC'Caption' 'Size'
¯2.6 ¯2.6
```

Note that the names of such Properties are reported as being *unused* if the argument to ⎕NC is simple.

```
      f.⎕NC 2 7⍴'CaptionSize   '
0 0
```

Defined Function (Name-Class 3.1)

Traditional APL defined functions have name-class 3.1.

```
     ∇ R←AVG X
[1]    R←(+/X)÷⍴X
     ∇
      AVG ⍳100
50.5

      ⎕NC'AVG'
3
      ⎕NC⊂'AVG'
3.1

      'MYSPACE'⎕NS 'AVG'
       MYSPACE.AVG ⍳100
50.5

      MYSPACE.⎕NC'AVG'
3
      ⎕NC⊂'MYSPACE.AVG'
3.1
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class.

```
      MEAN←AVG
      ⎕NC'AVG' 'MEAN'
3.1 3.1
```

Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, that is, 3.3.

```
      VMEAN←AVG∘,
      ⎕NC'AVG' 'VMEAN'
3.1 3.3
```

Dfn (Name-Class 3.2)

Dfns have name-class 3.2

```
      Avg←{(+/⍵)÷⍴⍵}

      ⎕NC'Avg'
3
      ⎕NC⊂'Avg'
3.2
```

Derived Function (Name-Class 3.3)

Names that reference a primitive or derived function have a name-class of 3.3.

```
      PLUS←+
      SUM←+/
      CUM←PLUS\
      ⎕NC'PLUS' 'SUM' 'CUM'
3.3 3.3 3.3
      ⎕NC 3 4⍴'PLUSSUM CUM '
3 3 3
```

Note the difference between the name-class of a name referring to a defined function (3.1) and that of a name referring to a defined function bound with an operator to form a derived function (3.3). Trains, being derived functions, also have nameclass 3.3.

```
     ∇ R←AVG X
[1]    R←(+/X)÷⍴X
     ∇

      MEAN←AVG
      VMEAN←AVG∘,

      negrec←-,÷

      ⎕NC'AVG' 'MEAN' 'VMEAN' 'negrec'
3.1 3.1 3.3 3.3
```

External Function (Name-Class 3.6)

Methods exposed by the Dyalog APL GUI and COM and .NET objects have name-class `¯3.6`. Methods exposed by External Functions created using `⎕NA` and `⎕SH` all have name-class `3.6`.

```
      'F'⎕WC'Form'

      F.⎕NC'GetTextSize' 'GetFocus'
¯3.6 ¯3.6

      'EX'⎕WC'OLEClient' 'Excel.Application'
      EX.⎕NC 'Wait' 'Save' 'Quit'
¯3.6 ¯3.6 ¯3.6

      ⎕USING←'System'
      dt←⎕NEW DateTime (2006 1 1)
      dt.⎕NC 'AddDays' 'AddHours'
¯3.6 ¯3.6
```

```
      'beep'⎕NA'user32|MessageBeep i'

      ⎕NC'beep'
3
      ⎕NC⊂'beep'
3.6
      'xutils'⎕SH''
      )FNS
avx     box     dbr     getenv  hex     ltom    ltov    mtol    ss
vtol
      ⎕NC'hex' 'ss'
3.6 3.6
```

Note that the names of such Methods are reported as being *unused* if the argument to `⎕NC` is simple.

```
      'F'⎕WC'Form'
      F.⎕NC↑'GetTextSize' 'GetFocus'
0 0
```

Operator (Name-Class 4.1)

Traditional Defined Operators have name-class 4.1.

```
      ∇FILTER∇
     ∇ VEC←(P FILTER)VEC  ⍝ Select from VEC those elts ..
[1]    VEC←(P¨VEC)/VEC    ⍝ for which BOOL fn P is true.
     ∇

      ⎕NC'FILTER'
4
      ⎕NC⊂'FILTER'
4.1
```

Dop (Name-Class 4.2)

Dops have name-class 4.2.

```
     pred←{⎕IO ⎕ML←1 3   ⍝ Partitioned reduction.
     ⊃⍺⍺/¨(⍺/⍳⍴⍺)⊂⍵
     }

      2 3 3 2 +pred ⍳10
3 12 21 19

      ⎕NC'pred'
4
      ⎕NC⊂'pred'
4.2
```

Derived Operator (Name-Class 4.3)

Derived operators include:

- A name referring to a monadic operator.
- A dyadic operator curried with its right-operand.

EXAMPLE

```
       each←¨
       each
¨
       ⎕NC ⊂'each'
4.3
```

```
      inv←*¯1
      inv
*¯1
      ⎕NC⊂'inv'
4.3
      c2f←(32∘+)∘(×∘1.8) ⍝ Centigrade to Fahrenheit
      f2c 0 100
32 212
      f2c inv 32 212     ⍝ Fahrenheit to Centigrade
0 100
```

External Event (Name-Class 8.6)

Events exposed by Dyalog APL GUI objects, COM and .NET objects have name-class `¯8.6`.

```
      f←⎕NEW'Form'('Caption' 'Dyalog GUI Form')
      f.⎕NC'Close' 'Configure' 'MouseDown'
¯8.6 ¯8.6 ¯8.6

      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      xl.⎕NL -8
 NewWorkbook  SheetActivate  SheetBeforeDoubleClick ...

      xl.⎕NC 'SheetActivate' 'SheetCalculate'
¯8.6 ¯8.6

    ⎕USING←'System.Windows.Forms,system.windows.forms.dll'
    ⎕NC,⊂'Form'
9.6
    Form.⎕NL -8
 Activated  BackgroundImageChanged  BackColorChanged ...
```

Namespace (Name-Class 9.1)

Plain namespaces created using ⎕NS, or fixed from a `:Namespace` script, have name-class 9.1.

```
      'MYSPACE' ⎕NS ''
      ⎕NC'MYSPACE'
9
      ⎕NC⊂'MYSPACE'
9.1
```

Note however that a namespace created by cloning, where the right argument to ⎕NS is a ⎕OR of a namespace, retains the name-class of the original space.

```
      'CopyMYSPACE'⎕NS ⎕OR 'MYSPACE'
      'CopyF'⎕NS ⎕OR 'F'⎕WC'Form'

      ⎕NC'MYSPACE' 'F'
9.1 9.2
      ⎕NC'CopyMYSPACE' 'CopyF'
9.1 9.2
```

The Name-Class of .NET namespaces (visible through ⎕USING) is also 9.1

```
      ⎕USING←''
      ⎕NC 'System' 'System.IO'
9.1 9.1
```

Instance (Name-Class 9.2)

Instances of Classes created using ⎕NEW, and GUI objects created using ⎕WC all have name-class 9.2.

```
      MyInst←⎕NEW MyClass
      ⎕NC'MyInst'
9
      ⎕NC⊂'MyInst'
9.2
      UrInst←⎕NEW ⎕FIX ':Class'  ':EndClass'
      ⎕NC 'MyInst' 'UrInst'
9.2 9.2

      'F'⎕WC 'Form'
      'F.B' ⎕WC 'Button'
      ⎕NC 2 3⍴'F  F.B'
9 9
      ⎕NC'F' 'F.B'
9.2 9.2
```

```
      F.⎕NC'B'
9
      F.⎕NC⊂,'B'
9.2
```

Instances of COM Objects whether created using ⎕WC or ⎕NEW also have name-class 9.2.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')
      'XL'⎕WC'OLEClient' 'Excel.Application'
      ⎕NC'xl' 'XL'
9.2 9.2
```

The same is true of Instances of .NET Classes (Types) whether created using ⎕NEW or .New.

```
      ⎕USING←'System'
      dt←⎕NEW DateTime (3↑⎕TS)
      DT←DateTime.New 3↑⎕TS
      ⎕NC 'dt' 'DT'
9.2 9.2
```

Note that if you remove the GUI component of a GUI object, using the Detach method, it reverts to a plain namespace.

```
      F.Detach
      ⎕NC⊂,'F'
9.1
```

Correspondingly, if you attach a GUI component to a plain namespace using the monadic form of ⎕WC, it morphs into a GUI object

```
      F.⎕WC 'PropertySheet'
      ⎕NC⊂,'F'
9.2
```

Class (Name-Class 9.4)

Classes created using the editor or ⎕FIX have name-class 9.4.

```
      )ED ⍥MyClass

:Class MyClass
    ∇ r←NameClass x
      :Access Public Shared
      r←⎕NC x
    ∇
:EndClass ⍝ MyClass

      ⎕NC 'MyClass'
9
      ⎕NC⊂'MyClass'
9.4
```

```
      ⎕FIX ':Class UrClass'  ':EndClass'
      ⎕NC 'MyClass' 'UrClass'
9.4 9.4
```

Note that the name of the Class is visible to a Public Method in that Class, or an Instance of that Class.

```
      MyClass.NameClass'MyClass'
9
      MyClass.NameClass⊂'MyClass'
9.4
```

Interface (Name-Class 9.5)

Interfaces, defined by `:Interface ... :EndInterface` clauses, have name-class 9.5.

```
:Interface IGolfClub
:Property Club
    ∇ r←get
    ∇
    ∇ set
    ∇
:EndProperty

∇ Shank←Swing Params
∇

:EndInterface ⍝ IGolfClub

      ⎕NC 'IGolfClub'
9
      ⎕NC ⊂'IGolfClub'
9.5
```

External Class (Name-Class 9.6)

External Classes (Types) exposed by .NET have name-class 9.6.

```
      ⎕USING←'System' 'System.IO'

      ⎕NC 'DateTime' 'File' 'DirectoryInfo'
9.6 9.6 9.6
```

Note that referencing a .NET class (type) with ⎕NC, fixes the name of that class in the workspace and obviates the need for APL to repeat the task of searching for and loading the class when the name is next used.

External Interface (Name-Class 9.7)

External Interfaces exposed by .NET have name-class 9.7.

```
      ⎕USING←'System.Web.UI,system.web.dll'

      ⎕NC 'IPostBackDataHandler' 'IPostBackEventHandler'
9.7 9.7
```

Note that referencing a .NET Interface with ⎕NC, fixes the name of that Interface in the workspace and obviates the need for APL to repeat the task of searching for and loading the Interface when the name is next used.

6.4.86 Native File Copy {R}←X ⎕NCOPY Y

This function copies native files and directories from one or more sources specified by `Y` to a destination specified by `X`. `⎕NCOPY` is similar to `⎕NMOVE` (see *Section 6.4.96*).

`X` is a character vector that specifies the name of the destination.

`Y` is a character vector that specifies the name of the source, or a vector of character vectors containing zero or more sources.

Source and destination path names may be full or relative (to the current working directory) path names which adhere to the operating system conventions.

If `X` specifies an existent directory then each source in `Y` is copied into that directory, otherwise `X` specifies the name of the copy. `X` must specify an existent directory if the source contains multiple names or if the **Wildcard** option is set.

The shy result `R` contains count(s) of top-level items copied. If `Y` is a single source name, `R` is a scalar otherwise it is a vector of the same length as `Y`.

Variant Options

`⎕NCOPY` may be applied using the *variant* operator with the options **Wildcard** (the Principal option), **IfExists**, **PreserveAttributes** and **ProgressCallback**.

Wildcard Option (Boolean)

0	The name or names in `Y` identifies a specific file name.
1	The name or names in `Y` that specify the *base name* and *extension* (see *NParts*), may also contain the wildcard characters "?" and "*". An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

Note that when **Wildcard** is 1, element(s) of `R` can be 0, 1 or `>1`. If **Wildcard** is 0, elements of `R` are always 1.

IfExists Option

The **IfExists** variant option determines what happens when a source file is to be copied to a target file that already exists. It does not apply to directories, only to the files within them.

Value	Description
'Error'	Existing files will not be overwritten and an error will be signalled. This is the default
`'Skip'`	Existing files will not be overwritten but the corresponding copy operation will be skipped (ignored).
`'Replace'`	Existing files will be overwritten.
`'ReplaceIfNewer'`	Existing files may be overwritten if, and only if, the corresponding source file is newer (more recently modified) than the existing one, otherwise it is skipped.

The following cases cause an error to be signalled regardless of the value of the **IfExists** variant.

- If the source specifies a directory and the destination specifies an existing file.
- If the source specifies a file and the same base name exists as a sub-directory in the destination.

PreserveAttributes Option (Boolean)

The **PreserveAttributes** variant option determines whether or not file attributes are preserved. It does not apply to directories, only to files.

0	file attributes are not preserved.
1	where possible, copied files will be given at least the same modification time as the source. Other file attributes will be preserved as permitted by the operating system and file system.

Note also that when files are copied across file systems, the different file systems may have different timestamp granularity and the timestamps may not be exactly the same.

Examples

There are a number of possibilities which are illustrated below. In all cases, if the source is a file, a copy of the file is created. If the source is a directory, a copy of the directory and all its contents is created.

Examples (single source, Wildcard is 0)

- The source name must be an existent file or directory.
- If the destination name does not exist but its path name does exist, the source is copied to the destination name.
- If the destination name is an existing directory the copy is created within that directory with the base name of the source.

```
      ⊃1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

⍝ Make a named back-up of the Session file
      ⊢'session.bak' ⎕NCOPY 'default.dlf'
```

```
1
      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
 ⍝ Copy the Session file to backups directory
      ⊢'backups'⎕NCOPY'default.dlf'
1
      ↑⊃0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
```

Examples (single source, Wildcard is 1)

- The source name may include wildcard characters which matches a number of existing files and/or directories. The destination name must be an existing directory.
- The files and/or directories that match the pattern specified by the source name are copied into the destination directory. If there are no matches, zero copies are made.

```
      ⊃1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Copy all files to backups directory
      ⊢'backups'(⎕NCOPY⍠'Wildcard' 1)'*.*'
3
```

```
      ↑⊃0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
backups/UserCommand20.cache
```

Examples (multiple sources, Wildcard is 0)

- Each source name must specify a single file or directory which must exist. The destination name must be an existing directory.
- Copies of each of the files and/or directories specified by the source base names are made in the destination directory.

```
      ⊃1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Copy 2 files to backups directory
      ⊢'backups'⎕NCOPY'default.dlf' 'def_uk.dse'
1 1
      ↑⊃0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
```

Examples (multiple sources, Wildcard is 1)

- The destination name must be an existing directory.
- Copies of each of the files and/or directories that match the patterns specified by the source names (if any) are made in the destination directory.

```
      ⍕1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Copy files to backups directory
      ⊢'backups'(⎕NCOPY⍠1)'d*' 'UserCommand20.cache'
2 1
      ↑⍕0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
backups/UserCommand20.cache
```

ProgressCallback Option

The **ProgressCallback** variant option is described in the *Programming: Native Files*. following is specific to ⎕NCOPY:

- The first element of the right argument to the callback function is the character vector '⎕NCOPY'.

Notes

- The special directories . and .. can never be copied into an existing directory.
- If any source name is a symbolic link it is dereferenced; that is, the source or directory it references is copied rather than the link itself.
- In the result R, a directory together with all its contents is counted once. A directory may be partially copied if the **IfExists** option is set to 'Replace' or 'ReplaceIfNewer').
- If an error occurs during the copy process then processing will immediately stop and an error will be signalled. The operation is not atomic; some items may be copied before this happens. In the event of an error there will be no result and therefore no indication of how many names were copied before the error occurred.

6.4.87 Native File Create {R}←X ⎕NCREATE Y

This function creates a new file. Under Windows the file is opened with mode 66 (see *Section 6.4.110*). Under non-Windows operating systems the current umask will specify the file permissions. The name of the new file is specified by the left argument X which must be a simple character vector or scalar containing a valid pathname for the file.

`Y` is 0 or a negative integer value that specifies an (unused) tie number by which the file may subsequently be referred. If `Y` is 0, the system allocates the first (closest to zero) available tie number which is returned as the result.

The shy result of `⎕NCREATE` is the tie number of the new file.

Variant Options

`⎕NCREATE` may be applied using the Variant operator with the options Unique and IfExists. There is no primary option.

Unique Option (Boolean)

0	the file named by `X` will be created
1	a uniquely named file will be created by extending the base name (see *File Name Parts*) with random characters. If a unique name cannot be created then an error will be signalled. The actual name of the file can be determined from `⎕NNAMES` or `⎕NINFO` .

IfExists Option (character vector)

Error	`⎕NCREATE` will generate a `FILE NAME ERROR` if the file already exists
`Replace`	`⎕NCREATE` will replace an existing file with an empty one of the same name.

Examples

```
      ⊢'myfile' ⎕NCREATE 0
¯1
      ⎕NUNTIE ¯1
      ⊢'myfile' ⎕NCREATE 0
FILE NAME ERROR: myfile: Unable to create file ("The file exists.")
      ⊢'myfile'⎕NCREATE 0
               ^

      ⊢'myfile' (⎕NCREATE⍠'IfExists' 'Replace') 0

¯1    ⍝ Note that it uses same tie number as before
```

```
      ⊢'myfile' (⎕NCREATE⍠('Unique' 1)) 0
¯2
      ⎕NNUMS,⎕NNAMES
¯1 myfile
¯2 myfile52c36z
```

Notes

- Setting IfExists to `Replace` has no effect when Unique is 1, because the file cannot already exist.
- The IfExists option does not affect the operation of *slippery ties*.

6.4.88 Native File Delete {R}←{X}⎕NDELETE Y

This function deletes files and directories.

`Y` is a character vector or scalar containing a single file or directory name, or a vector of character vectors containing zero or more file or directory names. Names must conform to the naming rules of the host Operating System.

The optional left argument `X` is a numeric scalar; valid values are shown in the following table. If omitted, its default value is 0.

0	Each file or directory with the given name must exist.
1	If the file or directory with the given name does not exist then no action is taken. The result `R` may be used to determine whether the file or directory was deleted or not.
2	If a name identifies a non-empty directory it, and all its contents, are to be deleted.
3	Combination of 1 and 2.

`R` is a numeric count of top-level entities deleted when processing the corresponding name in `Y`. If `Y` specifies a single name, `R` is a scalar. If `Y` is a vector of character vectors `R` is a vector with the same length as `Y`.

Variant Options

`⎕NDELETE` may be applied using the Variant operator with the Wildcard option.

Wildcard Option (Boolean)

0	The name or names in `Y` identifies a specific file name.
1	The name or names in `Y` that specify the *base name* and *extension* (see *File Name Parts*), may also contain the wildcard characters "?" and "*". An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

Note that when Wildcard is 1, element(s) of `R` can be 0 or `>1`. If Wildcard is 0, elements of `R` are always 1.

If `Y` specifies the name of a symbolic link, `⎕NDELETE` deletes that symbolic link; the target of the symbolic link is unaffected.

Examples

```
      ⎕NEXISTS'/Users/Pete/Documents/temp/t1/t2'
1
      ⊢⎕NDELETE'/Users/Pete/Documents/temp/t1/t2'
1
      ⊢⎕NDELETE'/Users/Pete/Documents/temp/t1/t2'
FILE NAME ERROR: Invalid file or directory name
("The system cannot find the file specified.")
      ⊢⎕NDELETE'/Users/Pete/Documents/temp/t1/t2'
     ^
```

```
      ⊢1 ⎕NDELETE'/Users/Pete/Documents/temp/t1/t2'
0
      ⊢⎕NDELETE 'temp1' 'temp2'
1 1
      ⊢⎕MKDIR'temp1' 'temp2'
1 1
       ⊢(⎕NDELETE⍠1)'t*'
2
```

```
      ⊢⎕MKDIR'temp1'
1
      ⊢'Hello World' ⎕NPUT 'temp1/hw.txt'
13
      ⊢⎕NDELETE 'temp1'
FILE ACCESS ERROR: temp1: Unable to delete directory ("The directory is
not empty.")
      ⊢⎕NDELETE'temp1'
        ^
      ⊢2 ⎕NDELETE 'temp1'
1
```

If the file is in use or the current user is not authorised to delete it, ⎕NDELETE will not succeed but will instead generate a FILE ACCESS ERROR.

Note

When multiple names are specified they are processed in the order given. If an error occurs at any point whilst deleting files or directories, processing will immediately stop and an error will be signalled. The operation is not atomic; the directory contents may be partially deleted before this happens. In the event of an error there will be no result and therefore no indication of how many files were deleted before the error occurred.

6.4.89 Native File Erase {R}←X ⎕NERASE Y

This function erases (deletes) a tied native file. Y is a negative integer tie number associated with a tied native file. X is a simple character vector or scalar containing the name of the same file and must be **identical** to the name used when it was opened by ⎕NCREATE or ⎕NTIE.

The shy result of ⎕NERASE is the tie number that the erased file had.

EXAMPLE

```
      file ⎕NERASE file ⎕NTIE 0
```

6.4.90 New Instance R←⎕NEW Y

⎕NEW creates a new instance of the Class, Dyalog GUI object, or .NET Type specified by Y.

`Y` must be a 1- or 2-item scalar or vector. The first item is a reference to a Class or to a .NET Type, or a character vector containing the name of a Dyalog GUI object.

The second item, if specified, contains the argument to be supplied to the Class or Type *Constructor* or a list of property/value pairs for a Dyalog GUI object.

The result `R` is a reference to a new instance of Class, Dyalog GUI object, or Type `Y`.

For further information, see *Interface Guide*.

Class Example

```
:Class Animal
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal

      Donkey←⎕NEW Animal 'Eeyore'
      Donkey
Eeyore
```

If `⎕NEW` is called with just a Class reference (that is, without parameters for the Constructor), the default constructor will be called. A default constructor is defined by a niladic function with the :Implements Constructor attribute. For example, the Animal Class may be redefined as:

```
:Class Animal
    ∇ NoName
      :Access Public
      :Implements Constructor
      ⎕DF 'Noname'
    ∇
    ∇ Name nm
      :Access Public
      :Implements Constructor
      ⎕DF nm
    ∇
:EndClass ⍝ Animal


      Horse←⎕NEW Animal
      Horse
Noname
```

.NET Examples

```
      ⎕USING←'System' 'System.Web.Mail,System.Web.dll'
      dt←⎕NEW DateTime (2006 1 1)
      msg←⎕NEW MailMessage
      ⎕NC 'dt' 'msg' 'DateTime' 'MailMessage'
9.2 9.2 9.6 9.6
```

Note that **.NET Types** are accessed as follows.

If the name specified by the first item of `Y` would otherwise generate a `VALUE ERROR`, and `⎕USING` has been set, APL attempts to load the Type specified by `Y` from the .NET assemblies (DLLs) specified in `⎕USING`. If successful, the name specified by `Y` is entered into the SYMBOL TABLE with a name-class of `9.6`. Subsequent references to that symbol (in this case `DateTime`) are resolved directly and do not involve any assembly searching.

Dyalog GUI Examples

```
      F←⎕NEW ⊂'Form'
      F
#.[Form]
```

To specify the initial values of any properties, `Y[2]` must be a vector (or scalar) of items each of which is of the form (PropertyName PropertyValue); the free-form syntax implemented by `⎕WC` and `⎕WS` is not allowed.

```
      ⎕NEW'Form'(⊂'Caption' 'Hello')
#.[Form]
```

```
      F←⎕NEW'Form'(('Caption' 'Hello')('Posn' (10 10)))
      F
#.[Form]
```

Note that as `⎕NEW` provides no facility to *name* a GUI object, the Event property should use the *onEvent* syntax so that a callback function (or the result of `⎕DQ`) receives a ref to the object. Otherwise, without the *onEvent* syntax, the first element of the argument to a callback function will contain a character vector such as `'[Form].[Button]'` which merely describes the type of the object but does not identify the object itself.

```
      cap←'Caption' 'Push Me'
      ev← 'Event' ('onSelect' 'foo')
      F.(B←⎕NEW'Button'#.(pos cap ev))
```

Note that you may not create an instance of OCXClass using `⎕NEW`.

6.4.91 Native File Exists `R←⎕NEXISTS Y`

This function reports whether or not files and directories exist.

`Y` is a character vector or scalar containing a single file/directory name, or a vector of character vectors containing zero or more file/directory names.

If `Y` specifies a single name, the result `R` is a scalar 1 if a file or directory exists or 0 if not. If `Y` is a vector of character vectors, `R` is a vector of 1s and 0s with the same length as `Y`.

Variant Options

`⎕NEXISTS` may be applied using the Variant operator with the Wildcard option.

Wildcard Option (Boolean)

0	The name or names in Y identifies a specific file name.
1	The name or names in Y that specify the *base name* and *extension* (see *NParts*), may also contain the wildcard characters "?" and "*". An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

If the Wildcard option is 1, R indicates whether or not one or more matches to the corresponding pattern in Y exist.

EXAMPLE

```
      ⎕←⎕MKDIR'/Users/Pete/Documents/temp/t1/t2'
1
      ⎕NEXISTS'/Users/Pete/Documents/temp/t1/t2'
1
      ⎕NEXISTS'/Users/Pete/Documents/temp/t1/t2/pd'
0

      ⊢⎕MKDIR'temp1' 'temp2'
1 1
      ⎕NEXISTS 'temp1' 'temp2' 'temp3'
1 1 0
      (⎕NEXISTS⍤1) 't*'
1
```

> **Note**
>
> If `Y` is a symbolic link, `⎕NEXISTS` will return 1 whether or not the target of the symbolic link exists.

6.4.92 Read Text File `R←{X} ⎕NGET Y`

This function reads the contents of the specified text file. See also *Section 6.4.100*.

`Y` is either a character vector/scalar containing the name of the file to be read, or a 2-item vector whose first item is the file name and whose second is an integer scalar specifying `flags` for the operation.

- if `flags` is `0` then `R[1]` is a character vector. This is the default.
- if `flags` is `1` then `R[1]` is a nested array of character vectors.
- if `flags` is `2` then `R[1]` is a matrix, with each row corresponding to a line in the text file specified within `X`.

The optional left-argument `X` is either

- a character vector that specifies the file-encoding as shown in the table below.
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

Table 6-2: *File Encodings*

Encoding	Description
`UTF-8`	The data is encoded as UTF-8 format.
`UTF-16LE`	The data is encoded as UTF-16 little-endian format.
`UTF-16BE`	The data is encoded as UTF-16 big-endian format.
`UTF-16`	The data is encoded as UTF-16 with the endianness of the host system (currently BE on AIX platforms, LE on all others).
`UTF-32LE`	The data is encoded as UTF-32 little-endian format.
`UTF-32BE`	The data is encoded as UTF-32 big-endian format.
`UTF-32`	The data is encoded as UTF-32 with the endianness of the host system (currently BE on AIX platforms, LE on all others).
`ASCII`	The data is encoded as 7-bit ASCII format.
`Windows-1252`	The data is encoded as 8-bit Windows-1252 format.
`ANSI`	ANSI is a synonym of Windows-1252.

The above UTF formats may be qualified with -BOM or -NOBOM (for example, UTF-8-BOM). See *Section 6.4.100*.

Whether or not `X` is specified, if the start of the file contains a recognised Byte Order Mark (BOM), the file is decoded according to the BOM. Otherwise, if `X` is specified the file is decoded according to the value of `X`. Otherwise, the file is examined to try to decide its encoding and is decoded accordingly.

The result `R` is a 3-element vector comprising `(content) (encoding) (newline)` where:

`content`	A simple character vector, or a vector of character vectors, according to the value of `flags` .
`encoding`	The encoding that was actually used to read the file. If this is a UTF format, it will always include the appropriate endianness (except for UTF-8 to which endianness doesn't apply) and a -BOM or -NOBOM suffix to indicate whether or not a BOM is actually present in the file. For example, UTF-16LE-BOM. If `X` specified a user-defined encoding as a 256-element numeric vector, `encoding` will be that same vector.
`newline`	Determined by the first occurrence in the file of one of the newline characters identified in the line separator table, or `0` if no such line separator is found.

If `content` is simple then all its line separators (listed in the table below) are replaced by (normalised to) `⎕UCS 10`, which in the Classic Edition must be in `⎕AVU` (else `TRANSLATION ERROR`).

If `content` is nested, it is formed by splitting the contents of the file on the occurrence of any of the line separators shown in the table below. These line separators are removed.

The 3rd element of the result `newline` is a numeric vector from the *Value* column of the table below corresponding to the first occurrence of any of the **newline characters** in the file. If none of these characters are present, the value is `0`.

Table 6-3: *Line separators:*

Value	Code	Description
newline characters		
13	`CR`	Carriage Return (U+000D)
10	`LF`	Line Feed (U+000A)
13 10	`CRLF`	Carriage Return followed by Line Feed
133	`NEL`	New Line (U+0085)
other line separator characters		
11	`VT`	Vertical Tab (U+000B)
12	`FF`	Form Feed (U+000C)
8232	`LS`	Line Separator (U+2028)
8233	`PS`	Paragraph Separator (U+2029)

6.4.93 Native File Information `R←{X}⎕NINFO Y`

This function queries or sets information about one or more files or directories. `Y` may be:

- a numeric scalar containing the tie number of a native file
- a character vector or scalar containing a file or directory name that conforms to the naming rules of the host Operating System.
- a vector of character vectors and/or tie numbers

Optionally, `X` specifies properties of the files/directories identifed in `Y`. `X` can be an array of any shape; the shape determines whether the specified properties are queried or set:

- If `X` is a simple numeric array, the properties are queried. In this case, the values of `X` correspond to properties of the file/directory specified in `Y` that are to be queried, as defined in the following table.
- If some or all of the elements in `X` are nested vectors, the properties are set. In this case, the values of `X` correspond to properties of the file/directory specified in `Y` that are to be set, as defined in the following table, with appropriate corresponding values to which those properties should be set. Not all file properties are settable.

If `X` is not defined, it is assumed to be `0`.

X	Property	Default	Settable
0	Name of the file or directory, as a character vector. If Y is a tie number then this is the name which the file was tied.		No
1	Type, as a numeric scalar: 0=Not known 1=Directory 2=Regular file 3=Character device 4=Symbolic link (only when Follow is 0) 5=Block device 6=FIFO (not Windows) 7=Socket (not Windows)	0	No
2	Size in bytes, as a numeric scalar	0	Yes
3	Last modification time, as a timestamp in ⎕TS format	7⍴0	No
4	Owner user id, as a character vector – on Windows a SID, on other platforms a numeric userid converted to character format	''	No
5	Owner name, as a character vector	''	No
6	Whether the file or directory is hidden (1) or not (0), as a numeric scalar. On Windows, file properties include a "hidden" attribute; on non-Windows platforms a file or directory is implicitly considered to be hidden if its name begins with a "."	¯1	Windows only
7	Target of symbolic link (when Type is 4)	''	No
8	Current position in the file. This identifies where ⎕NREAD would next read from or ⎕NAPPEND would next write to, and is only pertinent when the corresponding value in Y is a tie number. It is reported as 0 for named files and directories.	0	Yes
9	Last access time in ⎕TS format, when available	7⍴0	No
10	Creation time if available, otherwise the time of the last file status change in ⎕TS format	7⍴0	No
11	Whether the file or directory can (1) or cannot (0) be read (¯1 if unknown)	¯1	No
12	Whether the file or directory can (1) or cannot (0) be written (¯1 if unknown or for a directory under Windows)	¯1	No
13	Last modification time, as a UTC Dyalog Date Number.	0	Yes

X	Property	Default	Settable
14	Last access time, as a UTC Dyalog Date Number, when available.	0	Yes
15	Creation time if available, otherwise the time of the last file status change as a UTC Dyalog Date Number.	0	Windows only

The values in X are processed in ravel order. Duplicates are allowed.

The returned value R has the same shape as X (if the **Wildcard** variant option has been specified, the depth will change – see below). The content of R depends on whether properties were queried or set:

- If properties were queried:
  - each element of R is the value of the property identified by the corresponding value of X.
  - if a value in X is not defined in the table above, the corresponding R value is ⍬ (Zilde).
  - if more than one file/directory is specified in Y, the element of R corresponding to each element of X contains the property values for each of the files/directories specified in Y.
- If properties were set:
  - each element of R is the new value of the property identified by the corresponding value of X.
  - if the value in X does not specify a property that is defined in the table above and settable, or if the new value specified for the property is not valid, an error will be signalled.
  - if more than one file/directory is specified in Y, the element of R corresponding to each element of X contains the new property values for each of the files/directories specified in Y.

If a property value cannot be obtained, the default value (shown in the table above) is returned for that property.

If the Wildcard option is not enabled (the default) then Y specifies exactly one file or directory and must exist. In this case each element in R is a single property value for that file. If the name in Y does not exist, the function signals an error. On non-Windows platforms "*" and "?" are treated as normal characters. On Microsoft Windows an error will be signalled since neither are valid characters for file or directory names.

If the Wildcard option is enabled, zero or more files and/or directories may match the pattern in Y. In this case each element in R is a vector of property values for each of the files. Note that no error will be signalled if no files match the pattern.

When using the **Wildcard** option, matching of names is done case insensitively on Windows and macOS, and case sensitively on other platforms. The names '.' and '..' are excluded from any matches. The order in which the names match is not defined.

Variant Options

NINFO may be applied using the *variant* operator with the options **Wildcard** (the Principal option), **Recurse**, **Follow** and **ProgressCallback**.

Wildcard Option (Boolean)

0	The name or names in Y identifies a specific file name.
1	The name or names in Y that specify the *base name* and *extension* (see *NParts*), may also contain the wildcard characters "?" and "*". An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

Recurse Option

0	the name(s) in Y are searched for only in the corresponding specified directory.
1	the name(s) in Y are searched for in the corresponding specified directory as well as all sub-directories. If **Wildcard** is also 1, the wild card search is performed recursively.
1 n	the name(s) in Y are searched for in the corresponding specified directory as well as its sub-directories to the n th -level sub-directory. If n is 0, no sub-directories are searched. If n is ¯1 all sub-directories are searched.
2 (n)	same as 1 but if any unreadable directories are encountered they are skipped (whereas if **Recurse** is 1 (n), NINFO stops and generates an error).

Follow Option (Boolean)

`0`	the properties reported are those of the symbolic link itself
1	the properties reported for a symbolic link are those of the target of the symbolic link

ProgressCallback Option

The **ProgressCallback** variant option is described in the *Programming: Native Files*. The following is specific to `⎕NINFO`:

- The first element of the right argument to the callback function is the character vector `'⎕NINFO'`.
- The third element of the right argument (the information namespace) contains an extra field named `Info`, which is a vector with the same length as the `Last` field. Each element of the `Info` vector contains the information requested by the `⎕NINFO` call for the corresponding filename in `Last`.

Note

On platforms other than Microsoft Windows, file names are exposed by the operating system using UTF-8 encoding, which Dyalog translates internally to characters.

In the Unicode Edition, if the UTF-8 encoding is invalid, Dyalog replaces each offending byte with a unique Unicode symbol (in the *Low Surrogate Area* of the Unicode charts) that is mapped back to the original byte by the other system functions (including `⎕NTIE` and `⎕NDELETE`) that take native file names as arguments. The display of a file name containing these mapped bytes may appear strange.

In the Classic Edition, offending bytes are replaced by the `?` symbol, which means that the names reported do not accurately identify the files.

Examples

```
      (0 1 2) ⎕NINFO 'c:/Users/Pete/Documents'
┌→──────────────────────────────────────┐
│ ┌→──────────────────────┐             │
│ │c:/Users/Pete/Documents│ 1 163840    │
│ └───────────────────────┘             │
└∊──────────────────────────────────────┘
```

```
      ⎕NPARTS '' ⍝ current working directory
c:/Users/Pete/
      (⎕NINFO⍠1)'D*'
┌──────────────────────────────────────┐
│ ┌───────┬─────────┬─────────┬───────┐ │
│ │Desktop│Documents│Downloads│Dropbox│ │
│ └───────┴─────────┴─────────┴───────┘ │
└──────────────────────────────────────┘
```

```
      (⎕NINFO⍠1)'Documents/*.zip'
┌──────────────────────┐
│ ┌──────────────────┐ │
│ │Documents/dyalog.zip│ │
│ └──────────────────┘ │
└──────────────────────┘
```

```
      ⍪ (0,⍳6) ⎕NINFO 'Documents/dyalog.zip'
┌──────────────────────────────────────────────────┐
│Documents/dyalog.zip                              │
├──────────────────────────────────────────────────┤
│2                                                 │
├──────────────────────────────────────────────────┤
│3429284                                           │
├──────────────────────────────────────────────────┤
│2016 1 22 16 43 58 0                              │
├──────────────────────────────────────────────────┤
│S-1-5-21-2756282986-1198856910-2233986399-1001    │
├──────────────────────────────────────────────────┤
│HP/Pete                                           │
├──────────────────────────────────────────────────┤
│0                                                 │
└──────────────────────────────────────────────────┘
```

```
      ⊃1⎕NPARTS '' ⍝ current working directory
C:/Users/Pete/Documents/Dyalog APL-64 16.0 Unicode Files/
      (⎕NINFO⍠1)'*.*'
┌──────────────────────────────────────────────────────┐
│┌────────────┬──────────┬─────────┬──────────────────┐│
│|default.dlf|def_uk.dse|jsonx.dws|UserCommand20.cache||
│└────────────┴──────────┴─────────┴──────────────────┘│
└──────────────────────────────────────────────────────┘
```

```
      ⊢ ⎕MKDIR 'd1' 'd2'
1 1
      'a'∘⎕NPUT¨'find' 'd1/find' 'd1/nofind' 'd2/find'
      (⎕NINFO⍠'Recurse' 1)'find'
┌──────────────────────┐
│┌────────┬────────┬──┐│
│|d1/find|d2/find|find||
│└────────┴────────┴──┘│
└──────────────────────┘
```

The next set of examples illustrates the use of the **Recurse** variant option to limit the sub-directory depth.

```
      Y←'d:\bouzouki\*.*'
      ⍴⊃0(⎕NINFO⍠('Wildcard' 1)('Recurse' 0))Y
355
      ⍴⊃0(⎕NINFO⍠('Wildcard' 1)('Recurse' (1 0)))Y
355
      ⍴⊃0(⎕NINFO⍠('Wildcard' 1)('Recurse' (1 1)))Y
1333
      ⍴⊃0(⎕NINFO⍠('Wildcard' 1)('Recurse' (1 3)))Y
4223
```

The following expression will return all Microsoft Word documents (`.docx` and `.doc`) in the current directory, searching recursively through any sub-directories:

```
     (⎕NINFO⍠('Recurse' 1)('Wildcard' 1))'*.docx' '*.doc'
```

The following expression "touches" files, that is, it sets their last modification time to the current UTC time:

```
      (⊂13(1 ⎕DT'Z'))(⎕NINFO⍠1)'*.txt'
┌──────────────────────┐
│45719.53226 45719.53226│
└──────────────────────┘
```

> **Note**
>
> Of the file timestamps which are reported by the operating system, only the last modification time should be considered reliable and portable. Neither the access time or creation time are well supported across all platforms. Furthermore, they may not accurately reflect the actual time that the operation occurred.

6.4.94 Name List `R←{X}⎕NL Y`

`Y` must be a simple numeric scalar or vector containing one or more of the values for name-class. See also *Section 6.4.85*.

`X` is optional. If present, it must be a simple character scalar or vector. `R` is a list of the names of active objects whose name-class is included in `Y` in standard sorted order.

If *any* element of `Y` is negative, positive values in `Y` are treated as if they were negative, and R is a vector of character vectors. Otherwise, `R` is simple character matrix.

Furthermore, if `⎕NL` is being evaluated inside the namespace associated with a Class or an Instance of a Class, and any element of `Y` is negative, `R` includes the Public names exposed by the Base Class (if any) and all other Classes in the Class hierarchy.

If `X` is supplied, `R` contains only those names which begin with any character of `X`. Standard sorted order is in Unicode point order for Unicode editions, and in the collation order of `⎕AV` for Classic editions.

If an element of `Y` is an integer, the names of all of the corresponding sub-name-classes are included in `R`. For example, if `Y` contains the value 2, the names of all variables (name-class 2.1), fields (2.2), properties (2.3) and external or shared variables (2.6) are obtained. Otherwise, only the names of members of the corresponding sub-name-class are obtained.

Examples

```
      ⎕NL 2 3
A
FAST
FIND
FOO
V

      'AV' ⎕NL 2 3
A
V

      ⎕NL ¯9
 Animal  Bird  BirdBehaviour  Coin  Cylinder  DomesticParrot  Eeyore  Fi
shBehaviour  Nickel  Parrot  Penguin  Polly  Robin
      ⎕NL ¯9.3 ⍝ Instances
 Eeyore  Nickel  Polly  Robin
      ⎕NL ¯9.4 ⍝ Classes
 Animal  Bird  Coin  Cylinder  DomesticParrot  Parrot  Penguin
      ⎕NL ¯9.5 ⍝ Interfaces
 BirdBehaviour  FishBehaviour
```

⎕NL can also be used to explore Dyalog GUI Objects, .NET types and COM objects.

Dyalog GUI Objects

⎕NL may be used to obtain lists of the Methods, Properties and Events provided by Dyalog APL GUI Objects.

```
      'F' ⎕WC 'Form'
      F.⎕NL -2 ⍝ Properties
 Accelerator AcceptFiles  Active  AlphaBlend  AutoConf  Border  BCol  Ca
ption ...

      F.⎕NL -3 ⍝ Methods
 Animate  ChooseFont  Detach  GetFocus  GetTextSize  ShowSIP  Wait

      F.⎕NL -8 ⍝ Events
Close  Create  DragDrop  Configure  ContextMenu  DropFiles  DropObjects
 Expose  Help ...
```

.NET Classes (Types)

⎕NL can be used to explore .NET types.

When a reference is made to an undefined name, and ⎕USING is set, APL attempts to load the Type from the appropriate .NET Assemblies. If successful, the name is entered into the symbol table with name-class 9.6.

```
      ⎕USING←'System'
      DateTime
(System.DateTime)
      ⎕NL -9
 DateTime
      ⎕NC⊂'DateTime'
9.6
```

The names of the Properties and Methods of a .NET Type may then be obtained using ⎕NL.

```
      DateTime.⎕NL -2 ⍝ Properties
 MaxValue  MinValue  Now  Today  UtcNow

      DateTime.⎕NL -3 ⍝ Methods
 get_Now  get_Today  get_UtcNow  op_Addition  op_Equality  ...
```

In fact it is not necessary to make a separate reference first, because the expression `Type.⎕NL` (where `Type` is a .NET Type) is itself a reference to Type. So, (with ⎕USING still set to `'System'`):

```
      Array.⎕NL -3
 BinarySearch  Clear  Copy  CreateInstance  IndexOf  LastIndexOf  Revers
e  Sort

      ⎕NL -9
 Array  DateTime
```

Another use for ⎕NL is to examine .NET *enumerations*. For example:

```
      ⎕USING←'System.Windows.Forms,system.windows.forms.dll'

      FormBorderStyle.⎕NL -2
Fixed3D  FixedDialog  FixedSingle  FixedToolWindow  None  Sizable  Sizab
leToolWindow

      FormBorderStyle.FixedDialog.value__
3

      FormBorderStyle.({⍵,[1.5]⍪¨⍵,¨⊂'.value__'}⎕NL -2)
 Fixed3D            2
 FixedDialog        3
 FixedSingle        1
 FixedToolWindow    5
 None               0
 Sizable            4
 SizableToolWindow  6
```

COM Objects

Once a reference to a COM object has been obtained, ⎕NL may be used to obtain lists of its Methods, Properties and Events.

```
      xl←⎕NEW'OLEClient'(⊂'ClassName' 'Excel.Application')

      xl.⎕NL -2 ⍝ Properties
 _Default  ActiveCell  ActiveChart  ActiveDialog  ActiveMenuBar  ActiveP
rinter  ActiveSheet  ActiveWindow ...

      xl.⎕NL -3 ⍝ Methods
 _Evaluate  _FindFile  _Run2  _Wait  _WSFunction  ActivateMicrosoftApp
AddChartAutoFormat  AddCustomList  Browse  Calculate ...

      ⎕NL -9
 xl
```

6.4.95 Native File Lock {R}←X ⎕NLOCK Y

This function assists the controlled update of shared native files by locking a range of bytes.

Locking enables controlled update of native files by co-operating users. A process requesting a lock on a region of a file will be *blocked* until that region becomes available. A *write-lock* is exclusive, whereas a *read-lock* is shared. In other words, any byte in a file may be in one of only three states:

- Unlocked
- Write-locked by exactly one process.
- Read-locked by any number of processes.

`Y` must be a simple integer scalar or vector containing 1, 2 or 3 items namely:

1. Tie number
2. Offset (from 0) of first byte of region. Defaults to 0
3. Number of bytes to lock. Defaults to maximum possible file size

`X` must be a simple integer scalar or vector containing 1 or 2 items, namely:

1. Type: 0: Unlock, 1:Read lock, 2:Write lock.
2. Timeout: Number of seconds to wait for lock before generating a `TIMEOUT` error. Defaults to indefinite wait.

The shy result `R` is `Y`. To unlock the file, this value should subsequently be supplied in the right argument to `0 ⎕NLOCK`.

Examples

```
      2 ⎕NLOCK ¯1          ⍝ write-lock whole file
      0 ⎕NLOCK ¯1          ⍝ unlock whole file.
      1 ⎕NLOCK ¯1          ⍝ read (share) lock whole file.
      2 ⎕NLOCK¨⎕NNUMS      ⍝ write-lock all files.
      0 ⎕NLOCK¨⎕NNUMS      ⍝ unlock all files.

      1 ⎕NLOCK ¯1 12 1     ⍝ read-lock byte 12.
      1 ⎕NLOCK ¯1 0 10     ⍝ read-lock first 10 bytes.
      2 ⎕NLOCK ¯1 20       ⍝ write-lock from byte 20 onwards.
      2 ⎕NLOCK ¯1 10 2     ⍝ write-lock 2 bytes from byte 10
      0 ⎕NLOCK ¯1 12 1     ⍝ remove lock from byte 12.
```

To lock the region immediately beyond the end of the file prior extending it:

```
   ⎕←region←2 ⎕NLOCK ¯1, ⎕NSIZE ¯1 ⍝ write-lock from EOF.
¯1 1000
   ... ⎕NAPPEND ¯1                 ⍝ append bytes to file
   ... ⎕NAPPEND ¯1                 ⍝ append bytes to file

   0 ⎕NLOCK region                 ⍝ release lock.
```

The left argument may have a second optional item that specifies a *timeout* value. If a lock has not been acquired within this number of seconds, the acquisition is abandoned and a `TIMEOUT` error reported.

```
    2 10 ⎕NLOCK ¯1      ⍝ wait up to 10 seconds for lock.
```

Notes

- There is no *per-byte* cost associated with region locking. It takes the same time to lock/unlock a region, irrespective of that region's size.
- Different file servers implement locks in slightly different ways. For example on some systems, locks are *advisory*. This means that a write lock on a region precludes other locks intersecting that region, but doesn't stop reads or writes across the region. On the other hand, *mandatory* locks block both other locks *and* read/write operations. `⎕NLOCK` will just pass the server's functionality along to the APL programmer without trying to standardise it across different systems.
- All locks on a file will be removed by `⎕NUNTIE`.
- Blocked locking requests can be freed by a strong interrupt. Under Windows, this operation is performed from the Dyalog APL pop-up menu in the system tray.

Errors

- In this release, an attempt to unlock a region that contains bytes that have not been locked results in a `DOMAIN ERROR`.
- A `LIMIT ERROR` results if the operating system lock daemon has insufficient resources to honour the locking request.
- Some systems support only write locks. In this case an attempt to set a read lock will generate a `DOMAIN ERROR`, and it may be appropriate for the APL programmer to trap the error and apply a write lock.
- No attempt will be made to detect deadlock. Some servers do this and if such a condition is detected, a `DEADLOCK` error (1008) will be reported.

6.4.96 Native File Move {R}←X ⎕NMOVE Y

This function moves native files and directories from one or more sources specified by Y to a destination specified by X. ⎕NMOVE is similar to ⎕NCOPY (see *Section 6.4.86*).

When possible ⎕NMOVE *renames* files and directories, which effects a fast move when the source and destination are on the same file system. By default (see **RenameOnly** option below), if ⎕NMOVE is unable to rename files or directories, it instead copies them and deletes the originals.

X is a character vector that specifies the name of the destination.

Y is a character vector that specifies the name of the source, or a vector of character vectors containing zero or more sources.

Sources and destinations may be full or relative (to the current working directory) path names adhering to the operating system convention.

If Y specifies more than one source, X must be a character vector that specifies an existent directory to which each of the sources in Y is to be moved.

The shy result R contains count(s) of top-level items moved. If Y is a single source name, R is a scalar otherwise it is a vector of the same length as Y.

Variant Options

⎕NMOVE may be applied using the *variant* operator with the options **Wildcard** (the Principal option), **IfExists**, **RenameOnly** and **ProgressCallback**.

Wildcard Option (Boolean)

0	The name or names in Y identifies a specific file name.
1	The name or names in Y that specify the *base name* and *extension* (see *NParts*), may also contain the wildcard characters "?" and "*". An asterisk is a substitute for any 0 or more characters in a file name or extension; a question-mark is a substitute for any single character.

Note that when **Wildcard** is 1, element(s) of R can be 0 or >1. If **Wildcard** is 0, elements of R are always 1.

IfExists Option

The **IfExists** variant option determines what happens when a source file is to be copied to a target file that already exists. It does not apply to directories, only to the files within them.

Value	Description
'Error'	Existing files will not be overwritten and an error will be signalled.
`'Skip'`	Existing files will not be overwritten but the corresponding copy operation will be skipped (ignored).

The following cases cause an error to be signalled regardless of the value of the **IfExists** variant.

- If the source specifies a directory and the destination specifies an existing file.
- If the source specifies a file and the same base name exists as a sub-directory in the destination.

RenameOnly Option (Boolean)

The **RenameOnly** option determines what happens when it is not possible to rename the source.

0	The source will be copied and the original deleted
`1`	The move will fail

Examples

A number of possibilities exist, illustrated by the following examples. In all cases, if the source is a file, the file is moved. If the source is a directory, the directory and all of its contents are moved.

Single source, Wildcard is 0

- The source name must be an existent file or directory.
- If the destination name does not exist but its path name does exist, the source is moved to the destination name.

- If the destination name is an existing directory the source name is moved to that directory.

```
      ⍴1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

⍝ Rename the Session file
      ⊢'session.dlf' ⎕NMOVE 'default.dlf'
1
      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Move the Session file to backups directory
      ⊢'backups'⎕NMOVE'default.dlf'
1
      ↑⍴0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
```

Single source, Wildcard is 1

- The source name may include wildcard characters which matches a number of existing files and/or directories. The destination name must be an existing directory.
- The files and/or directories that match the pattern specified by the source name are moved into the destination directory. If there are no matches, zero copies are made.

```
       ⍴1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Move all files to backups directory
      ⊢'backups'(⎕NMOVE⍠'Wildcard' 1)'*.*'
3
      ↑⍴0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
backups/UserCommand20.cache
```

Multiple sources, Wildcard is 0

- Each source name must specify a single file or directory which must exist. The destination name must be an existing directory.
- Each of the files and/or directories specified by the source base names are moved to the destination directory.

```
      ⍴1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1

⍝ Move 2 files to backups directory
      ⊢'backups'⎕NMOVE'default.dlf' 'def_uk.dse'
1 1
      ↑⍴0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
```

Multiple sources, Wildcard is 1

- The destination name must be an existing directory.
- Each of the files and/or directories that match the patterns specified by the source names (if any) are moved to the destination directory.

```
      ⍴1 ⎕NPARTS ''
i:/Documents/Dyalog APL-64 17.0 Unicode Files/

      ⊢ ⎕MKDIR 'backups' ⍝ Make a backups directory
1
⍝ Move files to backups directory
      ⊢'backups'(⎕NMOVE⍠1)'d*' 'UserCommand20.cache'
2 1
      ↑⍴0 (⎕NINFO⍠1) 'backups\*'
backups/default.dlf
backups/def_uk.dse
backups/UserCommand20.cache
```

ProgressCallback Option

The **ProgressCallback** variant option is described in the *Programming: Native Files*. The following is specific to ⎕NMOVE:

- The first element of the right argument to the callback function is the character vector '⎕NMOVE'.

Note

When ⎕NMOVE copies and deletes files:

- The operation will take longer to complete.
- File modification times will be preserved but other attributes such as file ownership may be changed.
- Read permissions will be needed on all files within a directory which is moved.
- If the operation fails at any point and an error is signalled it is possible that there may be files and/or directories left duplicated in both the source and destination. It is not possible that a file or directory may be removed from the source without having been copied to the destination.

6.4.97 Native File Names R←⎕NNAMES

This niladic function reports the names of all currently open native files. R is a character matrix. Each row contains the name of a tied native file padded if necessary with blanks. The names are **identical** to those that were given when opening the files with ⎕NCREATE or ⎕NTIE. The rows of the result are in the order in which the files were tied.

6.4.98 Native File Numbers R←⎕NNUMS

This niladic function reports the tie numbers associated with all currently open native files. R is an integer vector of negative tie numbers. The elements of the result are in the order in which the files were tied.

6.4.99 File Name Parts R←{X} ⎕NPARTS Y

Splits a file or directory name into its constituent parts.

`Y` is a character vector or scalar containing a single name, or a vector of character vectors containing zero or more names. Names must conform to the file-naming rules of the host Operating System.

The file(s) need not exist; indeed this system function makes no attempt to identify or locate it/them.

The optional left-argument `X` specifies whether or not the name or names specified by `Y` are *normalised* before being processed. The default value 0 means no normalisation; 1 means normalise as follows:

- Pathnames are made absolute.
- On Windows, all "\" directory separators are changed to "/".
- The resultant name is simplified by removing extraneous directory separators etc. On Windows, this includes resolving occurrences of "." and ".." within the name. On non-Windows platforms single "." are removed. Note that ".." and symbolic links interact differently on Windows to other platforms; on other platforms they cannot be removed without reference to the file system itself and are left in place.

If `Y` is a scalar or vector, the result `R` is a 3-element vector of character vectors as follows:

`[1]`	*path*
`[2]`	*base name*
`[3]`	*extension*

The *path* identifies the directory in which the file exists.

The *base name* is the name of the file stripped of its path and extension, if any.

The *extension* is the file extension including the leading ".".

If `Y` is a vector of character vectors, `R` is a vector of 3-element character vectors and is the same length as `Y`.

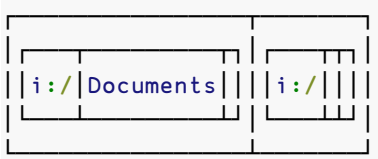
Examples

```
      ⎕CMD 'CD'⍵ Current working directory
c:\Users\Pete

      1 ⎕NPARTS 'a'
```

```
      1 ⎕NPARTS '\Users\Pete\Documents\dyalog.zip'
```

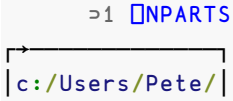
```
      ⎕'.'⎕WG'APLVersion'
AIX-64
      1 ⎕NPARTS'/home/andys/./..'
```

```
      1 ⎕NPARTS '.' '..'
```



Note that `⎕1 ⎕NPARTS ''` returns the current working directory.

```
      ⎕1 ⎕NPARTS ''
```



6.4.100 Write Text File {R}←X ⎕NPUT Y

This function writes character data to a text file. See also *Section 6.4.92*.

`Y` is either a simple character vector or scalar containing the name of the file to be written, or a 2-item vector whose first item is the file name and whose second is an integer scalar specifying `flags` for the operation.

If `flags` is 0 (the default value if omitted) the file will not be overwritten if it already exists and `NPUT` will signal an error. If `flags` is 1 the file will be overwritten. If flags is 2 the file will be appended to. That is:

flags	file does not exist	file exists
`0`	data is written to new file	error signalled, file is unchanged
`1`	data is written to new file	file is overwritten
`2`	data is written to new file	data is appended to file

The left-argument `X` is comprised of 1, 2 or 3 items which identify `(content) (encoding) (newline)` respectively.

`content` is either a vector of character vectors, each of which represents a line in the file to be written, a simple character vector, or a character matrix. If it is a matrix, then trailing spaces will be stripped from each row.

If specified, `encoding` is either:

- a character vector from the first column in the table *Section 6.4.92*. If `encoding` specifies a UTF format, it may be qualified with -BOM (for example, UTF-8-BOM), which causes a Byte Order Mark (BOM) to be written at the beginning of the file or -NOBOM which does not. If the -BOM or -NOBOM suffix is omitted, UTF-8 defaults to UTF-8-NOBOM, while the other UTF formats default to -BOM.
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

If omitted, `encoding` defaults to UTF-8-NOBOM.

Note: If a non-empty file is appended to:

- No BOM will be written, even if encoding specifies it.
- No check is made that the existing file content is text in the same encoding format.

If specified, `newline` is numeric and is either `0` or a scalar or vector from the column labelled *Value* in the **newline characters** section of the table *Section 6.4.92*. Any other value causes `DOMAIN ERROR`. If `newline` is omitted it defaults to `(13 10)` on Windows and `10` on other platforms.

In all cases, `newline` is appended if required to a simple vector or to each vector in a vector of vectors.

If content contains anything other than a character vector or scalar (or these, nested) then a `DOMAIN ERROR` is signalled.

If both `encoding` and `newline` are omitted `X` specifies only `content` and may be a simple character vector or a vector of character vectors.

The shy result `R` is the number of bytes written to the file.

Examples

```
      txt←'mene' 'mene' 'tekel' 'upharsin'

      ⍞←(⊂txt) ⎕NPUT 'writing.txt'
25
      ⊢(⊂'adding' '3' 'lines')⎕NPUT'writing.txt' 2
18

      LF←⎕UCS 10

      t←'adding',LF,'3',LF,'lines',LF
      ⊢t ⎕NPUT'writing.txt' 2
18
```

NEOL Option

The NEOL variant option specifies how embedded line separators are treated.

0	embedded line separator characters are preserved as is,and a `newline` is added to the last line if required.
1	every embedded LF is replaced by `newline`
2	every embedded line separator character is replaced by `newline`
¯1	same as 0 except that a `newline` is not added to the last line

Embedded line-separator examples

```
      LF CR←⎕UCS 10 13
      t←'adding',LF,'3',CR,'lines',CR,LF

      ⊢t ⎕NPUT'writing.txt' 2
17
      ⊢t(⎕NPUT⍠'NEOL' 0)'writing.txt' 2
16
      ⊢t(⎕NPUT⍠'NEOL' 2)'writing.txt' 2
18
```

`t` contains three lines each with different line endings: LF, CR and CRLF.

In the first example (NEOL is by default 1), only the LF is normalised so the written file contains lines ending with CRLF, CR and CRLF.

In the second example, none of the line endings are normalised so the written file contains lines ending with LF, CR and CRLF.

In the third example, all of the line endings are normalised so the written file contains lines ending with CRLF, CRLF and CRLF.

Note

If two or more APL processes separately write to a file using ⎕NPUT the behaviour is undefined. In particular, if multiple APLs use ⎕NPUT with flags set to 2 (append), it is not guaranteed that each ⎕NPUT operation will be atomic and all the text written to the file will be complete and/or contiguous.

6.4.101 Enqueue Event {R}←{X}⎕NQ Y

Windows only.

This system function generates an event or invokes a method.

While APL is executing, events occur "naturally" as a result of user action or of communication with other applications. These events are added to the event queue as and when they occur, and are subsequently removed and processed one by one by ⎕DQ. ⎕NQ provides an "artificial" means to generate an event and is analogous to ⎕SIGNAL.

If the left argument `X` is omitted or is 0, `⎕NQ` adds the event specified by `Y` to the bottom of the event queue. The event will subsequently be processed by `⎕DQ` when it reaches the top of the queue.

If `X` is 1, the event is actioned **immediately** by `⎕NQ` itself and is processed in exactly the same way as it would be processed by `⎕DQ`. For example, if the event has a callback function attached, `⎕NQ` will invoke it directly. See *Section 6.4.31* for further details. If the event generates any subsidiary events (for example, a KeyPress might generate a GotFocus), the subsidiary events are added to the event queue rather than being executed immediately.

Note that it is not possible for one thread to use `1 ⎕NQ` to send an event to another thread.

If `X` is 2 and the name supplied is the name of an event, `⎕NQ` performs the default processing for the event immediately, but does **not** invoke a callback function if there is one attached.

If `X` is 2 and the name supplied is the name of a (Dyalog APL) method, `⎕NQ` invokes the method. Its (shy) result is the result produced by the method.

If `X` is 3, `⎕NQ` invokes a method in an OLE Control. The (shy) result of `⎕NQ` is the result produced by the method.

If `X` is 4, `⎕NQ` signals an event from an ActiveXControl object to its host application. The (shy) result of `⎕NQ` is the result returned by the host application and depends upon the syntax of the event. This case is only applicable to ActiveXControl objects.

`Y` is a nested vector containing an event message. The first two elements of `Y` are:

`[1]`	Object	ref or character vector
`[2]`	Event	numeric scalar or character vector which specifies an event or method

`Y[1]` must specify an *existing* object. If not, `⎕NQ` terminates with a `VALUE ERROR`.

If `Y[2]` specifies a standard event type, subsequent elements must conform to the structure defined for that event type. If not, `⎕NQ` terminates with a `SYNTAX ERROR`. If additional elements (beyond those defined for the event type) are supplied this will not cause an error, but is not recommended because Dyalog may extend the event message in the future.

If `Y[2]` specifies a non-standard event type, `Y[3]` onwards (if present) may contain arbitrary information. Although any event type not listed herein may be used, numbers in the range 0-1000 are reserved for future extensions.

If `⎕NQ` is used monadically, or with a left argument of 0, its (shy) result is always an empty character vector. If a left argument of 1 is specified, `⎕NQ` returns `Y` unchanged or a modified `Y` if the callback function returns its modified argument as a result.

If the left argument is 2, `⎕NQ` returns either the value 1 or a value that is appropriate.

Examples

```
      ⍝ Send a keystroke ("A") to an Edit Field
      ⎕NQ TEST.ED 'KeyPress' 'A'

      ⍝ Iconify all top-level Forms
      {⎕NQ ⍵ 'StateChange' 1}¨'Form'⎕WN'.'

      ⍝ Set the focus to a particular field
      ⎕NQ TEST.ED3 40

      ⍝ Throw a new page on a printer
      1 ⎕NQ PR1 'NewPage'

      ⍝ Terminate ⎕DQ under program control

      'TEST'⎕WC 'Form' ... ('Event' 1001 1)
      ...
      ⎕DQ 'TEST'
      ...
      ⎕NQ TEST 1001  ⍝ From a callback

      ⍝ Call GetItemState method for a TreeView F.TV
      +2 ⎕NQ F.TV 'GetItemState' 6
96

      ⍝ Report where APL is installed
      +2 ⎕NQ'.' 'GetEnvironment' 'DYALOG'
C:\Program Files\Dyalog\Dyalog APL-64 15.0 Unicode
```

6.4.102 Nested Representation R←⎕NR Y

Y must be a simple character scalar or vector which represents the name of a function or a defined operator.

If Y is a name of a defined function or defined operator, R is a vector of text vectors. The first element of R contains the text of the function or operator header. Subsequent elements contain lines of the function or operator. Elements of R contain no unnecessary blanks, except for leading indentation of control structures and the blanks which precede comments.

If Y is the name of a variable, a locked function or operator, an external function or a namespace, or is undefined, R is an empty vector.

EXAMPLE

```
      ∇R←MEAN X    ⍝ Average
[1]   R←(+/X)÷⍴X
      ∇

      +F←⎕NR'MEAN'
  R←MEAN X    ⍝Average   R←(+/X)÷⍴X

      ⍴F
2
      ]display F
┌→──────────────────────────────────────┐
│ ┌→────────────────────┐  ┌→─────────┐ │
│ │ R←MEAN X    ⍝ Average│  │ R←(+/X)÷⍴X│ │
│ └─────────────────────┘  └──────────┘ │
└∊──────────────────────────────────────┘
```

The definition of ⎕NR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators. In these cases, the result of ⎕NR is identical to that of ⎕CR except that the representation of defined functions and operators is as described above.

EXAMPLE

```
      AVG←MEAN∘,

      +F←⎕NR'AVG'
   R←MEAN X    ⍝ Average   R←(+/X)÷⍴X  ∘,

      ⍴F
3
      ]display F
┌→──────────────────────────────────────────────┐
│ ┌→────────────────────────────────────┐       │
│ │ ┌→──────────────────┐  ┌→─────────┐ │ ∘ , │
│ │ │ R←MEAN X    ⍝ Average│  │ R←(+/X)÷⍴X│ │ - - │
│ │ └───────────────────┘  └──────────┘ │       │
│ └∊────────────────────────────────────┘       │
└∊──────────────────────────────────────────────┘
```

6.4.103 Native File Read R←⎕NREAD Y

This monadic function reads data from a native file. `Y` is a 3- or 4-element integer vector whose elements are as follows:

`[1]`	negative tie number,
`[2]`	conversion code (see below),
`[3]`	count. If this value is `¯1` , all of the elements defined by `Y[2]` are read from the position specified by `Y[4]` to the end of the file. This may result in the last few bytes in the file being ignored if they do not form a complete element.
`[4]`	start byte, counting from 0. If this value omitted or is `¯1` , data is read starting from the current position in the file (initially `0`).

Notes

`8 ⎕NINFO ⊃Y` can be used to report the current position of the file pointer.

`Y[2]` specifies conversion to an APL internal form as follows. Note that the internal formats for character arrays differ between the Unicode and Classic Editions.

If both `Y[3]` and `Y[4]` have the value `¯1`, then `⎕NREAD` reads data from the current position in the file to the end of the file.

`⎕NREAD` can be used with any file. However, calling `⎕NREAD` with at least one of `Y[3 4]` set to `¯1` is intended for regular files only; using on pipes, FIFOs or other special types of file is not recommended.

Table: Unicode Edition: Conversion Codes

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
80	count	8 bits character	count
82 [22]	count	8 bits character	count
83	count	8 bits integer	count
160	2 × count	16-bits character	count
163	2 × count	16 bits integer	count
320	4 × count	32-bits character	count
323	4 × count	32 bits integer	count
645	8 × count	64 bits floating	count
1287	16 × count	128 bits decimal	count
1289	16 × count	128 bits complex	count

Table: Classic Edition: Conversion Codes

[22] Conversion code 82 is permitted in the Unicode Edition for compatibility and causes 1-byte data on file to be translated (according to ⎕NXLATE) from ⎕AV indices into normal (Unicode) characters of type 80, 160 or 320.

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	8 × count
82	count	8 bits character	count
83	count	8 bits integer	count
163	2 × count	16 bits integer	count
323	4 × count	32 bits integer	count
645	8 × count	64 bits floating	count
1287	16 × count	128 bits decimal	count
1289	16 × count	128 bits complex	count

EXAMPLE

```
      DATA←⎕NREAD ¯1 160 (0.5×⎕NSIZE ¯1) 0 ⍝ Unicode
      DATA←⎕NREAD ¯1 82 (⎕NSIZE ¯1) 0      ⍝ Classic
      DATA←⎕NREAD ¯1 82 ¯1 0       ⍝ Shorter version
```

6.4.104 Native File Rename {R}←X ⎕NRENAME Y

⎕NRENAME is used to rename a native file.

Y is a negative integer tie number associated with a tied native file. X is a simple character vector or scalar containing a valid (and unused) file name.

The shy result of ⎕NRENAME is the tie number of the renamed file.

6.4.105 Native File Replace {R}←X ⎕NREPLACE Y

⎕NREPLACE is used to write data to a native file, replacing data which is already there.

X must be a simple homogeneous APL array containing the data to be written.

Y is a 2- or 3-element integer vector whose elements are as follows:

`[1]`	negative tie number,
`[2]`	start byte, counting from 0, at which the data is to be written; the value `¯1` causes the data to be written from the current position in the file (initially, 0).
`[3]`	conversion code (optional).

See *Section 6.4.103* for a list of valid conversion codes.

The shy result is the location of the internal file pointer which will be pointing to the end of the newly written data. Used, for example, in:

```
      ⍝ Replace sequentially from indx.
      {⍺ ⎕NREPLACE tie ⍵}/vec,indx
```

Note that `8 ⎕NINFO ⍠Y` can be used to report the current position of the file pointer.

Unicode Edition

Unless you specify the data type in `Y[3]`, a character array will by default be written using type 80.

If the data will not fit into the specified character width (bytes) `⎕NREPLACE` will fail with a `DOMAIN ERROR`.

As a consequence of these two rules, you must specify the data type (either 160 or 320) in order to write Unicode characters whose code-point is in the range 256-65535 and >65535 respectively.

EXAMPLE

```
      n←'test'⎕NTIE 0 ⍝ See "Example"

      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 7
ταβέρνα

      ⎕←'εστιατόριο'⎕NREPLACE n 3
DOMAIN ERROR
      ⎕←'εστιατόριο'⎕NREPLACE n 3
     ^

      ⎕←'εστιατόριο'⎕NREPLACE n 3 160
23
      ⎕NREAD n 80 3 0
abc
      ⎕NREAD n 160 10
εστιατόριο
```

For compatibility with old files, you may specify that the data be converted to type 82 on output. The conversion (to ⎕AV indices) will be determined by the local value of ⎕AVU.

6.4.106 Native File Resize {R}←X ⎕NRESIZE Y

This function changes the size of a native file.

Y is a negative integer tie number associated with a tied native file.

X is a single integer value that specifies the new size of the file in bytes. If X is smaller than the current file size, the file is truncated. If X is larger than the current file size, the file is extended and the value of additional bytes is undefined.

The shy result of ⎕NRESIZE is the tie number of the resized file.

6.4.107 Namespace {R}←{X}⎕NS Y

The ⎕NS system function makes it possible to create namespaces, copy elements from one namespace to another, and clone namespaces.

`Y` is one of the following:

- an vector of zero or more objects to be copied.
- an array containing references to, and/or *Section 6.4.116*s of, one or more namespaces.

If specified, `X` must be an array that identifies one or more namespaces. This means `X` must be one of:

- a simple character scalar or vector identifying the name of a namespace.
- a reference to a namespace.
- an array where each item is one of the above. If `X` refers to multiple namespaces, then `⎕NS` processes each item of `X` in ravel order, using the entire argument `Y`; this is equivalent to `X ⎕NS¨⊂Y`. If `X` is empty, no action is taken.

The result `R` is shy when the system function is invoked dyadically, otherwise its contents are determined by the value of `Y`.

Usage

`⎕NS` is used to create or populate one or more namespaces based on either a list of members to be copied into the target namespace(s), or a list of objects to be merged into the target namespace(s).

For simplicity, `X` is treated in the following as if it refers to only a single namespace, but is subject to the aforementioned equivalence of `X ⎕NS¨⊂Y`.

Case 1: Create or Populate Namespace from Member List

`Y` must be a simple character scalar, vector, matrix, or a nested vector of character vectors identifying zero or more workspace objects to be copied into `X`. The identifiers in `X` and `Y` can be simple names or compound names separated by `'.'` and including the names of the special namespaces `'#'`, `'##'` and `'⎕SE'`.

The treatment of `X` varies:

X	Treatment
Omitted (monadic ⎕NS)	Anonymous namespace created
Not in use	Namespace X created
Existing object	Used as-is
Existing non-object variable	Variable replaced with new namespace
Existing non-variable	DOMAIN ERROR

The objects identified in the list Y are copied into the namespace X.

If X is specified, the result R is the full name (starting with #. or ⎕SE.) of the namespace X. If X is omitted, the result R is a namespace reference to an unnamed namespace.

Examples

```
      ⎕←'X'⎕NS''                  ⍝ Create namespace X
#.X
      ⎕←'X'⎕NS'VEC' 'UTIL.DISP'   ⍝ Copy VEC and DISP to X
#.X
      )CS X                       ⍝ Change to namespace X
#.X
      ⎕←'Y'⎕NS'#.MAT' '##.VEC'    ⍝ Create #.X.Y and copy in
#.X.Y
      ⎕←'#.UTIL'⎕NS'Y.MAT'        ⍝ Copy MAT from Y to UTIL #.UTIL
#.UTIL
      ⎕←'#'⎕NS'Y'                 ⍝ Copy namespace Y to root
#
```

```
      ⎕←''⎕NS'#.MAT'            ⍝ Copy MAT to current space
#.X
      ⎕←''⎕NS''                 ⍝ Display current space
#.X
```

```
      NONAME←⎕NS ''            ⍝ Create unnamed namespace
      NONAME
#.[Namespace]
```

```
      DATA←⎕NS¨3⍴⊂''          ⍝ Create 3-element vector of
                              ⍝ distinct unnamed nspaces
      DATA
#.[Namespace]  #.[Namespace]  #.[Namespace]
      'one' 'two' ⎕NS 'DATA'
      one.⎕NL ¯2
DATA
```

Case 2: Create or Populate Namespace from Object List

`Y` is one or more references to, or `⎕OR`s of, namespaces.

If `Y` contains a reference to, or a `⎕OR` of, a *GUI* object, `X` must be a valid parent for the GUI object represented by `Y`, or the operation will fail with a `DOMAIN ERROR`. A maximum of one element of `Y` can represent a GUI object or the operation will fail with a `NONCE ERROR`.

If `Y` does not contain a reference to, or a `⎕OR` of, a *GUI* object, the result of the operation depends upon the existence of `X`:

- If `X` does not currently exist (name class is 0), `X` is created as a complete copy (clone) of the original namespace represented by `Y`. If `Y` contains a reference to, or the `⎕OR` of, a GUI object or of a namespace containing GUI objects, the corresponding GUI components of `Y` will be instantiated in `X`.
- If `X` is the name of an existing namespace (name class 9), the contents of `Y`, including any GUI components, are merged into `X`. Any items in `X` with corresponding names in `Y` (names with the same path in both `Y` and `X`) will be replaced by the names in `Y`, unless they have a conflicting name class in which case the existing items in `X` will remain unchanged. However, all GUI spaces in `X` will be stripped of their GUI components prior to the merge operation.

`Y` can also be a vector of namespaces, in which case each item of `Y` is processed as explained above, in ravel order. The effect is that the contents of all the namespaces are merged into the target namespace.

Examples

```
      original←⎕NS⍬
      original.(A B C)←1 2 3
      'new' ⎕NS ⎕OR'original'  ⍝ cloning a namespace from ⎕OR
      new.A
1
      cloned←⎕NS original  ⍝ cloning a namespace from reference
      cloned.D←4

      original.⎕NL ¯2
 A  B  C
      cloned.⎕NL ¯2
 A  B  C  D
```

```
      defaults←(
        name: '<no name>'
        age: '<no age>'
        phone: '<no phone>'
        email: '<no email>'
      )
      jack←(name: 'Jack' ⋄ email: 'jack@example.com')
      person←(age: 42 ⋄ phone: 12345678)
      show←⎕JSON⍠'Compact' 0

      show ⎕NS defaults jack  ⍝ merge defaults and jack
{
  "age": "<no age>",
  "email": "jack@example.com",
  "name": "Jack",
  "phone": "<no phone>"
}
      show ⎕NS defaults person  ⍝ merge defaults and person
{
  "age": 42,
  "email": "<no email>",
  "name": "<no name>",
  "phone": 12345678
}
```

Variant Option: Trigger

The `Trigger` variant option specifies whether any *Programming: Triggers* should be run for the modified variables in the target namespace that have triggers attached. The value must be a Boolean scalar. The default is 0, meaning that triggers are not run.

EXAMPLE

```
      ⎕VR 'trigger'
     ∇trigger arg
[1]   :Implements Trigger X,Y
[2]   ⎕←'Running trigger for: ',arg.Name
     ∇

      newValues←(Y: 1 ⋄ Z: 2)

      ⍝ ⎕NS without running triggers
      ⎕THIS ⎕NS newValues
      ⎕THIS ⎕NS⍠'Trigger' 0⊢newValues

      ⍝ ⎕NS running triggers
      ⎕THIS ⎕NS⍠'Trigger' 1⊢newValues
Running trigger for: Y
```

6.4.108 Namespace Indicator R←⎕NSI

`R` is a nested vector of character vectors containing the names of the spaces from which functions in the state indicator were called (`⍴⎕NSI←→⍴⎕RSI←→⍴⎕SI`).

`⎕RSI` and `⎕NSI` are identical except that `⎕RSI` returns refs to the spaces whereas `⎕NSI` returns their names. Put another way: `⎕NSI←→⍎¨⎕RSI`.

Note that `⎕NSI` contains the names of spaces *from which* functions were called not those *in which* they are currently running.

EXAMPLE

```
      )OBJECTS
xx      yy

      ⎕VR 'yy.foo'
     ∇ r←foo
[1]    r←⎕SE.goo
     ∇
      ⎕VR'⎕SE.goo'
     ∇ r←goo
[1]    r←⎕SI,[1.5]⎕NSI
     ∇

      )CS xx
#.xx
      calling←#.yy.foo
      ]display calling
┌→──────────────┐
↓ ┌→──┐ ┌→───┐ │
│ │goo│ │#.yy│ │
│ └───┘ └────┘ │
│ ┌→──┐ ┌→───┐ │
│ │foo│ │#.xx│ │
│ └───┘ └────┘ │
└∊──────────────┘
```

6.4.109 Native File Size R←⎕NSIZE Y

This reports the size of a native file.

`Y` is a negative integer tie number associated with a tied native file. The result `R` is the size of the file in bytes.

6.4.110 Native File Tie {R}←X ⎕NTIE Y

`⎕NTIE` opens a native file.

`X` is a simple character vector or scalar containing a valid pathname for an existing native file.

`Y` is a 1- or 2-element vector.

`Y[1]` is a negative integer value that specifies an (unused) tie number by which the file may subsequently be referred.

`Y[2]` is optional and specifies the mode in which the file is to be opened. This is an integer value calculated as the sum of 2 codes. The first code refers to the type of access needed from users who have already tied the native file. The second code refers to the type of access you wish to grant to users who subsequently try to open the file while you have it open.

If `Y[2]` is omitted, the system tries to open the file with the default value of 66 (read and write access for this process and for any subsequent processes that attempt to access the file). If this fails, the system attempts to open the file with the value 64 (read access for this process, read and write for subsequent processes).

Needed from existing users		Granted to subsequent users	
0	read access	0	see note [23]
1	write access	16	no access (exclusive)
2	read and write access	32	read access
		48	write access
		64	read and write access

On UNIX systems, the second column has no meaning and only the first code (`16|mode`) is passed to the `open(2)` call as the access parameter. See include file `fcntl.h` for details. See also *Section 6.4.95* which is not platform dependent.

`R` is the tie number by which the file may subsequently be referred. If `Y[1]` is a negative integer, then `R` is a shy result; if `Y[1]` is 0, `R` is an explicit result.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

[23] The original meaning of this value is no longer relevant. 0 now means the same as 16 (no access).

```
      tie←¯1+⌊/0,⎕NNUMS     ⍝ With next available number,
      file ⎕NTIE tie        ⍝ ... tie file.
```

to:

```
      tie←file ⎕NTIE 0      ⍝ Tie with first available no.
```

EXAMPLE

```
ntie←{                      ⍝ tie file and return tie no.
    ⍺←2+64                  ⍝ default all access.
    ⍵ ⎕NTIE 0 ⍺             ⍝ return new tie no.
}
```

> **Note**
>
> If the native file is already tied, executing `⎕NTIE` with the same or a different tie number simply re-ties it with the same or the new tie number. Re-tying a file with a tie number of 0, re-ties it with the same tie number. This feature can be used to re-tie the file using a different mode.

6.4.111 Null Item R←⎕NULL

This is a reference to a null item, such as may be returned across the COM interface to represent a null value. A null might be returned as the result of a .NET method or as the value of an empty cell in a spreadsheet

`⎕NULL` may be used in any context that accepts a namespace reference, in particular:

- As the argument to a defined function
- As an item of an array.
- As the argument to those primitive functions that take character data arguments, for example: `=, ≠, ≡, ≢, ,, ⍴, ⊃, ⊂`

EXAMPLE

```
      'EX'⎕WC'OLEClient' 'Excel.Application'
      WB←EX.Workbooks.Open 'simple.xls'

      (WB.Sheets.Item 1).UsedRange.Value2
[Null]  [Null]  [Null]  [Null]  [Null]
[Null]    Year  [Null]  [Null]  [Null]
[Null]    1999    2000    2001    2002
[Null]  [Null]  [Null]  [Null]  [Null]
Sales      100      76     120     150
[Null]  [Null]  [Null]  [Null]  [Null]
Costs       80      60     100     110
[Null]  [Null]  [Null]  [Null]  [Null]
Margin      20      16      20      40
```

To determine which of the cells are filled, you can compare the array with ⎕NULL.

```
      ⎕NULL≠¨(WB.Sheets.Item 1).UsedRange.Value2
0 0 0 0 0
0 1 0 0 0
0 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
0 0 0 0 0
1 1 1 1 1
```

6.4.112 Native File Untie {R}←⎕NUNTIE Y

This closes one or more native files. Y is a scalar or vector of negative integer tie numbers. The files associated with elements of Y are closed. Native file untie with a zero length argument (⎕NUNTIE 0) flushes all file buffers to disk - see *Section 6.4.68* for more explanation.

The shy result of ⎕NUNTIE is a vector of tie numbers of the files **actually untied**.

6.4.113 Native File Translate {R}←{X}⎕NXLATE Y

This associates a character translation vector with a native file or, if Y is 0, with the use by ⎕DR.

A translate vector is a 256-element vector of integers from 0-255. Each element maps the corresponding ⎕AV position onto an ANSI character code.

For example, to map ⎕AV[17+⎕IO] onto ANSI 'a' (code 97), element 17 of the translate vector is set to 97.

⎕NXLATE is a non-Unicode (Classic Edition) feature and is retained in the Unicode Edition only for compatibility.

Y is either a negative integer tie number associated with a tied native file or 0. If Y is negative, monadic ⎕NXLATE returns the current translation vector associated with the corresponding native file. If specified, the left argument X is a 256-element vector of integers that specifies a new translate vector. In this case, the old translate vector is returned as a shy result. If Y is 0, it refers to the translate vector used by ⎕DR to convert to and from character data.

The system treats a translate vector with value (⍳256)-⎕IO as meaning *no translation* and thus provides raw input/output bypassing the whole translation process.

The default translation vector established at ⎕NTIE or ⎕NCREATE time is derived from the mapping defined in the current output translation table (normally WIN.DOT) and maps alphabetic, numeric and most other characters in ⎕AV to their corresponding ANSI positions. However, some characters are not resolved by this process and it is recommended that users define translate vectors to cover all cases.

Unicode Edition

⎕NXLATE is relevant in the Unicode Edition only to process Native Files that contain characters expressed as indices into ⎕AV, such as files written by the Classic Edition.

In the Unicode Edition, when reading data from a Native File using conversion code 82, incoming bytes are translated first to ⎕AV indices using the translation table specified by ⎕NXLATE, and then to type 80, 160 or 320 using ⎕AVU. When writing data to a Native File using conversion code 82, characters are converted using these two translation tables in reverse.

6.4.114 Sign Off APL ⎕OFF

This niladic system function terminates the APL session, returning to the shell command level. The active workspace does not replace the last continuation workspace.

Although ⎕OFF is niladic, you may specify an optional integer I to the right of the system function which will be reported to the Operating System as the exit code. If I is an *expression* generating an integer, you should put the expression in parentheses. I must be in the range 0..255, but note that on UNIX processes use values greater than 127 to indicate the signal number which was used to terminate a process, and that currently APL itself generates values 0..8; this list may be extended in future. This list is documented in the *Dyalog for Microsoft Windows Installation and Configuration Guide: APL Exit Codes*.

6.4.115 Variant {R}←{X}(f ⎕OPT B)Y

⎕OPT is synonymous with the Variant Operator ⍠ and can be used in both Classic and Unicode Editions - unlike ⍠ which is valid in Unicode Editions only, and ⎕U2360 which is valid in Classic Editions only.

See *Section 4.4.22*.

6.4.116 Object Representation R←⎕OR Y

⎕OR converts a defined function, defined operator or namespace to a special form, described as its *object representation*, that may be assigned to a variable and/or stored on a component file[24] . Classes and Instances are however outside the domain of ⎕OR.

Taking the ⎕OR of a defined function or operator is an extremely fast operation as it simply changes the type information in the object's header, leaving its internal structure unaltered. Converting the object representation back to an executable function or operator using ⎕FX is also very fast.

However, the saved results of ⎕OR which were produced on a different hardware platform or using an older version of Dyalog APL may require a significant amount of processing when re-constituted using ⎕FX. For optimum performance, it is strongly recommended that you save ⎕ORs using the same version of Dyalog APL and on the same hardware platform that you will use to ⎕FX them.

⎕OR may also be used to convert a namespace (either a plain namespace or a named GUI object created by ⎕WC) into a form that can be stored in a variable or on a component file. The namespace may be reconstructed using ⎕NS or ⎕WC with its original

[24] ⎕OR and GUI objects stored in workspaces or in component files are not portable between 32-bit and 64-bit versions of Dyalog nor between different implementations (platforms) and are not backwards compatible

name or with a new one. ⎕OR may therefore be used to *clone* a namespace or GUI object.

Y must be a simple character scalar or vector which contains the name of an APL object.

If Y is the name of a variable, the result R is its value. In this case, R←⎕OR Y is identical to R←⍎Y.

Otherwise, R is a special form of the name Y, re-classified as a variable. The rank of R is 0 (R is scalar), and the depth of R is 1. These unique characteristics distinguish the result of ⎕OR from any other object. The Type of R (∊R) is itself. Note that although R is scalar, it may not be index assigned to an element of an array unless it is enclosed.

If Y is the name of a function or operator, R is in the domain of the monadic functions Same (⊣ and ⊢), Depth (≡), Disclose (⊃), Enclose (⊂), Rotate (⌽), Transpose (⍉), Index ([]), Indexing ([ ]), Format (⍕), Identity (+), Shape (⍴), Type (∊) and Unique (∪), of the dyadic functions Left (⊣), Right (⊢), Without (~), Index Of (⍳), Intersection (∩), Match (≡), Membership (∊), Not Match (≠) and Union (∪), and of the monadic system functions Canonical Representation (⎕CR), Cross-Reference (⎕REFS), Fix (⎕FX), Format (⎕FMT), Nested Representation (⎕NR) and Vector Representation (⎕VR).

Note that a ⎕OR object can be transmitted through an 'APL-style' TCP socket. This technique may be used to transfer objects including namespaces between APL sessions.

The object representation forms of namespaces produced by ⎕OR may not be used as arguments to any primitive functions. The only operations permitted for such objects (or arrays containing such objects) are ⎕EX, ⎕FAPPEND, ⎕FREPLACE, ⎕NS, and ⎕WC.

Example

```
      F←⎕OR ⎕FX'R←FOO' 'R←10'

      ⍴F

      ⍴⍴F
0
      ≡F
1
      F≡∊F
1
```

The display of the ⎕OR form of a function or operator is a listing of the function or operator. If the ⎕OR form of a function or operator has been enclosed, then the result will display as the name preceded by the symbol ∇. It is permitted to apply ⎕OR to a locked function or operator. In this instance the result will display as for the enclosed form.

Examples

```
      ∇F
      ∇ R←FOO
[1]     R←10
      ∇

      ⊂F
 ∇FOO

      ⎕LOCK'FOO'

      ⎕OR'FOO'
∇FOO
```

```
      A←⍳5

      A[3]←⊂F

      A
1 2 ∇FOO 4 5
```

For the ⎕OR forms of two functions or operators to be considered identical, their unlocked display forms must be the same, they must either both be locked or unlocked, and any monitors, trace and stop vectors must be the same.

Example

```
      F←⎕OR ⎕FX 'R←A PLUS B' 'R←A+B'

      F≡⎕OR 'PLUS'
1

      1 ⎕STOP 'PLUS'

      F≡⎕OR 'PLUS'
0
```

Namespace Examples

The following example sets up a namespace called `UTILS`, copies into it the contents of the `UTIL` workspace, then writes it to a component file:

```
      )CLEAR
clear ws
      )NS UTILS
#.UTILS
      )CS UTILS
#.UTILS
      )COPY UTIL
C:\WDYALOG\WS\UTIL saved Fri Mar 17 12:48:06 1995
      )CS
#
      'ORTEST' ⎕FCREATE 1
      (⎕OR'UTILS')⎕FAPPEND 1
```

The namespace can be restored with `⎕NS`, using either the original name or a new one:

```
      )CLEAR
clear ws
      'UTILS' ⎕NS ⎕FREAD 1 1
#.UTILS
      )CLEAR
clear ws
      'NEWUTILS' ⎕NS ⎕FREAD 1 1
#.NEWUTILS
```

This example illustrates how `⎕OR` can be used to clone a GUI object; in this case a Group containing some Button objects. Note that `⎕WC` will accept **only** a `⎕OR` object as its argument (or preceded by the "Type" keyword). You may not specify any other properties in the same `⎕WC` statement, but you must instead use `⎕WS` to reset them afterwards.

```
    'F'⎕WC'Form'
    'F.G1' ⎕WC 'Group' '&One' (10 10)(80 30)
    'F.G1.B2'⎕WC'Button' '&Blue' (40 10)('Style' 'Radio')
    'F.G1.B3'⎕WC'Button' '&Green' (60 10)('Style' 'Radio')
    'F.G1.B1'⎕WC'Button' '&Red' (20 10)('Style' 'Radio')
    'F.G2' ⎕WC ⎕OR 'F.G1'
    'F.G2' ⎕WS ('Caption' 'Two')('Posn' 10 60)
```

Note too that ⎕WC and ⎕NS may be used interchangeably to rebuild *pure* namespaces or GUI namespaces from a ⎕OR object. You may therefore use ⎕NS to rebuild a Form or use ⎕WC to rebuild a pure namespace that has no GUI components.

6.4.117 Search Path ⎕PATH

⎕PATH is a simple character vector representing a blank-separated list of namespaces. It is approximately analogous to the PATH variable in Windows or UNIX.

The ⎕PATH variable can be used to identify a namespace in which commonly used utility functions reside. Functions or operators (**NOT** variables) which are copied into this namespace and *exported* (see *Section 6.4.40*) can then be used directly from anywhere in the workspace without giving their full path names. ⎕PATH has Session scope.

EXAMPLE

To make the DISPLAY function available directly from within any namespace.

```
      ⍝ Create and reference utility namespace.
      ⎕PATH←'⎕SE.util'⎕NS''
      ⍝ Copy DISPLAY function from UTIL into it.
      'DISPLAY'⎕SE.util.⎕CY'UTIL'
      ⍝ (Remember to save the session to file).
```

In detail, ⎕PATH works as follows:

When a reference to a name cannot be found in the current namespace, the system searches for it from left to right in the list of namespaces indicated by ⎕PATH. In each namespace, if the name references a defined function (or operator) *and* the export type of that function is non-zero (see *Section 6.4.40*), then it is used to satisfy the reference. If the search exhausts all the namespaces in ⎕PATH without finding a qualifying reference, the system issues a VALUE ERROR in the normal manner.

The special character ↑ stands for the list of namespace ancestors:

```
      ## ##.## ##.##.## ...
```

In other words, the search is conducted upwards through enclosing namespaces, emulating the static scope rule inherent in modern block-structured languages.

Note that the ⎕PATH mechanism is used ONLY if the function reference cannot be satisfied in the current namespace. This is analogous to the case when the Windows or UNIX PATH variable begins with a '.'.

Examples

```
   ⎕PATH                Search in ...

1. '⎕SE.util'           Current space,   then
                        ⎕SE.util,        then
                        VALUE ERROR

2. '↑'                  Current space
                        Parent space: ##
                        Parent's parent space:  ##.##
                        ...
                        Root: # (or ⎕SE if current space
                                 was inside ⎕SE)
                        VALUE ERROR

3. 'util ↑ ⎕SE.util'    Current space
                        util (relative to current space)
                        Parent space: ##
                        ...
                        Root: # or ⎕SE
                        ⎕SE.util
                        VALUE ERROR
```

Note that ⎕PATH is a *session* variable. This means that it is workspace-wide and survives)LOAD and)CLEAR. It can of course, be localised by a defined function or operator.

⎕PATH does not support derived functions and will not be extended to support them; nor will it be extended to support other types of functions that may be developed in the future. ⎕PATH may therefore be considered an archaic feature.

6.4.118 Program Function Key R←{X}⎕PFKEY Y

⎕PFKEY is a system function that sets or queries the programmable function keys. ⎕PFKEY associates a sequence of keystrokes with a function key. When the user subsequently presses the key, it is as if he had typed the associated keystrokes one by one.

Note that Ride does not currently support the use of ⎕PFKEY; it is possible however to associate simple strings to function keys - see the Ride User Guide for more information.

`Y` is an integer scalar in the range 0-255 specifying a programmable function key. If `X` is omitted the result `R` is the current setting of the key. If the key has not been defined previously, the result is an empty character vector.

If `X` is specified it is a simple or nested character vector defining the new setting of the key. The value of `X` is returned in the result `R`.

The elements of `X` are either character scalars or 2-element character vectors which specify keycodes. See *User Interface: Keyboard Shortcuts*.

Programmable function keys are recognised in any of the three types of window (SESSION, EDIT and TRACE) provided by the Dyalog APL development environment. `⎕SR` operates with the 'raw' function keys and ignores programmed settings.

Note that key definitions can reference other function keys, such as "F1" or "F123".

The size of the buffer associated with `⎕PFKEY` is specified by the *pfkey_size* parameter.

Examples

```
      (')FNS',⊂'ER')⎕PFKEY 1
┌─┬─┬─┬─┬──┐
│)│F│N│S│ER│
└─┴─┴─┴─┴──┘
      (')VARS',⊂'ER')⎕PFKEY 2
┌─┬─┬─┬─┬─┬──┐
│)│V│A│R│S│ER│
└─┴─┴─┴─┴─┴──┘
     'F1' 'F2' ⎕PFKEY 3 ⍝ Does )FNS and )VARS
┌──┬──┐
│F1│F2│
└──┴──┘
```

The following expression defines the action for F12 to be "move the text to the right of the cursor to the left of the cursor".

```
     'Rl' 'CT' 'LL' 'PT'⎕PFKEY 12
┌──┬──┬──┬──┐
│Rl│CT│LL│PT│
└──┴──┴──┴──┘
```

6.4.119 Print Precision ⎕PP

⎕PP is the number of significant digits in the display of numeric output. ⎕PP may be assigned any integer value in the range 1 to 34. ⎕PP has Namespace scope.

⎕PP is used to format numbers displayed directly. It is an implicit argument of monadic function Format (⍕), monadic ⎕FMT and for display of numbers via ⎕ and ⍞ output. ⎕PP is ignored for the display of integers.

EXAMPLES

```
      ⎕PP←10

      ÷3 6
0.3333333333 0.1666666667

      ⎕PP←3

      ÷3 6
0.333 0.167
```

If ⎕PP is set to a value ≥17 (when ⎕FR is 645) or 34 (when ⎕FR is 1287), floating-point numbers may be converted between binary and character representation without loss of precision. Then, if ⎕CT is 0 (to ensure exact comparison), for any floating-point number N the expression N=⍎⍕N is true.

6.4.120 Profile Application {R}←{X}⎕PROFILE Y

⎕PROFILE facilitates the profiling of either CPU consumption or elapsed time for a workspace. It does so by retaining time measurements collected for APL functions/operators and function/operator lines. ⎕PROFILE is used to both control the state of profiling and retrieve the collected profiling data.

See also: Application Tuning Guide.

Y specifies the action to perform and any options for that action, if applicable. Y is case-insensitive. Note that the result R is in some cases shy.

Use	Description
`{state}←⎕PROFILE 'start' {timer}`	Turn profiling on using the specified timer or resume if profiling was stopped
`{state}←⎕PROFILE 'stop'`	Suspend the collection of profiling data
`{state}←⎕PROFILE 'clear'`	Turn profiling off, if active, and discard any collected profiling data
`{state}←⎕PROFILE 'calibrate'`	Calibrate the profiling timer
`state←⎕PROFILE 'state'`	Query profiling state
`data←⎕PROFILE 'data'`	Retrieve profiling data in flat form
`data←⎕PROFILE 'tree'`	Retrieve profiling data in tree form

`⎕PROFILE` has 2 states:

- active – the profiler is running and profiling data is being collected.
- inactive – the profiler is not running.

For most actions, the result of `⎕PROFILE` is its current state and contains:

`[1]`	character vector indicating the `⎕PROFILE` state having one of the values `'active'` or `'inactive'`
`[2]`	character vector indicating the timer being used having one of the values `'CPU'` or `'elapsed'`
`[3]`	call time bias in milliseconds. This is the amount of time, in milliseconds, that is consumed for the system to take a time measurement
`[4]`	timer granularity in milliseconds. This is the resolution of the timer being used

{state}←⎕PROFILE 'start'

Turn profiling on; `timer` is an optional case-independent character vector containing `'CPU'` or `'elapsed'` or `'none'` or `'coverage'`. If omitted, it defaults to `'CPU'`. If `timer` is `'none'`, `⎕PROFILE` records just the number of times each line of code is executed without incurring the timing overhead. If `timer` is `'coverage'`, `⎕PROFILE` only identifies which lines of code are executed without incurring the timing or counting overhead.

The first time a particular timer is chosen, ⎕PROFILE will spend 1000 milliseconds (1 second) to approximate the call time bias and granularity for that timer.

```
      ⊢⎕PROFILE 'start' 'CPU'
active  CPU  0.0001037499999 0.0001037499999
```

{state}←⎕PROFILE 'stop'

Suspends the collection of profiling data.

```
      ⊢⎕PROFILE 'stop'
inactive  CPU  0.0001037499999 0.0001037499999
```

{state}←⎕PROFILE 'clear'

Clears any collected profiling data and, if profiling is active, places profiling in an inactive state.

```
      ⊢⎕PROFILE 'clear'
inactive    0 0
```

{state}←⎕PROFILE 'calibrate'

Causes ⎕PROFILE to perform a 1000 millisecond calibration to approximate the call time bias and granularity for the current timer. Note, a timer must have been previously selected by using ⎕PROFILE 'start'.

⎕PROFILE will retain the lesser of the current timer values compared to the new values computed by the calibration. The rationale for this is to use the smallest possible values of which we can be certain.

```
      ⊢⎕PROFILE'calibrate'
active  CPU  0.0001037499997 0.0001037499997
```

state←⎕PROFILE 'state'

Returns the current profiling state.

```
      )CLEAR
clear ws
      ⎕PROFILE 'state'
 inactive     0 0

      ⎕PROFILE 'start' 'CPU'
 active  CPU  0.0001037499997 0.0001037499997
      ⎕PROFILE 'state'
 active  CPU  0.0001037499997 0.0001037499997
```

data←{X} ⎕PROFILE 'data'

Retrieves the collected profiling data. If the optional left argument `X` is omitted, the result is a matrix with the following columns:

`[;1]`	function name
`[;2]`	function line number or `0` for a whole function entry
`[;3]`	number of times the line or function was executed
`[;4]`	accumulated time (ms) for this entry exclusive of items called by this entry
`[;5]`	accumulated time (ms) for this entry inclusive of items called by this entry
`[;6]`	number of times the timer function was called for the exclusive time
`[;7]`	number of times the timer function was called for the inclusive time

Example: (numbers have been truncated for formatting)

```
      ⎕PROFILE 'data'
#.foo             1  1.04406  39347.64945   503 4080803 #.foo
1       1  0.12488     0.124887     1        1 #.foo       2    100
0.58851 39347.193900   200 4080500 #.foo       3    100  0.21340
0.213406   100     100 #.NS1.goo        100 99.44404   39346.6053 50300
4080300 #.NS1.goo  1    100  0.61679     0.616793   100     100
#.NS1.goo  2  10000 67.80292   39314.9642 20000 4050000 #.NS1.goo  3
10000 19.60274      19.6027 10000   10000
```

If `X` is specified it must be a simple vector of column indices. In this case, the result has the same shape as `X` and is a vector of the specified column vectors:

```
X ⎕PROFILE 'data' ↔ ↓[⎕IO](⎕PROFILE 'data'.htm)[;X]
```

If column 2 is included in the result, the value `¯1` is used instead of `0` to indicate a whole-function entry.

data←{X} ⎕PROFILE 'tree'

Retrieve the collected profiling data in tree format:

`[;1]`	depth level
`[;2]`	function name
`[;3]`	function line number or `0` for a whole function entry
`[;4]`	number of times the line or function was executed
`[;5]`	accumulated time (ms) for this entry exclusive of items called by this entry
`[;6]`	accumulated time (ms) for this entry inclusive of items called by this entry
`[;7]`	number of times the timer function was called for the exclusive time
`[;8]`	number of times the timer function was called for the inclusive time

The optional left argument is treated in exactly the same way as for `X ⎕PROFILE 'data'`.

EXAMPLE

```
      ⎕PROFILE 'tree'
0  #.foo               1     1.04406 39347.64945     503 4080803
1  #.foo       1       1     0.12488     0.12488       1       1
1  #.foo       2     100     0.58851 39347.19390     200 4080500
2  #.NS1.goo           100    99.44404 39346.60538   50300 4080300
3  #.NS1.goo   1       100     0.61679     0.61679     100     100
3  #.NS1.goo   2     10000    67.80292 39314.96426   20000 4050000
4  #.NS2.moo         10000 39247.16133 39247.16133 4030000 4030000
5  #.NS2.moo   1     10000    39.28315    39.28315   10000   10000
5  #.NS2.moo   2   1000000 36430.65236 36430.65236 1000000 1000000
5  #.NS2.moo   3   1000000  1645.36214  1645.36214 1000000 1000000
3  #.NS1.goo   3     10000    19.60274    19.60274   10000   10000
1  #.foo       3       100     0.21340     0.21340     100     100
```

Note that rows with an even depth level in column `[;1]` represent function summary entries and odd depth level rows are function line entries. Recursive functions will generate separate rows for each level of recursion.

Notes

Profile Data Entry Types

The results of `⎕PROFILE 'data'` and `⎕PROFILE 'tree'` have two types of entries; function summary entries and function line entries. Function summary entries contain `0` in the line number column, whereas function line entries contain the line number. Dfns line entries begin with 0 as they do not have a header line like traditional functions. The timer data and timer call counts in function summary entries represent the aggregate of the function line entries plus any time spent that cannot be directly attributed to a function line entry. This could include time spent during function initialisation, etc.

EXAMPLE

```
#.foo          1  1.04406 39347.649450   503 4080803
#.foo    1     1  0.12488     0.124887     1       1
#.foo    2   100  0.58851 39347.193900   200 4080500
#.foo    3   100  0.21340     0.213406   100     100
```

Timer Data Persistence

The profiling data collected is stored outside the workspace and will not impact workspace availability. The data is cleared upon workspace load, clear workspace, `⎕PROFILE 'clear'`, or interpreter sign off.

The PROFILE User Command

`]PROFILE` is a utility which implements a high-level interface to `⎕PROFILE` and provides reporting and analysis tools that act upon the profiling data. For further information, see Tuning Applications using the Profile User Command.

Using `⎕PROFILE` Directly

If you choose to use `⎕PROFILE` directly, the following guidelines and information may be of use to you.

Note: Running your application with ⎕PROFILE turned on incurs a significant processing overhead and will slow your application down.

Decide which timer to use

⎕PROFILE supports profiling of either CPU or elapsed time. CPU time is generally of more interest in profiling application performance.

Simple Profiling

To get a quick handle on the top CPU time consumers in an application, use the following procedure:

- Make sure the application runs long enough to collect enough data to overcome the timer granularity – a reasonable rule of thumb is to make sure the application runs for at least `(4000×4>⎕PROFILE 'state')` milliseconds.
- Turn profiling on with `⎕PROFILE 'start' 'CPU'`
- Run your application.
- Pause the profiler with `⎕PROFILE 'stop'`
- Examine the profiling data from `⎕PROFILE 'data'` or `⎕PROFILE 'tree'` for entries that consume large amounts of resource.

This should identify any items that take more than 10% of the run time.

To find finer time consumers, or to focus on elapsed time rather than CPU time, take the following additional steps prior to running the profiler:

Turn off as much hardware as possible. This would include peripherals, network connections, etc.

- Turn off as many other tasks and processes as possible. These include anti-virus software, firewalls, internet services, background tasks.
- Raise the priority on the Dyalog APL task to higher than normal, but in general avoid giving it the highest priority.
- Run the profiler as described above.

Doing this should help identify items that take more than 1% of the run time.

Advanced Profiling

The timing data collected by ⎕PROFILE is not adjusted for the timer's call time bias; in other words, the times reported by ⎕PROFILE include the time spent calling the timer function. One effect of this can be to make "cheap" lines that are called many times seem to consume more resource. If you desire more accurate profiling measurements, or if your application takes a short amount of time to run, you will probably want to adjust for the timer call time bias. To do so, subtract from the timing data the timer's' 'call time bias multiplied by the number of times the timer was called.

EXAMPLE

```
      CallTimeBias←3⊃⎕PROFILE 'state'
      RawTimes←⎕PROFILE 'data'
      Adjusted←RawTimes[;4 5]-RawTimes[;6 7]×CallTimeBias
```

6.4.121 Print Width ⎕PW

⎕PW is the maximum number of output characters per line before folding the display.

⎕PW may be assigned any integer value in the range 42 to 32767. Note that in versions of Dyalog APL prior to 13.0 ⎕PW had a minimum value of 30; this was increased to support 128-bit decimal values. ⎕PW has Session scope.

If an attempt is made to display a line wider than ⎕PW, then the display will be folded at or before the ⎕PW width and the folded portions indented 6 spaces. The display of a simple numeric array may be folded at a width less than ⎕PW so that individual numbers are not split.

⎕PW only affects output, either direct or through ⎕ output. It does not affect the result of the function Format (⍕), of the system function ⎕FMT, or output through the system functions ⎕ARBOUT and ⎕ARBIN, or output through ⍞.

Note that if the Auto_PW parameter (*Options/Configure/Session/Auto PW*) is set to 1, ⎕PW is automatically adjusted whenever the Session window is resized. In these circumstances, a value assigned to ⎕PW will only apply until the Session window is next resized.

Examples

```
      ⎕PW←42

      ⎕←3⍴÷3
0.3333333333 0.3333333333 0.3333333333
      0.3333333333
```

6.4.122 Replace R←{X}(A ⎕R B) Y

⎕R (Replace) and ⎕S (Search) are system operators which take search pattern(s) as their left arguments and transformation rule(s) as their right arguments; the derived function operates on text data to perform either a **search**, or a search and **replace** operation.

The search patterns may include *Regular Expressions* so that complex searches may be performed. ⎕R and ⎕S utilise the open-source regular-expression search engine PCRE, which is built into Dyalog APL and distributed according to the PCRE licence.

The transformation rules are applied to the text which matches the search patterns; they may be given as a simple character vector, numeric codes, or a function.

The two system operators, ⎕R for replace and ⎕S for search, are syntactically identical. With ⎕R, the input document is examined; text which matches the search pattern is amended and the remainder is left unchanged. With ⎕S, each match in the input document results in an item in the result whose type is dependent on the transformation specified. The operators use the Variant operator to set options.

A specifies one or more search patterns, being given as a single character, a character vector, a vector of character vectors or a vector of both characters and character vectors. See search pattern following.

B is the transformation to be performed on matches within the input document; it may be either one or more transformation patterns (specified as a character, a character vector, a vector of character vectors, or a vector of both characters and character vectors), one or more transformation codes (specified as a numeric scalar or a numeric vector) or a function; see transformation pattern, transformation codes and transformation function following.

Y specifies the input document; see input document below.

X optionally specifies an output stream; see output below.

`R` is the result value; see output below.

Examples of replace operations

```
      ('.at' ⎕R '\u0') 'The cat sat on the mat'
The CAT SAT on the MAT
```

In the search pattern the dot matches any character, so the pattern as a whole matches sequences of three characters ending 'at'. The transformation is given as a character string, and causes the entire matching text to be folded to upper case.

```
      ('\w+' ⎕R {⍵.Match}) 'The cat sat on the mat'
ehT tac tas no eht tam
```

The search pattern matches each word. The transformation is given as a function, which receives a namespace containing various variables describing the match, and it returns the match in reverse, which in turn replaces the matched text.

Examples of search operations

```
      STR←'The cat sat on the mat'
      ('.at' ⎕S '\u0') STR
CAT  SAT  MAT
```

The example is identical to the first, above, except that after the transformation is applied to the matches the results are returned in a vector, not substituted into the source text.

```
      ('.at' ⎕S {⍵.((1↑Offsets),1↑Lengths)}) STR
4 3  8 3  19 3
```

When searching, the result vector need not contain only text and in this example the function returns the numeric position and length of the match given to it; the resultant vector contains these values for each of the three matches.

```
      ('.at' ⎕S 0 1) STR
4 3  8 3  19 3
```

Here the transformation is given as a vector of numeric codes which are a short-hand for the position and length of each match; the overall result is therefore identical to the previous example.

These examples all operate on a simple character vector containing text, but the text may be given in several forms - character vectors, vectors of character vectors, and external data streams. These various forms constitute a 'document'. When the result also takes the form of a document it may be directed to a stream.

Input Document

The input document may be an array or a data stream.

When it is an array it may be given in one of two forms:

1. A character scalar or vector
2. A vector of character vectors

Currently, the only supported data stream is a native file, specified as tie number, which is read from the current position to the end. If the file is read from the start, and there is a valid Byte Order Mark (BOM) at the start of it, the data encoding is determined by this BOM. Otherwise, data in the file is assumed to be encoded as specified by the InEnc option.

Hint: once a native file has been read to the end by ⎕R or ⎕S it is possible to reset the file position to the start so that it may be read again using:

```
      {} ⎕NREAD tienum 82 0 0
```

The input document is comprised of lines of text. Line breaks may be included in the data:

Implicitly

- Between each item in the outer vector (type 2, above)

Explicitly, as

- carriage return
- line feed
- carriage return and line feed together, in that order
- vertical tab (U+000B)
- newline (U+0085)
- form Feed (U+000C)
- line Separator (U+2028)
- paragraph Separator (U+2029)

The implicit line ending character may be set using the EOL option. Explicit line ending characters may also be replaced by this character - so that all line endings are normalised - using the NEOL option.

The input document may be processed in **line** mode, **document** mode or **mixed** mode. In document mode and mixed mode, the entire input document, line ending characters included, is passed to the search engine; in line mode the document is split on line endings and passed to the search engine in sections without the line ending characters. The choice of mode affects both memory usage and behaviour, as documented in the section 'Line, document and mixed modes'.

Output

The format of the output is dependent on whether S or R are in use, whether an output stream is specified and, for R, the form of the input and whether the ResultText option is specified.

An output data stream may optionally be specified. Currently, the only supported data stream is a native file, specified as tie number, and all output will be appended to it. Data in the stream is encoded as specified by the OutEnc option. If this encoding specifies a Byte Order Mark and the file is initially empty then the Byte Order Mark will be written at the start. Appending to existing data using a different encoding is permitted but unlikely to produce desirable results. If an input stream is also used, care must be taken to ensure the input and output streams are not the same.

R

With no output stream specified and unless overridden by the ResultText option, the derived function result will be a document which closely matches the format of the input document, as follows:

A **character scalar or vector** input will result in a **character vector** output. Any and all line endings in the output will be represented by line ending characters within the character vector.

A **vector of character vectors** as input will result in a **vector of character vectors** as document output. Any and all line endings in the output document will be implied at the end of each character vector.

A **stream** as input will result in a **vector of character vectors** document output. Any and all line endings in the output document will be implied at the end of each character vector.

Note that the shape of the output document may be significantly different to that of the input document.

If the ResultText option is specified, the output type may be forced to be a **character vector** or **vector of character vectors** as described above, regardless of the input document.

With an output stream specified the text is appended to the stream. If the appended text does not end with a line ending character then the line ending character specified by the EOL option is also appended. The resulting length of the file is returned as a shy result.

□S

With no output stream specified, the result will be a vector containing one item for each match in the input document, of types determined by the transformation performed on each match.

With an output stream specified each match is appended to the stream. If any match does not end with a line ending character then the line ending character specified by the EOL option is also appended. The resulting length of the file is returned as a shy result. Only text may be written to the stream, which means:

- When a transformation function is used, the function may only generate a character vector result.
- Transformation codes may not be used.

Search pattern

For more information on the syntax of the search pattern, see the *Section 8.1*.

There may be multiple search patterns. If more than one search pattern is specified and more than one pattern matches the same part of the input document then priority is given to the pattern specified first.

Note that when anchoring a search to the beginning of a line, it is essential to use ^ (□UCS 94), not ∧ (□UCS 8743).

Transformation pattern

For each match in the input document, the transformation pattern causes the creation of text which, for R, replaces the matching text and, for S, generates one item in the result.

There may be either one transformation pattern, or the same number of transformation patterns as search patterns. If there are multiple search patterns and multiple transformation patterns then the transformation pattern used corresponds to the search pattern which matched the input text.

Transformation patterns may not be mixed with transformation codes or functions.

The following characters have special meaning:

Character	Meaning
%	acts as a placeholder for the entire line (line mode) or document (document mode or mixed mode) which contained the match
&	acts as a placeholder for the entire portion of text which matched
\n	represents a line feed character
\r	represents a carriage return
\0	equivalent to &
\ *n*	acts as a placeholder for the text which matched the first to ninth subpattern; *n* may be any single digit value from 1 to 9
(*n*)	acts as a placeholder for the text which matched the numbered subpattern; *n* may have an integer value from 0 to 63.
\< *name* >	acts as a placeholder for the text which matched the named subpattern
\	represents the backslash character
\%	represents the percent character
\&	represents the ampersand character
\x{ *nnnn* }	represents a Unicode code point; *nnnn* is a hexadecimal sequence of characters yielding a value between 0x1 and 0x10FFFF.

The above may be qualified so that matching text is folded, or mapped to upper- or lower-case, by using the **f**, **u**, and **l** modifiers respectively; the effect is as if the text was processed by □c. See *Section 6.4.14*.

Character sequences beginning with the backslash place the modifier after the backslash; character sequences with no leading backslash add both a backslash and the modifier to the start of the sequence, for example:

\u&	acts as a placeholder for the entire portion of text which matched, folded to upper case
\l0	equivalent to \l&

Character sequences beginning with the backslash other that those shown are invalid. All characters other than those shown are literal values and are included in the text without modification.

Transformation codes

The transformation codes are a numeric scalar or vector. Transformation codes may only be used with □s. For each match in the input document, a numeric scalar or vector of the same shape as the transformation codes is created, with the codes replaced with values as follows:

0	The offset from the start of the line (line mode) or document (document mode or mixed mode) of the start of the match.
1	The length of the match.
2	In line mode, the block number in the source document of the start of the match. The value is origin zero. In document mode or mixed mode this value is always zero.
3	The pattern number which matched the input document, origin zero.

Transformation Function

The transformation function is called for each match within the input document. The function is monadic and is passed a namespace, containing the following variables:

`Block`	The entire line (line mode) or document (document mode or mixed mode) in which the match was found.
`BlockNum`	With line mode, the block (line) number in the source document of the start of the match. The value is origin zero. With document mode or mixed mode the entire document is contained within one block and this value is always zero.
`Pattern`	The search pattern which matched.
`PatternNum`	The index-zero pattern number which matched.
`Match`	The text within Block which matched Pattern.
`Offsets`	A vector of one or more offsets relative to the start of Block. The first value is the offset of the entire match; any and all additional values are the offsets of the portions of the text which matched the subpatterns, in the order of the subpatterns within Pattern. `¯1` indicates no match, see below.
`Lengths`	A vector of one or more lengths, corresponding to each value in Offset. `¯1` indicates no match, see below.
`Names`	A vector of one or more character vectors corresponding to each of the values in Offsets, specifying the names given to the subpatterns within Pattern. The first entry (corresponding to the match) and all subpatterns with no name are included as length zero character vectors.
`ReplaceMode`	A Boolean indicating whether the function was called by `⍠'R'` (value 1) or `⍠'S'` (value 0).
`TextOnly`	A Boolean indicating whether the return value from the function must be a character vector (value 1) or any value (value 0).

The return value from the function is used as follows:

With `⍠'R'` the function must return a character vector. The contents of this vector are used to replace the matching text.

With `⍠'S'` the function may return no value. If it does return a value:

- When output is being directed to a stream it must be a character vector.
- Otherwise, it may be any value. The overall result of the derived function is the catenation of the enclosure of each returned value into a single vector.

The passed namespace exists over the lifetime of ⎕R or ⎕S; the function may therefore preserve state by creating variables in the namespace.

The function may itself call ⎕R or ⎕S.

There may be only one transformation function, regardless of the number of search patterns.

The locations of the match within Block and subpatterns within Match are given as offsets rather than positions, that is, the values are the number of characters preceding the data, and are not affected by the Index Origin.

The value of ¯1 may appear in both the Offsets and Fields items (in corresponding positions). They indicate that the subpattern to which they refer did not appear in the match at all.

Example

```
      {}('(A)|(B)'⎕R{⍬←⍵.(Offsets Lengths)⋄'x'})'ABC'
0 0  1 1
1 ¯1 1  1 ¯1 1
```

The pattern has two subpatterns - (A) and (B). Therefore Offsets and Lengths would be expected to have three elements each - one for the entire match, one for the first subpattern and one for the second subpattern. But these subpatterns have a | between them which means they are alternates - only one can match.

When ABC is searched the first match is the A in the first subpattern. The second subpattern does not feature. Offsets is 0 0 and Lengths is 1 1: the entire pattern matched from offset 0 length 1 and the first subpattern also matched from offset 0 length 1. The second subpattern did not feature in the match.

Note that ¯1 is only used as a "filler" when there are higher-numbered subpatterns that did match.

The second match is the B in the second subpattern. Offsets is 1 ¯1 1 and Lengths is 1 ¯1 1: the entire pattern matched from offset 1 length 1 and the second subpattern also matched from offset 1 length 1. The first subpattern did not feature in the match and this is indicated by the ¯1s. There has to be something between the offset/length for the entire pattern and the second subpattern.

Variant Options

Options are specified using the Variant operator. The Principal option is IC.

Default values are highlighted thus.

IC Option

When set, case is ignored in searches.

1	Matches are not case sensitive.
0	Matches are case sensitive.

EXAMPLE

```
      ('[AEIOU]' ⎕R 'X' ⍠ 'IC' 1) 'ABCDE abcde'
XBCDX XbcdX
      ('[AEIOU]' ⎕R 'X' ⍠ 1)'ABCDE abcde'
XBCDX XbcdX
```

Mode Option

Specifies whether the input document is interpreted in **line** mode, **document** mode or **mixed** mode.

L	When line mode is set, the input document is split into separate lines (discarding the line ending characters themselves), and each line is processed separately. This means that the ML option applies per line, and the '^' and '$' anchors match the start and end respectively of each line. Because the document is split, searches can never match across multiple lines, nor can searches for line ending characters ever succeed. Setting line mode can result in significantly reduced memory requirements compared with the other modes.
D	When document mode is set, the entire input document is processed as a single block. The ML option applies to this entire block, and the '^' and '$' anchors match the start and end respectively of the block - not the lines within it. Searches can match across lines, and can match line ending characters.
M	When mixed mode is set, the '^' and '$' anchors match the start and end respectively of each line, as if line mode is set, but in all other respects behaviour is as if document mode is set - the entire input document is processed in a single block.

EXAMPLES

```
      ('$' ⎕R '[Endline]' ⍠ 'Mode' 'L') 'ABC' 'DEF'
ABC[Endline]  DEF[Endline]

      ('$' ⎕R '[Endline]' ⍠ 'Mode' 'D') 'ABC' 'DEF'
ABC  DEF[Endline]

      ('$' ⎕R '[Endline]' ⍠ 'Mode' 'M') 'ABC' 'DEF'
ABC[Endline]  DEF[Endline]
```

DotAll Option

Specifies whether the dot ('.') character in search patterns matches line ending characters

0	The '.' character in search patterns matches most characters, but not line endings.
1	The '.' character in search patterns matches all characters.

This option is invalid in line mode, because line endings are stripped from the input document.

EXAMPLE

```
      ('.' ⎕R 'X' ⍠'Mode' 'D') 'ABC' 'DEF'
XXX  XXX
      ('.' ⎕R 'X' ⍠('Mode' 'D')('DotAll' 1)) 'ABC' 'DEF'
XXXXXXXX
```

EOL Option

Sets the line ending character which is implicitly present between character vectors, when the input document is a vector of character vectors.

CR	Carriage Return (U+000D)
LF	Line Feed (U+000A)
CRLF	Carriage Return followed by Line Feed
VT	Vertical Tab (U+000B)
NEL	New Line (U+0085)
FF	Form Feed (U+000C)
LS	Line Separator (U+2028)
PS	Paragraph Separator (U+2029)

In the Classic Edition, setting a value which is not in ⎕AVU may result in a `TRANSLATION ERROR`.

EXAMPLE

```
      ('\n' ⎕R'X' ⍠('Mode' 'D')('EOL' 'LF')) 'ABC' 'DEF'
ABCXDEF
```

Here, the implied line ending between 'ABC' and 'DEF' is '\n', not the default '\r\n'.

NEOL Option

Specifies whether explicit line ending sequences in the input document are normalised by replacing them with the character specified using the EOL option.

0	Line endings are not normalised.
1	Line endings are normalised.

EXAMPLE

```
      a←'ABC',(1↑2↓⎕AV),'DEF',(1↑3↓⎕AV),'GHI'
      ('\n'⎕S 0 ⍠ 'Mode' 'D' ⍠ 'NEOL' 1 ⍠ 'EOL' 'LF') a
3 7
```

'\n' has matched both explicit line ending characters in the input, even though they are different.

ML Option

Sets a limit to the number of processed pattern matches per line (line mode) or document (document mode and mixed mode).

Positive value n	Sets the limit to the first n matches.
0	Sets no limit.
Negative value ¯ n	Sets the limit to exactly the nth match.

EXAMPLES

```
      ('.' ⎕R 'x' ⍠ 'ML' 2) 'ABC' 'DEF'
xxC   xxF
      ('.' ⎕R 'x' ⍠ 'ML' ¯2) 'ABC' 'DEF'
AxC   DxF
      ('.' ⎕R 'x' ⍠ 'ML' ¯4 ⍠ 'Mode' 'D') 'ABC' 'DEF'
ABC   xEF
```

Greedy Option

Controls whether patterns are "greedy" (and match the maximum input possible) or are not (and match the minimum). Within the pattern itself it is possible to specify greediness for individual elements of the pattern; this option sets the default.

1	Greedy by default.
0	Not greedy by default.

EXAMPLES

```
      ('[A-Z].*[0-9]' ⎕R 'X' ⍠ 'Greedy' 1)'ABC123 DEF456'
X
      ('[A-Z].*[0-9]' ⎕R 'X' ⍠ 'Greedy' 0)'ABC123 DEF456'
X23 X56
```

OM Option

Specifies whether matches may overlap.

1	Searching continues for all patterns and then from the character following the *start* of the match, thus permitting overlapping matches.
0	Searching continues from the character following the *end* of the match.

This option may only be used with ⎕S. With ⎕R searching always continues from the character following the end of the match (the characters following the start of the match will have been changed).

EXAMPLES

```
      ('[0-9]+' ⎕S '\0' ⍠ 'OM' 0) 'A 1234 5678 B'
 1234  5678
      ('[0-9]+' ⎕S '\0' ⍠ 'OM' 1) 'A 1234 5678 B'
1234  234  34  4  5678  678  78  8
```

InEnc Option

This option specifies the encoding of the input stream when it cannot be determined automatically. It is either:

- a character vector that specifies the file-encoding as shown in the table below.
- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not

mapped to any character. Apart from ¯1, no value may appear in the table more than once.

When the stream is read from its start, and the start of the stream contains a recognised Byte Order Mark (BOM), the encoding is taken as that specified by the BOM and this option is ignored. Otherwise, the encoding is assumed to be as specified by this option.

UTF-8	The stream is processed as UTF-8 data. Note that ASCII is a subset of UTF-8, so this default is also suitable for ASCII data.
UTF-16	The stream is processed as UTF16 little-endian data on little-ended systems, or as UTF16 big-endian data on big-endian systems.
UTF-16LE	The stream is processed as UTF16 little-endian data.
UTF-16BE	The stream is processed as UTF16 big-endian data.
UTF-32	The stream is processed as UTF32 little-endian data on little-ended systems, or as UTF32 big-endian data on big-endian systems.
UTF-32LE	The stream is processed as UTF32 little-endian data.
UTF-32BE	The stream is processed as UTF32 big-endian data.
ASCII	The stream is processed as ASCII data. If the stream contains any characters outside of the ASCII range then an error is produced.
Windows-1252	The stream is processed as Windows-1252 (ANSI) data.
ANSI	Same as Windows-1252

For compatibility with the OutEnc option, the above UTF formats may be qualified with -BOM (for example, UTF-8-BOM) or -NOBOM. For input streams, the qualified and unqualified options are equivalent.

OutEnc Option

When the output is written to a stream, this option specifies how the data is to be encoded. It is either:

- a character vector that specifies the file-encoding as shown in the table below.

- a 256-element numeric vector that maps each possible byte value (0-255) to a Unicode code point (1st element = Unicode code point corresponding to byte value 0, and so on). ¯1 indicates that the corresponding byte value is not mapped to any character. Apart from ¯1, no value may appear in the table more than once.

Implied	If input came from a stream then the encoding format is the same as the input stream, otherwise UTF-8
UTF-8	The data is written in UTF-8 format.
UTF-16	The data is written in UTF16 little-endian format on little-ended systems, or in UTF16 big-endian format on big-endian systems.
UTF-16LE	The data is written in UTF-16 little-endian format.
UTF-16BE	The data is written in UTF-16 big-endian format.
UTF-32	The data is written in UTF32 little-endian format on little-ended systems, or in UTF32 big-endian format on big-endian systems.
UTF-32LE	The data is written in UTF-32 little-endian format.
UTF-32BE	The data is written in UTF-32 big-endian format.
ASCII	The data is written in ASCII format.
Windows-1252	The data is written in Windows-1252 (ANSI) format.
ANSI	Same as Windows-1252

The above UTF formats may be qualified with -BOM (for example, UTF-8-BOM) to specify that a Byte Order Mark should be written at the start of the stream or, -NOBOM that it should not. For files, this is ignored if the file already contains any data. If the -BOM or -NOBOM suffix is omitted, UTF-8 defaults to UTF-8-NOBOM, while the other UTF formats default to -BOM.

Enc Option

This option sets both InEnc and OutEnc simultaneously, with the same given value. Any option value accepted by those options except Implied may be given.

ResultText Option

For ⎕R, this option determines the format of the result.

Implied	The output will either be a **character vector** or **a vector of character vectors** , dependent on the input document type.
Simple	The output will be a **character vector** . Any and all line endings in the output will be represented by line ending characters within the character vector.
Nested	The output will be a **vector of character vectors** . Any and all line endings in the output document will be implied at the end of each character vector.

This option may only be used with ⎕R.

Examples

```
      ⎕UCS ¨ ('A' ⎕R 'x') 'AB' 'CD'
 120 66  67 68
      ⎕UCS ('A' ⎕R 'x' ⍠ 'ResultText' 'Simple') 'AB' 'CD'
120 66 13 10 67 68
```

UCP Option

This affects the way PCRE that processes \B, \b, \D, \d, \S, \s, \W, \w, and some of the POSIX character classes.

1	Unicode properties are used to classify characters.
0	Only ASCII characters are recognized.

Implementation Note: this option is implemented by setting or not setting the PCRE_UCP option when calling pcre_compile(). More information can be found in the PCRE documentation.

Examples

By default, the character ø (which is not an ASCII character) is considered to be a "non-word" character, so:

```
      ('\w'⍓S'\0')'Bjørn'  ⍝ identify "word" characters
B  j  r  n
      ('\W'⍓S'\0')'Bjørn'  ⍝ non-word" characters
ø
```

When UCP is set to 1, Unicode characters are matched as "word" characters (\w) too.

```
      ('\w'⍓S'\0' ⍠'UCP' 1)'Bjørn'
B  j  ø  r  n
```

Regex Option

This option may be used to disable regular expression matching which is enabled by default. It is a singleton Boolean value that applies to both search and transformation patterns, or a 2-element vector of Boolean values that applies to them separately.

1	Regular expression matching is applied.
0	regular expression matching is disabled.

EXAMPLES

```
      STR
The cat sat on the mat

      ('.at' ⍓S '\u0') STR
 CAT  SAT  MAT
      ('.at' ⍓S '\u0' ⍠('Regex' 0)) STR

      ('.at'⍓R'\u0')STR
The CAT SAT on the MAT
      ('.at'⍓R'\u0'⍠('Regex' (1 0)))STR
The \u0 \u0 on the \u0
```

Line, document and mixed modes

The Mode setting determines how the input document is packaged as a block and passed to the search engine. In line mode each line is processed separately; in document mode and mixed mode the entire document is presented to the search engine. This affects both the semantics of the search expression, and memory usage.

Semantic differences

- The ML option applies per block of data.
- In line mode, search patterns cannot be constructed to span multiple lines. Specifically, patterns that include line ending characters (such as '\r') will never match because the line endings are never presented to the search engine.
- By default the search pattern metacharacters '^' and '\$' match the start and end of the block of data. In line mode this is always the start and end of each line. In document mode this is the start and end of the document. In mixed mode the behaviour of '^' and '\$' are amended by setting the PCRE option 'MULTILINE' so that they match the start and end of each line within the document.

Memory usage differences

Blocks of data passed to the search engine are processed and stored in the workspace. Processing the input document in line mode limits the total memory requirements; in particular this means that large streams can be processed without holding all the data in the workspace at the same time.

Technical Considerations

□R and □S utilise the open-source regular-expression search engine PCRE, which is built into the Dyalog software and distributed according to the PCRE licence which is published separately.

Before data is passed to PCRE it is converted to UTF-8 format. This converted data is buffered in the workspace; processing large documents may have significant memory requirements. In line mode, the data is broken into individual lines and each is processed separately, potentially reducing memory demands.

It is possible to save a workspace with an active □R or □S on the stack and execution can continue when the workspace is reloaded with the same interpreter version. Later versions of the interpreter may not remain compatible and may signal a `DOMAIN ERROR` with explanatory message in the status window if it is unable to continue execution.

PCRE has a buffer length limit of 2^{31} bytes (2GB). UTF-8 encodes each character using between 1 and 6 bytes (typically 1 or 3). In the very worst case, where every character is encoded in 6 bytes, the maximum block length which can be searched would be 357,913,940 characters.

Further Examples

Several of the examples use the following vector as the input document:

```
      text
To be or not to be- that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles
```

Replace all upper and lower-case vowels by `'X'`:

```
      ('[aeiou]' ⎕R 'X' ⍠ 'IC' 1) text
TX bX Xr nXt tX bX- thXt Xs thX qXXstXXn:
WhXthXr 'tXs nXblXr Xn thX mXnd tX sXffXr
ThX slXngs Xnd XrrXws Xf XXtrXgXXXs fXrtXnX,
Xr tX tXkX Xrms XgXXnst X sXX Xf trXXblXs
```

Replace only the second vowel on each line by `'\VOWEL\'`:

```
      ('[aeiou]' ⎕R '\\VOWEL\\'⍠('IC' 1)('ML' ¯2)) text
To b\VOWEL\ or not to be- that is the question:
Wheth\VOWEL\r 'tis nobler in the mind to suffer
The sl\VOWEL\ngs and arrows of outrageous fortune,
Or t\VOWEL\ take arms against a sea of troubles
```

Case fold each word:

```
      ('(?<first>\w)(?<remainder>\w*)' ⎕R '\u<first>\l<remainder>') text
To Be Or Not To Be- That Is The Question:
Whether 'Tis Nobler In The Mind To Suffer
The Slings And Arrows Of Outrageous Fortune,
Or To Take Arms Against A Sea Of Troubles
```

Extract only the lines with characters 'or' (in upper or lower case) on them:

```
      ↑('or' ⎕S '%' ⍠ ('IC' 1)('ML' 1)) text
To be or not to be- that is the question:
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles
```

Identify which lines contain the word 'or' (in upper or lower case) on them:

```
      ('\bor\b'⎕S 2⍠('IC' 1)('ML' 1))text
0 3
```

Note the difference between the characters *'or'* (which appear in 'fortune') and the word *'or'*.

Place every non-space sequence of characters in brackets:

```
      ('[^\s]+' ⎕R '(&)' ) 'To be or not to be, that is the question'
(To) (be) (or) (not) (to) (be,) (that) (is) (the) (question)
```

Replace all sequences of one or more spaces by newline:

Note that the effect of this is dependent on the input format

Character vector input results in a single character vector output with embedded newlines:

```
      ]display ('\s+' ⎕R '\r') 'To be or not to be, that is the
question'
┌→────────┐
│To       │
│be       │
│or       │
│not      │
│to       │
│be,      │
│that     │
│is       │
│the      │
│question │
└─────────┘
```

A vector of two character vectors as input results in a vector of 10 character vectors output:

```
      ]display ('\s+' ⎕R '\r') 'To be or not to be,' 'that is the
question'
┌→──────────────────────────────────────────────────────────────────┐
│ ┌→┐ ┌→┐ ┌→┐ ┌→──┐ ┌→┐ ┌→──┐ ┌→────┐ ┌→┐ ┌→──┐ ┌→────────┐ │
│ │To│ │be│ │or│ │not│ │to│ │be,│ │that│ │is│ │the│ │question│ │
│ └──┘ └──┘ └──┘ └───┘ └──┘ └───┘ └────┘ └──┘ └───┘ └────────┘ │
└∊──────────────────────────────────────────────────────────────────┘
```

Change numerals to their expanded names, using a function:

```
     ∇r←f a;n
[1]   n←'zero' 'one' 'two' 'three' 'four'
[2]   n,←'five' 'six' 'seven' 'eight' 'nine'
[3]   r←' ',⊃(⍎a.Match)⍠n
     ∇
      verbose←('[0-9]' ⎕R f)
      verbose ⍕27×56×87
 one three one five four four
```

Swap 'red' and 'blue':

```
      ('red' 'blue' ⎕R 'blue' 'red') 'red hat blue coat'
blue hat red coat
```

Convert a comma separated values (CSV) file so that:

- dates in the first field are converted from European format to ISO, and
- currency values are converted from Deutsche Marks (DEM) to Euros (DEM 1.95583 to €1).

The currency conversion requires the use of a function. Note the nested use of ⎕R.

Input file:

```
01/03/1980,Widgets,DEM 10.20 02/04/1980,Bolts,DEM 61.75
17/06/1980,Nuts; special rate DEM 17.00,DEM 17.00
18/07/1980,Hammer,DEM 1.25
```

Output file:

```
1980-03-01,Widgets,€ 5.21 1980-04-02,Bolts,€ 31.57 1980-06-17,Nuts;
special rate DEM 17.00,€ 8.69 1980-07-18,Hammer,€ 0.63
```

```
     ∇ ret←f a;d;m;y;v
[1]    ⎕IO←0
[2]    :Select a.PatternNum
[3]    :Case 0
[4]        d m y←{a.Match[a.Offsets[⍵+1]+⍳a.Lengths[⍵+1]]}¨⍳3
[5]        ret←y,'-',m,'-',d,','
[6]    :Else
[7]        v←⍎a.Block[a.Offsets[1]+⍳a.Lengths[1]]
[8]        v÷←1.95583
[9]        ret←',€ ',('(\d+\.\d\d).*'⎕R'\1')⍕v
[10]   :EndSelect
     ∇
```

```
      in ← 'x.csv' ⎕NTIE 0
      out ← 'new.csv' ⎕NCREATE 0
      dateptn←'(\d{2})/(\d{2})/(\d{4}),'
      valptn←',DEM ([0-9.]+)'
      out (dateptn valptn ⎕R f) in
      ⎕NUNTIE¨in out
```

Create a simple profanity filter:

For the list of objectionable words

```
       profanity←'bleeding' 'heck'
```

first construct a pattern which will match the words:

```
      ptn←(('^' '$' '\r\n') ⎕R '\\b(' ')\\b' '|'
                           ⎕OPT 'Mode' 'D') profanity
      ptn
\b(bleeding|heck)\b
```

then a function that uses this pattern:

```
      sanitise←ptn ⎕R '****' ⎕OPT 1
      sanitise '"Heck", I said'
"****", I said
```

Replace the characters 'or' with '\u0' without having to escape the backslash:

Escaping transformation strings can be a daunting task. To avoid doing so, one can simply enclose the string in braces. This is not a special feature, but just a consequence of how transformation functions are used.

```
      ('to' ⎕R {'\u0'})text
To be or not \u0 be- that is the question:
Whether 'tis nobler in the mind \u0 suffer
The slings and arrows of outrageous fortune,
Or \u0 take arms against a sea of troubles
```

6.4.123 Cross References R←⎕REFS Y

Y must be a simple character scalar or vector, identifying the name of a function or operator, or the object representation form of a function or operator (see *Section 6.4.116*). R is a simple character matrix, with one name per row, of identified names in the function or operator in Y excluding distinguished names of system constants, variables or functions.

EXAMPLE

```
      ⎕VR'OPTIONS'
     ∇ OPTIONS;OPTS;INP
[1]   ⍝ REQUESTS AND EXECUTES AN OPTION
[2]    OPTS ←'INPUT' 'REPORT' 'END'
[3]   IN:INP←ASK'OPTION:'
[4]    →EX⍴⍨(⊂INP)∈OPTS
[5]    'INVALID OPTION. SELECT FROM',OPTS ⋄ →IN
[6]   EX:→EX+OPTS⍳⊂INP
[7]    INPUT ⋄ →IN
[8]    REPORT ⋄ →IN
[9]   END:
     ∇

    ⎕REFS'OPTIONS'
ASK
END
EX
IN
INP
INPUT
OPTIONS
OPTS
REPORT
```

If Y is locked or is an External Function, R contains its name only. For example:

```
      ⎕LOCK 'OPTIONS' ⋄ ⎕REFS 'OPTIONS'
OPTIONS
```

If `Y` is the name of a primitive, external or derived function, `R` is an empty matrix with shape 0 0.

6.4.124 Random Link ⎕RL

`⎕RL` is a 2-element vector. Its first element contains the base or *random number seed* and its second element is an integer that identifies the random number generator that is currently in use. Together these items define how the system generates random numbers using *Section 3.4.79* and *Section 3.4.7*.

In a `clear ws` `⎕RL` is `(⍬ 1)`. `⎕RL` has Namespace scope.

Random Number Seed

The facility to set the seed to a specific value provides the means to generate a repeatable sequence of random numbers, such as might be required by certain types of simulation modelling. This capability is not provided by RNG2.

If the seed is set to 0, the seed is set randomly but may be retrieved and subsequently re-assigned to create a repeatable sequence.

If the seed is set to `⍬`, Dyalog is able to take advantage of certain optimisations which deliver maximum performance. In this case, the actual seed in use is intentionally hidden and `⎕RL[1]` always reports `⍬`, regardless of the Random Number Generator in use.

Random Number Generators

The 3 random number generators are listed in the table below. The 4th column of the table contains the values of seeds that may be assigned to them.

Id	Name	Algorithm	Valid Seed Values
0	RNG0	Lehmer linear congruential generator.	`0` , `⍬` , or an integer in the range 1 to `¯2+2*31`
1	RNG1	Mersenne Twister.	`0` , `⍬` , an integer in the range 1 to `¯1+2*63` or a 625-element integer vector
2	RNG2	Operating System random number generator.	`⍬`

Note that assigning an invalid value to the seed will cause `DOMAIN ERROR`.

The default random number generator in a `CLEAR WS` is 1 (Mersenne Twister). This algorithm *RNG1* produces 64-bit values with good distribution.

The Lehmer linear congruential generator *RNG0* was the only random number generator provided in versions of Dyalog APL prior to Version 13.1. The implementation of this algorithm has several limitations including limited value range `(2*31)`, short period and non-uniform distribution (some values may appear more frequently than others). It is retained for backwards compatibility.

Under Windows, the Operating System random number generator algorithm *RNG2* uses the `rand_s()` function. Under UNIX/Linux it uses `/dev/urandom`.

Random Number Sequences

Random number sequences may be predictable or not and repeatable or not. A predictable and repeatable sequence is obtained by starting with the same specific value for the seed. A non-predictable sequence is obtained by starting with a seed which is itself chosen at random, but such a sequence is repeatable if the value of the seed (chosen at random) is visible. A non-predictable and non-repeatable sequence of random numbers is obtained where the initial seed is chosen completely at random and is unknown.

Using *RNG0* or *RNG1*:

- To obtain an entirely predictable random sequence, set the seed to a non-zero value
- To obtain a non-predictable, but repeatable sequence, set the seed to `0`
- To obtain a non-predictable, non-repeatable series of random numbers, set the seed to `⍬`

RNG2 does not support a user modifiable random number seed, so when using this scheme, it is not possible to obtain a repeatable random number series and the seed must always be `⍬`.

Implementation Note

`⎕RL` does not behave quite like a regular 2-element variable; it has its own rules relating to assignment and reference.

Reference

⎕RL *returns* a 2-element vector whose second element identifies the scheme in use (0, 1 or 2).

If ⎕RL[1] is set to ⍬, ⎕RL[1]always reports ⍬.

Otherwise if the seed ⎕RL[1] is set to a value other than ⍬:

- using *RNG0*, ⎕RL[1] is an integer which represents the *seed* for the next random number in the sequence.
- using *RNG1*, the system internally retains a block of 312 64-bit numbers which are used one by one to generate the results of roll and deal. When the first block of 312 have been used up, the system generates a second block. In this case, ⎕RL[1] is an integer vector of 32-bit numbers of length 625 (the first is an index into the block of 312) which represents the internal state of the random number generator. This means that, as with *RNG0*, you may save the value of ⎕RL in a variable and reassign it later.
- Using *RNG2*, the seed is purely internal and ⎕RL[1] is always zilde.

Assignment

⎕RL may only be assigned in its entirety. Indexed and selective assignment may not be used to assign values to individual elements.

To preserve compatibility with Versions of Dyalog prior to Version 15.0 (in which ⎕RL specifies just the seed) if the value assigned to ⎕RL represents a valid seed for the random number generator in use, it is taken to be the new seed. Otherwise, the value assigned to ⎕RL must be a 2-element vector, whose first item is the seed and whose second item is 0, 1 or 2 and specifies the random number generator to be used subsequently.

Examples (specific seeds for repeatable sequences)

```
      )CLEAR
clear ws
      ⎕RL←16807
      10?10
4 1 6 5 2 9 7 10 3 8
      5↑⊃⎕RL
10 0 16807 1819658750 ¯355441828
      X←?1000⍴1000
      5↑⊃⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303
```

```
      ⎕RL←16807
      10?10
4 1 6 5 2 9 7 10 3 8
      Y←?1000⍴1000
      X≡Y
1
      5↑⊃⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303
```

```
      ⎕RL←16807 0 ⍝ Select RNG0
      ⎕RL
16807 0
      ?9 9 9
2 7 5
      ?9
7
      ⎕RL
984943658 0

      ⎕RL←16807
      ?9 9 9
2 7 5
      ?9
7
      ⎕RL
984943658 0
```

```
      ⎕RL←16807 1 ⍝ Select RNG1
      5↑⊃⎕RL
100 ¯465541037 ¯1790786136 ¯205462449 996695303
```

Examples (0 seed)

When you set the seed to 0, a random seed is created for you:

```
      ⎕RL←0 0
      ⎕RL
865618822 0
      ⎕RL←0
      ⎕RL
1100783275 0
```

Setting the seed to 0 gives you a new, unpredictable random sequence yet it is repeatable because you can retrieve (and subsequently re-use) the actual seed after you set it:

```
      ?10⍴100
14 22 18 30 42 22 71 32 32 12
      ⎕RL←1100783275
      ?10⍴100
14 22 18 30 42 22 71 32 32 12
```

Example (zilde)

When you set the seed to zilde, you get the same random initialisation as setting it to 0 but you can't retrieve the actual value of the seed. When it is set to ⍬ it is subsequently reported as ⍬ and the internal value of the seed is hidden.

```
      ⎕RL←⍬
      ⎕RL
┌┬┐
│⍬│
└┴┘
```

6.4.125 Space Indicator R←⎕RSI

`R` is a vector of refs to the spaces from which functions in the state indicator were called `(⍴⎕RSI←→⍴⎕NSI←→⍴⎕SI)`.

`⎕RSI` and `⎕NSI` are identical except that `⎕RSI` returns refs to the spaces whereas `⎕NSI` returns their names. Put another way: `⎕NSI←→⍎¨⎕RSI`.

Note that ⎕RSI returns refs to the spaces *from which* functions were called not those *in which* they are currently running.

EXAMPLE

```
      )OBJECTS
xx      yy

      ⎕VR 'yy.foo'
     ∇ r←foo
[1]    r←⎕SE.goo
     ∇
      ⎕VR'⎕SE.goo'
     ∇ r←goo
[1]    r←⎕SI,[1.5]⎕RSI
     ∇

      )CS xx
#.xx
      calling←#.yy.foo
      ]display calling
┌→──────────┐
↓ ┌→──┐     │
│ │goo│ #.yy │
│ └───┘     │
│ ┌→──┐     │
│ │foo│ #.xx │
│ └───┘     │
└∊──────────┘
```

6.4.126 Response Time Limit ⎕RTL

A non-zero value in ⎕RTL places a time limit, in seconds, for input requested via ⍞, ⎕ARBIN, and ⎕SR. ⎕RTL may be assigned any integer in the range 0 to 32767. The value in a clear workspace is 0. ⎕RTL has Namespace scope.

EXAMPLE

```
      ⎕RTL←5 ⋄ ⍞←'FUEL QUANTITY?' ⋄ R←⍞
FUEL QUANTITY?
TIMEOUT
      ⎕RTL←5 ⋄ ⍞←'FUEL QUANTITY?' ⋄ R←⍞
```

6.4.127 Search R←{X}(A ⍧S B) Y

See *Section 6.4.122*.

6.4.128 Save Workspace {R}←{X}⍧SAVE Y

Y must be a simple character scalar or vector that identifies a full or relative path name to the file in which the workspace will be written.

Unless the path specified by Y is a full pathname, it is taken to be relative to the current working directory which may be obtained by the expression: ⊃1 ⍧NPARTS ''.

See *Programming: Workspaces* for the rules for specifying a workspace name.

The active workspace is saved with the file name specified by Y, whether or not a workspace file of that name already exists.

A DOMAIN ERROR is reported if the name in Y :

- is not a valid workspace name
- is not a valid file name
- refers to an unauthorised directory
- specifies an existing file that does not already contain a Dyalog workspace or session file

The shy result R is a simple Boolean scalar 1. However, when the workspace is subsequently loaded using ⍧LOAD and execution restarts, the result is 0, as described below.

The optional left argument X is either 0 or 1. If X is omitted or 1, the saved version of the workspace has execution suspended at the point of exit from the ⍧SAVE function. If the saved workspace is subsequently loaded by ⍧LOAD, execution is resumed, and the value 0 is returned if the result is used or assigned, or otherwise the result is suppressed. In this case, the latent expression value (⍧LX) is ignored.

If X is 0, the workspace is saved without any state indicator in effect. The effect is the same as if you first executed)RESET and then)SAVE. In this case, when the workspace is subsequently loaded, the value of the latent expression (⍧LX) is honoured if applicable.

As is the case for)SAVE (see *Section 7.2.24*), monadic ⍧SAVE will fail and issue DOMAIN ERROR if any threads (other than the root thread 0) are running or if there are any Edit

or Trace windows open. However, neither of these restrictions apply if the left argument X is 0.

Note that the values of all system variables (including ⎕SM) and all GUI objects are saved.

EXAMPLE

```
      (⊃'SAVED' 'ACTIVE' [⎕IO+⎕SAVE'TEMP']),' WS'
ACTIVE WS
      ⎕LOAD 'TEMP'
SAVED WS
```

Additional operations may be performed before saving the workspace. For further information, see *Section 4.4.9.43*.

6.4.129 Screen Dimensions R←⎕SD

⎕SD is a 2-element integer vector containing the number of rows and columns on the screen, or in the USER window.

For asynchronous terminals under UNIX, the screen size is taken from the terminal database *terminfo* or *termcap*.

In window implementations of Dyalog APL, ⎕SD reports the current size (in characters) of the USER window or the current size of the SM object, whichever is appropriate.

6.4.130 Session Namespace ⎕SE

⎕SE is a system namespace. Its GUI components (MenuBar, ToolBar, and so forth) define the appearance and behaviour of the APL Session window and may be customised to suit individual requirements.

⎕SE is maintained separately from the active workspace and is not affected by)LOAD or)CLEAR. It is therefore useful for containing utility functions. The contents of ⎕SE may be saved in and loaded from a .DSE file.

See *User Interface: Session Object* for further details.

6.4.131 Execute (UNIX) Command {R}←⎕SH Y

⎕SH executes a UNIX shell or a Windows Command Processor. ⎕SH is a synonym of ⎕CMD. Either function may be used in either environment (UNIX or Windows) with exactly the same effect. ⎕SH is probably more natural for the UNIX user. This section describes the behaviour of ⎕SH and ⎕CMD under UNIX. See *Section 6.4.17* for a discussion of the behaviour of these system functions under Windows.

The system commands *Section 7.2.25* and *Section 7.2.3* provide similar facilities. For further information, see *Section 7.2.25* and *Section 7.2.3*.

Y must be a simple character scalar or vector representing a UNIX shell command. R is a nested vector of character vectors.

Y may be any acceptable UNIX command. If the command does not produce any output, R is 0⍴⊂'' but the result is suppressed if not explicitly used or assigned. If the command has a non-zero exit code, then APL will signal a DOMAIN ERROR. If the command returns a result and has a zero exit code, then each element of R will be a line from the standard output (stdout) of the command. Output from standard error (stderr) is not captured unless redirected to stdout.

See also *Section 6.4.134*.

EXAMPLES

```
      ⎕SH'ls'
FILES WS temp

      ⎕SH 'rm WS/TEST'

      ⎕SH 'grep bin /etc/passwd ; exit 0'
bin:!:2:2::/bin:

      ⎕SH 'apl MYWS <inputfile >out1 2>out2 &'
```

Note

This function is disabled and instead generates a DOMAIN ERROR if the RIDE_SPAWNED parameter is non-zero. This is designed to prevent it being invoked from a Ride session which does not support this type of user interface. For further details, see the Ride User Guide.

6.4.132 Start UNIX Auxiliary Processor {R}←X ⎕SH Y

Used dyadically, ⎕SH starts an Auxiliary Processor. The effect, as far as the APL user is concerned, is identical under both Windows and UNIX although there are differences in the method of implementation. ⎕SH is a synonym of ⎕CMD Either function may be used in either environment (UNIX or Windows) with exactly the same effect. This section describes the behaviour of ⎕SH and ⎕CMD under UNIX. See *Section 6.4.18* for a discussion of the behaviour of these system functions under Windows.

Although it is still possible for users to create their own APs, Dyalog strongly recommends creating shared libraries/DLLs instead.

X must be a simple character vector. Y may be a simple character scalar or vector, or a nested character vector.

⎕SH loads the Auxiliary Processor from the file named by X using a search-path defined by the environment variable WSPATH.

The shy result R is the process id of the Auxiliary Processor task.

The effect of starting an AP is that one or more **external functions** are defined in the workspace. These appear as locked functions and may be used in exactly the same way as regular defined functions.

When an external function is used in an expression, the argument(s) (if any) are **piped** to the AP for processing. If the function returns a result, APL halts while the AP is processing and waits for the result. If not it continues processing in parallel.

The syntax of dyadic ⎕SH is similar to the UNIX execl(2) system call, where 'taskname' is the name of the auxiliary processor to be executed and arg0 through argn are the parameters of the calling line to be passed to the task, viz.

```
      'taskname' ⎕SH 'arg0' 'arg1' ... 'argn'
```

Examples

```
      'xutils' ⎕SH 'xutils' 'ss' 'dbr'
      '/bin/sh' ⎕SH 'sh' '-c' 'adb test'
```

Warning

Under macOS and Linux, if the configuration parameter **ENABLE_CEF** is 1, Auxiliary Processors cannot be used (they hang on error). The default value is 1 unless you are not running under a desktop (for example, you are running Dyalog in a PuTTY session when the default is 0).

6.4.133 Shadow Name {R}←⎕SHADOW Y

`Y` must be a simple character scalar, vector, or matrix, or a nested vector of character vectors or scalar characters, identifying one or more APL names. For a simple vector `Y`, names are separated by one or more blanks. For a matrix `Y`, each row is taken to be a single name.

Each valid name in `Y` is shadowed in the most recently invoked defined function or operator, as though it were included in the list of local names in the function or operator header. The class of the name becomes 0 (undefined). The name ceases to be shadowed when execution of the shadowing function or operator is completed. Shadow has no effect when the state indicator is empty.

The shy result `R` is a Boolean vector of 1s with the same length as the number of names in `Y`.

If a name is ill-formed, or if it is the name of a system constant or system function, `DOMAIN ERROR` is reported.

If the name of a top-level GUI object is shadowed, it is made inactive.

EXAMPLE

```
      ⎕VR'RUN'
     ∇ NAME RUN FN
[1]   ⍝ Runs function named <NAME> defined
[2]   ⍝ from representation form <FN>
[3]    ⎕SHADOW NAME
[4]    ⍎⎕FX FN
     ∇

      0 ⎕STOP 'RUN' ⍝ stop prior RUN exiting

      'FOO' RUN 'R←FOO' 'R←10'
10

RUN[0]

      )SINL
#.RUN[0]*       FOO     FN      NAME

      →⎕LC

      FOO
VALUE ERROR
      FOO
      ^
```

6.4.134 Execute External Program R←⎕SHELL Y

⎕SHELL executes an external program, either directly or using the operating system's shell.

Y describes the command to execute. Y is specified as either a character vector or a vector of character vectors. R contains information about the termination reason, along with output collected.

Although ⎕SHELL itself is cross-platform, the command specification depends on the platform's shell, the program being invoked, and the expected format of the command line arguments – ⎕SHELL calls are only as cross-platform as the programs they execute. In practice, many command line tools that are useful when building cross-platform applications, such as `git` and `aws-cli`, *are* cross-platform tools, which makes this less likely to be an issue.

Using the System's Shell

When `Y` is a character vector, the contents are executed using the system's shell. The shell being used can be changed using the `Shell` variant. The defaults are:

- Microsoft Windows: `PowerShell`
- Linux, macOS, AIX: `/bin/sh`

Example (Microsoft Windows)

```
      ⍝ SHELL 'Get-ChildItem -path C:\tmp'


    Directory: C:\tmp

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         19-11-2024     12:42                ciderTest
d-----         19-11-2024     12:58                test2
d-----         26-11-2024     12:56                ullu
-a----         20-12-2024     14:14            803 log.txt
-a----         26-11-2024     14:11             12 test.apls
```

Direct Execution

When `Y` is a vector of character vectors or an enclosed character vector, the first element of the array is treated as a program name, and the rest as individual arguments. The program is executed directly, without invoking the system's shell first. The program name can be an absolute path, a relative path, or the name of an executable in the current search path (operating-system specific). When the program is specified as a relative path, the name is resolved relative to the working directory, which can be set with the `WorkingDir` variant.

If the command does not depend on any shell features, then direct execution should be used as it has lower overhead.

> **Information**
>
> This difference in performance can be very significant, especially under Microsoft Windows where PowerShell is the default. Direct calls can be 5-10x faster.
>
> See the example below for information about how to use `cmd.exe` instead of PowerShell.

Return Value

The result is a five element nested vector:

```
(StreamData StreamIds ExitCode ExitReason Pid)←R
```

Within this:

- `StreamData` and `StreamIds`, describe the output that was collected while the program ran.
- `ExitCode` and `ExitReason` describe why the `⎕SHELL` call ended.
- `Pid` is either the positive integer process ID of the child process (if the `⎕SHELL` call ends before the child process has ended), or `¯1`.

Stream Data

A stream is a connection between the process and some resource, such as a file, or a pipe connecting two different processes. All running processes have a set of open, numbered, streams that they can read from or write to. Even though the streams are referenced by number, some streams have typical usage conventions:

Stream number	Name	Type	Description
0	Standard input	Input	In interactive programs running in a terminal window, input typed on the keyboard typically goes to standard input. In a shell pipeline, standard output of one program is connected to standard input of another program.
1	Standard output	Output	Used to print regular output data.
2	Standard error	Output	Used to print error messages or other information that is not part of the main output.

`⎕SHELL` provides great flexibility in what the streams should point to, by using the `Output` and `Input` variants. By default, the two output streams, 1 and 2, are configured such that the output produced on standard error is redirected to standard output and effectively merged together, and then turned into an APL array representing the lines of text.

The StreamData and StreamIds vectors have the same length. The StreamIds are always sorted in ascending order. Each index describes one collected output stream; StreamIds contains the integer stream IDs, and the elements of StreamData depend on how the output is collected (see the `Output` variant).

By default, `⍎⍕⎕SHELL cmd` returns a vector of character vectors representing the lines of the collected output text from the child process (both standard output and standard error).

Exit Reasons

The reasons why a call to `⎕SHELL` ends are described in the table below. `ExitReason` is an integer that describes the overall reason, and `ExitCode` provides additional information.

`ExitReason`	**Description**	`ExitCode`
0	The child process finished and exited normally.	The non-negative exit code.
1	The child process was terminated by a signal.	The negated signal number that caused the termination.
2	`⎕SHELL` timed out before the child process exited (see the `Timeout` option).	The constant `¯1006`.
3	`⎕SHELL` was interrupted by a weak interrupt in the IDE.	The constant `¯1002`.

> **Dyalog on Microsoft Windows**
>
> `ExitReason` cannot be 1 on Microsoft Windows.

Returning the exit code (instead of `⎕SHELL` producing some trappable error) makes it possible to access the other parts of the result, such as the error messages that were printed on the standard error stream. However, it is possible to turn non-successful exits into trappable errors using the `ExitCheck` variant.

When the `ExitReason` is 2 or 3, the child process had not stopped running before `⎕SHELL` stopped, which means it might still be running, and require some appropriate cleanup (see *8373⌶*). In practice, the child process often closes shortly after without the need for intervention, either due to the default signal being sent to it on non-Windows platforms (see the `Signal` option), or because the connected streams are closed on the `⎕SHELL` end.

Thread Switching

`⎕SHELL` is a thread switch point, which means the interpreter will run other APL threads while a long-running `⎕SHELL` call is in progress.

When an APL thread running `⎕SHELL` is terminated by *Section 6.4.158* or *Section 7.2.23*, the child process might be left running, as if `⎕SHELL` was interrupted.

Variant Options

SHELL supports the following variant options to control certain parts of the program execution context.

Output

The Output variant option controls output stream redirections. The value must describe a set of redirections, in one of the following formats:

- a two-column matrix.
- a two-element vector.
- a vector of two-element vectors.

Each row or two-element vector should be of the form (Stream Destination), in which: - Stream is the integer stream number. - Destination is one of the possible destinations described in the table below.

Any output the child process produces on its stream Stream will go to the specified destination.

Format	Description
`('Stream' n)`	Redirect the output to another stream with the ID `n`, which is also configured. For example, to redirect standard error to standard output, merging the two, the variant value would be `(2 ('Stream' 1))`.
`('File' n)`	Redirect the output to an open native file with tie number `n`. The file must be writable.
`('File' path)`	Redirect the output to an existing file with a path described by the character vector `path`. The output is appended to the end of the file.
`('Array' type)`	Redirect the output to the interpreter, which will convert it into a vector with data type `type`, and include it in the result of `⎕SHELL`. For example, to get standard output as a Boolean vector, the variant value would be `(1 ('Array' 11))`.
`'Array'`	Redirect the output to the interpreter, which will convert it to a vector of character vectors, split into the separate lines of text. Note that the line endings and text encoding are determined automatically, like when reading files using monadic `⎕NGET`. The nested array is included as part of the result of `⎕SHELL`.
`('Array' textEncoding)`	Redirect the output to the interpreter like with the `'Array'` format above, but use a specific text encoding, described by the character vector `textEncoding`. The possible values for `textEncodings` are those which are allowed as the left argument to dyadic `⎕NGET`.
`'Null'`	Redirect the output to the operating-system's null device, which effectively means the output is ignored.
`('Callback' fn encodingOrType)`	Redirect the output to the interpreter, which periodically will turn the data produced into an array, and run a callback function specified in `fn`. `fn` can either be a character vector, in which case it is the name of the function to run, or it can be a 2-element vector where the first element is the function name, and the second element is some arbitrary data which will be passed as the left argument to the callback function. When `encodingOrType` is a scalar integer, the conversion behaves as with `('Array' type)`, and otherwise it behaves as `('Array' textEncoding)`.

Format	Description
`('Callback' fn)`	This is identical to the other callback format, with the difference that the data is converted into lines of text in the same way as when the `'Array'` format is used, using heuristics to determine the line endings and text encoding.
`n`	When the destination is a scalar integer, it is treated as a shorthand for either `('Stream' n)` or `('File' n)`, depending on the sign of the integer. All non-negative numbers are treated as stream IDs, while negative numbers are treated as native file ties.

The default is is `0 2⍴0`, but see Default Redirections.

Callbacks

The callbacks supported by the `(Callback fn)` and `(Callback fn encodingOrType)` can be invoked either monadically or dyadically, depending on the format of `fn`:

- A simple character vector: the name of the function to call.
- A two-element vector `(name leftarg)`: the character vector name, and some arbitrary data to be passed as the left argument to the callback function.

When `encodingOrType` is a scalar integer, specifying a data type, the callback runs in *simple vector mode*, otherwise it runs in *line mode*. `(Callback fn)` without an explicit encoding is similar, but the text encoding is deduced automatically on the first call to the callback, and reused for subsequent calls.

The right argument is a namespace that contains the members described in the table below. The namespace is created once for each callback specification and updated just before each call.

Name	Description
`Data`	Initially contains `⍬`, and provides a name where the user code can store information between subsequent calls to the callback function.
`Done`	A Boolean scalar indicating that this is the last call to the callback for the given stream. Under normal circumstances where `ExitReason` is 0 or 1, it is guaranteed that there will be a single call with `Done=1`, even if no data was produced since the previous call.
`Encoding`	A character vector describing the text encoding that was used to decode the raw bytes. When the callback runs in simple vector mode, this member is an empty vector. It is most useful when the callback is specified as `('Callback' fn)`, without an explicit text encoding.
`Output`	The APL array produced from converting the bytes produced since the previous callback, according to the encoding or type specified. When the callback is running in line mode, it will also include any partial line from the end of the previous call's `Output`.
`Stream`	A scalar integer specifying the stream number. This allows the same callback function to be used on multiple output streams, and provides a way for the callback code to determine which stream it is currently processing.
`PartialLine`	A Boolean scalar indicating whether the last line in `Output` was terminated by a newline character or sequence (1 indicates that the line is partial, not terminated by a newline character or sequence). As partial lines are included again in the next call's `Output`, it is possible to process only the complete lines by conditionally ignoring the last element of `Output` when `PartialLine=1`. In simple vector mode, this is always 0.

The callback is invoked when:

- the internal buffer for the given stream is full.
- there is data in the internal buffer, but no I/O is immediately possible between the interpreter and the child process.
- there is data in the internal buffer, and the child process has finished running.

Depending on the type or encoding, some of these conditions could happen when the buffer contains a number of bytes that do not make up a full item, such as when the

buffer is full and the last byte is only one of multiple bytes needed to encode a UTF-8 character.

Input

The Input variant option controls input stream redirections. The value must describe a set of redirections, in one of the following formats:

- a two-column matrix.
- a two-element vector.
- a vector of two-element vectors.

Each row or two-element vector should be of the form `(Stream Source)`, in which: - `Stream` is the integer stream number. - `Source` is one of the possible sources described in the table below.

Any input the child process tries to read on its stream `Stream` will come from the specified source.

Format	Description
`('File' n)`	Input on the stream comes from an open native file with tie number `n`. The file must be readable.
`('File' path)`	Input comes from an existing file with a path described by the character vector `path`.
`('Array' Data Type)`	The APL array `Data` is raveled and converted to a vector of type `Type`. The input on the stream comes from the bytes of the converted array.
`('Array' Data)`	Shorthand for `('Array' Data (⎕DR Data))`. In Unicode interpreters when `Data` is a character array, this is the same as `('Array' Data 80)`.
`('Array' Data Encoding)`	This is similar to `('Array' Data Type)`, but `Data` must be either a character vector, or a vector of character vectors. Each character vector is converted to text in the given `Encoding`, and terminated by a newline character/character sequence. The `Encoding` must be one of those supported by `⎕NPUT`.
`('Array' Data Encoding Newline)`	Similar to `('Array' Data Encoding)`, but provides explicit control over the newline character/sequence. `Newline` must be one of the values supported by `⎕NPUT`.
`'Null'`	Input comes from the operating-system's null device, which effectively means no input is provided.
`('Token' n)`	`⎕SHELL` will periodically see if any tokens are available on the token number `n`, such as those produced by `X ⎕TPUT n`. When one becomes available, the token data `X` is parsed as one of the `('Array' ...)` sources, and appended to a buffer of data that the child process then sees on the specified stream. A token with no data, as constructed by a monadic `⎕TPUT` call, closes the stream. This, combined with running `⎕SHELL` on its own APL thread, provides a mechanism for feeding a child process input input dynamically.

The default is `0 2⍴0`, but see Default Redirections.

WorkingDir

The `WorkingDir` variant option sets the working directory of the child process. The value must be a character vector that refers to a valid directory, such as `'/tmp/somedir'` on Linux, or `'C:\tmp\somedir'` on Microsoft Windows.

The default is the current working directory of the interpreter.

InheritEnv

The `InheritEnv` variant option specifies whether the set of environment variables from the interpreter should be inherited by the child process. The value must be a Boolean scalar.

The default is `1`.

Env

The `Env` variant option specifies any additional environment variables and their values. If an environment variable that already exists in the set of inherited variables (see `InheritEnv`) is specified here, the value from `Env` takes precedence. The option value must be one of the following:

- a two-column matrix.
- a two-element vector.
- a vector of two-element vectors.

Each row or two-element vector consists of an environment variable name and its value, both specified as character vectors.

Example: Add three environment variables: `DAY=monday`, `MONTH=december`, and `WEEK=50`.

```
      ⍝ Using a two-column matrix and array notation
      env←[
        'DAY' 'monday'
        'MONTH' 'december'
        'WEEK' '50'
      ]
      ⎕SHELL⍠'Env' env⊢cmd
...

      ⍝ Using a nested vector
      env←('DAY' 'monday') ('MONTH' 'december') ('WEEK' '50')
      ⎕SHELL⍠'Env' env⊢cmd
...
```

All the environment variable names must be unique within the scope of the call to ⎕SHELL.

The default is `0 2⍴⊂⍬`.

Shell

The `Shell` variant option sets the shell to be used when `Y` is a character vector.

The value must be a character vector or a vector of character vectors with a length of at least 1.

> **Info**
>
> Shells typically takes some argument which specify that the next argument is a command to run, such as `/bin/bash -c` on Linux, but since the argument differs from shell to shell, it must be specified manually.

Example using cmd.exe (Microsoft Windows)

```
      ⎕SHELL⍠'Shell' ('cmd.exe' '/C')⊢'someCmd'
...
```

Example using bash (Linux)

```
      ⎕SHELL⍠'Shell' ('/bin/bash' '-c')⊢'someCmd'
...
```

There is no difference between the following two calls; the `Shell` variant option is simply a convenience.

```
      ⎕SHELL'/bin/bash' '-c' 'a b c'
      ⎕SHELL⍠'Shell' ('/bin/bash' '-c')⊢'a b c'
```

When the right argument of `⎕SHELL` is a nested vector, the `Shell` option has no effect.

The default depends on the operating-system:

- Microsoft Windows: `('C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe' '-Command')`.
- Linux, macOS, AIX: `('/bin/sh' '-c')`.

ExitCheck

The `ExitCheck` variant option specifies whether `⎕SHELL` should convert abnormal exit reasons and exit codes into a `DOMAIN ERORR`. The value must be a boolean scalar; if set to 1, a `DOMAIN ERROR` is reported if `ExitReason` and `ExitCode` are not both 0.

The default `0`.

Timeout

The `TimeOut` variant option specifies an upper limit for the duration of the `⎕SHELL` call. The value must be a non-negative numeric scalar representing the number of milliseconds. The value `0` allows the `⎕SHELL` call to run for as long as it needs.

The default is `0`.

Signal

The `Signal` variant option specifies a signal to send to the child process when it is being abandoned by `⎕SHELL`. The value must be either an integer representing a valid signal number, or `0` (which means no signal should be sent).

> **Dyalog on Microsoft Windows**
>
> On Microsoft Windows, the only valid signal number is `9`, which causes `⎕SHELL` to call `TerminateProcess()` on the child process, which is similar to signal `9` on the other platforms where it is `SIGKILL`.

The default depends on the operating-system:

- Microsoft Windows: `0`.
- Linux, macOS, AIX: the numeric value of `SIGTERM` which is the signal that asks the child process to shut itself down.

Window

> **Dyalog on Microsoft Windows**
>
> This option only has an effect on Windows.

The `Window` variant option specifies the initial window mode. The value must be a character vector containing one of the initial window parameters described on *Section 6.4.17*.

The default is `'Hidden'`.

See also *8373⌶*.

Default Redirections

Most programs assume that the three standard streams are configured when the program starts. For this reason, ⎕SHELL always sets up some redirections for those three, even when variant options have explicitly only set up some of them. For example, when ⎕SHELL⍠'Output' (1 ('File' '/tmp/log.txt'))⊢cmd is run, stream 0 and stream 2 are set up to their default values.

The default redirections for the two output streams are:

- (1 'Array'): Collect standard output as lines of text.
- (2 ('Stream' 1)): Send standard error to standard output, so it is collected together.

The default for the input stream is

- (0 'Null'): Provide no data on standard input.

> **Dyalog on AIX**
>
> The performance of ⎕SHELL on AIX can be improved using the *Installation/ Configuration: Environment Variables* configuration parameter; this configuration parameter is active by default.

6.4.135 State Indicator R←⎕SI

R is a nested vector of vectors giving the names of the functions or operators in the execution stack.

Example

```
      )SI
#.PLUS[2]*
.
#.MATDIV[4]
#.FOO[1]*
⍟

      ⎕SI
 PLUS   MATDIV   FOO

      (⍴⎕LC)=⍴⎕SI
1
```

If execution stops in a callback function, ⎕DQ will appear on the stack, and may occur more than once

```
      )SI
#.ERRFN[7]*
⎕DQ
#.CALC
⎕DQ
#.MAIN
```

To edit the function on the top of the stack:

```
      ⎕ED ⊃⎕SI
```

The name of the function which called this one:

```
      ⊃1↓⎕SI
```

To check if the function ΔN is pendent:

```
     ((⊂ΔN)∊1↓⎕SI)/'Warning : ',ΔN,' is pendent'
```

See also *Section 6.4.183*.

6.4.136 Signal Event {R}←{X}⎕SIGNAL Y

Y must be a scalar or vector.

If Y is a an empty vector nothing is signalled.

If `Y` is a vector of more than one element, all but the first element are ignored.

`R` has the same value as `Y`.

`Y←0` is a special form of `⎕SIGNAL`, the side effect of which is to reset the values of certain system constants. It is described further down this section.

If the first element of `Y` is a simple integer it is taken to be an event number. Permitted values are 0, 1-999 and 1006. `X` is an optional text message. If present, `X` must be a simple character scalar or vector, or an object reference. If `X` is omitted or is empty, the standard event message for the corresponding event number is assumed. See *Programming: Apl Errors*. If there is no standard message, a message of the form `ERROR NUMBER n` is composed, where `n` is the event number in `Y`. Values outside the permitted range will result in a `DOMAIN ERROR`.

If the first element of `Y` is a 2 column matrix or a vector of 2 element vectors of name/values pairs, then it is considered to be a set of values to be used to override the default values in a new instance of `⎕DMX`. Any other value for the first element of `Y` will result in a `DOMAIN ERROR`.

The names in the error specification must all appear in a system-generated `⎕DMX`, otherwise a `DOMAIN ERROR` will be issued. For each name specified, the default value in the new instance of `⎕DMX` is replaced with the value specified. `EN` must be one of the names in the error specification. Attempting to specify certain names, including `InternalLocation` and `DM`, will result in a `DOMAIN ERROR`. The value which is to be assigned to a name must be appropriate to the name in question.

Dyalog may enhance `⎕DMX` in future, thus potentially altering the list of valid and/or assignable names.

If the first element of `Y` is an array of name/value pairs then specifying any value for `X` will result in a `DOMAIN ERROR`.

The effect of the system function is to interrupt execution. The state indicator is cut back to exit from the function or operator containing the line that invoked `⎕SIGNAL`, or is cut back to exit the Execute (`⍎`) expression that invoked `⎕SIGNAL`. If executed within a nested dfn, the state indicator is cut back to exit from the capsule containing the line that invoked `⎕SIGNAL`. An error is then generated.

An error interrupt may be trapped if the system variable `⎕TRAP` is set to intercept the event. Otherwise, the standard system action is taken (which may involve cutting back the state indicator further if there are locked functions or operators in the state indicator). The standard event message is replaced by the text given in `X`, if present.

Example

```
      ⎕VR'DIVIDE'
     ∇ R←A DIVIDE B;⎕TRAP
[1]    ⎕TRAP←11 'E' '→ERR'
[2]    R←A÷B ⋄ →0
[3]   ERR:'DIVISION ERROR' ⎕SIGNAL 11
     ∇

      2 4 6 DIVIDE 0
DIVISION ERROR
      2 4 6 DIVIDE 0
     ^
```

If you are using the Microsoft .NET Framework, you may use ⎕SIGNAL to throw an exception by specifying a value of 90 in Y. In this case, if you specify the optional left argument X, it must be a reference to a .NET object that is or derives from the Microsoft .NET class System.Exception. The following example illustrates a *constructor* function CTOR that expects to be called with a value for ⎕IO (0 or 1)

```
     ∇ CTOR IO;EX
[1]    :If IO∊0 1
[2]        ⎕IO←IO
[3]    :Else
[4]        EX←ArgumentException.New'IO must be 0 or 1'
[5]        EX ⎕SIGNAL 90
[6]    :EndIf
     ∇
```

⎕SIGNAL 0: Reset error-related system constants

If Y is a simple integer with the value 0, ⎕SIGNAL does not interrupt execution, but merely returns the value 0. The side effect of calling ⎕SIGNAL 0 is to reset the values of ⎕DM, ⎕DMX, ⎕EN and ⎕EXCEPTION to their default values. ⎕SIGNAL 0 is the only form of ⎕SIGNAL which can be used to reset the aforementioned system constants; including a left argument or using a name/value pair right argument of ⎕SIGNAL will result in a DOMAIN ERROR.

EXAMPLE

```
      ÷0
DOMAIN ERROR: Divide by zero
      ÷0
     ^
      ⎕DM ⎕EN ⎕DMX
┌──────────────────────────────────┬──┬──────────────────────────────┐
│┌────────────┬────────┬─────────┐│11│ EM        DOMAIN ERROR       │
│|DOMAIN ERROR|      ÷0|      ^|||  | Message   Divide by zero      |
│└────────────┴────────┴─────────┘│  │                              │
└──────────────────────────────────┴──┴──────────────────────────────┘
      ⊢⎕SIGNAL 0
0
      ⎕DM ⎕EN ⎕DMX
┌─┬─┐
||0||
└─┴─┘
```

Further examples

Example 1

```
      'Hello'⎕SIGNAL 200
Hello
      'Hello'⎕SIGNAL 200
     ^
      ⎕DMX
 EM       Hello
 Message

      ⎕DM
 Hello        'Hello'⎕SIGNAL 200       ^
```

```
      ⎕SIGNAL⊂⊂('EN' 200)
ERROR 200
      ⎕SIGNAL⊂⊂('EN' 200)
     ^

      ⎕DMX
 EM       ERROR 200
 Message

      ⎕DM
 ERROR 200        ⎕SIGNAL⊂⊂('EN' 200)       ^
```

Example 2

```
      ⎕SIGNAL⊂('EN' 200)('Vendor' 'Andy')('Message' 'My error')
ERROR 200: My error
      ⎕SIGNAL⊂('EN' 200)('Vendor' 'Andy')('Message' 'My error')
     ^
```

```
      ⎕DMX
 EM       ERROR 200
 Message  My error
```

```
      ⍪⎕DMX.(EN EM Vendor)
       200
 ERROR 200
      Andy
```

Be aware of the following case, in which the argument has not been sufficiently nested:

```
      ⎕SIGNAL⊂('EN' 200)
DOMAIN ERROR: Unexpected name in signalled ⎕DMX specification
      ⎕SIGNAL⊂('EN' 200)
     ^
```

6.4.137 Size of Object R←⎕SIZE Y

`Y` must be a simple character scalar, vector or matrix, or a vector of character vectors containing a list of names. `R` is a simple integer vector of non-negative elements with the same length as the number of names in `Y`.

If the name in ⍵ identifies an object with an active referent, the workspace required in bytes by that object is returned in the corresponding element of ⍺. Otherwise, 0 is returned in that element of ⍺.

The result returned for an external variable is the space required to store the external array. The result for a system constant, variable or function is 0. The result returned for a GUI object gives the amount of workspace needed to store it, but excludes the space required for its children.

Note: Wherever possible, Dyalog APL *shares* the whole or part of a workspace object rather than generates a separate copy; however ⎕SIZE reports the size as though nothing is shared. ⎕SIZE also includes the space required for the interpreter's internal information about the object in question.

Examples

```
      ⎕VR 'FOO'
     ∇ R←FOO
[1]    R←10
     ∇

      A←⍳10

      'EXT/ARRAY' ⎕XT'E' ⋄ E←⍳20

      ⎕SIZE 'A' 'FOO' 'E' 'UND'
28 76 120 0
```

6.4.138 Screen Map ⎕SM

⎕SM is a system variable that defines a character-based user interface (as opposed to a graphical user interface). In versions of Dyalog APL that support asynchronous terminals, ⎕SM defines a **form** that is displayed on the **USER SCREEN**. The implementation of ⎕SM in "window" environments is compatible with these versions. In Dyalog APL/X, ⎕SM occupies its own separate window on the display, but is otherwise equivalent. In versions of Dyalog APL with GUI support, ⎕SM either occupies its own separate window (as in Dyalog APL/X) or, if it exists, uses the window assigned to the SM object. This allows ⎕SM to be used in a GUI application in conjunction with other GUI components.

⎕SM has workspace scope.

In general ⎕SM is a nested matrix containing between 3 and 13 columns. Each row of ⎕SM represents a **field**; each column a **field attribute**.

The columns have the following meanings:

Column	Description	Default
1	Field Contents	N/A
2	Field Position - Top Row	N/A
3	Field Position - Left Column	N/A
4	Window Size - Rows	0
5	Window Size - Columns	0
6	Field Type	0
7	Behaviour	0
8	Video Attributes	0
9	Active Video Attributes	¯1
10	Home Element - Row	1
11	Home Element - Column	1
12	Scrolling Group - Vertical	0
13	Scrolling Group - Horizontal	0

With the exception of columns 1 and 8, all elements in ⎕SM are integer scalar values.

Elements in column 1 (Field Contents) may be:

- A numeric scalar
- A numeric vector
- A 1-column numeric matrix
- A character scalar
- A character vector
- A character matrix (rank 2)
- A nested matrix defining a sub-form whose structure and contents must conform to that defined for ⎕SM as a whole. This definition is recursive. Note however that a sub-form must be a matrix - a vector is not allowed.

Elements in column 8 (Video Attributes) may be:

- An integer scalar that specifies the appearance of the entire field.
- An integer array of the same shape as the field contents. Each element specifies the appearance of the corresponding element in the field contents.

Screen Management (Async Terminals)

Dyalog APL for UNIX systems on tty devices (async terminals or on terminal emulators) manages two screens; the SESSION screen and the USER screen. If the SESSION screen is current, an assignment to ⎕SM causes the display to switch to the USER screen and show the form defined by ⎕SM.

Note that Ride does not directly support ⎕SM, although it is possible to display ⎕SM in the tty session to which a Ride client is connected.

If the USER screen is current, any change in the value of ⎕SM is immediately reflected by a corresponding change in the appearance of the display. However, an assignment to ⎕SM that leaves its value unchanged has no effect.

Dyalog APL automatically switches to the SESSION screen for default output, if it enters immediate input mode (6-space prompt), or through use of ⎕ or ⍞. This means that typing

```
      ⎕SM ← expression
```

in the APL session will cause the screen to switch first to the USER screen, display the form defined by ⎕SM, and then switch back to the SESSION screen to issue the 6-space prompt. This normally happens so quickly that all the user sees is a flash on the screen.To retain the USER screen in view it is necessary to issue a call to ⎕SR or for APL to continue processing. For example:

```
      ⎕SM ← expression  ⋄  ⎕SR 1
```

or

```
      ⎕SM ← expression  ⋄  ⎕DL 5
```

Screen Management (Window Versions)

In Dyalog APL/X, and optionally in Dyalog APL/W, ⎕SM is displayed in a separate **USER WINDOW** on the screen. In an end-user application this may be the only Dyalog APL

window. However, during development, there will be a SESSION window, and perhaps EDIT and TRACE windows too.

The USER Window will only accept input during execution of ⎕SR. It is otherwise "output-only". Furthermore, during the execution of ⎕SR it is the only active window, and the SESSION, EDIT and TRACE Windows will not respond to user input.

Screen Management (GUI Versions)

In versions of Dyalog APL that provide GUI support, there is a special SM object that defines the position and size of the window to be associated with ⎕SM. This allows character-mode applications developed for previous versions of Dyalog APL to be migrated to and integrated with GUI environments without the need for a total re-write.

Effect of Localisation

Like all system variables (with the exception of ⎕TRAP) ⎕SM is subject to "pass-through localisation". This means that a localised ⎕SM assumes its value from the calling environment. The localisation of ⎕SM does not, of itself therefore, affect the appearance of the display. However, reassignment of a localised ⎕SM causes the new form to overlay rather than replace whatever forms are defined further down the stack. The localisation of ⎕SM thus provides a simple method of defining pop-up forms, help messages, etc.

The user may edit the form defined by ⎕SM using the system function ⎕SR. Under the control of ⎕SR the user may change the following elements in ⎕SM which may afterwards be referenced to obtain the new values.

Column 1	Field Contents
Column 10	Home Element - Row (by scrolling vertically)
Column 11	Home Element - Column (by scrolling horizontally)

6.4.139 Screen Read `R←{X}⎕SR Y`

⎕SR is a system function that allows the user to edit or otherwise interact with the form defined by ⎕SM.

In versions of Dyalog APL that support asynchronous terminals, if the current screen is the SESSION screen, ⎕SR immediately switches to the USER SCREEN and displays the form defined by ⎕SM.

In Dyalog APL/X, ⎕SR causes the input cursor to be positioned in the USER window. During execution of ⎕SR, only the USER Window defined by ⎕SM will accept input and respond to the keyboard or mouse. The SESSION and any EDIT and TRACE Windows that may appear on the display are dormant.

In versions of Dyalog APL with GUI support, a single SM object may be defined. This object defines the size and position of the ⎕SM window, and allows ⎕SM to be used in conjunctions with other GUI components. In these versions, ⎕SR acts as a superset of ⎕DQ (see *Section 6.4.31*) but additionally controls the character-based user interface defined by ⎕SM.

Y is an integer vector that specifies the fields which the user may visit. In versions with GUI support, Y may additionally contain the names of GUI objects with which the user may also interact.

If specified, X may be an enclosed vector of character vectors defining EXIT_KEYS or a 2-element nested vector defining EXIT_KEYS and the INITIAL_CONTEXT.

The result R is the EXIT_CONTEXT.

Thus the 3 uses of ⎕SR are:

```
EXIT_CONTEXT←⎕SR FIELDS

EXIT_CONTEXT←(⊂EXIT_KEYS)⎕SR FIELDS

EXIT_CONTEXT←(EXIT_KEYS)(INITIAL_CONTEXT)⎕SR FIELDS
```

FIELDS

If an element of Y is an integer scalar, it specifies a field as the index of a row in ⎕SM (if ⎕SM is a vector it is regarded as having 1 row).

If an element of Y is an integer vector, it specifies a sub-field. The first element in Y specifies the top-level field as above. The next element is used to index a row in the form defined by ⊃⎕SM[Y[1];1] and so forth.

If an element of Y is a character scalar or vector, it specifies the name of a top-level GUI object with which the user may also interact. Such an object must be a "top-level"

object, that is, the `Root` object (`'.'`) or a `Form` or pop-up `Menu`. This feature is implemented ONLY in versions of Dyalog APL with GUI support.

EXIT_KEYS

Each element of `EXIT_KEYS` is a 2-character code from the Input Translate Table for the keyboard. If the user presses one of these keys, `⎕SR` will terminate and return a result.

If `EXIT_KEYS` is not specified, it defaults to:

```
      'ER' 'EP' 'QT'
```

which (normally) specifies , and .

INITIAL_CONTEXT

This is a vector of between 3 and 6 elements with the following meanings and defaults:

Element	Description	Default
1	Initial Field	N/A
2	Initial Cursor Position - Row	N/A
3	Initial Cursor Position - Col	N/A
4	Initial Keystroke	`''`
5	(ignored)	N/A
6	Changed Field Flags	0

Structure of INITIAL_CONTEXT

`INITIAL_CONTEXT[1]` specifies the field in which the cursor is to be placed. It is an integer scalar or vector, and must be a member of `Y`. It must not specify a field which has `BUTTON` behaviour (64), as the cursor is not allowed to enter such a field.

`INITIAL_CONTEXT[2 3]` are integer scalars which specify the initial cursor position within the field in terms of row and column numbers.

`INITIAL_CONTEXT[4]` is either empty, or a 2-element character vector specifying the initial keystroke as a code from the Input Translate Table for the keyboard.

`INITIAL_CONTEXT[5]` is ignored. It is included so that the `EXIT_CONTEXT` result of one call to `⎕SR` can be used as the `INITIAL_CONTEXT` to a subsequent call.

`INITIAL_CONTEXT[6]` is a Boolean scalar or vector the same length as `Y`. It specifies which of the fields in `Y` has been modified by the user.

EXIT_CONTEXT

The result `EXIT_CONTEXT` is a 6 or 9-element vector whose first 6 elements have the same structure as the `INITIAL_CONTEXT`. Elements 7-9 **only** apply to those versions of Dyalog APL that provide mouse support.

Element	Description
1	Final Field
2	Final Cursor Position - Row
3	Final Cursor Position - Col
4	Terminating Keystroke
5	Event Code
6	Changed Field Flags
7	Pointer Field
8	Pointer Position - Row
9	Pointer Position - Col

Structure of the Result of `⎕SR`

`EXIT_CONTEXT[1]` contains the field in which the cursor was when `⎕SR` terminated due to the user pressing an exit key or due to an event occurring. It is an integer scalar or vector, and a member of `Y`.

`EXIT_CONTEXT[2 3]` are integer scalars which specify the row and column position of the cursor within the field `EXIT_CONTEXT[1]` when `⎕SR` terminated.

`EXIT_CONTEXT[4]` is a 2-element character vector specifying the last keystroke pressed by the user before `⎕SR` terminated. Unless `⎕SR` terminated due to an event, `EXIT_CONTEXT[4]` will contain one of the exit keys defined by `X`. The keystroke is defined in terms of an Input Translate Table code.

`EXIT_CONTEXT[5]` contains the **sum** of the event codes that caused `⎕SR` to terminate. For example, if the user pressed a mouse button on a `BUTTON` field (event code 64) **and** the current field has `MODIFIED` behaviour (event code 2) `EXIT_CONTEXT[5]` will have the value 66.

`EXIT_CONTEXT[6]` is a Boolean scalar or vector the same length as `Y`. It specifies which of the fields in `Y` has been modified by the user during **this** `⎕SR`, ORed with `INITIAL_CONTEXT[6]`. Thus if the `EXIT_CONTEXT` of one call to `⎕SR` is fed back as the `INITIAL_CONTEXT` of the next, `EXIT_CONTEXT[6]` records the fields changed since the start of the process.

EXIT_CONTEXT (Window Versions)

`⎕SR` returns a 9-element result **ONLY** if it is terminated by the user pressing a mouse button. In this case:

`EXIT_CONTEXT[7]` contains the field over which the mouse pointer was positioned when the user pressed a button. It is an integer scalar or vector, and a member of `Y`.

`EXIT_CONTEXT[8 9]` are integer scalars which specify the row and column position of the mouse pointer within the field `EXIT_CONTEXT[7]` when `⎕SR` terminated.

> **Note**
>
> This function is disabled and instead generates a `DOMAIN ERROR` if the RIDE_SPAWNED parameter is non-zero. This is designed to prevent it being invoked from a Ride session which does not support this type of user interface. For further details, see the Ride User Guide.

6.4.140 Source `R←⎕SRC Y`

`⎕SRC` returns the script that defines the scripted object `Y`.

`Y` must be a reference to a scripted object. Scripted objects include Classes, Interfaces and scripted Namespaces.

`R` is a vector of character vectors containing the script that was used to define `Y`.

```
      )ED ⍥MyClass
```

```
:Class MyClass
∇ r←foo arg
:Access public shared
r←1+arg
∇
:EndClass

      z←⎕SRC MyClass
      ⍴z
6
      ⍴⍴z
 14  15  27  13  5  9
      ⍪z
 :Class MyClass
     ∇ r←foo arg
       :Access public shared
       r←1+arg
     ∇
 :EndClass
```

> **Note**
>
> The only two ways to permanently alter the source of a scripted object are to change the object in the editor, or by refixing it using `⎕FIX` . A useful technique to ensure that a scripted object is in sync with its source is to `⎕FIX ⎕SRC ref` , where `ref` is an object reference.

6.4.141 State Indicator Stack R←⎕STACK

`R` is a two-column matrix, with one row per entry in the state indicator.

Column 1 :`⎕OR` form of user defined functions or operators on the state indicator. Space (`⎕UCS 32`) for entries that are not user defined functions or operators.

Column 2 :Indication of the type of the item on the stack.

space	user defined function or operator
⍣	execute level
⍞	evaluated input
*	desk calculator level
⍞DQ	in callback function
other	primitive operator

Example

```
      )SI
#.PLUS[2]*
.
#.MATDIV[4]
#.FOO[1]*
⍣

      ⎕STACK
        *
∇PLUS
        .
∇MATDIV
        *
∇FOO
        ⍣
        *

      ⍴⎕STACK
8 2

      (⍴⎕LC)=1↑⍴⎕STACK
0
```

Pendent defined functions and operators may be edited in Dyalog APL with no resulting SI damage. However, only the visible definition is changed; the pendent version on the stack is retained until its execution is complete. When the function or operator is displayed, only the visible version is seen. Hence ⎕STACK is a tool which allows the user to display the form of the actual function or operator being executed.

Example

To display the version of MATDIV currently pendent on the stack:

```
      ⊃⎕STACK[4;1]
     ∇ R←A MATDIV B
[1]   ⍝ Divide matrix A by matrix B
[2]    C←A⌹B
[3]   ⍝ Check accuracy
[4]    D←⌊0.5+A PLUS.TIMES B
     ∇
```

6.4.142 State of Object R←⎕STATE Y

Y must be a simple character scalar or vector which is taken to be the name of an APL object or a system variable. The result returned is a nested vector of 4 elements as described below. ⎕STATE supplies information about shadowed or localised objects that is otherwise unobtainable.

1⊃R	Boolean vector, element set to 1 if and only if this level shadows Y . Note: (⍴1⊃R)=⍴⎕LC
2⊃R	Numeric vector giving the stack state of this name as it entered this level. Note: (⍴2⊃R)=⍴⎕LC 0=not on stack 1=suspended 2=pendent (may also be suspended)
3⊃R	Numeric vector giving the name classification of Y as it entered this level. Note: (⍴3⊃R)=+/1⊃R
4⊃R	Vector giving the contents of Y before it was shadowed at this level. Note: (⍴4⊃R)=+/0≠3⊃R

Example

```
      ⎕FMT∘⎕OR¨'FN1' 'FN2' 'FN3'
    ∇ FN1;A;B;C      ∇ FN2;A;C              ∇ FN3;A
[1]   A←1        [1]   A←'HELLO'        [1]   A←100
[2]   B←2        [2]   B←'EVERYONE'     [2]   ∘
[3]   C←3        [3]   C←'HOW ARE YOU?'      ∇
[4]   FN2        [4]  FN3
    ∇                ∇

      )SI
#.FN3[2]*
#.FN2[4]
#.FN1[4]

      ⎕STATE 'A'
 1 1 1  0 0 0  2 2 0   HELLO  1

       ⎕FMT∘⎕OR¨'foo' 'goo'
      ∇ foo;⎕IO       ∇ goo;⎕IO
 [1]    ⎕IO←0    [1]    ⎕IO←1
 [2]    goo      [2]    ⎕STATE'⎕IO'
      ∇               ∇

       foo
 1 1  0 0  ¯1 ¯1  0 1
```

6.4.143 Set Stop {R}←X ⎕STOP Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. X must be a simple non-negative integer scalar or vector. R is a simple integer vector of non-negative elements. X identifies the numbers of lines in the function or operator named by Y on which a stop control is to be placed. Numbers outside the range of line numbers in the function or operator (other than 0) are ignored. The number 0 indicates that a stop control is to be placed immediately prior to exit from the function or operator. If X is empty, all existing stop controls are cancelled. The value of X is independent of ⎕IO.

R is a vector of the line numbers on which a stop control has been placed in ascending order. The result is suppressed unless it is explicitly used or assigned.

Examples

```
      ⊢(0,⍳10) ⎕STOP 'FOO'
0 1
```

Existing stop controls in the function or operator named by `Y` are cancelled before new stop controls are set:

```
      ⊢1 ⎕STOP 'FOO'
1
```

All stop controls may be cancelled by giving `X` an empty vector:

```
      ⍴'' ⎕STOP 'FOO'
0

      ⍴⍬ ⎕STOP 'FOO'
0
```

Attempts to set stop controls in a locked function or operator are ignored.

```
      ⎕LOCK'FOO'

      ⊢0 1 ⎕STOP'FOO'
```

The effect of ⎕STOP when a function or operator is invoked is to suspend execution at the beginning of any line in the function or operator on which a stop control is placed immediately before that line is executed, and immediately before exiting from the function or operator if a stop control of 0 is set. Execution may be resumed by a branch expression. A stop control interrupt (1001) may also be trapped - see *Section 6.4.165*.

EXAMPLE

```
      ⎕FX'R←FOO' 'R←10'

      0 1 ⎕STOP'FOO'

      FOO
FOO[1]

      R
VALUE ERROR
      R
      ^

      →1
FOO[0]

      R
10

      →⎕LC
10
```

6.4.144 Query Stop R←⎕STOP Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. R is a simple non-negative integer vector of the line numbers of the function or operator named by Y on which stop controls are set, shown in ascending order. The value 0 in R indicates that a stop control is set immediately prior to exit from the function or operator.

EXAMPLE

```
      ⎕STOP'FOO'
0 1
```

6.4.145 Set Access Control R←X ⎕SVC Y

This system function sets access control on one or more shared variables.

Y is a character scalar, vector, or matrix containing names of shared variables. Each name may optionally be paired with its surrogate. If so, the surrogate must be separated from the name by at least one space.

`X` may be a 4-element Boolean vector which specifies the access control to be applied to all of the shared variables named in `Y`. Alternatively, `X` may be a 4-column Boolean matrix whose rows specify the access control for the corresponding name in `Y`. `X` may also be a scalar or a 1-element vector. If so, it treated as if it were a 4-element vector with the same value in each element.

Each shared variable has a current access control vector which is a 4-element Boolean vector. A 1 in each of the four positions has the following impact :

`[1]`	You cannot **set** a new value for the shared variable until after an intervening use or set by your partner.
`[2]`	Your partner cannot **set** a new value for the shared variable until after an intervening use or set by you.
`[3]`	You cannot **use** the value of the shared variable until after an intervening set by your partner.
`[4]`	Your partner cannot **use** the value of the shared variable until after an intervening set by you.

The effect of `⎕SVC` is to reset the access control vectors for each of the shared variables named in `Y` by OR-ing the values most recently specified by your partner with the values in `X`. This means that you cannot reset elements of the control vector which your partner has set to 1.

Note that the initial value of your partner's access control vector is normally 0 0 0 0. However, if it is a non-APL client application that has established a hot DDE link, its access control vector is defined to be 1 0 0 1. This inhibits either partner from setting the value of the shared variable twice, without an intervening use (or set) by the other. This prevents loss of data which is deemed to be desirable from the nature of the link. (An application that requests a hot link is assumed to require every value of the shared variable, and not to miss any). Note that APL's way of inhibiting another application from setting the value twice (without an intervening use) is to delay the acknowledgement of the DDE message containing the second value until the variable has been used by the APL workspace. An application that waits for an acknowledgement will therefore hang until this happens. An application that does not wait will carry on obliviously.

The result `R` is a Boolean vector or matrix, corresponding to the structure of `X`, which contains the new access control settings. If `Y` refers to a name which is not a shared variable, or if the surrogate name is mis-spelt, the corresponding value in `R` is `4⍴0`.

EXAMPLES

```
      1 0 0 1 ⎕SVC 'X'
1 0 0 1

      1 ⎕SVC 'X EXTNAME'
1 1 1 1

      (2 4⍴1 0 0 1 0 1 1 0) ⎕SVC ↑'ONE' 'TWO'
1 1 1 1
0 1 1 0
```

6.4.146 Query Access Control R←⎕SVC Y

This system function queries the access control on one or more shared variables.

Y is a character scalar, vector, or matrix containing names of shared variables. Each name may optionally be paired with its surrogate. If so, the surrogate must be separated from the name by at least one space.

If Y specifies a single name, the result R is a Boolean vector containing the current effective access control vector. If Y is a matrix of names, R is a Boolean matrix whose rows contain the current effective access control vectors for the corresponding row in Y.

For further information, see the preceding section on setting the access control vector.

EXAMPLE

```
      ⎕SVC 'X'
0 0 0 0
```

6.4.147 Shared Variable Offer R←X ⎕SVO Y

This system function offers to share one or more variables with another APL workspace or with another application. Shared variables are implemented using Dynamic Data Exchange (**DDE**) and may be used to communicate with any other application that supports this protocol. See *Interface Guide* for further details.

Y is a character scalar, vector or matrix. If it is a vector it contains a name and optionally an external name or surrogate. The first name is the name used internally in the current workspace. The external name is the name used to make the connection with the partner and, if specified, must be separated from the internal name by one or more

blanks. If the partner is another application, the external name corresponds to the DDE **item** specified by that application. If the external name is omitted, the internal name is used instead. The internal name must be a valid APL name and be either undefined or be the name of a variable. There are no such restrictions on the content of the external name.

Instead of an external name, `Y` may contain the special symbol `'⍏'` separated from the (internal) name by a blank. This is used to implement a mechanism for sending `DDE_EXECUTE` messages, and is described at the end of this section.

If `Y` is a scalar, it specifies a single 1-character name. If `Y` is a matrix, each row of `Y` specifies a name and an optional external name as for the vector case.

The left argument `X` is a character vector or matrix. If it is a vector, it contains a string that defines the **protocol**, the **application** to which the shared variable is to be connected, and the **topic** of the conversation. These three components are separated by the characters `':'` and `'|'` respectively. The protocol is currently always `'DDE'`, but future implementations of Dyalog APL may support additional communications protocols if applicable. If `Y` specifies more than one name, `X` may be a vector or a matrix with one row per row in `Y`.

If the shared variable offer is a general one (server), `X`, or the corresponding row of `X`, should just contain `'DDE:'`. In this case, Dyalog automatically defines the application name and topic to be `dyalog` and `⎕WSID` respectively.

The result `R` is a numeric scalar or vector with one element for each name in `Y` and indicates the "degree of coupling". A value of 2 indicates that the variable is fully coupled (via a warm or hot DDE link) with a shared variable in another APL workspace, or with a DDE item in another application. A value of 1 indicates that there is no connection, or that the second application rejected a warm link. In this case, a transfer of data may have taken place (via a cold link) but the connection is no longer open. Effectively, APL treats an application that insists on a cold link as if it immediately retracts the sharing after setting or using the value, whichever is appropriate.

Examples

```
      'DDE:' ⎕SVO 'X'
1

      'DDE:' ⎕SVO 'X SALES_92'
1

      'DDE:' ⎕SVO ↑'X SALES_92' 'COSTS_92'
1 1

      'DDE:DYALOG|SERV_WS' ⎕SVO 'X'
2

      'DDE:EXCEL|SHEET1' ⎕SVO 'DATA R1C1:R10C12'
2
```

A special syntax is used to provide a mechanism for sending `DDE_EXECUTE` messages to another application. This case is identified by specifying the `'⍟'` symbol in place of the external name. The subsequent assignment of a character vector to a variable shared with the external name of `'⍟'` causes the value of the variable to be transmitted in the form of a `DDE_EXECUTE` message. The value of the variable is then reset to 1 or 0 corresponding to a positive or negative acknowledgement from the partner. In most (if not all) applications, commands transmitted in `DDE_EXECUTE` messages must be enclosed in square brackets `[]`. For details, see the relevant documentation for the external application.

Examples

```
      'DDE:EXCEL|SYSTEM' ⎕SVO 'X ⍟'
2

      X←'[OPEN("c:\mydir\mysheet.xls")]'
      X
1

      X←'[SELECT("R1C1:R5C10")]'
      X
1
```

6.4.148 Query Degree of Coupling R←⎕SVO Y

This system function returns the current degree of coupling for one or more shared variables.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks.

If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

If Y specifies a single name, the result R is a 1-element vector whose value 0, 1 or 2 indicates its current degree of coupling. If Y specifies more than one name, R is a vector whose elements indicate the current degree of coupling of the variable specified by the corresponding row in Y. A value of 2 indicates that the variable is fully coupled (via a warm or hot DDE link) with a shared variable in another APL workspace, or with a DDE item in another application. A value of 1 indicates that you have offered the variable but there is no such connection, or that the second application rejected a warm link. In this case, a transfer of data may have taken place (via a cold link) but the connection is no longer open. A value of 0 indicates that the name is not a shared variable.

EXAMPLES

```
      ⎕SVO 'X'
2
      ⎕SVO ↑'X SALES' 'Y' 'JUNK'
2 1 0
```

6.4.149 Shared Variable Query R←⎕SVQ Y

This system function is implemented for compatibility with other versions of APL but currently performs no useful function. Its purpose is to obtain a list of outstanding shared variable offers made to you, to which you have not yet responded.

Using DDE as the communication protocol, it is not possible to implement ⎕SVQ effectively.

6.4.150 Shared Variable Retract Offer R←⎕SVR Y

This system function terminates communication via one or more shared variables, or aborts shared variable offers that have not yet been accepted.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks. If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

The result R is vector whose length corresponds to the number of names specified by Y, indicating the level of sharing of each variable after retraction.

See *Section 6.4.151* for further information on the possible states of a shared variable.

6.4.151 Shared Variable State R←⎕SVS Y

This system function returns the current state of one or more shared variables.

Y is a character scalar, vector or matrix. If it is a vector it contains a shared variable name and optionally its external name or surrogate separated from it by one of more blanks. If Y is a scalar, it specifies a single 1-character name. If Y is a matrix, each row of Y specifies a name and an optional external name as for the vector case.

If Y specifies a single name, the result R is a 4-element vector indicating its current state. If Y specifies more than one name, R is a matrix whose rows indicate the current state of the variable specified by the corresponding row in Y.

There are four possible shared variable states:

0011	means that you and your partner are both aware of the current value, and neither has since reset it. This is also the initial value of the state when the link is first established.
1010	means that you have reset the shared variable and your partner has not yet used it. This state can only occur if both partners are APL workspaces.
0101	means that your partner has reset the shared variable but that you have not yet used it.
0000	the name is not that of a shared variable

Examples

```
      ⎕SVS 'X'
0 1 0 1

      ⎕SVS ↑'X SALES' 'Y' 'JUNK'
0 0 1 1
1 0 1 0
0 0 0 0
```

6.4.152 Allocate Token Range {R}←{X} ⎕TALLOC Y

`Y` is either a single integer or a 2-element vector. The first (or only) item in `Y` is 0, 1, 2 or `¯1` and indicates the type of operation to perform. If it is 1, then the optional second item is a character vector.

The optional left argument `X` identifies an existing allocated range of token numbers `n`. `X` must be a scalar greater than or equal to `n`, but must be less than `n+1`.

Allocation (First element of `Y` is 1)

If the first element of `Y` is 1, the result `R` is a positive integer that identifies a range of numbers that may be used as token types for `⎕TPUT` and `⎕TGET`. That range is defined as the set of floating-point numbers between `R` and `R+1` (but not the integer end-points). Negated values of these number may also be used.

In this case, the optional `Y[2]` is an arbitrary character vector that serves as a description for the allocated range of tokens.

De-allocation (`Y` is `¯1`)

If `Y` is `¯1`, `⎕TALLOC` releases a previously allocated range of tokens identified by the left-argument `X`. The result `R` is a shy `0`.

To succeed, this range must have previously been allocated, not freed by de-allocation, and must be inactive, that is, its tokens must not currently be in the token pool or in use by a `⎕TGET`. If not, `⎕TALLOC` will signal a `DOMAIN ERROR`.

A de-allocated range becomes free for subsequent re-allocation by `⎕TALLOC`.

Querying a description (`Y` is 0)

`Y` is 0, ⎕TALLOC returns a non-shy result `R` containing the description for a currently allocated range of tokens identified by the left-argument `X`.

If `X` does not represent a currently allocated range, ⎕TALLOC will signal a `DOMAIN ERROR`.

If `X` is omitted, the result `R` is a vector of 2-element vectors identifying the range and description of all currently allocated ranges.

Descriptions that were not defined are returned as empty character vectors.

Querying the Token Pool (`Y` is 2)

`Y` is 2, ⎕TALLOC returns a non-shy result `R` containing the list of tokens in the token pool that fall in the range specified by the left-argument `X`.

Examples

```
      ⎕←trg←⎕TALLOC 1 'cats'
1
      ⎕TALLOC 0
┌─────────┐
│ ┌─┬────┐ │
│ │1│cats│ │
│ └─┴────┘ │
└─────────┘
     ⎕TPUT trg+.1 .2 .3
     ⎕TPUT -trg+.9
     ⎕TPOOL
1.1 1.2 1.3 ¯1.9

     ⎕TGET trg+.1 .2 .3 .9

     1 ⎕TALLOC ¯1 ⍝ Try to de-allocate the range
DOMAIN ERROR
     1 ⎕TALLOC ¯1
       ^
     1 ⎕TALLOC 2  ⍝ Failed due to ¯1.9 token
¯1.9
     ⎕TGET ¯1.9   ⍝ Remove the inexhaustible ¯1.9 token
     1 ⎕TALLOC 2

     1 ⎕TALLOC ¯1 ⍝ De-allocation now works
```

6.4.153 Terminal Control (⎕ML)

⎕TC is a deprecated feature and is replaced by ⎕UCS (see note).

⎕TC is a simple three element vector. If ⎕ML<3 this is ordered as follows:

⎕TC[1]	Backspace
⎕TC[2]	Linefeed
⎕TC[3]	Newline

Note that ⎕TC≡⎕AV[1+⍳3] for ⎕ML<3 .

If ⎕ML≥3the order of the elements of ⎕TC is instead compatible with IBM's APL2:

⎕TC[1]	Backspace
⎕TC[2]	Newline
⎕TC[3]	Linefeed

Elements of ⎕TC beyond 3 are not defined but are reserved.

Note

With the introduction of ⎕UCS in Version 12.0, the use of ⎕TC is discouraged and it is strongly recommended that you generate control characters using ⎕UCS instead. This recommendation holds true even if you continue to use the Classic Edition.

Control Character	Old	New
Backspace	⎕TC[1]	⎕UCS 8
Linefeed	⎕TC[2] (⎕ML<3) ⎕TC[3] (⎕ML≥3)	⎕UCS 10
Newline	⎕TC[3] (⎕ML<3) ⎕TC[2] (⎕ML≥3)	⎕UCS 13

6.4.154 Thread Child Numbers R←⎕TCNUMS Y

Y must be a simple array of integers representing thread numbers.

The result R is a simple integer vector of the child threads of each thread of Y.

EXAMPLES

```
      ⎕TCNUMS 0
2 3

      ⎕TCNUMS 2 3
4 5 6 7 8 9
```

6.4.155 Get Tokens {R}←{X} ⎕TGET Y

Y must be a simple numeric scalar or vector that specifies one or more tokens, each with a specific non-zero token type, that are to be retrieved from the pool. Non-integer values in Y must fall within a range that has been allocated using ⎕TALLOC.

X is an optional time-out value in seconds.

Shy result R is a scalar or vector containing the values of the tokens of type Y that have been retrieved from the token pool.

Note that types of the tokens in the pool may be positive or negative, and the elements of Y may also be positive or negative.

A request (⍞TGET) for a *positive* token will be satisfied by the presence of a token in the pool with the same positive or negative type. If the pool token has a positive type, it will be removed from the pool. If the pool token has a negative type, it will remain in the pool. N*egatively* typed tokens will therefore satisfy an infinite number of requests for their positive equivalents. Note that a request for a positive token will remove one if it is present, before resorting to its negative equivalent

A request for a negative token type will only be satisfied by the presence of a negative token type in the pool, and that token will be removed.

If, when a thread calls ⍞TGET, the token pool satisfies **all** of the tokens specified by Y, the function returns immediately with a (shy) result that contains the values associated with the pool tokens. Otherwise, the function will block (wait) until **all** of the requested tokens are present or until a time-out (as specified by X) or a weak interrupt occurs.

For example, if the pool contains only tokens of type 2:

```
    ⍞TGET 2 4          ⍝ blocks waiting for a 4-token ...
```

The ⍞TGET operation is atomic in the sense that no tokens are taken from the pool until **all** of the requested types are present. While this last example is waiting for a 4-token, other threads could take any of the remaining 2-tokens.

Note also, that repeated items in the right argument are distinct. The following will block until there are at least 3 × 1.9-tokens in the pool:

```
    ⍞TGET 3/1.9        ⍝ wait for 3 × 2-tokens ...
```

The pool is administered on a first-in-first-out basis. This is significant only if tokens of the same type are given distinct values. For example:

```
    ⎕TGET ⎕TPOOL              ⍝ empty pool.

    'ABCDE'⎕TPUT¨2 2 3 2 3   ⍝ pool some tokens.

    ⊢⎕TGET 2 3
AC

    ⊢⎕TGET 2 3
BE
```

`R` is an empty numeric vector `⍬` (zilde) if a timeout or a weak interrupt occurs.

Beware - the following statement will wait forever and can only be terminated by an interrupt.

```
    ⎕TGET 0        ⍝ wait forever ...
```

Note too that if a thread waiting to `⎕TGET` tokens is `⎕TKILL`ed, the thread disappears without removing any tokens from the pool. Conversely, if a thread that has removed tokens from the pools is `⎕TKILL`ed, the tokens are not returned to the pool.

6.4.156 This Space R←⎕THIS

`⎕THIS` returns a reference to the current namespace, that is, to the space in which it is referenced.

If `NC9` is a reference to any object whose name-class is `9`, then:

```
      NC9≡NC9.⎕THIS
1
```

Examples

```
      ⎕THIS
#
      'X'⎕NS ''
      X.⎕THIS
#.X
     'F'⎕WC'Form'
     'F.B'⎕WC'Button'
      F.B.⎕THIS
#.F.B

      Polly←⎕NEW Parrot
      Polly.⎕THIS
#.[Parrot]
```

An Instance may use `⎕THIS` to obtain a reference to its own Class:

```
    Polly.(⊃⊃⎕CLASS ⎕THIS)
#.Parrot
```

or a function (such as a Constructor or Destructor) may identify or enumerate all other Instances of the same Class:

```
      Polly.(⍴⎕INSTANCES⊃⊃⎕CLASS ⎕THIS)
1
```

6.4.157 Current Thread Identity R←⎕TID

`R` is a simple integer scalar whose value is the number of the current thread.

Examples

```
      ⎕TID     ⍝ Base thread number
0

      ⍎&'⎕TID' ⍝ Thread number of async ⍎.
1
```

6.4.158 Kill Thread {R}←{X}⎕TKILL Y

Y must be a simple array of integers representing thread numbers to be terminated. X is a Boolean single, defaulting to 1, which indicates that all descendant threads should also be terminated.

The shy result R is a vector of the numbers of all threads that have been terminated.

The **base thread** 0 is always excluded from the cull.

EXAMPLES

```
      ⎕TKILL 0             ⍝ Kill background threads.

      ⎕TKILL ⎕TID          ⍝ Kill self and descendants.

      0 ⎕TKILL ⎕TID        ⍝ Kill self only.

      ⎕TKILL ⎕TCNUMS ⎕TID ⍝ Kill descendants.
```

6.4.159 Current Thread Name ⎕TNAME

The system variable ⎕TNAME reports and sets the name of the current APL thread. This name is used to identify the thread in the Tracer.

The default value of ⎕TNAME is an empty character vector. ⎕TNAME has workspace scope.

You may set ⎕TNAME to any valid character vector, but it is recommended that control characters (such as ⎕AV[⎕IO]) be avoided.

EXAMPLE

```
      ⎕TNAME←'Dylan'
      ⎕TNAME
Dylan
```

6.4.160 Thread Numbers R←⎕TNUMS

⎕TNUMS reports the numbers of all current threads.

R is a simple integer vector of the base thread and all its living descendants.

EXAMPLE

```
      ⎕TNUMS
0 2 4 5 6 3 7 8 9
```

6.4.161 Token Pool R←⎕TPOOL

R is a simple scalar or vector containing the token types for each of the tokens that are currently in the token pool.

The following (⎕ML←0) function returns a 2-column snapshot of the contents of the pool. It does this by removing and replacing all of the tokens, restoring the state of the pool exactly as before. Coding it as a single expression guarantees that snap is atomic and cannot disturb running threads.

```
      snap←{((⎕TGET ⍵)){(⍵↑⍺) ⋄ ⍺ ⎕TPUT¨⍵}⍵}

      snap ⎕TPOOL   snap ⎕TPOOL
┌─────┬──────────┐
│1.1  │1.1       │
├─────┼──────────┤
│1.2  │1.2       │
├─────┼──────────┤
│1.3  │1.3       │
├─────┼──────────┤
│¯1.9 │ no limit │
└─────┴──────────┘
```

See also: *Section 6.4.152*.

6.4.162 Put Tokens {R}←{X} ⎕TPUT Y

Y must be a simple numeric scalar or vector of non-zero token types. Non-integer values in Y must fall within a range that has been allocated using ⎕TALLOC.

X is an optional array of values to be stored in each of the tokens specified by Y.

Shy result R is a vector of thread numbers (if any) unblocked by the ⎕TPUT.

Examples

```
      ⎕TPUT 2 3 2        ⍝ put a 2-token, a 3-token and
                           another 2-token into the pool.

      88 ⎕TPUT 2         ⍝ put another 2-token into the pool
                           this token has the value 88.

      'Hello'⎕TPUT ¯1.9  ⍝ put a ¯1.9-token into the pool
                           with the value 'Hello'.
```

If `X` is omitted, the *value* associated with each of the tokens added to the pool is the same as its *type*.

Note that you cannot put a 0-token into the pool; 0-s are removed from `Y`.

6.4.163 Set Trace {R}←X ⎕TRACE Y

`Y` must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. `X` must be a simple non-negative integer scalar or vector.

`X` identifies the numbers of lines in the function or operator named by `Y` on which a trace control is to be placed. Numbers outside the range of line numbers in the function or operator (other than 0) are ignored. The number 0 indicates that a trace control is to be placed immediately prior to exit from the function or operator. The value of `X` is independent of `⎕IO`.

`R` is a simple integer vector of non-negative elements indicating the lines in the function or operator on which a trace control has been placed.

Example

```
      +(0,⍳10) ⎕TRACE'FOO'
0 1
```

Existing trace controls in the function or operator named by `Y` are cancelled before new trace controls are set:

```
      + 1 ⎕TRACE'FOO'
1
```

All trace controls may be cancelled by giving `X` an empty vector:

```
      ⍴⍬ ⎕TRACE 'FOO'
0
```

Attempts to set trace controls in a locked function or operator are ignored.

```
      ⎕LOCK 'FOO'
      +1 ⎕TRACE 'FOO'
```

The effect of trace controls when a function or operator is invoked is to display the result of each complete expression for lines with trace controls as they are executed, and the result of the function if trace control 0 is set. If a line contains expressions separated by ⋄, the result of each complete expression is displayed for that line after execution.

The result of a complete expression is displayed even where the result would normally be suppressed. In particular:

- the result of a branch statement is displayed;
- the result (*pass-through value*) of assignment is displayed;
- the result of a function whose result would normally be suppressed is displayed;

For each traced line, the output from ⎕TRACE is displayed as a two element vector, the first element of which contains the function or operator name and line number, and the second element of which takes one of two forms.

- The result of the line, displayed as in standard output.
- → followed by a line number.

EXAMPLE

```
      ⎕VR 'DSL'
     ∇ R←DSL SKIP;A;B;C;D
[1]    A←2×3+4
[2]    B←(2 3⍴'ABCDEF')A
[3]    →NEXT×⍳SKIP
[4]    'SKIPPED LINE'
[5]   NEXT:C←'one' ⋄ D←'two'
[6]   END:R←C D
     ∇

      (0,⍳6) ⎕TRACE 'DSL'

      DSL 1
 DSL[1]  14
 DSL[2]   ABC   14
          DEF
 DSL[3]  →5
 DSL[5]  one
 DSL[5]  two
 DSL[6]   one   two
 DSL[0]   one   two
 one  two
```

6.4.164 Query Trace R←⎕TRACE Y

Y must be a simple character scalar or vector which is taken to be the name of a visible defined function or operator. R is a simple non-negative integer vector of the line numbers of the function or operator named by Y on which trace controls are set, shown in ascending order. The value 0 in R indicates that a trace control is set to display the result of the function or operator immediately prior to exit.

EXAMPLE

```
      ⎕TRACE'DSL'
0 1 2 3 4 5 6
```

6.4.165 Trap Event ⎕TRAP

This is a non-simple vector. An item of ⎕TRAP specifies an action to be taken when one of a set of events occurs. ⎕TRAP has workspace scope.

An item of ⎕TRAP is a 2 or 3 element vector whose items are simple scalars or vectors in the following order:

1. an integer vector whose value is one or more event codes selected from the list in the Figure on the following two pages.
2. a character scalar whose value is an action code selected from the letters `C`, `E`, `N` or `S`.
3. if element 2 is the letter `C` or `E`, this item is a character vector forming a valid APL expression or series of expressions separated by ⋄. Otherwise, this element is omitted.

An EVENT may be an APL execution error, an interrupt by the user or the system, a control interrupt caused by the ⎕STOP system function, or an event generated by the ⎕SIGNAL system function.

When an event occurs, the system searches for a trap definition for that event. The most local ⎕TRAP value is searched first, followed by successive shadowed values of ⎕TRAP, and finally the global ⎕TRAP value. Separate actions defined in a single ⎕TRAP value are searched from **left to right**. If a trap definition for the event is found, the defined action is taken. Otherwise, the normal system action is followed.

The ACTION code identifies the nature of the action to be taken when an associated event occurs. Permitted codes are interpreted as follows:

Table 6-4: *Trappable Event Codes*

`C`	Cutback	The state indicator is 'cut back' to the environment in which the ⎕TRAP is locally defined (or to immediate execution level). The APL expression in element 3 of the same ⎕TRAP item is then executed.
`E`	Execute	The APL expression in element 3 of the same ⎕TRAP item is executed in the environment in which the event occurred.
`N`	Next	The event is excluded from the current ⎕TRAP definition. The search will continue through further localised definitions of ⎕TRAP .
`S`	Stop	Stops the search and causes the normal APL action to be taken in the environment in which the event occurred.

Code	Event
0	Any event in range 1-999
1	WS FULL
2	SYNTAX ERROR
3	INDEX ERROR
4	RANK ERROR
5	LENGTH ERROR
6	VALUE ERROR
7	FORMAT ERROR
10	LIMIT ERROR
11	DOMAIN ERROR
12	HOLD ERROR
16	NONCE ERROR
18	FILE TIE ERROR
19	FILE ACCESS ERROR
20	FILE INDEX ERROR
21	FILE FULL
22	FILE NAME ERROR
23	FILE DAMAGED
24	FILE TIED
25	FILE TIED REMOTELY
26	FILE SYSTEM ERROR
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
31	FILE TIE QUOTA USED UP
32	FILE NAME QUOTA USED UP
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE

Code	Event
38	FILE COMPONENT DAMAGED
52	FIELD CONTENTS RANK ERROR
53	FIELD CONTENTS TOO MANY COLUMNS
54	FIELD POSITION ERROR
55	FIELD SIZE ERROR
56	FIELD CONTENTS/TYPE MISMATCH
57	FIELD TYPE/BEHAVIOUR UNRECOGNISED
58	FIELD ATTRIBUTES RANK ERROR
59	FIELD ATTRIBUTES LENGTH ERROR
60	FULL-SCREEN ERROR
61	KEY CODE UNRECOGNISED
62	KEY CODE RANK ERROR
63	KEY CODE TYPE ERROR
70	FORMAT FILE ACCESS ERROR
71	FORMAT FILE ERROR
72	NO PIPES
76	PROCESSOR TABLE FULL
84	TRAP ERROR
90	EXCEPTION
92	TRANSLATION ERROR
200-499	Reserved for distributed auxiliary processors
500-999	User-defined events
1000	Any event in range 1001-1008
1001	Stop vector

Code	Event
1002	Weak interrupt
1003	INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE (Dyalog APL/X, Dyalog APL/W)
1008	DEADLOCK

See *Programming: Trap* for an alternative 'control structured' error trapping mechanism.

Examples

```
      ⎕TRAP←⊂(3 4 5) 'E' 'ERROR' ⋄ ⍴⎕TRAP
1

      ⎕TRAP
  3 4 5 E ERROR
```

Items may be specified as scalars. If there is only a single trap definition, it need not be enclosed. However, the value of ⎕TRAP will be rigorously correct:

```
      ⎕TRAP←11 'E' '→LAB'

      ⎕TRAP
  11 E →ERR

      ⍴⎕TRAP
1
```

The value of ⎕TRAP in a clear workspace is an empty vector whose prototype is

`0⍴⊂⍬ '' ''`. A convenient way of cancelling a ⎕TRAP definition is:

```
      ⎕TRAP←0⍴⎕TRAP
```

Event codes 0 and 1000 allow all events in the respective ranges 1-999 and 1000-1006 to be trapped. Specific event codes may be excluded by the N action (which must precede the general event action):

```
      ⎕TRAP←(1 'N')(0 'E' '→GENERR')
```

The 'stop' action is a useful mechanism for cancelling trap definitions during development of applications.

The 'cut-back' action is useful for returning control to a known point in the application system when errors occur. The following example shows a function that selects and executes an option with a general trap to return control to the function when an untrapped event occurs:

```
     ∇ SELECT;OPT;⎕TRAP
[1]   ⍝ Option selection and execution
[2]   ⍝ A general cut-back trap
[3]    ⎕TRAP←(0 1000)'C' '→ERR'
[4]   INP:⍞←'OPTION : ' ⋄ OPT←(OPT≠' ')/OPT←9↓⍞
[5]    →EX⍴⍨(⊂OPT)∈Options ⋄ 'INVALID OPTION' ⋄ →INP
[6]   EX:⍎OPT ⋄ →INP
[7]   ERR:ERROR⍙ACTION ⋄ →INP
[8]   END:
     ∇
```

User-defined events may be signalled through the ⎕SIGNAL system function. A user-defined event (in the range 500-999) may be trapped explicitly or implicitly by the event code 0.

EXAMPLE

```
      ⎕TRAP←500 'E' '''USER EVENT 500 - TRAPPED'''

      ⎕SIGNAL 500
USER EVENT 500 - TRAPPED
```

6.4.166 Token Requests R←⎕TREQ Y

Y is a simple scalar or vector of thread numbers.

R is a vector containing the concatenated token requests for all the threads specified in Y. This is effectively the result of catenating all of the right arguments together for all threads in Y that are currently executing ⎕TGET.

EXAMPLE

```
    ⎕TREQ ⎕TNUMS    ⍝ tokens required by all threads.
```

6.4.167 Timestamp R←⎕TS

This is a seven element vector which identifies the clock time set on the particular installation as follows:

⎕TS[1]	Year
⎕TS[2]	Month
⎕TS[3]	Day
⎕TS[4]	Hour
⎕TS[5]	Minute
⎕TS[6]	Second
⎕TS[7]	Millisecond

EXAMPLE

```
      ⎕TS
1989 7 11 10 42 59 123
```

Note that on some systems, where time is maintained only to the nearest second, a zero is returned for the seventh (millisecond) field.

6.4.168 Wait for Threads to Terminate R←⎕TSYNC Y

Y must be a simple array of thread numbers.

If Y is a simple scalar, R is an array, the result (if any) of the thread.

If Y is a simple non-scalar, R has the same shape as Y, and result is an array of enclosed thread results.

Examples

```
      dup←{⍵ ⍵}          ⍝ Duplicate

      ⎕←dup&88           ⍝ Show thread number
11
88 88

      ⎕TSYNC dup&88      ⍝ Wait for result
88 88

      ⎕TSYNC,dup&88
 88 88

      ⎕TSYNC dup&1 2 3
 1 2 3  1 2 3

      ⎕TSYNC dup&¨1 2 3
 1 1  2 2  3 3
```

Deadlock

The interpreter detects a potential deadlock if a number of threads wait for each other in a cyclic dependency. In this case, the thread that attempts to cause the deadlock issues error number `1008: DEADLOCK`.

```
     ⎕TSYNC ⎕TID       ⍝ Wait for self
DEADLOCK
      ⎕TSYNC ⎕TID
       ^

      ⎕EN
1008
```

Potential Value Error

If any item of `Y` does not correspond to the thread number of an active thread, or if any subject thread terminates without returning a result, then `⎕TSYNC` does not return a result. This means that, if the calling context of the `⎕TSYNC` requires a result, for

example: `rslt←⎕TSYNC tnums`, a `VALUE ERROR` will be generated. This situation can occur if threads have completed before `⎕TSYNC` is called.

```
      ⎕←÷&4           ⍝ thread (3) runs and terminates.
3
0.25
      ⎕TSYNC 3        ⍝ no result required: no prob
      ⎕←⎕TSYNC 3      ⍝ context requires result
VALUE ERROR

      ⎕←⎕TSYNC {}&0   ⍝ non-result-returning fn: no result.
VALUE ERROR
```

Coding would normally avoid such an inconvenient `VALUE ERROR` either by arranging that the thread-spawning and `⎕TSYNC` were on the same line:

```
      rslt ← ⎕TSYNC myfn&¨ argvec
```

or

```
      tnums←myfn&¨ argvec ⋄ rslt←⎕TSYNC tnums
```

or by error-trapping the `VALUE ERROR`.

6.4.169 Unicode Convert `R←{X} ⎕UCS Y`

`⎕UCS` converts (Unicode) characters into integers and vice versa.

The optional left argument `X` is a character vector containing the name of a variable-length Unicode encoding scheme which must be one of:

- 'UTF-8'
- 'UTF-16'
- 'UTF-32'

If not, a `DOMAIN ERROR` is issued.

If `X` is omitted, `Y` is a simple character or integer array, and the result `R` is a simple integer or character array with the same rank and shape as `Y`.

If `X` is specified, `Y` must be a simple character or integer vector, and the result `R` is a simple integer or character vector.

Monadic ⎕UCS

Used monadically, ⎕UCS simply converts characters to Unicode code points and vice-versa.

With a few exceptions, the first 256 Unicode code points correspond to the ANSI character set.

```
      ⎕UCS 'Hello World'
72 101 108 108 111 32 87 111 114 108 100

      ⎕UCS 2 11⍴72 101 108 108 111 32 87 111 114 108 100
Hello World
Hello World
```

The code points for the Greek alphabet are situated in the 900's:

```
      ⎕UCS 'καλημέρα'
954 945 955 951 956 941 961 945
```

Unicode also contains the APL character set. For example:

```
      ⎕UCS 123 40 43 47 9077 41 247 9076 9077 125
{(+/⍵)÷⍴⍵}
```

Dyadic ⎕UCS

Dyadic ⎕UCS is used to translate between Unicode characters and one of three standard variable-length Unicode encoding schemes, UTF-8, UTF-16 and UTF-32. These represent a Unicode character string as a vector of 1-byte (UTF-8), 2-byte (UTF-16) and 4-byte (UTF-32) signed integer values respectively.

```
      'UTF-8' ⎕UCS 'ABC'
65 66 67
      'UTF-8' ⎕UCS 'ABCÆØÅ'
65 66 67 195 134 195 152 195 133
      'UTF-8' ⎕UCS 195 134, 195 152, 195 133
ÆØÅ
      'UTF-8' ⎕UCS 'γεια σου'
206 179 206 181 206 185 206 177 32 207 131 206 191 207 133
      'UTF-16' ⎕UCS 'γεια σου'
947 949 953 945 32 963 959 965
      'UTF-32' ⎕UCS 'γεια σου'
947 949 953 945 32 963 959 965
```

Because integers are *signed*, numbers greater than 127 will be represented as 2-byte integers (type 163), and are thus not suitable for writing directly to a native file. To write the above data to file, the easiest solution is to use ⎕UCS to convert the data to 1-byte characters and append this data to the file:

```
      (⎕UCS 'UTF-8' ⎕UCS 'ABCÆØÅ') ⎕NAPPEND tn
```

Note regarding UTF-16: For most characters in the first plane of Unicode (0000-FFFF), UTF-16 and UCS-2 are identical. However, UTF-16 has the potential to encode all Unicode characters, by using more than 2 bytes for characters outside plane 1.

```
      'UTF-16' ⎕UCS 'ABCÆØÅ𝄞𝄵'
65 66 67 198 216 197 9042 9035
      ⎕←unihan←⎕UCS (2×2*16)+⍳3 ⍝ x20001-x20003

      'UTF-16' ⎕UCS unihan
55360 56321 55360 56322 55360 56323
```

Translation Error

⎕UCS will generate a DOMAIN ERROR if the argument cannot be converted. Additionally, in the Classic Edition, a TRANSLATION ERROR is generated if the result is not in ⎕AV or the numeric argument is not in ⎕AVU.

6.4.170 Using (Microsoft .NET Search Path) ⎕USING

⎕USING specifies a list of Microsoft .NET Namespaces that are to be searched for a reference to a .NET class. ⎕USING has Namespace scope.

⎕USING is a vector of character vectors each element of which contains 1 or 2 comma-delimited strings. The first string specifies the name of a .NET namespace; the second specifies the *pathname* of an assembly file. This may be a full pathname or a relative one, but must include the file extension (`.dll`). If just the file name is specified, it is assumed to be located in the standard .NET Framework directory that was specified when the .NET Framework was installed (for example, `C:\Windows\Microsoft.NET\Framework64\v4.0.30319`)

It is convenient to treat .NET namespaces and assemblies in pairs. For example:

```
⎕USING←'System,mscorlib.dll'
  ⎕USING,←⊂'System.Windows.Forms,System.Windows.Forms.dll'  ⎕USING,←⊂'Sy
stem.Drawing,System.Drawing.dll'
```

Note that because Dyalog APL automatically loads `mscorlib.dll` (which contains the most commonly used classes in the `System` Namespace), it is not actually necessary to specify it explicitly in ⎕USING.

⎕USING has Namespace scope, that is, each Dyalog namespace, class or instance has its own value of ⎕USING that is initially inherited from its parent space but which may be separately modified. ⎕USING may also be localised in a function header, so that different functions can declare different search paths for .NET namespaces/assemblies.

If ⎕USING is empty (`⎕USING←0⍴⊂''`), APL will not search for .NET classes in order to resolve names which would otherwise give a `VALUE ERROR`.

Assigning a simple character vector to ⎕USING is equivalent to setting it to the enclose of that vector. The statement (`⎕USING←''`) does not empty ⎕USING, it sets it to a single empty element, which gives access to `mscorlib.dll` and the Bridge DLL without a namespace prefix.

Notes

- The value of ⎕USING is only used when an object is instantiated. Changing the value of ⎕USING has no effect on objects that have already been instantiated in the active workspace.
- When a workspace containing .Net objects is saved, the names of the Net objects are saved with it but they are not automatically re-instantiated when the workspace is loaded or copied. A reference to such an orphaned object will report `(NULL)`.
- Some functionality might work with .NET Framework or .NET but not both, for example, SharpPlot requires the .NET Framework and does not work with .NET itself.

Examples

```
  ⎕USING←'System'
  ]display ⎕USING
.→---------.
| .→-----. |
| |System| |
| '------' |
'∈---------'

⎕USING,←⊂'System.Windows.Forms,System.Windows.Forms.dll'
⎕USING,←⊂'System.Drawing,System.Drawing.dll'
```

An Assembly may contain top-level classes which are not packaged into .NET Namespaces. In this case, you omit the Namespace name. For example:

```
  ⎕USING←,⊂',.\LoanService.dll'
```

6.4.171 Value Get R←{X}⎕VGET Y

⎕VGET enables values to be read for names in a source namespace or source namespaces. Optionally, a fallback value can be used if the name requested is undefined.

Y specifies the names. It must be one of the following:

- a matrix of names or a matrix of names and a value vector – see Case 1: Name Matrix.
- a vector of names or name-value pairs – see Case 2: Vector of Names.
- a vector of nameclasses – see Case 3: Nameclasses.

All specified names must be either undefined, or have an array value in the source namespace(s). If Y specifies a matrix or a vector of names, fallback values to use in cases where a name has no value can also be specified to prevent a VALUE ERROR from being generated.

If specified, X must be an array that identifies one or more source namespaces. This means that X must be one of:

- a simple character scalar or vector identifying the name of a namespace.
- a reference to a namespace.

- an array in which each item is one of the above. If `X` refers to multiple namespaces, then `⎕VGET` processes each item of `X` in ravel order, using the entire right argument `Y`; this is equivalent to `X ⎕VGET¨⊂Y`.

When `X` is an empty array, the prototype of the empty result `R` depends on the fallback values, if specified. If the prototype of `X` is an instance of a class that can be instantiated, such as an instance of a class with a niladic or no constructors, then a new instance of the class is made and the prototype of the result is determined by the values found within the new instance.

The namespace(s) referenced must already exist, or a `VALUE ERROR` is generated.

If `X` is not specified, the source namespace is the current namespace.

The result `R` depends on the format of `Y`.

See also *Section 6.4.173*.

Case 1: Name Matrix

Names are specified as rows in a character matrix. `Y` must be either:

- a character matrix, where each row is a name.
- a two element vector, where the first item is a character matrix of names and the second item is a specification of fallback values.

The fallback values must be one of the following:

- a vector with as many elements as there are names in the matrix.
- a scalar value that is the fallback value for all names.

The result `R` is a vector of the values from the corresponding names or fallback values.

EXAMPLES

Multiple names without fallback:

```
      (name1 name2 name3 longer_name)←(1 2 3) () 'APL' 42
      names←↑'name1' 'name2' 'name3' 'longer_name'
      names
name1
name2
name3
longer_name
      ⎕VGET names
 1 2 3  #.[Namespace]  APL  42
```

Multiple names with a different fallback for each name:

```
      name2←100
      names←↑'name1' 'name2' 'name3'
      names
name1
name2
name3
      defaults←1 2 3
      ⎕VGET names defaults
1 100 3
```

Multiple names with the same fallback for all of them:

```
      persons←(
        (name: 'Jack' ⋄ age: 36)
        (name: 'Peter' ⋄ email: 'peter@example.com')
        (phone: 12345678 ⋄ email: 'susan@example.com')
      )
      persons
 #.[Namespace]  #.[Namespace]  #.[Namespace]
      names←↑'name' 'age' 'email' 'phone'
      names
name
age
email
phone


⍝ Lookup information about each person, with a default for missing data
      ↑persons ⎕VGET names (⊂'<no data>')
 Jack              36  <no data>          <no data>
 Peter      <no data>  peter@example.com  <no data>
 <no data>  <no data>  susan@example.com   12345678
```

Case 2: Vector of Names

Names are specified as character vectors or scalars. `Y` must be one of the following:

- a single name: `R` is the value of that name in the source namespace.
- a single enclosed name: `R` is also the value of the name, but enclosed.
- a single enclosed name-value pair, which is a two-element vector consisting of a character vector name and a fallback value for that name: `R` is the value of the name, or the fallback value in case the name has no value.
- a nested vector where each item is either a name, or a name value pair: `R` is a vector with the same length as `Y`, with the values from the corresponding names, or fallback values.

Examples

Single name:

```
      (ns1 ns2)←()()
      (ns1 ns2).name1←'ABC' 'DEF'

      ns1 ⎕VGET 'name1'
ABC
      ns1 'ns2' ⎕VGET 'name1'
 ABC  DEF
```

Single name enclosed:

```
      name1←'APL'
      ⎕VGET ⊂'name1'
 APL
      ≢⍴⎕VGET ⊂'name1'
0
```

Multiple names without fallback:

```
      (name1 name2 name3)←(1 2 3) () 'APL'

      ⎕VGET 'name1' 'name2' 'name3'
 1 2 3  #.[Namespace]  APL
```

Single name with fallback:

```
      ns←⍬
      ns ⎕VGET ⊂'name1' 'default'
default
```

Multiple names with fallback for some:

```
      (name1 name2)←'APL' 123
      ⎕VGET ('name1' 1) 'name2' ('name3' 3)
 APL  123 3
      ⎕EX'name1'
      ⎕VGET ('name1' 1) 'name2' ('name3' 3)
1 123 3
```

Multiple names with a different fallback for each of them:

```
      name2←100
      ⎕VGET ('name1' 1) ('name2' 2) ('name3' 3)
1 100 3
```

See Case 1: Name Matrix for an example of multiple names with the same fallback value for all of them.

Case 3: Nameclasses

`Y` must be a numeric scalar or vector, where each item is a nameclass (see *Section 6.4.85*).

If any of the numbers in `Y` are negative, the result `R` is a vector of name-value pairs, one for each existing name in the source namespace with a nameclass from `Y`. Otherwise, `R` is a 2-element nested vector, where the first element is a character matrix of names and the second element is a vector of values. In both cases, `R` is suitable as an argument for *Section 6.4.171* and *Section 6.4.173*.

Section 6.4.85 always reports the names of fields in a class as having nameclass `2` (`2.2` with the sub-class), even when the name has no value (might expect `0`) or the field is a namespace reference (might expect `9`). *Section 6.4.171* with a right argument of `2` will only include fields that have values that are not references, while a right argument of `9` will include fields that are references. With a right argument of `2.2`, *Section 6.4.171* will return all fields that are not undefined.

EXAMPLES

Name value pairs:

```
      ns←⍬
      ns.ns1←⎕SE
      ns.ns2←#
      ns.a1←1 2 3
      ns.a2←'APL'
      ns ⎕VGET ¯2
 a1  1 2 3    a2  APL
      ns ⎕VGET ¯2 ¯9
 a1  1 2 3    a2  APL    ns1  ⎕SE    ns2  #
      ↑⎕SE ⎕VGET ¯8
onClose            0
onCreate            Dyalog.Callbacks.SECreate
onFontOK           0
onFontCancel       0
onWorkspaceLoaded   Dyalog.Callbacks.WSLoaded
onSessionPrint     0
onSessionTrace     0

      ÷0
DOMAIN ERROR: Divide by zero
      ÷0
      ^
      ↑⎕DMX ⎕VGET ¯2
Category                                        General
DM                          DOMAIN ERROR        ÷0        ^
EM                                             DOMAIN ERROR
EN                                                      11
ENX                                                      1
HelpURL           https://help.dyalog.com/dmx/20.0/General/1
InternalLocation                                scalm.cpp  356
Message                                         Divide by zero
OSError                                                 0 0
Vendor                                                  Dyalog
```

Name matrix and value vector:

```
      ]box on
Was OFF
      (name1 name2 name3)←'APL' (1 2 3) ⎕SE
      ⎕VGET 2 9
┌─────┬──────────────────┐
│name1│┌───┬─────┬─────┐│
│name2│⎕APL⎕1 2 3⎕ ⎕SE ⎕│
│name3│└───┴─────┴─────┘│
└─────┴──────────────────┘
```

6.4.172 Vector Representation R←⎕VR Y

Y must be a simple character scalar or vector which represents the name of a function or defined operator.

If Y is the name of a defined function or defined operator, R is a simple character vector containing a character representation of the function or operator with each line except the last terminated by the newline character (⎕UCS ⎕AVU[4]).

Its display form is as follows:

1. the header line starts at column 8 with the ∇ symbol in column 6,
2. the line number for each line of the function starts in column 1,
3. the statement contained in each line starts at column 8 except for labelled lines or lines beginning with ⍝ which start at column 7,
4. the header line and statements contain no redundant blanks beyond column 7 except that the ⋄ separator is surrounded by single blanks, control structure indentation is preserved and comments retain embedded blanks as originally defined,
5. the last line shows only the ∇ character in column 6.

If Y is the name of a variable, a locked function or operator, an external function, or is undefined, R is an empty vector.

EXAMPLE

```
      ρV←⎕VR'PLUS'
128

      V
     ∇ R←{A}PLUS B
[1]   ⍝ MONADIC OR DYADIC +
[2]    →DYADICρ⍨2=⎕NC'A' ⋄ R←B ⋄ →END
[3]   DYADIC:R←A+B ⋄ →END
[4]   END:
     ∇
```

The definition of ⎕VR has been extended to names assigned to functions by specification (←), and to local names of functions used as operands to defined operators. In these cases, the result of ⎕VR is identical to that of ⎕CR except that the representation of defined functions and operators is as described above.

EXAMPLE

```
      AVG←MEAN∘,

      +F←⎕VR'AVG'
      ∇ R←MEAN X    ⍝ Arithmetic mean
[1]     R←(+/X)÷ρX
     ∇ ∘,

      ρF
3

      ]display F
┌→──────────────────────────────────────────┐
│ ┌→──────────────────────────────┐         │
│ │     ∇ R←MEAN X    ⍝ Arithmetic mean│ ∘ , │
│ │[1]    R←(+/X)÷ρX               │ - - │
│ │     ∇                          │     │
│ └~──────────────────────────────┘         │
└∊──────────────────────────────────────────┘
```

6.4.173 Value Set {R}←{X}⎕VSET Y

⎕VSET enables values to be set for names in a target namespace or target namespaces.

`Y` specifies the names and the values to set for them. It must be one of the following:

- a nested vector or scalar, where each element is a name-value pair. The name must be a simple character vector.
- a two-element nested array, where the first element is a matrix of names and the second element is a vector or scalar of value(s). If multiple names are specified and the value is a scalar, the same value is used for all names.

All names must have nameclass 0, 2, 8 or 9 in the target namespace(s). For more information on nameclasses, see *Section 6.4.85*.

If specified, `X` must be an array that references one or more namespaces. This means that `X` must be one of:

- a simple character scalar or vector identifying the name of a namespace.
- a reference to a namespace.
- an array in which each item is one of the above. If `X` refers to multiple namespaces, then `⎕VSET` processes each item of `X` in ravel order, using the entire right argument `Y`; this is equivalent to `X ⎕VSET¨⊂Y`.

The namespace(s) referenced must already exist, or a `VALUE ERROR` is generated.

If `X` is not specified, the target namespace is the current namespace.

The result `R` is a shy reference to the target namespace(s).

See also *Section 6.4.171*.

Examples

Name value pairs:

```
      ⎕VSET ('name1' 123) ('name2' (1 2 'hello'))
      name1
123
      name2
1 2  hello

      (ns1 ns2 ns3)←()()()
      ns1 'ns2' ns3 ⎕VSET ('X1' 'X value') ('Y1' 'Y value')
      (ns1 ns2 ns3).(X1 Y1)
  X value  Y value    X value  Y value    X value  Y value
```

Name matrix and value vector:

```
      names←↑'name1' 'name2' 'name3'
      names
name1
name2
name3
      values←1 2 3
      ⎕VSET names values
      name1 name2 name3
1 2 3
```

Single name-value pair:

```
      ⎕VSET ⊂'name1' (10 20 30)
      ⎕VSET ,⊂'name2' (40 50 60)
      name1
10 20 30
      name2
40 50 60
```

Multiple names, with a single value:

```
      names←↑'name1' 'name2' 'name3'
      value←'APL'

      ⎕VSET names (⊂value)
      name1 name2 name3
 APL  APL  APL
```

Variant Option: Trigger

The `Trigger` variant option specifies whether any *Programming: Triggers* should be run for the modified variables in the target namespace that have triggers attached. The value must be a Boolean scalar. The default is `1`, meaning that triggers are run.

EXAMPLE

```
      ⎕VR 'trigger'
     ∇trigger arg
[1]   :Implements Trigger name1,name3
[2]   ⎕←'Running trigger for: ',arg.Name
     ∇

      ⍝ name1 has a trigger, name2 does not
      name1←1
Running trigger for: name1
      name2←2

      ⍝ With the trigger option disabled, triggers are not run
      ⎕VSET⍠'Trigger' 0⊢('name1' 1) ('name2' 2) ('name3' 3)

      ⍝ With the trigger option enabled, triggers are run
      ⎕VSET⍠'Trigger' 1⊢('name1' 1) ('name2' 2) ('name3' 3)
Running trigger for: name1
Running trigger for: name3

      ⍝ Without the trigger option, triggers are run (the default)
      ⎕VSET ('name1' 1) ('name2' 2) ('name3' 3)
Running trigger for: name1
Running trigger for: name3
```

6.4.174 Verify & Fix Input `R←{X}⎕VFI Y`

`Y` must be a simple character scalar or vector. `X` is optional. If present, `X` must be a simple character scalar or vector. `R` is a nested vector of length two whose first item is a simple logical vector and whose second item is a simple numeric vector of the same length as the first item of `R`.

`Y` is the character representation of a series of numeric constants. If `X` is omitted, adjacent numeric strings are separated by one or more blanks. Leading and trailing blanks and separating blanks in excess of one are redundant and ignored. If `X` is present, `X` specifies one or more alternative separating characters. Blanks in leading and trailing positions in `Y` and between numeric strings separated also by the character(s) in `X` are redundant and ignored. Leading, trailing and adjacent occurrences of the character(s) in `X` are not redundant. The character 0 is implied in `Y` before a leading character, after a trailing character, and between each adjacent pair of characters specified by `X`.

The length of the items of R is the same as the number of identifiable strings (or implied strings) in Y separated by blank or the value of X. An element of the first item of R is 1 where the corresponding string in Y is a valid numeric representation, or 0 otherwise. An element of the second item of R is the numeric value of the corresponding string in Y if it is a valid numeric representation, or 0 otherwise.

Examples

```
      ⎕VFI '2 -2 ¯2'
 1 0 1  2 0 ¯2

      ⎕VFI '12.1 1E1 1A1 ¯10'
 1 1 0 1  12.1 10 0 ¯10

      ⊃(//⎕VFI'12.1 1E1 1A1 ¯10')
12.1 10 ¯10

      ','⎕VFI'3.9,2.4,,76,'
 1 1 1 1 1  3.9 2.4 0 76 0

      '⋄'⎕VFI'1 ⋄ 2 3 ⋄ 4 '
 1 0 1  1 0 4
      (⍬ ⍬)≡⎕VFI''
1
```

6.4.175 Workspace Available R←⎕WA

This is a simple integer scalar. It identifies the total available space in the active workspace area given as the number of bytes it could hold.

A side effect of using ⎕WA is an internal reorganisation of the workspace and process memory, as follows:

1. Any un-referenced memory is discarded. This process, known as *garbage collection*, is required because whole cycles of refs can become un-referenced.
2. Numeric arrays are *demoted* to their tightest form. For example, a simple numeric array that happens to contain only values 0 or 1, is demoted or *squeezed* to have a ⎕DR type of 11 (Boolean).
3. All remaining used memory blocks are copied to the low-address end of the workspace, leaving a single free block at the high-address end. This process is known as *compaction*.
4. All memory allocated is returned to the Operating System except the space required for the compacted workspace, plus a working overhead based on the

configured maximum workspace size (**MAXWS**). If the compacted workspace occupies more than 1/16 of **MAXWS** then the overhead is 1/16 of MAXWS, otherwise it is 1/64 of **MAXWS**. On a Windows system, you can see the process size changing by using Task Manager.

EXAMPLE

```
      ⎕WA
261412
```

See also: *Section 4.4.9.28*

6.4.176 Window Create Object {R}←{X}⎕WC Y

Windows only.

This system function creates a GUI **object**. `Y` is either a vector which specifies **properties** that determine the new object's appearance and behaviour, or a ref to or the `⎕OR` of a GUI object that exists or previously existed. `X` is a character vector which specifies the name of the new object, and its position in the object hierarchy.

If `X` is omitted, `⎕WC` attaches a GUI component to the current namespace, retaining any functions, variables and other namespaces that it may contain. Monadic `⎕WC` is discussed in detail at the end of this section.

If `Y` is a nested vector each element specifies a property. The `Type` property (which specifies the class of the object) **must** be specified. Most other properties take default values and need not be explicitly stated. Properties (including `Type`) may be declared either positionally or with a keyword followed by a value. Note that `Type` must always be the first property specified. Properties are specified positionally by placing their values in `Y` in the order prescribed for an object of that type.

If `Y` is a ref or the result of `⎕OR`, the new object is a complete copy of the other, including any child objects, namespaces, functions and variables that it contained at that time.

The shy result `R` is the full name (starting `#.` or `⎕SE.`) of the namespace `X`.

An object's name is specified by giving its full pathname in the object hierarchy. At the top of the hierarchy is the `Root` object whose name is `"."`. Below `"."` there may be one or more "top-level" objects. The names of these objects follow the standard rules for other APL objects as described in *Programming: Names*

Names for sub-objects follow the same rules except that the character "`.`" is used as a delimiter to indicate parent/child relationships.

The following are examples of legal and illegal names:

Legal	Illegal
`FORM1`	`FORM 1`
`form_23`	`form#1`
`Form1.Gp`	`11_Form`
`F1.g2.b34`	`Form+1`

If `X` refers to the name of an APL variable, label, function, or operator, a `DOMAIN ERROR` is reported. If `X` refers to the name of an existing GUI object or namespace, the existing one is replaced by the new one. The effect is the same as if it were deleted first.

If `Y` refers to a non-existent property, or to a property that is not defined for the type of object `X`, a `DOMAIN ERROR` is reported. A `DOMAIN ERROR` is also reported if a value is given that is inconsistent with the corresponding property. This can occur for example, if `Y` specifies values positionally and in the wrong order.

A "top-level" object created by `⎕WC` whose name is localised in a function/operator header, is deleted on exit from the function/operator. All objects, including sub-objects, can be deleted using `⎕EX`.

GUI objects are named **relative** to the current namespace, so the following examples are equivalent:

```
      'F1.B1' ⎕WC 'Button'
```

is equivalent to :

```
      )CS F1
#.F1
      'B1' ⎕WC 'Button'
      )CS
#
```

is equivalent to :

```
      'B1' F1.⎕WC 'Button'
```

Examples

```
⍝ Create a default Form called F1

      'F1' ⎕WC 'Form'

⍝ Create a Form with specified properties (by position)
⍝   Caption = "My Application"  (Title)
⍝   Posn    = 10 30  (10% down, 30% across)
⍝   Size    = 80 60  (80% high, 60% wide)

      'F1' ⎕WC 'Form' 'My Application' (10 30)(80 60)
```

```
⍝ Create a Form with specified properties (by keyword)
⍝   Caption = "My Application"  (Title)
⍝   Posn    = 10 30  (10% down, 30% across)
⍝   Size    = 80 60  (80% high, 60% wide)

      PROPS←⊂'Type' 'Form'
      PROPS,←⊂'Caption' 'My Application'
      PROPS,←⊂'Posn' 10 30
      PROPS,←⊂'Size' 80 60
      'F1' ⎕WC PROPS

⍝ Create a default Button (a pushbutton) in the Form F1

      'F1.BTN' ⎕WC 'Button'

⍝ Create a pushbutton labelled "Ok"
⍝ 10% down and 10% across from the start of the FORM
⍝ with callback function FOO associated with EVENT 30
⍝ (this event occurs when the user presses the button)

      'F1.BTN'⎕WC'Button' '&Ok' (10 10)('Event' 30 'FOO')
```

Monadic ⎕WC is used to *attach* a GUI component to an existing object. The existing object must be a pure namespace or a GUI object. The operation may be performed by changing space to the object or by running ⎕WC *inside* the object using the *dot* syntax. For example, the following statements are equivalent.

```
      )CS F
#.F
      ⎕WC 'Form'  ⍝ Attach a Form to this namespace

      )CS
#
      F.⎕WC'Form' ⍝ Attach a Form to namespace F
```

6.4.177 Window Get Property R←{X}⎕WG Y

Windows only.

This system function returns property values for a GUI object.

`X` is a namespace reference or a character vector containing the name of the object. `Y` is a character vector or a vector of character vectors containing the name(s) of the properties whose values are required. The result `R` contains the current values of the specified properties. If `Y` specifies a single property name, a single property value is returned. If `Y` specifies more than one property, `R` is a vector with one element per name in `Y`.

If `X` refers to a non-existent GUI name, a `VALUE ERROR` is reported. If `Y` refers to a non-existent property, or to a property that is not defined for the type of object `X`, a `DOMAIN ERROR` is reported.

GUI objects are named **relative** to the current namespace. A null value of `X` (referring to the namespace in which the function is being evaluated) may be omitted. The following examples are equivalent:

```
      'F1.B1' ⎕WG 'Caption'
      'B1' F1.⎕WG 'Caption'
      '' F1.B1.⎕WG 'Caption'
      F1.B1.⎕WG 'Caption'
```

Examples

```
      'F1' ⎕WC 'Form' 'TEST'

      'F1' ⎕WG 'Caption'
TEST

      'F1' ⎕WG 'MaxButton'
1

      'F1' ⎕WG 'Size'
50 50

      ]display 'F1' ⎕WG 'Caption' 'MaxButton' 'Size'
┌→──────────────────┐
│ ┌→──┐   ┌→────┐ │
│ │TEST│ 1 │50 50│ │
│ └───┘   └~────┘ │
└∊──────────────────┘
```

6.4.178 Window Child Names R←{X}⎕WN Y

Windows only.

This system function reports the GUI objects whose parent is `Y`.

If `Y` is a name (that is, is a character vector) then the result `R` is a vector of character vectors containing the names of the named direct GUI children of `Y`.

If `Y` is a reference then the result `R` is a vector of references to the direct GUI children of `Y`, named or otherwise.

The optional left argument `X` is a character vector which specifies the `Type` of GUI object to be reported; if `X` is not specified, no such filtering is performed.

Names of objects further down the tree are not returned, but can be obtained by recursive use of `⎕WN`.

If `Y` refers to a namespace with no GUI element, a `VALUE ERROR` is reported.

Note that `⎕WN` reports **only** those child objects visible from the current thread.

GUI objects are named **relative** to the current namespace. The following examples are equivalent:

```
      ⎕WN 'F1.B1'
      F1.⎕WN 'B1'
      F1.B1.⎕WN ''
```

EXAMPLE

```
      f←⎕NEW⊂'Form'
      f.n←⎕NS''                    ⍝ A non-GUI object
      f.l←f.⎕NEW⊂'Label'           ⍝ A reference to a Label
      'f.b1'⎕WC'Button'            ⍝ A named Button
      f.(b2←⎕NEW ⊂'Button')        ⍝ A reference to a Button
      ⎕WN 'f'
[Form].b1
      ⎕WN f
#.[Form].[Label]  #.[Form].b1  #.[Form].[Button]
      'Button' ⎕WN f
#.[Form].b1  #.[Form].[Button]
```

6.4.179 Window Set Property {R}←{X}⎕WS Y

Windows only.

This system function resets property values for a GUI object.

`X` is a namespace reference or a character vector containing the name of the object. `Y` defines the property or properties to be changed and the new value or values. If a single property is to be changed, `Y` is a vector whose first element `Y[1]` is a character vector containing the property name. If `Y` is of length 2, `Y[2]` contains the corresponding property value. However, if the property value is itself a numeric or nested vector, its elements may be specified in `Y[2 3 4 ...]` instead of as a single nested element in `Y[2]`. If `Y` specifies more than one property, they may be declared either positionally or with a keyword followed by a value. Properties are specified positionally by placing their values in `Y` in the order prescribed for an object of that type. Note that the first property in `Y` must always be specified with a keyword because the `Type` property (which is expected first) may not be changed using `⎕WS`.

If `X` refers to a non-existent GUI name, a `VALUE ERROR` is reported. If `Y` refers to a non-existent property, or to a property that is not defined for the type of object `X`, or to a property whose value may not be changed by `⎕WS`, a `DOMAIN ERROR` is reported.

The shy result `R` contains the previous values of the properties specified in `Y`.

GUI objects are named **relative** to the current namespace. A null value of `⍬` (referring to the namespace in which the function is being evaluated) may be omitted. The following examples are equivalent:

```
      'F1.B1' ⎕WS 'Caption' '&Ok'
      'B1' F1.⎕WS 'Caption' '&Ok'
      '' F1.B1.⎕WS 'Caption' '&Ok'
      F1.B1.⎕WS 'Caption' '&Ok'
```

EXAMPLES

```
      'F1' ⎕WC 'Form'  ⍝ A default Form

      'F1' ⎕WS 'Active' 0

      'F1' ⎕WS 'Caption' 'My Application'

      'F1' ⎕WS 'Posn' 0 0

      'F1' ⎕WS ('Active' 1)('Event' 'Configure' 'FOO')

      'F1' ⎕WS 'Junk' 10
DOMAIN ERROR

      'F1' ⎕WS 'MaxButton' 0
DOMAIN ERROR
```

6.4.180 Workspace Identification ⎕WSID

This is a simple character vector. It contains the identification name of the active workspace. If a new name is assigned, that name becomes the identification name of the active workspace, provided that it is a correctly formed name.

See *Programming: Workspaces* for workspace naming conventions.

It is useful, though not essential, to associate workspaces with a specific directory in order to distinguish workspaces from other files.

The value of ⎕WSID in a clear workspace is `'CLEAR WS'`. ⎕WSID has workspace scope.

EXAMPLE

```
      ⎕WSID
CLEAR WS

      ⎕WSID←'ws/mywork        (UNIX)

      ⎕WSID←'B:\WS\MYWORK'    (Windows)
```

6.4.181 Window Expose ⎕WX

⎕WX is a system variable with the value 0, 1or 3. ⎕WX has Namespace scope.

Considered as a sum of bit flags, the first bit in ⎕WX specifies (a); the second (b) as follows:

1. whether or not the names of properties, methods and events provided by a Dyalog APL GUI object are exposed
2. certain aspects of behaviour of .NET and COM objects

If ⎕WX is 1 (1st bit is set), the names of properties, methods and events are exposed as reserved names in GUI namespaces and can be accessed directly by name. This means that the same names may not be used for global variables in GUI namespaces.

If ⎕WX is 0, these names are hidden and may only be accessed indirectly using ⎕WG and ⎕WS.

If ⎕WX is 3 (2nd bit is also set) COM and .NET objects adopt the behaviour introduced in Version 11, as opposed to the behaviour in previous versions of Dyalog APL.

Any attempt to set ⎕WX to 2 generates a `DOMAIN ERROR`.

Note that it is the value of ⎕WX in the object itself, rather than the value of ⎕WX in the calling environment, that determines its behaviour.

When you create an object, its ⎕WX (like any other system variable) is initially inherited from its parent.

If the value of ⎕WX of a GUI object is initially 0, it will not expose its members. If you subsequently change it from 0 to 1, it will expose them. If you change its ⎕WX back to 0, it will not expose any yet-unexposed members, although already-exposed members will continue to be exposed.

The value of ⎕WX in a clear workspace is defined by the default_wx parameter (see *Installation/Configuration: Configuration Parameters*) which itself defaults to 3.

⎕WX has namespace scope and may be localised in a function header. This allows you to create a utility namespace or utility function in which the exposure of objects is known and determined, regardless of its global value in the workspace.

Notes

- The visibility of the properties and methods of the Root object are not controlled by ⎕WX but by the **PropertyExposeRoot** parameter. For further information, see *Installation/Configuration: Configuration Parameters*.
- ⎕WX is retained for backwards compatibility and should be considered as deprecated. Dyalog recommends ⎕WX be set to 3 and never changed.

6.4.182 XML Convert `R←{X} ⎕XML Y`

⎕XML converts an XML string into an APL array or converts an APL array into an XML string.

Options for ⎕XML are specified using the Variant operator ⍠ or by the optional left argument X. The former is recommended but the older mechanism using the left argument is still supported.

For conversion *from* XML, Y is a character vector containing an XML string. The result R is a 5 column matrix whose columns are made up as follows:

Column	Description
1	Numeric value which indicates the level of nesting
2	Element name, other markup text, or empty character vector when empty
3	Character data or empty character vector when empty
4	Attribute name and value pairs, (`0 2⍴⊂''`) when empty
5	A numeric value which indicates what the row contains

The values in column 5 have the following meanings:

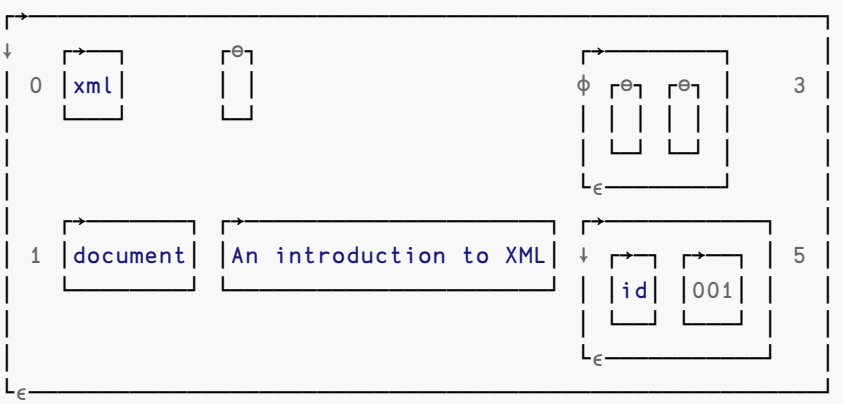
Value	Description
1	Element
2	Child element
4	Character data
8	Markup not otherwise defined
16	Comment markup
32	Processing instruction markup

These values are additive. For example, a value of 5 in column 5 means that the row contains both an element (value 1) and character data (value 4).

EXAMPLE

```
      x←'<xml><document id="001">An introduction to XML'
      x,←'</document></xml>'
```

```
      ]display v←⎕XML x
```



For conversion *to* XML, `Y` is a 3, 4 or 5 column matrix and the result `R` is a character vector. The columns of `Y` have the same meaning as those described above for the result of converting *from* XML.

EXAMPLE

```
      ⎕XML v
<xml>
  <document id="001">An introduction to XML</document>
</xml>
```

Introduction to XML and Glossary of Terms

XML is an open standard, designed to allow exchange of data between applications. The full specification [25] describes functionality, including processing directives and other directives, which can transform XML data as it is read, and which a full XML processor would be expected to handle.

The ⎕XML function is designed to handle XML to the extent required to import and export APL data. It favours speed over complexity - some markup is tolerated but largely ignored, and there are no XML query or validation features. APL applications which require processing, querying or validation will need to call external tools for this, and finally call ⎕XML on the resulting XML to perform the transformation into APL arrays.

XML grammar such as processing instructions, document type declarations etc. may optionally be stored in the APL array, but will not be processed or validated. This is principally to allow regeneration of XML from XML input which contains such structures, but an APL application could process the data if it chose to do so.

The XML definition uses specific terminology to describe its component parts. The following is a summary of the terms used in this section:

Character Data

Character data consists of free-form text. The free-form text should not include the characters '>', '<' or '&', so these must be represented by their entity references ('>', '<' and '&' respectively), or numeric character references.

Entity References and Character References

Entity references are named representations of single characters which cannot normally be used in character data because they are used to delimit markup, such as > for '>'. Character references are numeric representations of any character, such as ⍰ for space. Note that character references always take values in the Unicode code space, regardless of the encoding of the XML text itself.

25 http://www.w3.org/TR/2008/REC-xml-20081126/

⎕XML converts entity references and all character references which the APL character set is able to represent into their character equivalent when generating APL array data; when generating XML it converts any or all characters to entity references as needed.

There is a predefined set of entity references, and the XML specification allows others to be defined within the XML using the `<!ENTITY >` markup. ⎕XML does not process these additional declarations and therefore will only convert the predefined types.

Whitespace

Whitespace sequences consist of one or more spaces, tabs or line-endings. Within character data, sequences of one or more whitespace characters are replaced with a single space when this is enabled by the whitespace option. Line endings are represented differently on different systems (0x0D 0x0A, 0x0A and 0x0D are all used) but are normalized by converting them all to 0x0A before the XML is parsed, regardless of the setting of the whitespace option.

Elements

An element consists of a balanced pair of tags or a single empty element tag. Tags are given names, and start and end tag names must match.

An example pair of tags, named TagName is

```
<TagName></TagName>
```

This pair is shown with no content between the tags; this may be abbreviated as an empty element tag as

```
<TagName/>
```

Tags may be given zero or more attributes, which are specified as name/value pairs; for example

```
<TagName AttName="AttValue">
```

Attribute values may be delimited by either double quotes as shown or single quotes (apostrophes); they may not contain certain characters (the delimiting quote, '&' or '<') and these must be represented by entity or character references.

The content of elements may be zero or more mixed occurrences of character data and nested elements. Tags and attribute names *describe* data, attribute values and the content within tags contain the data itself. Nesting of elements allows structure to be defined.

Because certain markup which describes the format of allowable data (such as element type declarations and attribute-list declarations) is not processed, no error will be reported if element contents and attributes do not conform to their restricted declarations, nor are attributes automatically added to tags if not explicitly given.

Attributes with names beginning **xml:** are reserved. Only **xml:space** is treated specially by ⎕XML. When converting both from and to XML, the value for this attribute has the following effects on space normalization for the character data within this element and child elements within it (unless subsequently overridden):

- **default** – space normalization is as determined by the **whitespace** option.
- **preserve** - space normalization is disabled – all whitespace is preserved as given.
- **any other value** – rejected.

Regardless of whether the attribute name and value have a recognised meaning, the attribute will be included in the APL array / generated XML. Note that when the names and values of attributes are examined, the comparisons are case-sensitive and take place after entity references and character references have been expanded.

Comments

Comments are fully supported markup. They are delimited by '' and all text between these delimiters is ignored. This text is included in the APL array if markup is being preserved, or discarded otherwise.

CDATA Sections

CDATA Sections are fully supported markup. They are used to delimit text within character data which has, or may have, markup text in it which is not to be processed as such. They and are delimited by ''. CDATA sections are never recorded in the APL array as markup when XML is processed – instead, that data appears as character data. Note that this means that if you convert XML to an APL array and then convert this back to XML, CDATA sections will not be regenerated. It is, however, *possible* to generate CDATA sections in XML by presenting them as markup.

Processing Instructions

Processing Instructions are delimited by '<&' and '&>' but are otherwise treated as other markup, below.

Other markup

The remainder of XML markup, including document type declarations, XML declarations and text declarations are all delimited by '<!' and '>', and may contain nested markup. If markup is being preserved the text, including nested markup, will appear as a single row in the APL array. `⎕XML` does not process the contents of such markup. This has varying effects, including but not limited to the following:

- No validation is performed.
- Constraints specified in markup such element type declarations will be ignored and therefore syntactically correct elements which fall outside their constraint will not be rejected.
- Default attributes in attribute-list declarations will not be automatically added to elements.
- Conditional sections will always be ignored.
- Only standard, predefined, entity references will be recognized; entity declarations which define others entity references will have no effect.
- External entities are not processed.

Conversion from XML

- The level number in the first column of the result `R` is 0 for the outermost level and subsequent levels are represented by an increase of 1 for each level. Thus, for
- An introduction to XML
- The *xml* element is at level 0 and the *document id* element is at level 1. The text within the *document id* element is at level 2.
- Each tag in the XML contains an element name and zero or more attribute name and value pairs, delimited by '<' and '>' characters. The delimiters are not included in the result matrix. The element name of a tag is stored in column 2 and the attribute(s) in column 4.
- All XML markup other than tags are delimited by either '<!' and '>', or '<?' and '>' characters. By default these are not stored in the result matrix but the **markup** option may be used to specify that they are. The elements are stored in their

entirety, except for the leading and trailing '<' and '>' characters, in column 2. Nested constructs are treated as a single block. Because the leading and trailing '<' and '>' characters are stripped, such entries will always have either '!' or '&' as the first character.

- Character data itself has no tag name or attributes. As an optimisation, when character data is the sole content of an element, it is included with its parent rather than as a separate row in the result. Note that when this happens, the level number stored is that of the parent; the data itself implicitly has a level number one greater.
- Attribute name and value pairs associated with the element name are stored in the fourth column, in an (*n x 2*) matrix of character values, for the *n* (including zero) pairs.
- Each row is further described in the fifth column as a convenience to simplify processing of the array (although this information could be deduced). Any given row may contain an entry for an element, character data, markup not otherwise defined, a comment or a processing instruction. Furthermore, an element will have zero or more of these as children. For all types except elements, the value in the fifth column is as shown above. For elements, the value is computed by adding together the value of the row itself (1) and those of its children. For example, the value for a row for an element which contains one or more sub-elements and character data is 7 – that is 1 (element) + 2 (child element) + 4 (character data). It should be noted that:
- Odd values always represent elements. Odd values other than 1 indicate that there are children.
- Elements which contain just character data (5) are combined into a single row as noted previously.
- Only immediate children are considered when computing the value. For example, an element which contains a sub-element which in turn contains character data does not itself contain the character data.
- The computed value is derived from what is actually preserved in the array. For example, if the source XML contains an element which contains a comment, but comments are being discarded, there will be no entry for the comment in the array and the fifth column for the element will not indicate that it has a child comment.

Conversion to XML

Conversion to XML takes an array with the format described above and generates XML text from it. There are some simplifications to the array which are accepted:

- The fifth column is not needed for XML generation and is effectively ignored. Any numeric values are accepted, or the column may be omitted altogether. If the fifth column is omitted then the fourth column may also be omitted.
- For the fourth column, if there are no attributes in a particular row then the `(0 2⍴⊂'')` may be abbreviated as `⍬` (zilde). If there is only one attribute then a 2-element vector can be specified.
- Data in the third column and attribute values in the fourth column (if present) may be provided as either character vectors or numeric values. Numeric values are implicitly formatted as if `⎕PP` was set to 17.

The following validations are performed on the data in the array:

- All elements within the array are checked for type.
- Values in column 1 must be non-negative and start from level 0, and the increment from one row to the next must be ≤ +1.
- Tag names in column 2 and attribute names in column 4 (if present) must conform to the XML name definition.

Then, character references and entity references are emitted in place of characters where necessary, to ensure that valid XML is generated. However, markup, if present, is *not* validated and it is possible to generate invalid XML if care in not taken with markup constructs.

Options

There are 3 options which may be specified using the Variant operator `⍠` (recommended) or by the optional left argument `X` (retained for backwards compatibility). The names are different and are case-sensitive; they must be spelled exactly as shown below.

Option names for Variant	Option names for left argument
Whitespace	whitespace
Markup	markup
UnknownEntity	unknown-entity

The values of each option are tabulated below. In each case the value of the option for Variant is given first, followed by its equivalent for the optional left argument in brackets; for example, **UnknownEntity (unknown-entity)**.

Note that the default value is shown first, and that the option names and values are case-sensitive.

If options are specified using the optional left argument, `X` specifies a set of option/value pairs, each of which is a character vector. `X` can be a 2-element vector, or a vector of 2-element character vectors. In the examples below, this method is illustrated by the equivalent expression written as a comment, following the recommended approach using the Variant operator `⍠`:

```
      ]display (⎕XML⍠'Whitespace' 'Strip')eg
      ⍝       'whitespace' 'strip' ⎕XML eg
```

Errors detected in the input arrays or options will all cause `DOMAIN ERROR`.

Whitespace (whitespace)

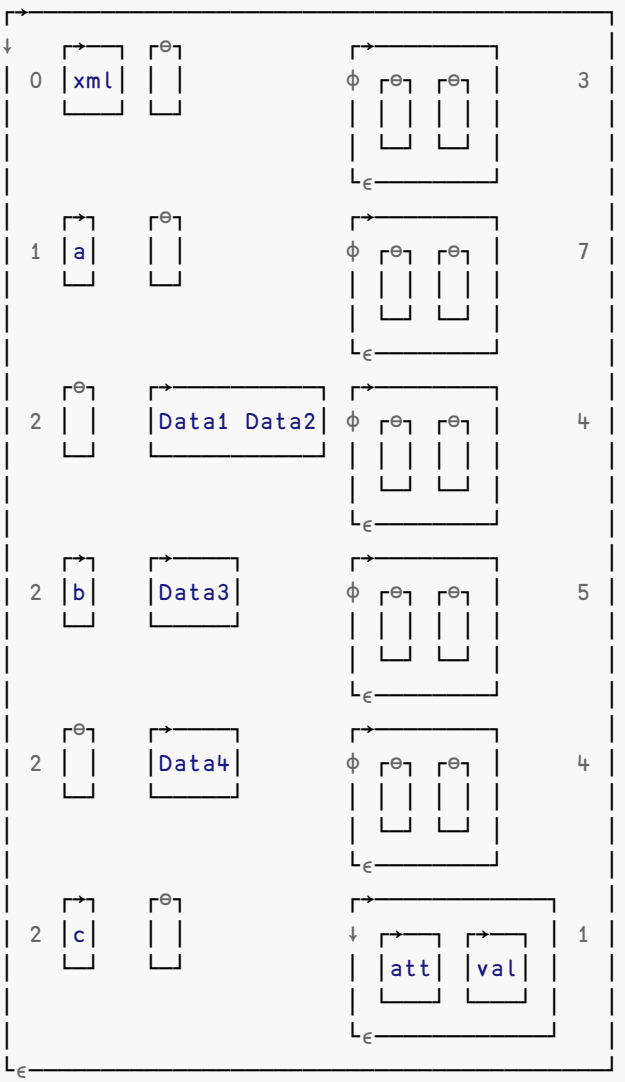
When converting from XML `Whitespace` specifies the default handling of white space surrounding and within character data. When converting to XML `Whitespace` specifies the default formatting of the XML. Note that attribute values are not comprised of character data so white space in attribute values is always preserved.

Converting from XML	
Strip (strip)	All leading and trailing whitespace sequences are removed; remaining whitespace sequences are replaced by a single space character
`Trim(trim)`	All leading and trailing whitespace sequences are removed; all remaining white space sequences are handled as preserve
`Preserve(preserve)`	Whitespace is preserved as given except that line endings are represented by Linefeed (`⎕UCS 10`)
Converting to XML	
Strip (strip)	All leading and trailing whitespace sequences are removed; remaining whitespace sequences within the data are replaced by a single space character. XML is generated with formatting and indentation to show the data structure
`Trim(trim)`	Synonymous with `strip`
`Preserve(preserve)`	White space in the data is preserved as given, except that line endings are represented by Linefeed (`⎕UCS 10`). XML is generated with no formatting and indentation other than that which is contained within the data

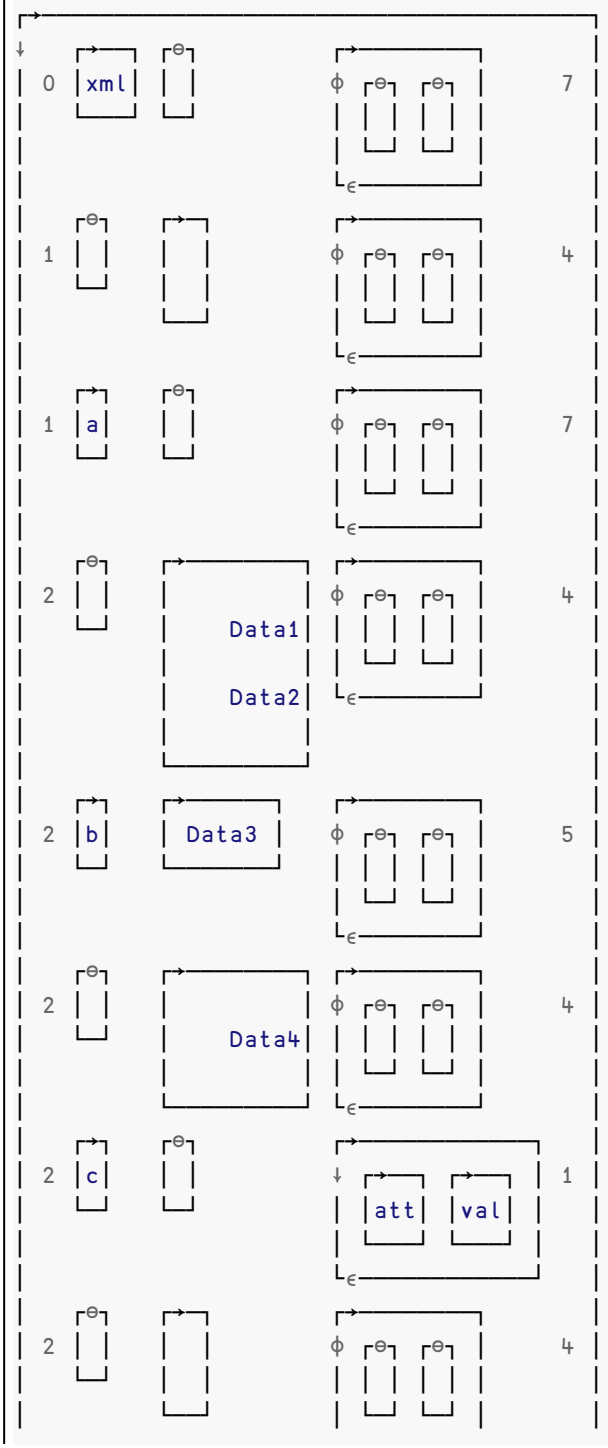
```
      ]display eg
┌→──────────────────┐
│<xml>              │
│  <a>              │
│    Data1          │
│    <!-- Comment -->│
│    Data2          │
│    <b> Data3 </b>  │
│    Data4          │
│    <c att="val"/>  │
│  </a>             │
│</xml>             │
└───────────────────┘
```

Converting from XML

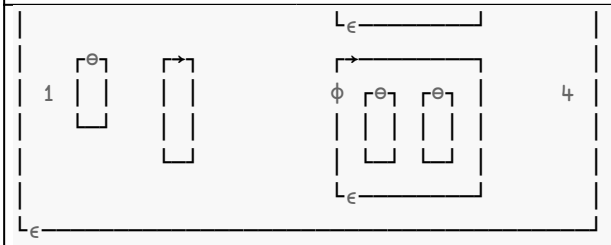
```
      ]display (⎕XML⍠'Whitespace' 'Strip')eg
      ⍝      'whitespace' 'strip' ⎕XML eg
```



```
      ]display (⎕XML⍠'Whitespace' 'Preserve')eg
      ⍝          'whitespace' 'preserve' ⎕XML eg
```



Converting from XML



Markup (markup)

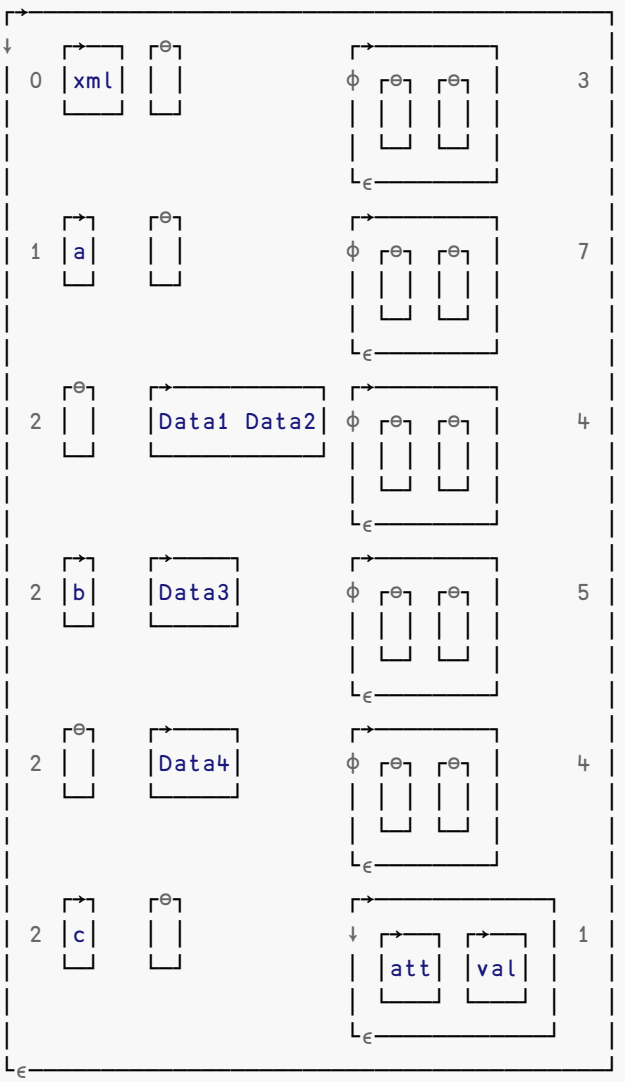
When converting from XML, `Markup` determines whether markup (other than entity tags) appears in the output array or not. When converting to XML `Markup` has no effect.

Converting from XML	
Strip (strip)	Markup data is not included in the output array
`Preserve(preserve)`	Markup text appears in the output array, without the leading '<' and trailing '>' in the tag (2 nd) column

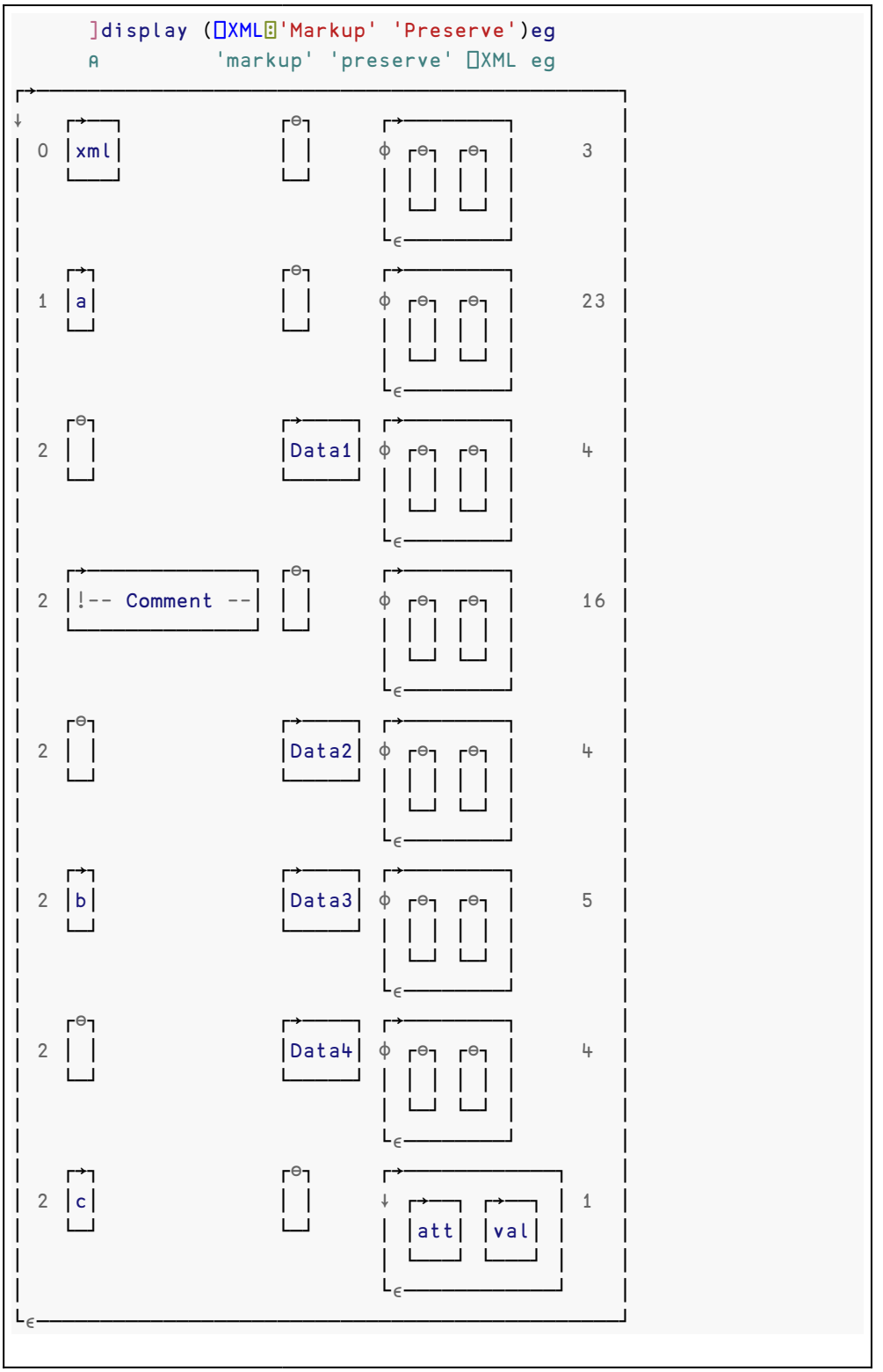
```
      ]display eg
┌→─────────────────┐
│<xml>             │
│  <a>             │
│    Data1         │
│    <!-- Comment -->│
│    Data2         │
│    <b> Data3 </b>  │
│    Data4         │
│    <c att="val"/>  │
│  </a>            │
│</xml>            │
└──────────────────┘
```

Converting from XML

```
      ]display (⎕XML⍠'Markup' 'Strip')eg
      ⍝      'markup' 'strip' ⎕XML eg
```



```
      ]display (⎕XML⍠'Markup' 'Preserve')eg
      ⍝          'markup' 'preserve' ⎕XML eg
```



UnknownEntity (unknown-entity)

When converting from XML, this option determines what happens when an unknown entity reference, or a character reference for a Unicode character which cannot be represented as an APL character, is encountered. In Classic versions of Dyalog APL that is any Unicode character which does not appear in ⎕AVU. When converting to XML, this option determines what happens to Esc characters (⎕UCS 27) in data.

Converting from XML	
Replace (replace)	The reference is replaced by a single '?' character
`Preserve(preserve)`	The reference is included in the output data as given, but with the leading '&' replaced by Esc (⎕UCS 27)
Converting to XML	
Replace (replace)	Esc (⎕UCS 27) is preserved
`Preserve(preserve)`	Esc (⎕UCS 27) is replaced by '&'

6.4.183 Extended State Indicator R←⎕XSI

R is a nested vector of character vectors giving the full path names of the functions or operators in the execution stack. Note that if a function has changed space, its original (home) space is reported, rather than its current one.

EXAMPLE

In the following, function `foo` in namespace `x` has called `goo` in namespace `y`. Function `goo` has then changed space (⎕CS) to namespace `z` where it has been suspended:

```
      )SI
[z] y.goo[2]*
x.foo[1]
```

⎕XSI reports the full path name of each function:

```
      ⎕XSI
 #.y.goo  #.x.foo
```

This can be used for example, to edit all functions in the stack, irrespective of the current namespace by typing: ⎕ED ⎕XSI

See also *Section 6.4.135*.

6.4.184 Set External Variable {R}←X ⎕XT Y

Y must be a simple character scalar or vector which is taken to be a variable name. X must be a simple character scalar or vector which is taken to be a file reference. The name given by Y is identified as an EXTERNAL VARIABLE associated with an EXTERNAL ARRAY whose value may be stored in file identified by X. The shy result R has the same value as X.

If Y is the name of a defined function or operator, a label or a namespace in the active workspace, a DOMAIN ERROR is reported.

Attempts to assign namespace references or the ⎕OR of namespaces to an external array will result in a DOMAIN ERROR.

EXAMPLE

```
      'EXT\ARRAY' ⎕XT 'V'
```

If the file reference does not exist, the external variable has no value until a value is assigned:

```
      V
VALUE ERROR
      V
      ^
```

A value assigned to an external variable is stored in file space, not within the workspace:

```
      ⎕WA
2261186

      V←⍳100000

      ⎕WA
2261186
```

There are no specific restrictions placed on the use of external variables. They must conform to the normal requirements when used as arguments of functions or as operands of operators. The essential difference between a variable and an external variable is that an external variable requires only temporary workspace for an operation to accommodate (usually) a part of its value.

Examples

```
      V←⍳5
      +/V
15

      V[3]←⊂'ABC'

      V
1 2  ABC  4 5

      ⍴⍴V
     3
```

Assignment allows the structure or the value of an external variable to be changed without fully defining the external array in the workspace.

Examples

```
      V,←⊂2 4⍴⍳8

      ⍴V
6

      V[6]
1 2 3 4
5 6 7 8

      V[1 2 4 5 6]×←10

      V
10 20  ABC  40 50  10 20 30 40
                   50 60 70 80
```

An external array is (usually) preserved in file space when the name of the external variable is disassociated from the file. It may be re-associated with any valid variable name.

Example

```
      ⎕EX'V'

      'EXT\ARRAY'⎕XT'F'

      F
10 20  ABC  40 50  10 20 30 40
                   50 60 70 80
```

In UNIX versions, if `X` is an empty vector, the external array is associated with a temporary file which is erased when the array is disassociated.

Example

```
      ''⎕XT'TEMP'

      TEMP←⍳10

      +/TEMP×TEMP
385

      ⎕EX'TEMP'
```

An external array may be erased using the native file function: `⎕NERASE`.

In a multi-user environment (UNIX or a Windows LAN) a new file associated with an external array is created with access permission for owner read/write. An existing file is opened for exclusive use (by the owner) if the permissions remain at this level. If the access permissions allow any other users to read and write to the file, the file is opened for shared use. In UNIX versions, access permissions may be modified using the appropriate Operating System command, or in Windows using the supplied function `XVAR` from the UTIL workspace.

> **Information**
>
> Support for external variables has been deprecated, and they are scheduled for removal in a future release. For information on how to identify uses of external variables in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

6.4.185 Query External Variable R←⎕XT Y

Y must be a simple character scalar or vector which is taken to be a variable name. R is a simple character vector containing the file reference of the external array associated with the variable named by Y, or the null vector if there is no associated external array.

EXAMPLE

```
      ⎕XT'V'
EXT\ARRAY

      ⍴⎕XT'G'
0
```

> **Information**
>
> Support for external variables has been deprecated, and they are scheduled for removal in a future release. For information on how to identify uses of external variables in your existing codebase, see the *v20.0 Release Notes: Deprecated Functionality*.

7 System Commands

7.1 Introduction

System commands are **not** executable APL expressions. They provide services or information associated with the workspace and the **external environment**.

Command Presentation

System commands may be entered from immediate execution mode or in response to the prompt ⍞: within evaluated input. All system commands begin with the symbol), known as a right parenthesis. All system commands may be entered in upper or lower case.

Table 7-1: *System Commands*

Command	Description
)CLASSES	List classes
)CLEAR	Clear the workspace
)CMD Y	Execute a Windows Command
)CONTINUE	Save a Continue workspace and terminate APL
)COPY {Y}	Copy objects from another workspace
)CS {Y}	Change current namespace
)DROP {Y}	Drop named workspace
)ED Y	Edit object(s)
)ERASE Y	Erase object(s)
)EVENTS	List events of GUI namespace or object
)FNS {Y}	List user defined Functions
)HOLDS	Display Held tokens
)LIB {Y}	List workspaces in a directory
)LOAD {Y}	Load a workspace
)METHODS	List methods in GUI namespace or object
)NS {Y}	Create a global Namespace
)OBJECTS {Y}	List global namespaces
)OBS {Y}	List global namespaces (alternative form)
)OFF	Terminate the APL session
)OPS {Y}	List user defined Operators
)PCOPY {Y}	Perform Protected Copy of objects
)PROPS	List properties of GUI namespace or object
)RESET	Reset the state indicator
)SAVE {Y}	Save the workspace
)SH {Y}	Execute a (UNIX) Shell command
)SI	State Indicator
)SIC	Clear State Indicator

Command	Description
)SINL	State Indicator with local Name Lists
)TID {Y}	Switch current Thread Identity
)VARS {Y}	List user defined global Variables
)WSID {Y}	Workspace Identification
)XLOAD Y	Load a workspace; do not execute ⎕LX
{ } indicates that the parameter(s) denoted by Y are optional.	

7.2 System Commands (A-Z)

7.2.1 List Classes)CLASSES

This command lists the names of APL Classes in the active workspace.

EXAMPLE

```
      )CLEAR
clear ws
      )ED ⍥MyClass

:Class MyClass
    ∇ Make Name
      :Implements Constructor
      ⎕DF Name
    ∇
:EndClass ⍝ MyClass

      )CLASSES
MyClass
      )COPY ⎕⎕ YourClass
.\⎕⎕ saved Sun Jan 29 18:32:03 2006
      )CLASSES
MyClass YourClass
      ⎕NC 'MyClass' 'YourClass'
9.4 9.4
```

7.2.2 Clear Workspace)CLEAR

This command clears the active workspace and gives the report "`clear ws`". The active workspace is lost. The name of a clear workspace is `CLEAR WS`. *System Variables* are initialised with their default values.

In GUI implementations of Dyalog, `)CLEAR` expunges all GUI objects, discards any unprocessed events in the event queue and resets the properties of the `Root` object `'.'` to their default values.

Apart from .NET objects, the contents of the session namespace `⎕SE` are not affected. .NET objects in `⎕SE` are disconnected from .NET because `)CLEAR` closes the current .NET AppDomain.

7.2.3 Windows Command Processor)CMD cmd

This command allows Windows Command Processor or UNIX shell commands to be given from APL. `)CMD` is a synonym of `)SH`. Either command may be given in either environment (Windows or UNIX) with exactly the same effect. `)CMD` is probably more natural for the Windows user. This section describes the behaviour of `)CMD` and `)SH` under Windows. See *Section 7.2.25* for a discussion of the behaviour of these commands under UNIX.

The system functions *Section 6.4.131* and *Section 6.4.17* provide similar facilities but may be executed from within APL code. For further information, see *Section 6.4.131* and *Section 6.4.17*.

Note that under Windows, you may not execute `)CMD` without a command. If you wish to, you can easily open a new Command Prompt window outside APL.

EXAMPLE

```
     )CMD dir
 Volume in drive C is OS
 Volume Serial Number is B438-9B76

 Directory of C:\Users\Pete\Documents\Dyalog APL-64 17.0 Unicode Files

23/06/2018  15:59    <DIR>          .
23/06/2018  15:59    <DIR>          ..
23/06/2018  14:53           181,488 default.dlf
13/06/2018  20:13         1,262,296 def_uk.dse
14/06/2018  14:36           108,976 UserCommand20.cache
               3 File(s)      1,552,760 bytes
               2 Dir(s)  101,371,437,056 bytes free
```

If **cmd** issues prompts and expects user input, it is **ESSENTIAL** to explicitly redirect input and output to the console. If this is done, APL detects the presence of a ">" in the command line and runs the command processor in a visible window and does not direct output to the pipe. If you fail to do this your system will appear to hang because there is no mechanism for you to receive or respond to the prompt.

EXAMPLE

```
      )CMD DATE <CON >CON
```

(Command Prompt window appears)

```
Current date is Wed 19-07-1995
Enter new date (dd-mm-yy): 20-07-95
```

(Command Prompt window disappears)

Implementation Notes

The argument of `)CMD` is simply passed to the appropriate command processor for execution and its output is received using an *unnamed pipe*.

By default, `)CMD` will execute the string `('cmd.exe /c',Y)` where `Y` is the argument given to `)CMD`. However, the implementation permits the use of alternative command processors as follows:

Before execution, the argument is prefixed and postfixed with strings defined by the APL parameters CMD_PREFIX and CMD_POSTFIX. The former specifies the name of

your command processor and any parameters that it requires. The latter specifies a string which may be required. If CMD_PREFIX is not defined, it defaults to the name defined by the environment variable COMSPEC followed by "\c". If COMSPEC is not defined, it defaults to COMMAND.COM or CMD.EXE as appropriate. If CMD_POSTFIX is not defined, it defaults to an empty vector.

> **Note**
>
> note This function is disabled and instead generates a `DOMAIN ERROR` if the RIDE_SPAWNED parameter is non-zero. This is designed to prevent it being invoked from a Ride session which does not support this type of user interface. For further details, see the Ride User Guide.

7.2.4 Save Continuation `)CONTINUE`

This command saves the active workspace in the current working directory and ends the Dyalog session. The name of the workspace file is CONTINUE in upper-case with the extension defined by the *Installation/Configuration: Configuration Parameters*.

Note that the values of all system variables (including `⎕SM`) and GUI objects are also saved in `CONTINUE`.

When a `CONTINUE` workspace is loaded, the latent expression (if any) is NOT executed.

7.2.5 Copy Workspace `)COPY {ws {nms}}`

This command brings all or selected global objects `nms` from a stored workspace (or session file) with the given name. A stored workspace is one which has previously been saved with the system command `)SAVE` or the system function `⎕SAVE`.

See *Programming: Workspaces* for the rules for specifying a workspace name.

If the list of names is excluded, all defined objects (including namespaces) are copied.

If the workspace name identifies a valid, readable workspace, the system reports the workspace name, "`saved`" and the date and time when the workspace was last saved.

Examples

```
      )COPY WS/UTILITY
WS/UTILITY saved Mon Nov  1 13:11:19 1992

      )COPY TEMP ⎕LX FOO X A.B.C
./TEMP saved Mon Nov  1 14:20:47 1992
not found X
```

Copied objects are defined at the global level in the active workspace. Existing global objects in the active workspace with the same name as a copied object are replaced. If the copied object replaces either a function in the state indicator, or an object that is an operand of an operator in the state indicator, or a function whose left argument is being executed, the original object remains defined until its execution is completed or it is no longer referenced by an operator in the state indicator. If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports `ws not found`.

You may copy an object from a namespace by specifying its full pathname. The object will be copied to the current namespace in the active workspace, losing its original parent and gaining a new one in the process. You may only copy a GUI object into a namespace that is a suitable parent for that object. For example, you could only copy a Group object from a saved workspace if the current namespace in the active workspace is itself a Form, SubForm or Group.

If the workspace name identifies a file that is not a workspace, the system reports `bad ws`.

If the source workspace is too large to be loaded, the system reports `ws too large`.

When copying data between Classic and Unicode Editions, `)COPY` will fail with `TRANSLATION ERROR` if *any* object in the source workspace fails conversion between Unicode and `⎕AV` indices, whether or not that object is specified by `nms`. See *Section 6.4.12* for further details.

If `ws` is omitted, the file open dialog box is displayed and all objects copied from the selected workspace.

If the list of names is included, the names of system variables may also be included and copied into the active workspace. The global referents will be copied.

If an object is not found in the stored workspace, the system reports `not found` followed by the name of the object.

Dependent Objects

If the list of names includes the name of:

- an Instance of a Class but not the Class itself
- a Class but not a Class upon which it depends
- an array or a namespace that contains a ref to another namespace, but not the namespace to which it refers

the dependent object(s) **will also be copied** but will be **unnamed** and **hidden**. In such as case, the system will issue a warning message.

For example, if a saved workspace named CFWS contains a Class named `#.CompFile` and an Instance (of `CompFile`) named `icf`,

```
      )COPY CFWS icf
.\CFWS saved Fri Mar 03 10:21:36 2006
copied object created an unnamed copy of class #.CompFile
```

The existence of a hidden copy can be confusing, especially if it is a hidden copy of an object which had a name which is in use in the current workspace. In the above example, if there is a class called `CompFile` in the workspace into which `icf` is copied, the copied instance may *appear* to be an instance of the *visible* `CompFile`, but it will actually be an instance of the hidden `CompFile` - which may have very different (or perhaps worse: very slightly different) characteristics to the named version.

If you copy a Class without copying its Base Class, the Class can be used (it will use the invisible copy of the Base Class), but if you edit the Class, you will either be unable to save it because the editor cannot find the Base Class, or - if there is a visible Class of that name in the workspace - it will be used as the Base Class.

In the latter case, the invisible copy which was brought in by `)COPY` will now disappear, since there are no longer any references to it - and if these two Base Classes were different, the behaviour of the derived Class will change (and any changes made to the invisible Base Class since it was copied will be lost).

Referenced Objects

If you copy a Class or a namespace that is referenced by a Class as its Base Class or via a `:Include` statement, the referring Class will continue to refer to the original definition of the copied name which will be retained for that purpose. The Class can be made to refer to the copied definition of that name by refixing it.

Example

The current workspace has a class named `pete` whose Base class is called `base`.

```
:Class pete: base
:EndClass

:Class base
    ∇ r←foo n
      :Access Public
      r←'Original'n
    ∇
:EndClass
```

A second workspace named `copy.dws` contains a different version of the `base` class:

```
:Class base
    ∇ r←foo n
      :Access Public
      r←'Copied'n
    ∇
:EndClass

      )COPY copy.dws base
copy.dws saved Thu Nov 22 16:24:27 2018
      inst←⎕NEW pete
      inst.foo 1
Original  1

      ⎕FIX ⎕SRC pete
      inst←⎕NEW pete
      inst.foo 1
Copied  1
```

Copying Objects from Session Files

You may also copy objects from session (.dse) files, although with certain restrictions.

Example

```
        )COPY C:\Users\Pete\Desktop\pete.dse ⎕SE.UCMD
C:\Users\Pete\Desktop\pete.dse saved Wed Oct 14 ...
```

- You can copy a function or variable from any namespace under a saved ⎕SE, no matter what your current namespace is.
- You can copy any namespace without GUI from a saved ⎕SE, no matter what you current namespace is.
- You can copy any namespace with GUI from a saved ⎕SE as long as your current namespace is the same as the parent namespace of the namespace you are trying to copy

Examples

In the following example, the *not copied* cases occur because the current namespace is not an appropriate parent for the object in question.

```
      )CS #
#
      )COPY C:\Users\...\pete.dse ⎕SE.Dyalog.Callbacks
C:\Users\...\pete.dse saved Wed Oct 14 15:31:14 2015
```

```
       )COPY C:\Users\...\pete.dse ⎕SE.cbbot
C:\Users\...\pete.dse saved Wed Oct 14 15:31:14 2015
not copied cbbot
```

```
       )CS ⎕SE
⎕SE
      )COPY C:\Users\...\pete.dse ⎕SE.cbbot
C:\Users\...\pete.dse saved Wed Oct 14 15:31:14 2015

      )COPY C:\Users\...\pete.dse ⎕SE.cbbot.bandsb1
C:\Users\...\pete.dse saved Wed Oct 14 15:31:14 2015
not copied bandsb1
```

7.2.6 Change Space)CS {nm}

`)CS` changes the current space to the **global** namespace `nm`.

If no `nm` is given, the system changes to the top level (Root) namespace. If `nm` is not the name of a global namespace, the system reports the error message `Namespace does not exist`.

`name` may be either a simple name or a compound name separated by '`.`', including one of the special names `'#'` (Root) or `'##'` (Parent).

EXAMPLES

```
      )CS
#
      )CS X
#.X
      )CS Y.Z
#.X.Y.Z
      )CS ##
#.X.Y
      )CS #.UTIL
#.UTIL
```

7.2.7 Drop Workspace)DROP {ws}

This command removes the specified workspace from disk storage.

See *Programming: Workspaces* for the rules for specifying a workspace name.

If `ws` is omitted, a file open dialog box is displayed to elicit the workspace name.

EXAMPLE

```
      )DROP WS/TEMP
Thu Sep 17 10:32:18 1998
```

7.2.8 Edit Object)ED nms

`)ED` invokes the Dyalog editor and opens an Edit window for each of the objects specified in `nms`.

If a name includes the slash character (`\` or `/`) it is taken to be the name of a file. See *User Interface: Editing Scripts And Text Files*.

If a name is followed by a line number specified in square brackets, the Editor positions the cursor in the corresponding line. There must not be a space between the last character of the name and the `[`.

If a name is not already defined, it is assumed to be a function or operator unless `type` is also specified.

You can specify the type of a new object by prefixing its name with a single symbol:

∇	function or operator
⍟	Namespace script
○	Class script
◦	Interface
→	simple character vector
∈	vector of character vectors
–	character matrix
◇	array: use array notation

If an existing object is named, any type specification in the command is ignored unless the type is ◇.

If the type is ◇ and the name is either undefined or of an array, the Editor opens in array-notation mode. The resulting array can be of any type or structure.

The first object named becomes the top window on the stack. See the *Dyalog for Microsoft Windows UI Guide* or the *Dyalog for UNIX UI Guide* for details.

EXAMPLES

```
)ED MYFUNCTION

)ED MYFUNCTION[1000] YOURFUNCTION[3]

)ED ∇FOO -MAT ∈VECVEC
```

Objects specified in `nms` that cannot be edited are silently ignored. Objects qualified with a namespace path (for example, `a.b.c.foo`) are silently ignored if the namespace does not exist.

7.2.9 Erase Object)ERASE nms

This command erases named global defined objects (functions, operators, variables, namespaces and GUI objects) from the active workspace or current namespace.

If a named object is a function or operator in the state indicator, or the object is an operand of an operator in the state indicator, or the object is a function whose left

argument is being executed, the object remains defined until its execution is completed or it is no longer referenced by an operator in the state indicator. However, the name is available immediately for other uses.

If a named object is a GUI object, the object and all its children are deleted and removed from the screen.

If an object is not erased for any reason, the system reports `not found` followed by the name of the object.

Erasing objects such as external functions may have other implications: see *Section 6.4.39* for details.

EXAMPLE

```
      )ERASE FOO A ⎕IO
not found ⎕IO
```

7.2.10 List Events)EVENTS

The `)EVENTS` system command lists the Events that may be generated by the object associated with the current space.

For example:

```
      ⎕CS 'BB' ⎕WC 'BrowseBox'

      )EVENTS
Close   Create  FileBoxCancel   FileBoxOK
```

`)EVENTS` produces no output when executed in a pure (non-GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )EVENTS
```

7.2.11 List Global Defined Functions)FNS {nm}

This command displays the names of global defined functions in the active workspace or current namespace. Names are displayed in `⎕AV` collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

EXAMPLES

```
      )FNS
ASK DISPLAY GET PUT ZILCH
      )FNS G
GET PUT ZILCH
```

7.2.12 Display Held Tokens)HOLDS

System command)HOLDS displays a list of tokens which have been acquired or requested by the :Hold control structure.

Each line of the display is of the form:

```
token:  acq  req  req ...
```

Where acq is the number of *the* thread that has acquired the token, and req is the number of *a* thread which is requesting it. For a token to appear in the display, a thread (and only one thread) must have acquired it, whereas any number of threads can be requesting it.

EXAMPLE

Thread 300's attempt to acquire token 'blue' results in a deadlock:

```
300:DEADLOCK
Sema4[1] :Hold 'blue'
         ^


      )HOLDS
blue:   100
green:  200     100
red:    300     200     100
```

- Blue has been acquired by thread 100.
- Green has been acquired by 200 and requested by 100.
- Red has been acquired by 300 and requested by 200 and 100.

The following cycle of dependencies has caused the deadlock:

```
Thread 300 attempts to acquire blue,      300 → blue
which is owned by 100,                     ↑      ↓
which is waiting for red,                 red ←  100
which is owned by 300.
```

7.2.13 List Workspace Library)LIB {dir}

This command lists the names of Dyalog workspaces contained in the given directory.

EXAMPLE

```
      )LIB WS
MYWORK TEMP
```

If a directory is not given, the workspaces on the user's APL workspace path (**WSPATH**) are listed. In this case, the listing is divided into sections identifying the directories concerned. The current directory is identified as ".".

EXAMPLE

```
      )LIB
.
        PDTEMP   WORK    GRAPHICS

c:\Dyalog\ws
        display groups
```

7.2.14 Load Workspace)LOAD {ws}

This command causes the named stored workspace to be loaded. The current active workspace is lost.

`ws` specifies a file name. If no file extension is given, it is implied. See *Installation/ Configuration: Wsext*.

If `ws` is a full or relative pathname, only the specified directory is examined. If not, the APL workspace path (`WSPATH`) is traversed in search of the named workspace. A stored workspace is one which has previously been saved with the system command `)SAVE` or the system function `⎕SAVE`. Under Windows, if `ws` is omitted, the File Open dialog box is displayed.

If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports `ws not found`. If the workspace name identifies a file or directory that is not a workspace, the system reports `wsname is not a ws`. If successfully loaded, the system reports workspace name `saved`, followed by the date and time when the workspace was last saved. If the workspace is too large to be loaded into the APL session, the system reports `ws too large`. After loading the workspace, the current namespace is set to # and the latent expression (`⎕LX`) is

executed unless APL was invoked with the **-x** option. If the workspace was saved with a suspension, typing the expression →1+⎕LC will resume execution and switch back into the namespace associated with the suspended function.

If the workspace contains any GUI objects whose `Visible` property is 1, these objects will be displayed. If the workspace contains a non-empty ⎕SM but does not contain an SM GUI object, the form defined by ⎕SM will be displayed in a window on the screen.

Holding the Ctrl key down while entering a `)LOAD` command or selecting a workspace from the session file menu causes the incoming latent expression to be *traced*.

Holding the Shift key down while selecting a workspace from the session file menu will *prevent* execution of the latent expression.

EXAMPLE

```
      )LOAD dfns
C:\Program Files\Dyalog\Dyalog APL-64 ...

An assortment of D Functions and Operators.

      tree #                ⍝ Workspace map.
      ↑¯10↑↓attrib ⎕NL 3 4  ⍝ What's new?
      notes find 'Word'     ⍝ Apropos "Word".
```

7.2.15 List Methods)METHODS

The `)METHODS` system command lists the Methods that apply to the object associated with the current space.

For example:

```
      ⎕CS 'F' ⎕WC 'Form'
      )METHODS
Animate ChooseFont   Detach  GetFocus    GetTextSize Wait
```

`)METHODS` produces no output when executed in a pure (non-GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )METHODS
```

7.2.16 Create Namespace)NS {nm}

)NS creates a **global** namespace and displays its full name, nm.

nm may be either a simple name or a compound name separated by '.', including one of the special names '#' (Root) or '##' (Parent).

If name does not start with the special Root space identifier '#', the new namespace is created relative to the current one.

If name is already in use for a workspace object other than a namespace, the command fails and displays the error message Name already exists.

If name is an existing namespace, no change occurs.

)NS with no nm specification displays the current namespace.

EXAMPLES

```
      )NS
#

      )NS W.X
#.W.X

      )CS W.X
#.W.X

      )NS Y.Z
#.W.X.Y.Z

      )NS
#.W.X
```

7.2.17 List Global Namespaces)OBJECTS {nm}

This command displays the names of global **namespaces** in the active workspace. Names are displayed in the ⎕AV collating order. If a name is included after the command, only those names starting at or after the given name in collating order are displayed. Namespaces are objects created using ⎕NS,)NS or ⎕WC and have name class 9.

Note:)OBS can be used as an **alternative** to)OBJECTS

EXAMPLES

```
      )OBJECTS
FORM1   UTIL    WSDOC   XREF

      )OBS W
WSDOC   XREF
```

7.2.18 List Global Namespaces)OBS {nm}

This command is the same as the)OBJECTS command. See *Section 7.2.17*.

7.2.19 Sign Off APL)OFF

This command terminates the APL session, returning to the Operating System command processor or shell.

7.2.20 List Global Defined Operators)OPS {nm}

This command displays the names of global defined operators in the active workspace or current namespace. Names are displayed in ⎕AV collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

EXAMPLES

```
      )OPS
AND DOIF DUAL ELSE POWER

      )OPS E
ELSE POWER
```

7.2.21 Protected Copy)PCOPY {ws {nms}}

This command brings all or selected global objects from a stored workspace with the given name provided that there is no existing global usage of the name in the active workspace. A stored workspace is one which has previously been saved with the system command)SAVE or the system function ⎕SAVE.

)PCOPY does not copy ⎕SM. This restriction may be removed in a later release.

If the workspace name is not valid or does not exist or if access to the workspace is not authorised, the system reports "`ws not found`". If the workspace name identifies a file that is not a workspace, or is a workspace with an invalid version number (one that is greater than the version of the current APL) the system reports "`bad ws`".

See *Programming: Workspaces* for the rules for specifying a workspace name.

If the workspace name is the name of a valid, readable workspace, the system reports the workspace name, "`saved`", and the date and time that the workspace was last saved.

If the list of names is excluded, all global defined objects (functions and variables) are copied. If an object is not found in the stored workspace, the system reports "`not found`" followed by the name of the object. If an object cannot be copied into the active workspace because there is an existing referent, the system reports "`not copied`" followed by the name of the object.

For further information, see *Section 6.4.23*.

EXAMPLES

```
      )PCOPY WS/UTILITY
WS/UTILITY saved Mon Nov  1 13:11:19 1993
not copied COPIED IF
not copied COPIED JOIN

      )PCOPY TEMP FOO X
./TEMP saved Mon Nov  1 14:20:47 1993
not found X
```

> **Warning**
>
> If a workspace full condition occurs during the execution of `)PCOPY` the state of the active workspace is unpredictable.

7.2.22 List Properties)PROPS

The `)PROPS` system command lists the Properties of the object associated with the current space.

For example:

```
      ⎕CS 'BB' ⎕WC 'BrowseBox'

      )PROPS
BrowseFor       Caption ChildList       Data    Event
EventList       HasEdit KeepOnClose     MethodList
PropList        StartIn Target  Translate       Type
```

)PROPS produces no output when executed in a pure (non GUI) namespace, for example:

```
      ⎕CS 'X' ⎕NS ''
      )PROPS
```

7.2.23 Reset State Indicator)RESET {n}

This command cancels all suspensions recorded in the state indicator and discards any unprocessed events in the event queue.

The optional parameter n specifies that only the top n suspensions are to be cleared.

)RESET also performs an internal re-organisation of the workspace and process memory. See *Section 6.4.175* for details.

EXAMPLE

```
      )SI
#.FOO[1]*
⍎
#.FOO[1]*

      )RESET

      )SI
```

7.2.24 Save Workspace)SAVE {-force} {ws}

This command compacts (see *Section 6.4.175* for details) and saves the active workspace.

If specified, ws is a full or relative path name to the file in which the workspace will be written. If ws is omitted, it defaults to ⎕WSID. Unless the path specified by ws or ⎕WSID is

a full pathname, it is taken to be relative to the current working directory which may be obtained by the expression: `⍕1 ⎕NPARTS ''`.

If `ws` specifies a file name other than that implied by `⎕WSID`, the specified file must not already exist unless the **force** parameter is specified.. If `ws` is omitted or resolves to the same file as `⎕WSID`, an existing stored workspace with the same name will be replaced.

See *Programming: Workspaces* for the rules for specifying a workspace name.

If an extension is not specified, an extension is added according to the *Installation/ Configuration: Configuration Parameters*..

A workspace may not be saved if any threads (other than the root thread 0) are running or if there are any Edit or Trace windows open. Otherwise, the workspace is saved with its state of execution intact, however certain operations may be performed before it is saved. For further information, see *Section 4.4.9.43*.

`)SAVE` may fail with one of the following error messages:

`unacceptable char`	The given workspace name was ill-formed
`not saved this ws is WSID`	An attempt was made to change the name of the workspace for the save, and the renamed workspace already existed. This error can be overridden by specifying the `-force` argument.
`not saved this ws is CLEAR WS`	The active workspace was `CLEAR WS` and no attempt was made to change the name.
`Can't save - file could not be created.`	The workspace name supplied did not represent a valid file name for the current Operating System.
`cannot create`	The user does not have access to create the file OR the workspace name conflicts with an existing non-workspace file.
`cannot save with windows open`	A workspace may not be saved if trace or edit windows are open.
`Cannot overwrite old workspace`	An attempt was made to save the workspace with a newer version of the interpreter than that with which it was originally saved. This error can be avoided by specifying the `-force` argument.

After a successful save, the system reports the workspace name, followed by the word "`saved`" and the current time and date; and if `ws` specified a new name, `⎕WSID` is assigned that name.

Example

```
      )SAVE MYWORK
./MYWORK saved Thu Sep 17 10:32:20 1998
```

Note that any time prior to executing `)SAVE`, the active workspace may be renamed by the system command `)WSID` or by assigning a name to the system variable `⎕WSID`.

A stored workspace may subsequently be loaded with the system command `)LOAD` or the system function `⎕LOAD`, and objects may be copied from a stored workspace with the system commands `)COPY` or `)PCOPY` or the system function `⎕CY`.

7.2.25 Execute (UNIX) Command)SH {cmd}

This command allows WINDOWS or UNIX shell commands to be given from APL. `)SH` is a synonym of `)CMD`. Either command may be given in either environment (Windows or UNIX) with exactly the same effect. `)SH` is probably more natural for the UNIX user. This section describes the behaviour of `)SH` and `)CMD` under UNIX. See *Section 7.2.3* for a discussion of their behaviour under Windows.

The system functions *Section 6.4.131* and *Section 6.4.17* provide similar facilities but may be executed from within APL code. For further information, see *Section 6.4.131* and *Section 6.4.17*.

`)SH` allows UNIX shell commands to be given from APL. The argument must be entered in the appropriate case (usually lower-case). The result of the command, if any, is displayed.

`)SH` causes Dyalog to invoke the `system()` library call. The shell which is used to run the command is therefore the shell which `system()` is defined to call. For example, under AIX this would be `/usr/bin/sh`.

When the shell is closed, control returns to APL. See *Dyalog for UNIX UI Guide* for further information.

The parameters CMD_PREFIX and CMD_POSTFIX may be used to execute a different shell under the shell associated with `system()`.

Example

```
      )SH ps -u andys | grep -v ps
  UID      PID    TTY  TIME CMD
 6179  9437326  pts/0  0:00 ksh
 6179 10223736  pts/0  0:00 dyalog
 6179 10354810  pts/0  0:00 sh
 6179 10879188  pts/0  0:00 ksh
 6179 11665660      -  0:00 sshd
```

> **Note**
>
> This function is disabled and instead generates a `DOMAIN ERROR` if the RIDE_SPAWNED parameter is non-zero. This is designed to prevent it being invoked from a Ride session which does not support this type of user interface. For further details, see the Ride User Guide.

7.2.26 State Indicator `)SI {n} {-tid=tn}`

This command displays the contents of the state indicator in the active workspace. The state indicator identifies those operations which are suspended or pendent for each suspension.

The optional parameter `n` specifies that only the first `n` or last `¯n` lines of the output should be displayed.

The optional parameter `-tid=tn` specifies that the state indicator is to be displayed only for thread number `tn`.

The list consists of a line for each suspended or pendent operation beginning with the most recently suspended function or operator. Each line may be:

- The name of a defined function or operator, followed by the line number at which the operation is halted, and followed by the `*` symbol if the operation is suspended. The name of the function or operator is its full pathname relative to the root namespace `#`. For example, `#.UTIL.PRINT`. In addition, the display of a function or operator which has dynamically changed space away from its origin is prefixed with its current space. For example, `[⎕SE] TRAV`.
- A primitive operator symbol.
- The Execute function symbol (`⍎`).
- The Evaluated Input symbol (`⎕`).
- The System Function `⎕DQ` or `⎕SR` (occurs when executing a callback function).

Examples

```
      )SI
#.PLUS[2]*
.
#.MATDIV[4]
#.FOO[1]*
⍏
```

This example indicates that at some point function `FOO` was executed and suspended on line 1. Subsequently, function `MATDIV` was invoked, with a function derived from the Inner Product or Outer Product operator (`.`) having defined function `PLUS` as an operand.

In the following, function `foo` in namespace `x` has called `goo` in namespace `y`. Function `goo` has then changed space (`)CS`) to namespace `z` where it has been suspended:

```
      )SI
[z] y.goo[2]*
x.foo[1]
```

Threads

In a multi-threading application, where parent threads spawn child threads, the state indicator assumes the structure of a branching tree. Branches of the tree are represented by indenting lines belonging to child threads. For example:

```
      )SI
·   #.Calc[1]
&5
·   ·   #.DivSub[1]
·   &7
·   ·   #.DivSub[1]
·   &6
·   #.Div[2]*
&4
#.Sub[3]
#.Main[4]
```

Here, `Main` has called `Sub`, which has spawned threads `4` and `5` with functions: `Div` and `Calc`. Function `Div`, after spawning `DivSub` in each of threads `6` and `7`, has been suspended at line [2].

The state indicator for a particular thread `tn` may be displayed by specifying the parameter `-tid=tn`.

```
      ⎕←foo&¨10 10 10 10
┌→──────────┐
│9 10 11 12│
└~──────────┘
       )SI
·   #.foo[1]
&9
·   #.foo[1]
&10
·   #.foo[1]
&11
·   #.foo[1]
&12
      )SI -tid=11
#.foo[1]
```

7.2.27 Clear State Indicator)SIC

This command is a synonym for `)RESET`. See *Section 7.2.23*.

7.2.28 State Indicator & Name List)SINL

This command displays the contents of the state indicator together with local names. The display is the same as for `)SI` (see above) except that a list of local names is appended to each defined function or operator line.

EXAMPLE

```
      )SINL
#.PLUS[2]*        B       A       R       DYADIC  END
.
#.MATDIV[4]       R       END     I       J       ⎕TRAP
#.FOO[1]* R
⍝
```

7.2.29 Thread Identity)TID {tid}

`)TID` associates the Session window with the specified thread so that expressions that you subsequently execute in the Session are executed in the context of that thread.

If you attempt to `)TID` to a thread that is paused or running, that thread will, if possible, be interrupted by a strong interrupt. If the thread is in a state which it would be inappropriate to interrupt (for example, if the thread is executing an external function), the system reports:

```
      Can't switch, this thread is n
```

If no thread number is given, `)TID` reports the number of the current thread.

Examples

```
      ⍝ State indicator
      )SI
·   #.print[1]
&3
·   ·   #.sub_calc[2]*
·   &2
·   #.calc[1]
&1

      ⍝ Current thread
      )TID
is 2

      ⍝ Switch suspension to thread 3
      )TID 3
was 2

      ⍝ State indicator
      )SI
·   #.print[1]*
&3
·   ·   #.sub_calc[2]
·   &2
·   calc[1]
&1

      ⍝ Attempt to switch to pendent thread 1
      )TID 1
Can't switch, this thread is 3
```

7.2.30 List Global Defined Variables)VARS {nm}

This command displays the names of global defined variables in the active workspace or current namespace. Names are displayed in ⎕AV collation order. If a name is included after the command, only those names starting at or after the given name in collation order are displayed.

Examples

```
      )VARS
A  B  F  TEMP VAR

      )VARS F
F TEMP VAR
```

7.2.31 Workspace Identification)WSID {ws}

This command displays or sets the name of the active workspace.

If a workspace name is not specified,)WSID reports the name of the current active workspace. The name reported is the full path name, including directory references.

If a workspace name is given, the current active workspace is renamed accordingly. The previous name of the active workspace (excluding directory references) is reported. See *Programming: Workspaces* for the rules for specifying a workspace name.

Examples

```
      )LOAD WS/TEMP
WS/TEMP saved Thu Sep 17 10:32:19 1998

      )WSID
is WS/TEMP

      )WSID WS/KEEP
was WS/TEMP

      )WSID
WS/KEEP
```

7.2.32 Load without Latent Expression)XLOAD {ws}

This command causes the named stored workspace to be loaded. The current active workspace is lost.

)XLOAD is identical in effect to)LOAD except that)XLOAD does **not** cause the expression defined by the latent expression □LX in the saved workspace to be executed.

8 Appendices

8.1 PCRE Overview

PCRE (Perl Compatible Regular Expressions) is an *open source* library used by the ⎕R and ⎕S system operators. The regular expression syntax which the library supports is not unique to APL nor is it an integral part of the language.

The Dyalog interpreter now supports PCRE2. For more information about PCRE2, see the full syntax and semantics and quick-reference pages.