



The tool of thought for expert programming

Dyalog™ for Windows

Language Reference

Version: 14.0

Dyalog Limited

email: support@dyalog.com

<http://www.dyalog.com>

Dyalog is a trademark of Dyalog Limited

Copyright © 1982-2014 by Dyalog Limited

All rights reserved.

Version: 14.0

Revision: 20150120

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

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose.

Dyalog Limited reserves the right to revise this publication without notification.

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.

UNIX is a registered trademark of The Open Group.

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

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Array Editor is copyright of davidliebtog.com

All other trademarks and copyrights are acknowledged.

Contents

Chapter 1: Primitive Functions	1
Key to Notation	1
Migration Level	1
Scalar Functions	2
Mixed Functions	5
Conformability	8
Fill Elements	8
Axis Operator	9
Functions (A-Z)	9
Abort	10
Add	11
And, Lowest Common Multiple	12
Assignment	13
Assignment (Indexed)	16
Assignment (Selective)	21
Binomial	23
Branch	24
Catenate/Laminate	26
Catenate First	28
Ceiling	28
Circular	29
Conjugate	30
Deal	30
Decode	31
Depth	33
Direction (Signum)	34
Disclose	35
Divide	36
Drop	37
Drop with Axes	38
Enclose	39
Enclose with Axes	40
Encode	41
Enlist	43
Equal	44
Excluding	45
Execute (Monadic)	46
Execute (Dyadic)	46
Expand	47
Expand First	48

Exponential	48
Factorial	48
Find	49
First	50
Floor	50
Format (Monadic)	51
Format (Dyadic)	55
Grade Down (Monadic)	57
Grade Down (Dyadic)	58
Grade Up (Monadic)	60
Grade Up (Dyadic)	62
Greater	63
Greater Or Equal	64
Identity	64
Index	65
Index with Axes	68
Index Generator	69
Index Of	70
Indexing	73
Intersection	77
Left	78
Less	79
Less Or Equal	79
Logarithm	80
Magnitude	80
Match	81
Matrix Divide	82
Matrix Inverse	84
Maximum	85
Membership	85
Minimum	85
Minus	85
Mix	86
Multiply	91
Nand	91
Natural Logarithm	92
Negative	92
Nor	92
Not	93
Not Equal	93
Not Match	94
Or, Greatest Common Divisor	95
Partition	96
Partitioned Enclose	98
Pi Times	99
Pick	99
Plus	100
Power	100

Ravel	101
Ravel with Axes	101
Reciprocal	104
Replicate	104
Reshape	106
Residue	106
Reverse	107
Reverse First	107
Right	107
Roll	108
Rotate	109
Rotate First	110
Same	110
Shape	111
Split	112
Subtract	112
Table	113
Take	114
Take with Axes	115
Tally	116
Times	116
Transpose (Monadic)	116
Transpose (Dyadic)	117
Type	118
Union	118
Unique	119
Without	119
Zilde	119
Chapter 2: Primitive Operators	121
Operator Syntax	121
Axis Specification	123
Operators (A-Z)	124
Assignment (Modified)	124
Assignment (Indexed Modified)	125
Assignment (Selective Modified)	126
Axis (with Monadic Operand)	127
Axis (with Dyadic Operand)	128
Commute	131
Composition (Form I)	132
Composition (Form II)	133
Composition (Form III)	134
Composition (Form IV)	134
Each (with Monadic Operand)	135
Each (with Dyadic Operand)	136
I-Beam	137
Inner Product	138

Key	139
Outer Product	143
Power Operator	144
Rank	146
Reduce	149
Reduce First	151
Reduce N-Wise	151
Scan	152
Scan First	153
Spawn	154
Variant	155
 Chapter 3: The I-Beam Operator	 159
I-Beam	159
Inverted Table Index Of	161
Unsqueeze Type	163
Syntax Colouring	164
Compress Vector of Short Integers	165
Serialise/Deserialise Array	167
Number of Threads	168
Parallel Execution Threshold	168
Thread Synchronisation Mechanism	168
Update Function Time Stamp	169
Memory Manager Statistics	170
Specify Workspace Available	173
Update DataTable	174
Read DataTable	177
Data Binding	180
Flush Session Caption	186
Close All Windows	187
Export To Memory	187
Set Workspace Save Options:	187
Expose Root Properties	188
Disable Component Checksum Validation	189
Fork New Task (UNIX only)	190
Change User (UNIX only)	191
Reap Forked Tasks (UNIX only)	192
Signal Counts (UNIX only)	194
Random Number Generator	195
 Chapter 4: System Functions	 197
System Constants	199
System Variables	200
System Operators	202
System Namespaces	202
System Functions Categorised	203

Character Input/Output	213
Evaluated Input/Output	215
Underscored Alphabetic Characters	217
Alphabetic Characters	217
Account Information	218
Account Name	218
Arbitrary Output	219
Attributes	220
Atomic Vector	224
Atomic Vector - Unicode	224
Base Class	227
Class	228
Clear Workspace	230
Execute Windows Command	231
Start Windows Auxiliary Processor	235
Canonical Representation	236
Change Space	238
Comparison Tolerance	241
Copy Workspace	242
Digits	244
Decimal Comparison Tolerance	244
Display Form	245
Division Method	248
Delay	249
Diagnostic Message	249
Extended Diagnostic Message	250
Dequeue Events	255
Data Representation (Monadic)	258
Data Representation (Dyadic)	259
Edit Object	260
Event Message	261
Event Number	261
Exception	262
Expunge Object	263
Export Object	265
File Append Component	266
File System Available	266
File Check and Repair	267
File Copy	270
File Create	271
File Drop Component	273
File Erase	274
File History	274
File Hold	276
Fix Script	277
Component File Library	278
Format (Monadic)	279
Format (Dyadic)	280

File Names	287
File Numbers	288
File Properties	289
Floating-Point Representation	293
File Read Access	295
File Read Component Information	296
File Read Components	297
File Rename	298
File Replace Component	299
File Resize	300
File Size	301
File Set Access	301
File Share Tie	302
Exclusive File Tie	304
File Untie	305
Fix Definition	305
Instances	306
Index Origin	307
Key Label	308
Line Count	308
Load Workspace	309
Lock Definition	310
Latent Expression	311
Map File	311
Migration Level	313
Set Monitor	315
Query Monitor	316
Name Association	317
Native File Append	345
Name Classification	346
Native File Create	357
Native File Erase	357
New Instance	358
Name List	359
Native File Lock	363
Native File Names	365
Native File Numbers	365
Enqueue Event	366
Nested Representation	368
Native File Read	369
Native File Rename	371
Native File Replace	371
Native File Resize	372
Create Namespace	373
Namespace Indicator	375
Native File Size	375
Native File Tie	376
Null Item	377

Native File Untie	378
Native File Translate	378
Sign Off APL	379
Variant	379
Object Representation	380
Search Path	384
Program Function Key	386
Print Precision	387
Profile Application	388
Print Width	395
Cross References	396
Replace	397
Random Link	416
Space Indicator	418
Response Time Limit	419
Search	419
Save Workspace	419
Screen Dimensions	420
Session Namespace	420
Execute (UNIX) Command	421
Start UNIX Auxiliary Processor	422
State Indicator	423
Shadow Name	424
Signal Event	425
Size of Object	428
Screen Map	429
Screen Read	432
Source	436
State Indicator Stack	437
State of Object	439
Set Stop	440
Query Stop	441
Set Access Control	442
Query Access Control	443
Shared Variable Offer	444
Query Degree of Coupling	446
Shared Variable Query	447
Shared Variable Retract Offer	447
Shared Variable State	448
Terminal Control	449
Thread Child Numbers	450
Get Tokens	450
This Space	452
Current Thread Identity	453
Kill Thread	453
Current Thread Name	454
Thread Numbers	454
Token Pool	454

Put Tokens	455
Set Trace	456
Query Trace	457
Trap Event	458
Token Requests	462
Time Stamp	463
Wait for Threads to Terminate	464
Unicode Convert	465
Using (Microsoft .NET Search Path)	468
Vector Representation	469
Verify & Fix Input	470
Workspace Available	471
Windows Create Object	472
Windows Get Property	475
Windows Child Names	476
Windows Set Property	477
Workspace Identification	478
Window Expose	479
XML Convert	480
Extended State Indicator	495
Set External Variable	496
Query External Variable	498
 Chapter 5: System Commands	 499
Introduction	499
List Classes	501
Clear Workspace	501
Windows Command Processor	502
Save Continuation	503
Copy Workspace	504
Change Space	506
Drop Workspace	506
Edit Object	507
Erase Object	508
List Events	508
List Global Defined Functions	509
Display Held Tokens	510
List Workspace Library	511
Load Workspace	512
List Methods	513
Create Namespace	513
List Global Namespaces	514
List Global Namespaces	514
Sign Off APL	514
List Global Defined Operators	514
Protected Copy	515
List Properties	516

Reset State Indicator	516
Save Workspace	516
Execute (UNIX) Command	518
State Indicator	519
Clear State Indicator	520
State Indicator & Name List	520
Thread Identity	521
List Global Defined Variables	522
Workspace Identification	522
Load without Latent Expression	523
 Appendices: PCRE Specifications	525
Appendix A - PCRE Syntax Summary	526
Appendix B - PCRE Regular Expression Details	533
 Symbolic Index	571
 Index	579

Chapter 1:

Primitive Functions

Key to Notation

The following definitions and conventions apply throughout this manual:

f	A function, or an operator's left argument when a function.
g	A function, or an operator's right argument when a function.
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.

Migration Level

ML determines the degree of migration of the Dyalog APL language towards IBM's APL2. Unless otherwise stated, the manual assumes **ML** has a value of 1.

Scalar Functions

There is a class of primitive functions termed scalar functions. This class is identified in [Table 1](#) below. Scalar functions are **pervasive**, i.e. their properties apply at all levels of nesting. Scalar functions have the following properties:

Table 1: Scalar Primitive Functions

Symbol	Monadic	Dyadic
+	Identity	Plus (Add)
-	Negative	Minus (Subtract)
×	Direction (Signum)	Times (Multiply)
÷	Reciprocal	Divide
	Magnitude	Residue
⌊	Floor	Minimum
⌈	Ceiling	Maximum
*	Exponential	Power
⊗	Natural Logarithm	Logarithm
∘	Pi Times	Circular
!	Factorial	Binomial
~	Not	\$
?	Roll	\$
€	Type (See Enlist)	\$

Symbol	Monadic	Dyadic
\wedge		And
\vee		Or
∇		Nand
$\nabla\vee$		Nor
$<$		Less
\leq		Less Or Equal
$=$		Equal
\geq		Greater Or Equal
$>$		Greater
\neq		Not Equal
\$ Dyadic form is not scalar		

Monadic Scalar Functions

- The function is applied independently to each simple scalar in its argument.
- The function produces a result with a structure identical to its argument.
- When applied to an empty argument, the function produces an empty result. With the exception of $+$ and ϵ , the type of this result depends on the function, not on the type of the argument. By definition $+$ and ϵ return a result of the same type as their arguments.

Example

```

      ÷2 (1 4)
0.5  1 0.25

```

Dyadic Scalar Functions

- The function is applied independently to corresponding pairs of simple scalars in its arguments.
- A simple scalar will be replicated to conform to the structure of the other argument. If a simple scalar in the structure of an argument corresponds to a non-simple scalar in the other argument, then the function is applied between the simple scalar and the items of the non-simple scalar. Replication of simple scalars is called scalar extension.
- A simple unit is treated as a scalar for scalar extension purposes. A unit is a single element array of any rank. If both arguments are simple units, the argument with lower rank is extended.
- The function produces a result with a structure identical to that of its arguments (after scalar extensions).
- If applied between empty arguments, the function produces a composite structure resulting from any scalar extensions, with type appropriate to the particular function. (All scalar dyadic functions return a result of numeric type.)

Examples

```
2 3 4 + 1 2 3
3 5 7
```

```
2 (3 4) + 1 (2 3)
3 5 7
```

```
(1 2) 3 + 4 (5 6)
5 6 8 9
```

```
10 × 2 (3 4)
20 30 40
```

```
2 4 = 2 (4 6)
1 1 0
```

```
(1 1p5) - 1 (2 3)
4 3 2
```

```
1↑''+ι0
0
1↑(0p<' '(0 0))×''
0 0 0
```

Note: The Axis operator applies to all scalar dyadic functions.

Mixed Functions

Mixed rank functions are summarised in [Table 2](#). For convenience, they are subdivided into five classes:

Table 2: Mixed rank functions

Structural	These functions change the structure of the arguments in some way.
Selection	These functions select elements from an argument.
Selector	These functions identify specific elements by a Boolean map or by an ordered set of indices.
Miscellaneous	These functions transform arguments in some way, or provide information about the arguments.
Special	These functions have special properties.

In general, the structure of the result of a mixed primitive function is different from that of its arguments.

Scalar extension may apply to some, but not all, dyadic mixed functions.

Mixed primitive functions are not pervasive. The function is applied to elements of the arguments, not necessarily independently.

Examples

```

      'CAT' 'DOG' 'MOUSE' 1< 'DOG'
2
      3↑ 1 'TWO' 3 'FOUR'
1 TWO 3

```

In the following tables, note that:

- `[]` Implies axis specification is optional
- `$` This function is in another class

Table 3: Structural Primitive Functions

Symbol	Monadic	Dyadic
ρ	\$	Reshape
$,$	Ravel []	Catenate/Laminate[]
$\overline{\cdot}$	Table	Catenate First / Laminate []
ϕ	Reverse []	Rotate []
Θ	Reverse First []	Rotate First []
Φ	Transpose	Transpose
\uparrow	Mix/Disclose (First) []	\$
\downarrow	Split []	\$
\subset	Enclose []	Partitioned Enclose []
ϵ	Enlist (See Type)	\$

Table 4: Selection Primitive Functions

Symbol	Monadic	Dyadic
\supset	Disclose /Mix	Pick
\uparrow	\$	Take []
\downarrow	\$	Drop []
$/$		Replicate []
∇		Replicate First []
\backslash		Expand []
$\backslash\downarrow$		Expand First []
\sim	\$	Without (Excluding)
\cap		Intersection
\cup	Unique	Union
\neg	Same	Left
\vdash	Identity	Right

Table 5: Selector Primitive Functions

Symbol	Monadic	Dyadic
ι	Index Generator	Index Of
ϵ	\$	Membership
Δ	Grade Up	Grade Up
Ψ	Grade Down	Grade Down
$?$	\$	Deal
$\underline{\epsilon}$		Find

Table 6: Miscellaneous Primitive Functions

Symbol	Monadic	Dyadic
ρ	Shape	\$
\equiv	Depth	Match
\neq	Tally	Not Match
$\underline{\epsilon}$	Execute	Execute
Φ	Format	Format
\perp		Decode (Base)
\top		Encode (Representation)
\boxdiv	Matrix Divide	Matrix Inverse

Table 7: Special Primitive Functions

Symbol	Monadic	Dyadic
\rightarrow	Abort	
\rightarrow	Branch	
\leftarrow	\$	Assignment
$[I]\leftarrow$	\$	Assignment(Indexed)
$(I)\leftarrow$		Assignment(Selective)
$[]$		Indexing

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 ($\llcorner Y$ for array Y). The Type function (ϵ) replaces a numeric value with zero and a character value with ' '.
 The Disclose function (\rhd) returns the first item of an array. If the array is empty, $\rhd Y$ 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 (\backslash or \backslash), Replicate ($/$ or $/$), Reshape (ρ) and Take (\uparrow).

Examples

```

       $\epsilon \iota 5$ 
0 0 0 0 0

       $\epsilon \rhd (\iota 3) ('ABC')$ 
0 0 0

       $\llcorner \epsilon \rhd (\iota 3) ('ABC')$ 
0 0 0

       $\llcorner \epsilon \rhd \llcorner (\iota 3) ('ABC')$ 
0 0 0

      A  $\leftarrow$  'ABC' (1 2 3)
      A  $\leftarrow$  OpA
       $\llcorner \epsilon \rhd A$ 

      ' '  $\llcorner \epsilon \rhd A$ 
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 2p16
1 2
5 6
```

```
1 2 3+[2]2 3p10 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.

Functions (A-Z)

Scalar and mixed primitive functions are presented in alphabetical order of their descriptive names as shown in Figures 3(i) and 3(ii) respectively. Scalar functions are described in terms of single element arguments. The rules for extension are defined at the beginning of this chapter.

The class of the function is identified in the heading block. The valence of the function is implied by its syntax in the heading block.

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 [4](#). 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)ρ0 ♦ 'ACCOUNT NOT AUTHORISED' ♦ →
      ▽

      VALIDATE
ACCOUNT NOT AUTHORISED

      1↑□AI
52

```

Add **$R \leftarrow X + Y$**

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 Plus.

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
```

And, Lowest Common Multiple

 $R \leftarrow X \wedge Y$

Case 1: 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

```

Case 2: 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` is an implicit argument.

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

```


Assignment

$X \leftarrow Y$

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 .

The assignment arrow (or specification arrow) is often read as 'Is' or 'Gets'.

Examples

```

      A ← 2.3
      A
2.3

```

```

      A ← 1 3
      A
1 2 3

```

More than one name may be specified in X by using vector notation. If so, Y must be a vector or a scalar. If Y is a scalar, its value is assigned to all names in X . If Y is a vector, each element of Y is assigned to the corresponding name in X .

Examples

```

      A B ← 2
      A
2

```

```

      B
2

```

```

      P □ IO Q ← 'TEXT' 1 (1 2 3)
      P
TEXT
      □ IO
1
      Q
1 2 3

```

For compatibility with IBM's APL2, the list of names specified in X may be enclosed in parentheses.

Examples

```
(A B C)←1 2 3
(D E)←'Hello' 'World'
```

Multiple assignments are permitted. The value of Y is carried through each assignment:

```
I←J←K←0
0 0 0 I,J,K
```

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←{ (+/ω) ÷ ρω }
```

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 ''

9      N ← ns1
      □ NC 'N'          A name class of a scalar ref

9      F ← f1
      □ NC 'F'          A name class of a scalar ref

9      refs ← N F        A vector of refs.
      □ NC 'refs'       A 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

Assignment (Indexed)

 $\{R\} \leftarrow X[I] \leftarrow Y$

Indexed Assignment is the Assignment function modified by the Indexing function. The phrase $[I] \leftarrow$ 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 $(EXP\ X)[I] \leftarrow Y$, see [Assignment \(Selective\) on page 21](#). 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.

$\square IO$ 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 $1:pR$. Elements of X identified by index positions I are replaced by corresponding elements of Y .

Examples

```

      +A←15
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 3ρ'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 $\iota(\rho X)[K]$ is implied:

```

      C[;1;2 3]←2 2ρ112 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 2ρ(1 3)(2 4)(2 1)(1 4)]←2 2ρ103 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[←⍷]←←'VECTOR' ♦ S
VECTOR
      S[←⍷]←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

      ⍷ρC
1 1 1 2 1 3 1 4
2 1 2 2 2 3 2 4

      C[1 1⍷⍷ρC]←1 2 ♦ C
1 12 13 14
21 2 23 24

      C[2 1⍷⍷ρC]←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 3p16)(2 2p'SMITH' 'JONES' 'SAM' 'BILL')
```

```
      D
1 2 3 SMITH JONES
4 5 6 SAM    BILL
```

```
≡J←c2 (1 2)
-3
```

```
      D[J]←c'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[c2 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:

```
c2 1 ↔ c(c2), (c1)
```

Note that for any array **A**, **A[c0]** represents a scalar quantity, which is the whole of **A**, so:

```
      A←5p0
      A
0 0 0 0 0
      A[c0]←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 $(EXP\ X)[I] \leftarrow Y$.

```

MAT←4 3p'Hello' 'World'
(2↑MAT)[1 2;]←'#'
MAT
##llo ##rld ##llo
##rld ##llo ##rld
Hello World Hello
World Hello World

MAT←4 3p'Hello' 'World'
□ML←1 A ∈ is Enlist
(∈MAT)[2×ι[0.5×p∈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#

```


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 [Assignment \(Indexed\) on page 16](#). **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**.

The following functions may appear in the selection expression. Where appropriate these functions may be used with axis **[]** and with the Each operator **⋅**.

Functions for Selective Assignment

↑	Take
↓	Drop
,	Ravel
;	Table
φ	Reverse, Rotate
ρ	Reshape
⊃	Disclose, Pick
⊔	Transpose (Monadic and Dyadic)
/	Replicate
\	Expand
⌈	Index
∈	Enlist ($\boxed{\text{ML}} \geq 1$)

Note: Mix and Split (monadic **↑** and **↓**), Type (monadic **∈** when $\boxed{\text{ML}} < 1$) and Membership (dyadic **∈**) may not be used in the selection expression.

Examples

```
A←'HELLO'
((A∈'AEIOU')/A)←'★'
```

```
A
H★LL★
```

```
Z←3 4p⋅12
(5↑,Z)←0
```

```
Z
0 0 0 0
0 6 7 8
9 10 11 12
```

```

MAT←3 3p19
(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∈``c'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 3p'Hello' 'World'
(⊖2↑``MAT[;1 3])←'$'
MAT
Hel$$ World Hel$$
Wor$$ Hello Wor$$
Hel$$ World Hel$$
Wor$$ Hello Wor$$

```

Binomial

$R \leftarrow X!Y$

X and Y may be any numbers except that if Y is a negative integer then X must be a whole number (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 for positive integer arguments:

$$X!Y \leftrightarrow (!Y) \div (!X) \times !Y - X$$

For other arguments, results are derived smoothly from the Beta function:

$$\text{Beta}(X, Y) \leftrightarrow \div Y \times (X-1)!X+Y-1$$

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

```

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
→t0	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.
→TESTρL1	Similar to above.
TEST↑L1	Similar to above.
→L1ρ~TEST	Similar to above.
→L1[ιTEST	Similar to above but only if $\Box IO \leftrightarrow 1$.
→L1×ιTEST	Similar to above but only if $\Box IO \leftrightarrow 1$.
→(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.
→NϕL1,L2,L3	Unconditional branch to the first label after rotation.

A branch expression may occur within a statement including ♦ separators:

```

[5]      →NEXTρ~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
```

Catenate/Laminate

$R \leftarrow X, [K]Y$

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 $R \leftarrow X \overline{Y}$ 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 **lamine**.

Catenation with Integer or Implied Axis Specification

The arrays X and Y are joined along the required axis to form array R . A scalar or 1-element vector 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 4p 'THISWEEK') 5 = '
THIS
WEEK
====

      S,[1]+7S+2 3p16
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 K th axis. The new axis has a length of 2. K must exceed $\square IO$ (the index origin) minus 1, and K must be less than $\square IO$ plus the greater of the ranks of X and Y . A scalar or 1-element vector 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
-----

```

Catenate First **$R \leftarrow X; [K] Y$**

The form $R \leftarrow X; Y$ implies catenation along the first axis whereas the form $R \leftarrow X, Y$ implies catenation along the last axis (columns). See [Catenate/Laminate above](#).

Ceiling **$R \leftarrow \lceil Y$**

Ceiling is defined in terms of Floor as $\lceil Y \leftrightarrow -\lfloor -Y$

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 [Floor on page 50](#).

\square_{CT} is an implied argument of Ceiling.

Circular

 $R \leftarrow X \circ Y$

Y must be numeric. X must be an integer in the range $-12 \leq X \leq 12$. 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.

$(-X) \circ Y$	X	$X \circ Y$
$(1-Y^2)^{.5}$	0	$(1-Y^2)^{.5}$
$\text{Arcsin } Y$	1	$\text{Sine } Y$
$\text{Arccos } Y$	2	$\text{Cosine } Y$
$\text{Arctan } Y$	3	$\text{Tangent } Y$
$(Y+1) \times ((Y-1) \div Y+1) \times 0.5$	4	$(1+Y^2)^{.5}$
$\text{Arcsinh } Y$	5	$\text{Sinh } Y$
$\text{Arccosh } Y$	6	$\text{Cosh } Y$
$\text{Arctanh } Y$	7	$\text{Tanh } Y$
$-8 \circ Y$	8	$(-1+Y^2)^{.5}$
Y	9	a
$+Y$	10	$ Y $
$Y \times 0J1$	11	b
$*Y \times 0J1$	12	θ

Examples

```
0 -1 o 1
0 1.570796327
```

```
1o(PI+o1)÷2 3 4
1 0.8660254038 0.7071067812
```

```
2oPI÷3
0.5
9 11o3.5J-1.2
3.5 -1.2
```

```
9 11o.03.5J-1.2 2J3 3J4
3.5 2 3
-1.2 3 4
```

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.

Note that if **Y** is nested, the function has to process the entire array in case any item is complex.

Examples

```

      +3j4
3J-4
      +1j2 2j3 3j4
1J-2 2J-3 3J-4

      3j4++3j4
6
      3j4×+3j4
25

      +A←ι5
1 2 3 4 5

      +□EX'A'
1

```

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 $X \leq Y$.

R is an integer vector obtained by making **X** random selections from ιY 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 Generator on page 195](#) and [Random Link on page 416](#).

Decode

$R \leftarrow X \downarrow Y$

`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:

$$pR \leftrightarrow (\neg 1 \downarrow pX), 1 \downarrow pY$$

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):

$$Y[1]X^{n-1} + Y[2]X^{n-2} + \dots + Y[n]X^0$$

Examples

```

      2 1 1 2 3 4
26    3 1 1 2 3 4
58    1j1 1 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 1p2 10
2
10

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

```

Depth

 $(\square ML)$ $R \leftarrow \equiv Y$

Y may be any array. R is the 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 $\square ML < 2$ and Y is not of uniform depth then R is negated.

If $\square ML < 2$, a negative value of R indicates non-uniform depth.

Examples

```

       $\equiv 1$ 
0
       $\equiv 'A'$ 
0
       $\equiv 'ABC'$ 
1
       $\equiv 1 \ 'A'$ 
1

       $\square ML \leftarrow 0$ 

       $\equiv A \leftarrow (1 \ 2)(3 \ (4 \ 5))$  A Non-uniform array
-3
       $\equiv ''A$  A A[1] is uniform, A[2] is non-uniform
1 -2
       $\equiv ''''A$ 
0 0 0 1

       $\square ML \leftarrow 2$ 

       $\equiv A$ 
3
       $\equiv ''A$ 
1 2
       $\equiv ''''A$ 
0 0 0 1
```

Direction (Signum) **$R \leftarrow \times Y$**

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 $Y \div |Y|$.

Examples

```

      ×-15.3 0 101
-1 0 1

```

```

      ×3j4 4j5
0.6J0.8 0.6246950476J0.7808688094

```

```

      {ω÷|ω}3j4 4j5
0.6J0.8 0.6246950476J0.7808688094

```

```

      |×3j4 4j5
1 1

```

Disclose**(\square ML)** **$R \leftrightarrow Y$ or $R \leftrightarrow Y$**

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

If $\square ML < 2$, Disclose is represented by the symbol: \rightarrow .

If $\square ML \geq 2$, Disclose is represented by the symbol: \uparrow .

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 .

Disclose is the inverse of Enclose. The identity $R \leftrightarrow \rightarrow < R$ holds for all R . Disclose is also referred to as First.

Examples

```

1       $\rightarrow 1$ 

2       $\rightarrow 2 \ 4 \ 6$ 

       $\rightarrow$  'MONDAY' 'TUESDAY'
MONDAY

1  2  3   $\rightarrow (1 \ (2 \ 3)) (4 \ (5 \ 6))$ 

0       $\rightarrow \iota 0$ 

1       $\rightarrow$  ' '=>' '

0  0  0   $\rightarrow 1 \downarrow < 1, < 2 \ 3$ 
```

Divide **$R \leftarrow X \div Y$**

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 $\square DIV$ is an implicit argument of Divide.

If $\square DIV=0$ and $Y=0$ then if $X=0$, the result of $X \div Y$ is 1; if $X \neq 0$ then $X \div Y$ is a **DOMAIN ERROR**.

If $\square DIV=1$ and $Y=0$, the result of $X \div Y$ 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-11.25 20J25
```

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


Drop **$R \leftarrow X \downarrow Y$**

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 $(\rho X) \rho 1$. 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 I th 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 I th axis, the result is an empty array with zero length along that axis.

Examples

```

BOARD      4↓ 'OVERBOARD'

OVER      -5↓ 'OVERBOARD'

0         ρ10↓ 'OVERBOARD'

ONE        M
FAT
FLY        0 -2↓M

O
F
F

ON         -2 -1↓M

FAT        1↓M
FLY

M3←-2 3 4ρ□A

QRST       1 1↓M3
UVWX

ABCD       -1 -1↓M3
EFGH

```

Drop with Axes

$$R \leftarrow X \downarrow [K] Y$$

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 :

$$\rho \rho R \leftrightarrow \rho \rho Y$$

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

$$(\rho R)[,K] \leftrightarrow 0[(\rho Y)[,K] - |,X$$

Examples

```

      1←M←2 3 4p124
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

```

Enclose**R←cY**

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

```

      c1
1
      c' A '
A
      c1 2 3
1 2 3
      c1, c'CAT'
1 CAT
      c2 4p18
1 2 3 4
5 6 7 8
      c10
      c<10
      c<10
10

```

Enclose with Axes

$$R \leftarrow c[K]Y$$

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:

$$\rho R \leftrightarrow (\rho Y)[(\iota \rho R) \sim K]$$

The shape of each element of R is the shape of the K 'th axes of Y :

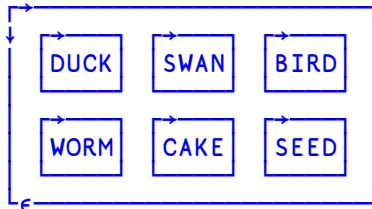
$$\rho \rho R \leftrightarrow (\rho Y)[, K]$$

Examples

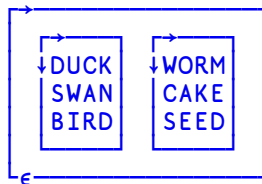
```
]display A←2 3 4p'DUCKSWANBIRDWORMCAKESEED'
```



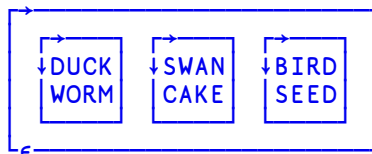
```
]display c[3]A
```



```
]display c[2 3]A
```



```
]display c[1 3]A
```



Encode **$R \leftarrow X \tau Y$**

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 $(\rho X), \rho Y$ (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 $(\times / X) | Y$. 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

```

```

      A75
0 0 0
1 0 0
0 0 0
0 0 0
1 0 0
0 1 0
1 1 4
1 3 11

```

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

Examples

```

      0 12 1.25 10.5
1      10
0.25 0.5

```

```

      4 1313?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

```

Enlist**($\square ML \geq 1$)** **$R \leftarrow \epsilon Y$**

Migration level must be such that $\square ML \geq 1$ (otherwise see [Type on page 118](#)).

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

Examples

```

       $\square ML \leftarrow 1$           a Migration level 1
      MAT  $\leftarrow 2$  2p 'MISS' 'IS' 'SIP' 'PI'  $\diamond$  MAT
MISS   IS
SIP    PI
       $\epsilon$  MAT
MISSISSIPPI

```

```

      M  $\leftarrow 1$  (2 2p2 3 4 5) (6(7 8))
      M
1  2 3 6 7 8
   4 5
       $\epsilon$  M
1 2 3 4 5 6 7 8

```

Equal

$R \leftarrow X = Y$

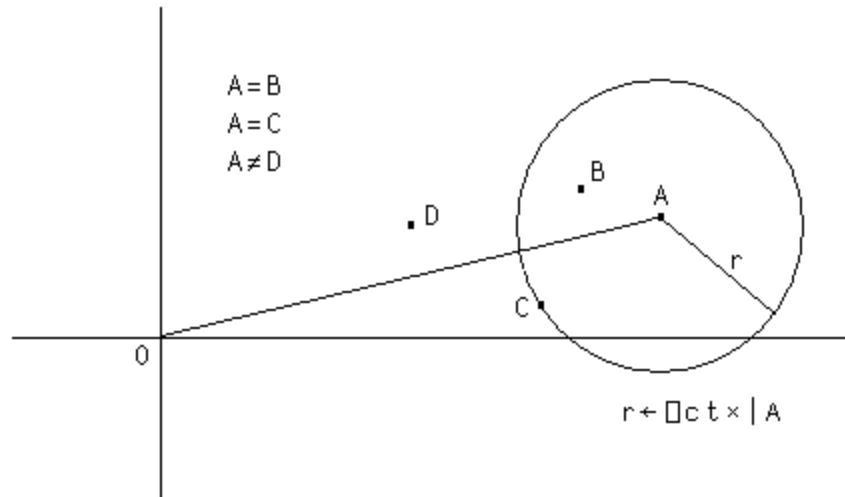
Y may be any array. X may be any array. R is Boolean. $\square CT$ is an implicit argument of Equal.

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 $\square CT \times (|X| \vee |Y|)$.

For complex numbers $X = Y$ is 1 if the magnitude of $X - Y$ does not exceed $\square CT$ 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 $\square CT$ times the larger magnitude.



Examples

```

3=3.1 3 ^-2 ^-3
0 1 0 0

a<-2+0j1×CT
a
2J1E^-14
a=2j.000000000000001 2j.000000000000001
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=1.000000000000001
1

1=1.00000001
0

```

Excluding**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 $X \equiv Y$ returns 1 for those elements.

CT is an implicit argument of **Excluding**. **Excluding** is also known as **Without**.

Examples

```

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

5 10 15~110
15

```

For performance information, see *Programmer's Guide: Search Functions and Hash Tables*.

Execute (Monadic) **$R \leftarrow \underline{Y}$**

Y must be a simple character scalar or vector. If Y is an empty vector, it is treated as an empty character vector. Y is taken to be an APL statement to be executed. R is the result of the last-executed expression. If the expression has no value, then \underline{Y} has no value. If Y is an empty vector or a vector containing only blanks, then \underline{Y} has no value.

If Y contains a branch expression which evaluates to a non-empty result, R does not yield a result. Instead, the branch is effected in the environment from which the Execute was invoked.

Examples

```

       $\underline{2+2}$ 
4
       $4=\underline{2+2}$ 
1
      A
1 2 3
4 5 6
       $\underline{A}$ 
1 2 3
4 5 6
       $\underline{A\leftarrow 2|^{-1}\uparrow\Box TS} \diamond \rightarrow 0\rho\tilde{A} \diamond A$ 
0
      A
0

```

Execute (Dyadic) **$R \leftarrow X \underline{Y}$**

Y must be a simple character scalar or vector. If Y is an empty vector, it is treated as an empty character vector. X must be a namespace reference or a simple character scalar or vector representing the name of a namespace. Y is then taken to be an APL statement to be executed in namespace X . R is the result of the last-executed expression. If the expression has no value, then $X \underline{Y}$ has no value.

Example

```

 $\Box SE \underline{\Box NL 9}$ 

```

Expand **$R \leftarrow X \backslash [K] Y$**

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 $R \leftarrow X \backslash Y$ 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 $+/\uparrow |X|$.

Examples

```

0          0\10
0

          1 -2 3 -4 5\ 'A'
A   AAA          AAAAAA

          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

```

Expand First **$R \leftarrow X \backslash Y$**

The form $R \leftarrow X \backslash Y$ implies expansion along the first axis whereas the form $R \leftarrow X \setminus Y$ implies expansion along the last axis (columns). See [Expand above](#).

Exponential **$R \leftarrow * Y$**

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

Example

```
*1 0
2.718281828 1

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

1+*0j1 A Euler Identity
0
```

Factorial **$R \leftarrow ! Y$**

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 . For non-integral values of Y , $!Y$ is equivalent to the gamma function of $Y+1$.

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
```

Find **$R \leftarrow X \in Y$**

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 .

$\square CT$ and $\square DCT$ are implicit arguments to 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
```

First	(\lfloorML)	$R \leftarrow Y$ or $R \leftarrow \uparrow Y$
--------------	---------------------------------	--

See function [Disclose on page 35](#).

Floor	$R \leftarrow \lfloor Y$
--------------	--

Y must be numeric.

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

Examples

```

 $\lfloor -2.3 \ 0.1 \ 100 \ 3.3$ 
 $-3 \ 0 \ 100 \ 3$ 

```

```

 $\lfloor 0.5 + 0.4 \ 0.5 \ 0.6$ 
 $0 \ 1 \ 1$ 

```

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

```

 $\lfloor 1j3.2 \ 3.3j2.5 \ -3.3j^{-2.5}$ 
 $1j3 \ 3j2 \ -3j^{-3}$ 

```

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

```

 $\nabla \text{ fl} \leftarrow \text{CpxFloor } \text{cpxs}; a; b$ 
[1]  A Complex floor of scalar complex number (a+ib)
[2]  a b  $\leftarrow$  9 11 o c p x s
[3]  :If 1 > (a -  $\lfloor$ a) + b -  $\lfloor$ b
[4]      fl  $\leftarrow$  ( $\lfloor$ a) + 0j1  $\times$   $\lfloor$ b
[5]  :Else
[6]      :If (a -  $\lfloor$ a) < b -  $\lfloor$ b
[7]          fl  $\leftarrow$  ( $\lfloor$ a) + 0j1  $\times$  1 +  $\lfloor$ b
[8]      :Else
[9]          fl  $\leftarrow$  (1 +  $\lfloor$ a) + 0j1  $\times$   $\lfloor$ b
[10]     :EndIf
[11] :EndIf
 $\nabla$ 

```

```

 $\text{CpxFloor}'' 1j3.2 \ 3.3j2.5 \ -3.3j^{-2.5}$ 
 $1j3 \ 3j2 \ -3j^{-3}$ 

```

ϵ_{CT} is an implicit argument of Floor.

Format (Monadic) **$R \leftarrow \text{f} Y$**

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 $\square PPW$. If Y is a simple character array, then R is Y .

Example

```
+B←fA←2 6p '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 $\square PP$. $\square PP$ is ignored when formatting integers.

Examples

```

      □PP←5
0      ρC←f10
      ρC←f10
2      C
10     C
      ρC←f12.34
5      C
12.34
      f123456789
123456789
      f123.456789
123.46
```

Scaled notation is used if the magnitude of the non-integer number is too large to represent with $\square PP$ 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

```

pC←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:

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

```

```

      2 2 29
pC←2 2 3pC

```

```

      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
```

```
4      ρA
```

```
≡A
```

```
3
```

```
ρB
```

```
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 \lceil 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+ι4) (2 2 3⍥'CDEFGHIJKLMN')

      C
1  AB 2 3 CDE
      4 5 FGH

      IJK
      LMN

      ρC
4

      ≡C
-2

      D
1  AB 2 3 CDE
      4 5 FGH

      IJK
      LMN

      ρD
5 16

      ≡D
1

```

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

```

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

```

□PP is an implicit argument of Monadic Format.

Format (Dyadic) **$R \leftarrow X \text{ } \overline{\text{f}} \text{ } Y$**

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 $1 \uparrow \rho \rho Y$ and $\neg 1 \uparrow \rho R$ is $\neg 1 \uparrow \rho Y$. If any element of Y is complex, dyadic $\overline{\text{f}}$ reports a **DOMAIN ERROR**.

Conformability requires that if X has more than two elements, then ρX must be $2 \times \neg 1 \uparrow \rho Y$. If X contains one element, it is extended to $(2 \times \neg 1 \uparrow \rho Y) \rho 0, X$. If X contains 2 elements, it is extended to $(2 \times \neg 1 \uparrow \rho Y) \rho X$.

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 ρ 6
1      2      3
4      5      6

      4 0 ̄ 1.1 2 ̄ 4 2.5 4 7
1      2 ̄ 4      3

```

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

Example

```

      4 1 ̄ 1.1 2 ̄ 4 2.5 4 7
1.1 2.0 ̄ 4.0 2.5

```

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

Example

```

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

```

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.

Example

```

3 2 3p10 15.2346 -17.1 2 3 4
10.000 15.235 -17.100
2.000 3.000 4.000

```

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

Example

```

3 0 6 2 3 2p10.1 15 1001 22.357 101 1110.1
10 15.00
*** 22.36
101*****

```

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

Example

```

26 2*100
1267650600228229_____ . _____
-
p26 2*100
59
0 20 3
0.3333333333333333_____
0 -20 3
3.3333333333333333_____ E-1

```

The shape of R is the same as the shape of Y except that the last dimension of Y 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.

```

p5 2 3 4p124
2 3 20

```

Grade Down (Monadic)

 $R \leftarrow \Psi Y$

Y must be a simple character or simple numeric array of rank greater than 0. R is an integer vector being the permutation of $\uparrow 1 \uparrow \rho Y$ that places the sub-arrays of Y along the first axis in descending order. The indices of any set of identical sub-arrays in Y occur in R in ascending order.

If Y is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

Example

```

      M
2 5 3 2
3 4 1 1
2 5 4 5
2 5 3 2
2 5 3 4

      ΨM
2 3 5 1 4

      M[ΨM; ]
3 4 1 1
2 5 4 5
2 5 3 4
2 5 3 2
2 5 3 2

```

If Y is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in $\Box AV$ (Classic Edition).

$\Box IO$ is an implicit argument of Grade Down.

Note that character arrays sort differently in the Unicode and Classic Editions.

Example

```

      M
Goldilocks
porridge
Porridge
3 bears

```

Unicode Edition	Classic Edition
<pre> ΨM 2 3 1 4 </pre>	<pre> ΨM 3 1 4 2 </pre>
<pre> M[ΨM;] porridge Porridge Goldilocks 3 bears </pre>	<pre> M[ΨM;] Porridge Goldilocks 3 bears porridge </pre>

Grade Down (Dyadic)

$R \leftarrow X \Psi Y$

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 $1 \uparrow \rho Y$ containing the permutation of $1 \uparrow \rho Y$ 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:

$$X \Psi Y \leftrightarrow \Psi X \iota Y$$

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:

abc
ABA

Right argument:

ab
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

```

      pS1
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.

Grade Up (Monadic)**R←▲Y**

Y must be a simple character or simple numeric array of rank greater than 0. R is an integer vector being the permutation of $\iota 1 \uparrow \rho Y$ that places the sub-arrays along the first axis in ascending order.

If Y is a numeric array of rank greater than 1, the elements in each of the sub-arrays along the first axis are compared in ravel order with greatest weight being given to the first element and least weight being given to the last element.

Examples

```
      22.5 1 15 3 -4
5 2 4 3 1
```

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

```
2 3 5
1 2 6
```

```
2 3 4
5 2 4
```

```
      M
3 2 1
```

If `Y` is a character array, the implied collating sequence is the numerical order of the corresponding Unicode code points (Unicode Edition) or the ordering of characters in `AV` (Classic Edition).

`IO` is an implicit argument of `Grade Up`

Note that 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

Grade Up (Dyadic)

 $R \leftarrow X \uparrow Y$

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 $\iota 1 \uparrow p Y$ 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:

$$X \uparrow Y \leftrightarrow \uparrow X \iota Y$$

If X is a higher-rank array, each axis of X represents a grading attribute in increasing order of importance. 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 2p'ABBA') ⍤ 'AB'[:5 2p2] ⍤ A and B are equivalent
t
1 2 3 4 5
```

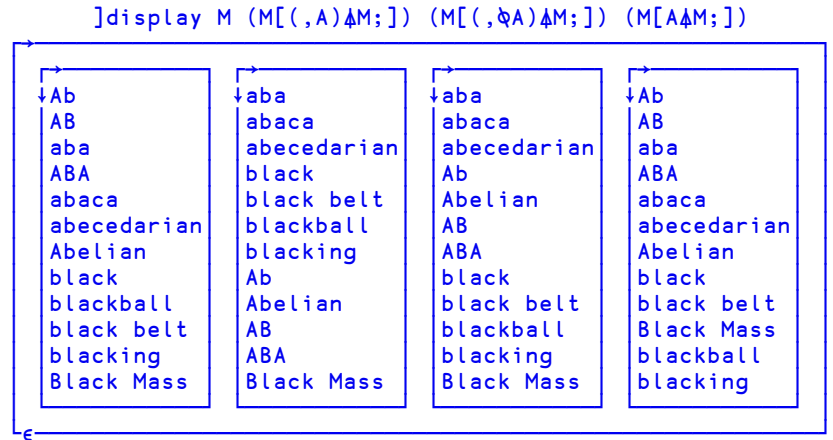
```
]display A←2 14p' abcdegiklmnrt ABCDEGIKLMNRT'
```

```
→
↓ abcdegiklmnrt
  ABCDEGIKLMNRT
```

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

```
]display M←↑V
```

```
→
↓ Ab
  AB
  aba
  ABA
  abaca
  abecedarian
  Abelian
  black
  blackball
  black belt
  blackening
  Black Mass
```



Greater

$R \leftarrow X > Y$

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 is an implicit argument of Greater.

Examples

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

CT←1E-10

```
1 1.00000000001 1.000000001 > 1
0 0 1
```

Greater Or Equal

 $R \leftarrow X \geq Y$

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.

$\square CT$ is an implicit argument of Greater Or Equal.

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

```

Identity

 $R \leftarrow Y$

Y may be any array. The result R is the argument Y .

Example

```

      ⊢ 'abc' 1 2 3
abc 1 2 3

```

Index

 $R \leftarrow \{X\} \boxed{Y}$ **Dyadic case**

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

$$(I \ J \ \dots \ \boxed{Y}) \equiv Y[I;J;\dots]$$

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:

$$(\rho X \boxed{Y}) \equiv \uparrow, / \rho^{\text{``}} X$$

Index is sometimes referred to as *squad indexing*.

Note that index may be used with selective specification.

\boxed{IO} is an implicit argument of index.

Examples

```

      IO←1

      VEC←111 222 333 444
      3⊖VEC
333      (←4 3)⊖VEC
444 333      (←2 3ρ3 1 4 1 2 3)⊖VEC
333 111 444
111 222 333

      MAT←10⊥∘∘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
      ρ(2 1ρ1)(3 4ρ2)⊖MAT
2 1 3 4
      ρ⊖⊖MAT
0 0
      (3(2 1)⊖MAT)←0 ⋄ MAT      A Selective assignment.
11 12 13 14
21 22 23 24
0 0 33 34

```

Monadic case

If Y is an array, Y is returned.

If Y is a ref to an instance of a Class with a Default property, all elements of the Default property are returned. For example, if `Item` is the default property of `MyClass`, and `imc` is an Instance of `MyClass`, then by definition:

```
imc.Item≡[]imc
```

NONCE ERROR is reported if the Default Property is Keyed, because in this case APL has no way to determine the list of all the elements.

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 $\rho, \uparrow, \downarrow, \succ$) as opposed to functions that operate on the *values* of the index set (functions such as $+, \lceil, \lfloor, \rho''$), 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 a Class with a Default Numbered Property, the expression:

```
1↑ϕ[]CompFile
```

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

Note that similarly, the expression

```
10000ρ[]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.

Index with Axes

$$R \leftarrow \{X\} \llbracket K \rrbracket Y$$

X must be a scalar or vector of depth ≤ 2 , of integers each $\geq \llbracket IO \rrbracket$. 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 :

$$(\rho, X) \equiv \rho, K$$

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:

$$I \ J \ \llbracket [1 \ 3] \rrbracket Y \leftrightarrow Y[I; ; J]$$

Note that index with axis may be used with selective specification. $\llbracket IO \rrbracket$ is an implicit argument of index with axis.

Examples

$\llbracket IO \rrbracket \leftarrow 1$

$\llbracket \leftarrow CUBE \leftarrow 10 \rrbracket \llbracket 2 \ 3 \ 4 \rrbracket$

```
111 112 113 114
121 122 123 124
131 132 133 134
```

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

$2 \llbracket [1] \rrbracket CUBE$

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

$2 \llbracket [3] \rrbracket CUBE$

```
112 122 132
212 222 232
```

$CUBE[; ; 2] \equiv 2 \llbracket [3] \rrbracket CUBE$

```
1
(1 3)4 \llbracket [2 3] \rrbracket CUBE
114 134
214 234
```

$CUBE[; 1 \ 3; 4] \equiv (1 \ 3)4 \llbracket [2 \ 3] \rrbracket CUBE$

```
1
```



```

      (2(1 3)[][1 3]CUBE)←0 ♦ CUBE is 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

```

Index Generator

$R \leftarrow \iota Y$

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:

$$\iota Y \leftrightarrow (\iota Y)[\iota Y]$$

$\square IO$ is an implicit argument of Index Generator. This function is also known as Interval.

Examples

```

      □IO
1      ρι0
0      ι5
1 2 3 4 5

      ι2 3
1 1 1 2 1 3
2 1 2 2 2 3

      ⍒A←2 4ρ'MAINEXIT'
MAIN
EXIT
      A[ιρA]
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

```

Index Of

$R \leftarrow X \text{ } \iota Y$

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

Vector Left Argument

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 $\text{⍺IO} + \text{⍺} \rho X$.

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

⍺IO and $\text{⍺CT}/\text{⍺DCT}$ 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

```

Higher-Rank Left Argument

If X is a higher rank array, 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 $1 \downarrow \rho X$. In this case, the shape of the result R is $(1 - \rho \rho X) \downarrow \rho Y$.

If a sub-array of Y cannot be found in X , then the corresponding element of R will be $\square_{IO+\rho X}$.

Examples

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

X ⍳ 1 2 3 4
1

Y ← 2 4 ρ 1 2 3 4 9 10 11 12
Y
1 2 3 4
9 10 11 12

X ⍳ Y
1 3
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
```

More Examples

```

      x
United Kingdom
Germany
France
Italy
United States
Canada
Japan
Canada
France

      y
United Kingdom
Germany
France
Italy
USA

Canada
Japan
China
India
Deutschland

      px
9 14

      py
2 5 14

      xty
1 2 3 4 10
6 7 10 10 10

      xtx
1 2 3 4 5 6 7 6 3

```

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

```

      ytx
LENGTH ERROR
      ytx
^

```

For performance information, see *Programmer's Guide: Search Functions and Hash Tables*.

Indexing

$$R \leftarrow X[Y]$$

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 X is determined by the index specification.

This form of Indexing, using brackets, does not follow the normal syntax of a dyadic function. For an alternative method of indexing, see [Index on page 65](#).

$\square IO$ 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 $1 \rho X$.

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 3ρ1 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 4ρ10×18
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 4p10×124
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 $1(\rho X)[K]$ 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 \mathbf{Y} is a non-simple array. Each item identifies a single element of \mathbf{X} by a set of indices with one element per axis of \mathbf{X} in row-major order.

Examples

```

      M
10 20 30 40
50 60 70 80

```

```

      M[c1 2]
20

```

```

      M[2 2p<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[3p<10]
ZZZ

```

Simple and Choose indexing are indistinguishable for vector \mathbf{X} :

```

      V<-10 20 30 40

      V[c2]
20

      c2
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 3pG,('MNO' 5)('PQR' 6)
G
ABC 1 DEF 2 GHI 3
JKL 4 MNO 5 PQR 6

DEF G[((1 2)1)((2 3)2)]
6

G[2 2p<(2 2)2]
5 5
5 5
ABC G[<<1 1]
1

ABC G[<1 1]
1

V←,G
ABC V[<<1]
1

ABC V[<1]
1

ABC V[1]
1

```


Intersection

 $R \leftarrow X \cap Y$

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 $X \equiv Y$ returns 1 for those items.

$\square CT$ is an implicit argument of Intersection.

Examples

```
'ABRA' n 'CAR'
ARA

1 'PLUS' 2 n 5
1 2
```

For performance information, see *Programmer's Guide: Search Functions and Hash Tables*.

Left

 $R \leftarrow X \rightarrow Y$

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 \rightarrow 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 A first row
scent
→ / mat A first column
scarf

→ / [2] 2 3 4 p 2 4 A first row from each plane
1 2 3 4
13 14 15 16
```

Similarly, with expansion:

```
→ \ mat
sssss
cccc
aaaaa
rrrrr
fffff
→ \ mat
scent
scent
scent
scent
scent
```

Less **$R \leftarrow X < Y$**

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.

$\square CT$ is an implicit argument of Less.

Examples

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

```
 $\square CT \leftarrow 1E^{-10}$ 
```

```
1 0.999999999999 0.9999999999 < 1
0 0 1
```

Less Or Equal **$R \leftarrow X \leq Y$**

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.

$\square CT$ is an implicit argument of Less Or Equal.

Examples

```
2 4 6 8 10 ≤ 6
1 1 1 0 0
```

```
 $\square CT \leftarrow 1E^{-10}$ 
```

```
1 1.000000000001 1.00000001 ≤ 1
1 1 0
```

Logarithm

 $R \leftarrow X \oslash Y$

Y must be a positive numeric array. X must be a positive numeric array. X cannot be 1 unless Y is also 1. R is the base X logarithm of Y .

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

$$X \oslash Y \leftrightarrow (\oslash Y) \div \oslash X$$

Examples

```

      10⊗100 2
2 0.3010299957

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

      1 ⊗ 1
1
      2 ⊗ 1
0

```

Magnitude

 $R \leftarrow |Y$

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 $(a + ib)$ is defined to be $\sqrt{a^2 + b^2}$

Examples

```

      |2 -3.4 0 -2.7
2 3.4 0 2.7

      |3j4
5

```

Match **$R \leftarrow X \equiv Y$**

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).

$\square CT$ is an implicit argument of Match.

Examples

```

      0  $\equiv$  1 0
1      ' '  $\equiv$  1 0
0      A
THIS
WORD

      A  $\equiv$  2 4p 'THISWORD'
1
      A  $\equiv$  1 1 0
0
      +B  $\leftarrow$  A A
THIS THIS
WORD WORD

      A  $\equiv$  B
1

      (0pA)  $\equiv$  0pB
0

      ' '  $\Rightarrow$  0pB
1 1 1 1
1 1 1 1

      ' '  $\Rightarrow$  0pA
1

```

Matrix Divide

 $R \leftarrow X \oslash Y$

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 $Y + . \times R$ is X .

R is determined such that $(X - Y + . \times R)^2$ is minimised.

The shape of R is $(1 \downarrow pY), 1 \downarrow pX$.

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

$P \oslash Q$
14.941 -2.9609

Example: linear regression on complex numbers

$x \leftarrow j \sqrt{50} + ?2 \ 13 \ 4 p 100$
 $y \leftarrow (x + .x3 \ 4 \ 5 \ 6) + j \sqrt{0.0001} x \sqrt{50} + ?2 \ 13 p 100$
 ρx
 13 4
 ρy
 13
 $y \oslash x$
 3J0.000011066 4J-0.000018499 5J0.000005745 6J0.000050328
 A i.e. $y \oslash x$ recovered the coefficients 3 4 5 6

Matrix Inverse

 $R \leftarrow \text{inv}(Y)$

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, $R + . \times Y$ is an identity matrix.

The shape of R is $\phi p Y$.

Examples

```

      M
2 -3
4 10

      +A+inv(M)
0.3125 0.09375
-0.125 0.0625

```

Within calculation accuracy, $A + . \times M$ is the identity matrix.

```

      A+.xM
1 0
0 1

```

```

      j+{a+0 0 a+0J1xw}
      x+j^50+?2 5 5p100
      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
      px
5 5
      id+{o.=i1w} A identity matrix of order w
      [/,| (id 1tpx) - x+.xinvx
3.66384E-16

```


Maximum **$R \leftarrow X \uparrow Y$**

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

```

Membership **$R \leftarrow X \in Y$**

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 $X \equiv Y$ returns 1 for those elements.

`CT` is an implicit argument 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 *Programmer's Guide: Search Functions and Hash Tables*.

Minimum **$R \leftarrow X \downarrow Y$**

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

```

Minus **$R \leftarrow X - Y$**

See [Subtract on page 112](#).

Mix	($\square ML$)	$R \leftarrow \uparrow[K]Y$ or $R \leftarrow \Rightarrow[K]Y$
-----	------------------	---

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

If $\square ML < 2$, Mix is represented by the symbol: \uparrow .

If $\square ML \geq 2$, Mix is represented by the symbol: \Rightarrow .

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 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. pR will be some permutation of $(pY), py$.

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

1. For all values of $\square ML$, 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 py inserted between the $\lfloor K$ th and the $\lceil K$ th axes of Y .
2. If $\square ML \geq 2$, 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 $\lceil K$ (in the fractional case).
3. If $\square ML \geq 2$, K may be a vector, with the same length as py , each element of which specifies the position in the result of the corresponding axis of the y .

If K is absent, the axes of y appear as the last axes of the result.

Simple Vector Examples

In this example, the shape of \mathbf{Y} is 3, and the shape of \mathbf{y} is 2. So the shape of the result will be a permutation of 2 and 3, i.e. in this simple example, either $(2\ 3)$ or $(3\ 2)$.

If K is omitted, the shape of the result is $(\mathbf{pY}), \mathbf{py}$.

```

      ↑(1 2)(3 4)(5 6)
1  2
3  4
5  6

```

If K is between 0 and 1, the shape of the result is $(\mathbf{py}), \mathbf{pY}$ because (\mathbf{py}) is inserted between the 0th and the 1st axis of the result, i.e. at the beginning.

```

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

```

If K is between 1 and 2, the shape of the result is $(\mathbf{pY}), \mathbf{py}$ because (\mathbf{py}) is inserted between the 1st and 2nd axis of the result, i.e. at the end. This is the same as the case when K is omitted.

```

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

```

If $\square ML \geq 2$ an integer K may be used instead (Note that \triangleright is used instead of \uparrow).

```

      □ML←3
      ▷(1 2)(3 4)(5 6)
1  2
3  4
5  6
      ▷[1](1 2)(3 4)(5 6)
1  3  5
2  4  6
      ▷[2](1 2)(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)(3 4)(5)

1 3 5
0 4 0

```

More Simple Vector Examples:

```

      ]box on
Was OFF
      ML←3
      ⇒('andy' 19)('geoff' 37)('pauline' 21)

```

andy	19
geoff	37
pauline	21

```

      ⇒[1]('andy' 19)('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 concatenating it with its prototype.

Example (Matrix of Vectors)

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 $(5\ 4\ 3\ 2)$.

$Y \leftarrow 5\ 4 \rho(1\ 20) \times c\ 3\ 2 \rho 1$
 Y

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

```

      ρ>Y
5 4 3 2
      ρ>[1]Y
3 2 5 4
      ρ>[2]Y
5 3 2 4
      ρ>[3]Y
5 4 3 2
      ρ>[4]Y
INDEX ERROR
      ρ>[4]Y
      ^

```

Note that $\rho>[4]Y$ 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:

```

3 5 2  ρ>[1 3]Y
      4
3 5 4  ρ>[1 4]Y
      2
5 3 4  ρ>[2 4]Y
      2
5 2 4  ρ>[4 2]Y
      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 3ρ10×18)
Y

```

1	2	3	4	5	10	20	30
					40	50	60

```

3 2 4  ρ>Y
      ρ>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 1ρ1**) to conform with the 3rd item which is rank 2. It is then extended in shape to become (**2 4↑1 1ρ1**) 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.

Multiply **$R \leftarrow X \times Y$**

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 Times.

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

```

Nand **$R \leftarrow X \tilde{\wedge} Y$**

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

```

Natural Logarithm

 $R \leftarrow \odot Y$

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

Example

```

      *1 2
0 0.6931471806

      *2 2p0j1 1j2 2j3 3j4
0.000000000J1.570796327 0.8047189562J1.107148718
1.282474679J0.9827937232 1.6094379120J0.927295218

```

Negative

 $R \leftarrow -Y$

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

```

Nor

 $R \leftarrow X \tilde{\vee} Y$

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

```


Not **$R \leftarrow \sim Y$**

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

```

Not Equal **$R \leftarrow X \neq Y$**

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” result, determined as follows:

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

$\square CT$ is an implicit argument of Not Equal.

Examples

```

      1 2 3 ≠ 1.1 2 3
1 0 0

```

```

□CT←1E-10

```

```

      1≠1 1.000000000001 1.0000001
0 0 1

```

```

      1 2 3 ≠ 'CAT'
1 1 1

```

Not Match

 $R \leftarrow X \neq Y$

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).

$\square CT$ is an implicit argument of Not Match.

Examples

```

0      0 ≠ 1 0
      ' ' ≠ 1 0
1
      1 ← A ← c(1 3) 'ABC'
1 2 3   ABC
      A ≠ (1 3) 'ABC'
1
      A ≠ c(1 3) 'ABC'
0
      0 ≠ 0 p A
1
      (1 ↑ 0 p A) ≠ c(0 0 0) ' '
1

```

Or, Greatest Common Divisor

 $R \leftarrow X \vee Y$

Case 1: 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

```

Case 2: X and Y are numeric (non-Boolean)

R is the Greatest Common Divisor of X and Y .

Examples

```

      15 1 2 7 ∨ 35 1 4 0
5 1 2 7

```

```

      rational ← {tw 1 ÷ c1 ∨ w}  A rational (□CT) approximation
                                A 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

```

□CT is an implicit argument in case 2.

Partition**($\square ML \geq 3$)** **$R \leftarrow X \subset [K]Y$**

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.

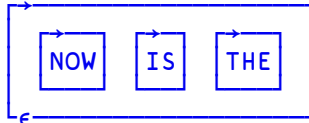
R is an array of the elements of Y partitioned according to X .

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.

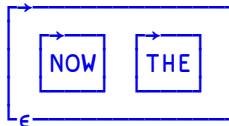
Examples

```
 $\square ML \leftarrow 3$ 
```

```
]display 1 1 1 2 2 3 3 3<'NOWISTHE'
```

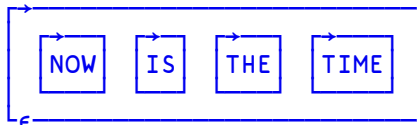


```
]display 1 1 1 0 0 3 3 3<'NOWISTHE'
```



```
TEXT<' NOW IS THE TIME '
```

```
]display (' '≠TEXT)⊂TEXT
```



```
]display CMAT← $\square$ FMT(' ',ROWS),COLS;NMAT
```

	Jan	Feb	Mar
Cakes	0	100	150
Biscuits	0	0	350
Buns	0	1000	500

```
]display (v≠' '≠CMAT)<CMAT      A Split at blank cols.
```

	Jan	Feb	Mar
Cakes	0	100	150
Biscuits	0	0	350
Buns	0	1000	500

```
]display N←4 4p16
```

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

```
]display 1 1 0 1<N
```

1 2	4
5 6	8
9 10	12
13 14	16

```
]display 1 1 0 1<[1]N
```

1 5	2 6	3 7	4 8
13	14	15	16

Partitioned Enclose**(\square ML < 3)** **$R \leftarrow X \llcorner [K] Y$**

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

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.

X must have the same length as the K th 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 of Y .

R is a vector of items selected from Y . The sub-arrays identified along the K th axis of Y at positions corresponding to each 1 in X up to the position before the next 1 in X (or the last element of X) become the successive items of R . The length of R is $\sum X$ (after possible extension).

Examples

```

      0 1 0 0 1 1 0 0 0 c19
2 3 4 5 6 7 8 9

```

```

      1 0 1 c[1] 3 4p12
1 2 3 4 9 10 11 12
5 6 7 8

```

```

      1 0 0 1 c[2]3 4p12
1  2  3    4
5  6  7    8
9 10 11   12

```

Pi Times**R←○Y**

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

Example

```

○0.5 1 2
1.570796327 3.141592654 6.283185307

○0J1
0J3.141592654

*○0J1 A Euler
-1

```

Pick**R←X>Y**

Y may be any array.

X is a scalar or vector of indices of **Y**, viz. $\text{⍳} \rho Y$.

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.

$\square \text{IO}$ is an implicit argument of Pick.

Examples

```

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

G←2 3ρG,('MNO' 5)('PQR' 6)

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

JKL ((←2 1),1)>G

JKL ((←2 1)>G
      4

```

```

K      ((2 1)1 2)>G
      (5p<10)>10
10

```

Plus

R←X+Y

See [Add](#) on page 11.

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, 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

```


Ravel **$R \leftarrow Y$**

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

```

```

      p,10
1

```

Ravel with Axes **$R \leftarrow [K]Y$**

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.

```

ppR ↔ 1+ppY
pR  ↔ (1,pY)[ΔK,ιppY]

```

Examples

```

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

      MAT ← 3 4 ρ 12
      ρ, [0.5] MAT
1 3 4
      ρ, [1.5] MAT
3 1 4
      ρ, [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, $R \leftrightarrow Y$.

$$\rho \rho R \leftrightarrow 1 + (\rho \rho Y) - \rho, K$$

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
      ρM
      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
      ρ,[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
      ρ,[2 3]M
2 12

```

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

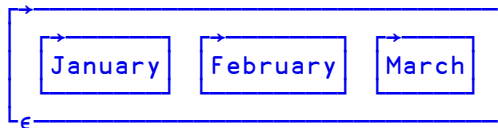
$\rho R \leftrightarrow (\rho Y), 1$

Examples

```

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

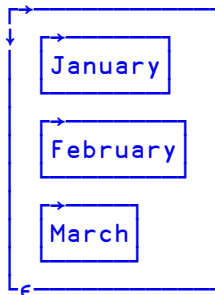
```



```

]display ,[i0]Q1

```



Reciprocal

R ← ÷ Y

Y must be a numeric array. **R** is numeric. **R** is the reciprocal of **Y**; that is $1 \div Y$. If $\square \text{DIV}=0$, $\div 0$ results in a **DOMAIN ERROR**. If $\square \text{DIV}=1$, $\div 0$ returns 0.

$\square \text{DIV}$ 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

```

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 **K**th 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 **K**th axis in the order of occurrence. The shape of **R** is the shape of **Y** except that the length of the (implied) **K**th axis is **+ / |X|** (after possible extension).

This function is sometimes called Compress when **X** is Boolean.

Examples

$$\begin{array}{ccccccc} & & 1 & 0 & 1 & 0 & 1/\iota 5 \\ 1 & 3 & 5 & & & & \end{array}$$

$$\begin{array}{ccccccccccccccc} & & & 1 & ^{-2} & 3 & ^{-4} & 5/\iota 5 \\ 1 & 0 & 0 & 3 & 3 & 3 & 0 & 0 & 0 & 0 & 5 & 5 & 5 & 5 & 5 \end{array}$$

$$\begin{array}{ccc} & & M \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}$$

$$\begin{array}{ccc} & 2 & 0 & 1/M \\ 1 & 1 & 3 \\ 4 & 4 & 6 \end{array}$$

$$\begin{array}{ccc} & 0 & 1/M \\ 4 & 5 & 6 \end{array}$$

$$\begin{array}{ccc} & 0 & 1/[1]M \\ 4 & 5 & 6 \end{array}$$

If Y is a singleton ($1 = \times/p, Y$) its value is notionally extended to the length of X along the specified axis.

$$\begin{array}{ccc} & 1 & 0 & 1/4 \\ 4 & 4 & & \\ & 1 & 0 & 1/,3 \\ 3 & 3 & & \\ & 1 & 0 & 1/1 & 1p5 \\ 5 & 5 & & \end{array}$$

Reshape **$R \leftarrow X_{\rho} Y$**

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 ($\epsilon \in Y$). If X contains at least one zero, then R is empty. If X is an empty vector, then R is scalar.

Examples

```

      2 3p18
1 2 3
4 5 6

```

```

      2 3p14
1 2 3
4 1 2

```

```

      2 3p10
0 0 0
0 0 0

```

Residue **$R \leftarrow X | Y$**

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 $Y - X \times \lfloor Y \div X + 0 = X$.

\square_{CT} is an implicit argument 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 ($\square_{\text{UCS } 166}$, U+00A6) is not interpreted as Residue.

Reverse **$R \leftarrow \phi[K]Y$**

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 $R \leftarrow \Theta Y$ implies the first axis.

R is the array Y rotated about the K th or implied axis.

Examples

```

      ϕ1 2 3 4 5
5 4 3 2 1

```

```

      M
1 2 3
4 5 6

```

```

      ϕM
3 2 1
6 5 4

```

```

      ΘM
4 5 6
1 2 3

```

```

      ϕ[1]M
4 5 6
1 2 3

```

Reverse First **$R \leftarrow \Theta[K]Y$**

The form $R \leftarrow \Theta Y$ implies reversal along the first axis. See [Reverse above](#).

Right **$R \leftarrow X \vdash Y$**

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

Example

```

      42 ⊢ 'abc' 1 2 3
abc 1 2 3

```

Note that when \vdash 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  A last row
fleet
      +/mat  A last column
tenet

      +/[2]2 3 4p124 A last row from each plane
  9 10 11 12
21 22 23 24

```

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 $1:Y$ 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, i.e. $(0 < R[I] < 1)$.

`⎕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 [16807](#) for more information.

Examples

```

      ?9 9 9
2 7 5
      ?3p0
0.3205466592 0.3772891947 0.5456603511

```


Rotate **$R \leftarrow X\phi[K]Y$**

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 $R \leftarrow X\phi Y$ implies the first axis.

If Y is a scalar, it is treated as a one-element vector. X must have the same shape as the rank of Y excluding the K th dimension. If X is a scalar or one-element vector, it will be extended to conform. If Y is a vector, then X may be a scalar or a one-element vector.

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
      Iϕ[2]M
  1  6  7  4
  5  2  3  8

  9 14 11 16
13 10 15 12

```

```

      J
2  -3
3  -2

      JϕM
3  4  1  2
6  7  8  5

12  9 10 11
15 16 13 14

```

Rotate First **$R \leftarrow X \ominus [K] Y$**

The form $R \leftarrow X \ominus Y$ implies rotation along the first axis. See [Rotate above](#).

Same **$R \leftarrow \neg Y$**

Y may be any array.

The result R is the argument Y .

Examples

```

      -'abc' 1 2 3
abc 1 2 3

```

Shape

 $R \leftarrow \rho Y$

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 $\rho\rho Y$.

Examples

```

      ρ10
      ρ'CAT'
3
      ρ3 4ρ12
3 4
      +G←(2 3ρ16)('CAT' 'MOUSE' 'FLEA')
1 2 3   CAT  MOUSE  FLEA
4 5 6
      ρG
2
      ρρG
1
      ρ∘∘G
2 3   3
      ρ∘∘∘G
      3 5 4

```

Split **$R \leftarrow \downarrow[K]Y$**

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 K th axis. R is a scalar if Y is a scalar. Otherwise R is an array whose rank is $\text{rank } Y - 1$ and whose shape is $(\text{shape } Y)[K]$.

Examples

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

```
↓2 5ρ10
1 2 3 4 5 6 7 8 9 10
```

```
↓[1]2 5ρ10
1 6 2 7 3 8 4 9 5 10
```

Subtract **$R \leftarrow X - Y$**

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 Minus.

Example

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

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

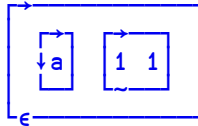
Table **$R \leftarrow \overline{\cdot} Y$**

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

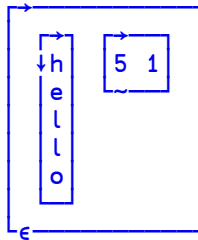
Table has been implemented according to the Extended APL Standard (ISO/IEC 13751:2001).

Examples

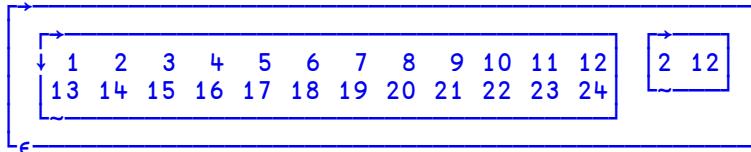
```
]display {ω (ρω)} ⍤ 'a'
```



```
]display {ω (ρω)} ⍤ 'hello'
```



```
]display {ω (ρω)} ⍤ 2 3 4⍤24
```



Take **$R \leftarrow X \uparrow Y$**

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 $(p, X)p1$. 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 ($\epsilon \in Y$).

Examples

```
5↑'ABCDEF'
ABCDE
```

```
5↑1 2 3
1 2 3 0 0
```

```
~5↑1 2 3
0 0 1 2 3
```

```
5↑(13) (14) (15)
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 4p⊞A
1↑M3
```

```
ABCD
EFGH
IJKL
~1↑M3
```

```
MNOP
QRST
UVWX
```

Take with Axes

 $R \leftarrow X \uparrow [K] Y$

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 :

$$\rho \rho R \leftrightarrow \rho \rho Y$$

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

$$(\rho R)[,K] \leftrightarrow |,X$$

Examples

```

      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

```

Tally **$R \leftarrow \#Y$**

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

Tally returns the number of major cells of Y . 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 $\{ \theta \rho (\rho \omega), 1 \}$.

Examples

```

#2 3 4 p 10
2
#2
1
#0
0

```

Note that $\#V$ is useful for returning the length of vector V as a scalar. (In contrast, ρV is a one-element vector.)

Times **$R \leftarrow X \times Y$**

See [Multiply on page 91](#).

Transpose (Monadic) **$R \leftarrow \phi Y$**

Y may be any array. R is an array of shape $\phi \rho Y$, similar to Y with the order of the axes reversed.

Examples

```

      M
1 2 3
4 5 6

      ϕM
1 4
2 5
3 6

```


Transpose (Dyadic)

 $R \leftarrow X \Phi Y$

Y may be any array. X must be a simple scalar or vector whose elements are included in the set $\iota \rho \rho Y$. Integer values in X may be repeated but all integers in the set $\iota \uparrow / X$ 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.

$\square IO$ 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

```

Type**($\square ML < 1$)** **$R \leftarrow \epsilon Y$**

Migration level must be such that $\square ML < 1$ (otherwise ϵ means Enlist. See [Enlist on page 43](#)).

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

```

       $\epsilon(2 \ 3p16)(1 \ 4p\text{'TEXT'})$ 
0 0 0
0 0 0

      ' '= $\epsilon$ 'X'
1

```

Union **$R \leftarrow X \cup Y$**

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 $X \equiv Y$ returns 1 for those items.

$\square CT$ is an implicit argument of Union.

Examples

```

      'WASH'  $\cup$  'SHOUT'
WASHOUT

      'ONE' 'TWO'  $\cup$  'TWO' 'THREE'
ONE TWO THREE

```

For performance information, see *Programmer's Guide: Search Functions and Hash Tables*.

Unique **$R \leftarrow uY$**

Y must be a vector. R is a vector of the elements of Y omitting non-unique elements after the first.

$\square CT$ is an implicit argument of Unique.

Examples

```

      u 'CAT' 'DOG' 'CAT' 'MOUSE' 'DOG' 'FOX'
CAT   DOG  MOUSE   FOX

```

```

      u 22 10 22 22 21 10 5 10
22 10 21 5

```

Without **$R \leftarrow X \sim Y$**

See [Excluding on page 45](#).

Zilde **$R \leftarrow \emptyset$**

The empty vector ($\mathbf{t0}$) may be represented by the numeric constant \emptyset called ZILDE.

Chapter 2:

Primitive Operators

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 *Programmer's Guide: 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

```

      +/15
15
      (*2)13
1 4 9

      PLUS ← + ♦ TIMES ← ×
      1 PLUS.TIMES 2
2

      □NL 2
A
X
      □EX''↓□NL 2
      □NL 2

```

Monadic Operators

Like primitive functions, monadic operators can be:

- named
- enclosed within parentheses
- displayed in the session

Examples

```

..      ⍳ ← each ← (")      A name and display
      shape←p
      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

```

× -1      ⍳ ← inv ← ×-1      A produces monadic inverse operator
+ \inv 1 2 3      A scan-inverse
1 1 1
      lim ← ×≡      A power-limit
      1 +÷lim 1      A Phi
1.61803

```

Axis Specification

Some operators may include an axis specification. Axis is itself an operator. However the effect of axis is described for each operator where its specification is permitted. $\square IO$ is an implicit argument of the function derived from the Axis operator.

The description for each operator follows in alphabetical sequence. The valence of the derived function is specifically identified to the right of the heading block.

Table 8: Primitive Operators

Class of Operator	Name	Producing Monadic derived function	Producing Dyadic derived function
Monadic	Assignment		$Xf \leftarrow Y$
	Assignment		$X[I]f \leftarrow Y$
	Assignment		$(EXP \ X)f \leftarrow Y$
	Commute		$Xf \rightsquigarrow Y$
	Each	$f''Y$	$Xf''Y$
	I-Beam	$A \mp Y$	
	Reduction	$f/Y \ [\]$	
		$f \nabla Y \ [\]$	
	Scan	$f \setminus Y \ [\]$	
		$f \backslash Y \ [\]$	
	Spawn	$f \& Y$	$Xf \& Y$
Dyadic	Axis	$f[B]Y$	$Xf[B]Y$
	Composition	$f \circ gY$	$Xf \circ gY$
		$A \circ gY$	
		$(f \circ B)Y$	
	Inner Product		$Xf . gY$
	Outer Product		$X \circ . gY$
	Power	$f \times gY$	$Xf \times gY$
	Variant	$f \boxminus gY$	$Xf \boxminus gY$
[] Indicates optional axis specification			

Operators (A-Z)

Monadic and Dyadic primitive operators are presented in alphabetical order of their descriptive names as shown in [Table 8](#) above.

The valence of the operator and the derived function are implied by the syntax in the heading block.

Assignment (Modified)

 $\{R\} \leftarrow X f \leftarrow Y$

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 whose items are 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/¯2←vec>0 ♦vec
3 5 1 4 2

```


Assignment (Indexed Modified)

$$\{R\} \leftarrow X[I] f \leftarrow Y$$

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 .

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 indexed elements of X , that is $X[I]$, to the result of $X[I] f Y$. This result must have the same shape as $X[I]$.

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

```

If an index is repeated, function f will be applied to the successive values of the indexed elements of X , taking the index occurrences in left-to-right order.

Example

```

      B←5p0
      B[2 4 1 2 1 4 2 4 1 3]++1
      B
3 3 1 3 0

```

Assignment (Selective Modified) **$\{R\} \leftarrow (EXP \ X) f \leftarrow Y$**

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 [Assignment \(Selective\) on page 21](#) for a list of allowed selection functions.) The selected elements of X must be 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 selected elements of X to the result of $X[I]fY$ where $X[I]$ defines the elements of X selected by EXP .

Example

```

      A
12 36 23 78 30

      ((A>30)/A) x← 100
      A
12 3600 23 7800 30

```

Axis (with Monadic Operand)

R←f[B]Y

f must be a monadic primitive mixed function taken from those shown in [Table 9](#) 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. Axis does not follow the normal syntax of an operator.

Table 9: Primitive monadic mixed functions with optional axis.

Function	Name	Range of B
ϕ or \ominus	Reverse	$B \in \mathbb{I} \rho \rho Y$
\uparrow	Mix	$(0 \neq 1 B) \wedge (B > \square IO - 1) \wedge (B < \square IO + \rho \rho Y)$
\downarrow	Split	$B \in \mathbb{I} \rho \rho Y$
,	Ravel	fraction, or zero or more axes of Y
\Leftarrow	Enclose	$(B \equiv 1 0) \vee (\wedge / B \in \mathbb{I} \rho \rho Y)$

In most cases, B must be an integer which identifies a specific axis of Y. However, when f is the Mix function (\uparrow), B is 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 (\Leftarrow), B can be a **vector** of two or more axes.

$\square IO$ is an implicit argument of the derived function which determines the meaning of B.

Examples

```

       $\phi[1]2\ 3\rho 6$ 
4 5 6
1 2 3

       $\uparrow[.1]'ONE'\ 'TWO'$ 
OT
NW
EO
```

Axis (with Dyadic Operand)

$$R \leftarrow X f [B] Y$$

f must be a dyadic primitive scalar function, or a dyadic primitive mixed function taken from [Table 10](#) below. B must be a numeric scalar or vector. X and Y may be any arrays whose items are appropriate to function f . Axis does not follow the normal syntax of an operator.

Table 10: Primitive dyadic mixed functions with optional axis.

Function	Name	Range of B
$/$ or \neq	Replicate	$B \in \mathbb{I} \rho \rho Y$
\backslash or ∇	Expand	$B \in \mathbb{I} \rho \rho Y$
\subset	Partitioned Enclose	$B \in \mathbb{I} \rho \rho Y$
ϕ or \ominus	Rotate	$B \in \mathbb{I} \rho \rho Y$
$,$ or $;$	Catenate	$B \in \mathbb{I} \rho \rho Y$
$,$ or $;$	Laminate	$(0 \neq 1 B) \wedge (B > \square IO - 1) \wedge (B < \square IO + (\rho \rho X) \uparrow \rho \rho Y)$
\uparrow	Take	zero or more axes of Y
\downarrow	Drop	zero 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 (\uparrow) and Drop (\downarrow), B can be a **vector** of two or more axes.

$\square IO$ is an implicit argument of the derived function which determines the meaning of B .

Examples

```

1 4 5 =[1] 3 2p16
1 0
0 1
1 0

2 ^2 1/[2]2 3p'ABCDEF'
AA C
DD F

'ABC',[1.1]'='
A=
B=
C=

'ABC',[0.1]'='
ABC
===

[]IO←O

'ABC',[-0.5]'='
ABC
===

```

Axis with Scalar Dyadic Functions

The axis operator $[X]$ 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 $(\rho L) \leftrightarrow (\rho H)[X]$. Each element of L is replicated along the remaining $(\rho H) \sim X$ 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 $(\rho L) \leftrightarrow (\rho R)[X]$.

Examples

```

      mat
10 20 30
40 50 60

```

```

      mat+[1]1 2      A add along first axis
11 21 31
42 52 62

```

```

      mat+[2]1 2 3    A 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

```

Commute

 $\{R\} \leftarrow \{X\} \quad f \sim Y$

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 YfX . The derived function need not return a result.

If left argument X is omitted, the right argument Y is duplicated in its place, i.e.

$$f \sim Y \leftrightarrow Y \quad f \sim Y$$

Examples

```

      N
3 2 5 4 6 1 3

```

```

      N/2 | N
3 5 1 3

```

```

      p3
3 3 3

```

```

mean<+/(÷°p) A mean of a vector
mean 10
5.5

```

The following statements are equivalent:

```

F/2←I
F←F/2I
F←I/F

```

Commute often eliminates the need for parentheses

Composition (Form I)

 $\{R\} \leftarrow f \circ gY$

f may be any monadic function. g may be any monadic function which returns a result. Y may be any array whose items are appropriate to function g . The items of gY must be appropriate to function f .

The derived function is equivalent to $f gY$. The derived function need not return a result.

Composition allows functions to be *glued* together to build up more complex functions.

Examples

```

RANK ← p°p
RANK ∷ 'JOANNE' (2 3p16)
1 2

```

```

+ / ° i ``2 4 6
3 10 21

```

```

□VR 'SUM'
▽ R←SUM X
[1] R←+/X
▽

```

```

SUM ° i ``2 4 6
3 10 21

```


Composition (Form II)

 $\{R\} \leftarrow A \circ g Y$

g may be any dyadic function. A may be any array whose items are appropriate to function g . Y may be any array whose items are appropriate to function g .

The derived function is equivalent to $A g Y$. The derived function need not return a result.

Examples

```

      2 2∘p `` 'AB'
AA  BB
AA  BB

      SINE ← 1∘∘

      SINE 10 20 30
-0.5440211109 0.9129452507 -0.9880316241

```

The following example uses Composition Forms I and II to list functions in the workspace:

```

      □NL 3
ADD
PLUS

      □∘←∘□VR``↓□NL 3
▽ ADD X
[1]   →LABρ~0≠□NC'SUM' ♦ SUM←0
[2]   LAB:SUM←SUM++/X
▽
▽ R←A PLUS B
[1]   R←A+B
▽

```

Composition (Form III)

$$\{R\} \leftarrow (f \circ B) Y$$

f may be any dyadic function. B may be any array whose items are appropriate to function f . Y may be any array whose items are appropriate to function f .

The derived function is equivalent to YfB . The derived function need not return a result.

Examples

```
(*o.5)4 16 25
2 4 5
```

```
SQRT ← *o.5
```

```
SQRT 4 16 25
2 4 5
```

The parentheses are required in order to distinguish between the operand B and the argument Y .

Composition (Form IV)

$$\{R\} \leftarrow Xf \circ gY$$

f may be any dyadic function. g may be any monadic function which returns a result. Y may be any array whose items are appropriate to function g . Also gY must return a result whose items are appropriate as the right argument of function f . X may be any array whose items are appropriate to function f .

The derived function is equivalent to $XfgY$. The derived function need not return a result.

Examples

```
+o÷/40p1          A Golden Ratio! (Bob Smith)
1.618033989
```

```
0,o,i''i5
0 1 0 1 2 0 1 2 3 0 1 2 3 4 0 1 2 3 4 5
```

Each (with Monadic Operand)

 $\{R\} \leftarrow f \cdot Y$

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' (ι3))('DICK' (ι4))('HARRY' (ι5))
      ρG
3
      ρ⋅⋅G
2 2 2
      ρ⋅⋅⋅G
3 3 4 4 5 5
      +⊖FX⋅⋅('FOO1' 'A←1')('FOO2' 'A←2')
FOO1 FOO2

```

Each (with Dyadic Operand)

 $\{R\} \leftarrow X f \cdot Y$

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 first items 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
      1ϕ⌞G
2 3  1  5 6  4 9 8 10

      1ϕ⌞⌞G
1  3 2  4 6 5  8 9 10

      1ϕ⌞⌞⌞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

```

I-Beam **$R \leftarrow \{X\} (A \pm) Y$**

I-Beam is a monadic operator that provides a range of system related services.

WARNING: Although documentation is provided for I-Beam functions, any service provided using 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 . Y is an array that supplies further information about what is to be done.

X may or may not be required depending on A .

R is the result of the derived function.

For further information, see [I-Beam on page 159](#).

Inner Product

$$R \leftarrow X f . g Y$$

f must be a dyadic function. g may be any dyadic function which returns a result. The last axis of X must have the same length as the first axis of Y .

The result of the derived function has shape $(\neg 1 \downarrow \rho X), 1 \downarrow \rho Y$. Each item of R is the result of $f/xg\ y$ where x and y are typical vectors taken from all the combinations of vectors along the last axis of X and the first axis of Y respectively.

Function f (and the derived function) need not return a result in the exceptional case when $2 = \neg 1 \uparrow \rho X$. In all other cases, function f must return a result.

If the result of $xg\ y$ is empty, for any x and y , a **DOMAIN ERROR** will be reported unless function f is a primitive scalar dyadic function with an identity element shown in [Identity Elements on page 149](#).

Examples

```
1 2 3+.×10 12 14
76
```

```
1 2 3 PLUS.TIMES 10 12 14
76
```

```
+/1 2 3×10 12 14
76
```

```
      NAMES
HENRY
WILLIAM
JAMES
SEBASTIAN
```

```
      NAMES^.='WILLIAM '
0 1 0 0
```

Key **$R \leftarrow \{X\} f \boxtimes Y$**

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 \boxtimes ¹ 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 $\iota \neq Y$ having that key. $f \boxtimes Y$ is the same as $Y \ f \boxtimes \iota \neq Y$.

Key is similar to the GROUP BY clause in SQL.

Example

```
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
```

In this example, both arrays are vectors so their major cells are their elements. The function $\{\alpha:'\omega\}$ is applied between the unique elements in `suits` ('Spades' 'Hearts' 'Clubs') and the elements in `cards` grouped according to their corresponding elements in `suits`, i.e. ('2' 'Ace'), ('Queen' 'Jack') and ('4').

¹The symbol \boxtimes is not available in Classic Edition, and the Key operator is instead represented by \boxtimes U2338

Monadic Example

```

      {α ω} ⍱ suits A indices of unique major cells
Spades  1 3
Hearts  2 5
Clubs   4

```

```

      {α,≠ω} ⍱ suits A count of unique major cells
Spades  2
Hearts  2
Clubs   1

```

Further Examples

x is a vector of stock codes, y is a corresponding matrix of values.

```

      px
10
      py
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 $\{α ω\}$ 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{α ω}⍱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 $\{\alpha(+\omega)\}$, which delivers the stock codes and the corresponding totals in y :

```

      x{α(+ω)}⊞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{+ω}⊞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

```

      {α ω} ⊞ suits A indices of unique major cells
Spades  1 3
Hearts  2 5
Clubs   4

      {α,≠ω} ⊞ suits A 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 A 12, some repeat
Pete Jay Bob Pete Pete Jay Jim Pete Pete Jim
Pete Pete
```

```
(unames)°.≡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←↓(unames)°.≡names
]disp b/'c12
```

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

```
+/'b/'cscores
399 85 71 109
```

```
]disp {cω}≡ names
```

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

```
names {+/ω}≡ scores
399 85 71 109
```

Outer Product

 $\{R\} \leftarrow X \circ .g Y$

g may be any dyadic function. The left operand of the operator is the symbol \circ . 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 $(\rho X), \rho Y$. Each element of R is the item returned by function g when applied to the particular combination of elements of X and Y .

Examples

```

      1 2 3 ◦ . × 10 20 30 40
10 20 30 40
20 40 60 80
30 60 90 120

```

```

      1 2 3 ◦ . ρ 'AB'
A      B
AA     BB
AAA    BBB

```

```

      1 2 ◦ . , 1 2 3
1 1 1 2 1 3
2 1 2 2 2 3

```

```

      (1 3) ◦ . = 1 3
1 0 0
0 1 0
0 0 1

```

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 .

Power Operator

$\{R\} \leftarrow \{X\} (f \star g) Y$

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. In particular, if g is $=$ or \equiv , 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 :

$$X\ (f\ \star\ g)\ Y \rightarrow (X \circ f\ \star\ g)\ Y$$

A *negative* right operand g applies the *inverse* of the operand function f , $(|g)$ 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

Defined, dynamic and some primitive functions do not have an inverse. In this case, a negative argument g generates **DOMAIN ERROR**.

Examples

$(, \circ c \circ, \star (1 \equiv, \text{vec})) \text{vec}$

A ravel-enclose if simple.

$a\ b\ c \leftarrow 1\ 0\ 1 \{ (c \star \alpha) \omega \} `` abc$

A enclose first and last.

$\text{cap} \leftarrow \{ (\alpha \star \alpha) \omega \}$

A conditional application.

$a\ b\ c \leftarrow 1\ 0\ 1 < \text{cap} `` abc$

A enclose first and last.

```

succ←1∘+           A successor function.

(succ×4)10         A fourth successor of 10.
14
(succ×-3)10        A third predecessor of 10.
7
1∘÷×=1            A fixpoint: golden mean.
1.618033989

f←(32∘+)∘(×∘1.8)  A Fahrenheit from Celsius.
f 0 100
32 212

c←f×-1            A c is Inverse of f.
c 32 212          A Celsius from Fahrenheit.
0 100

invs←{(α×-1)ω}    A inverse operator.

+\\invs 1 3 6 10   A scan inverse.
1 2 3 4

2∘↓invs 9          A decode inverse.
1 0 0 1

dual←{ωω×-1 αα ωω ω} A dual operator.

mean←{(+/ω)÷ρω}   A mean function.

mean dual∘ 1 2 3 4 5 A geometric mean.
2.605171085

+//dual÷ 1 2 3 4 5 A parallel resistance.
0.4379562044

mean dual(×~)1 2 3 4 5 A root-mean-square.
3.31662479

&dual↑ 'hello' 'world' A 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

```

Rank

$$R \leftarrow \{X\} f \ddot{\circ} k Y$$

If X is omitted, f may be any monadic function that returns a result. Y may be any array.

The Rank operator $\ddot{\circ}^1$ applies function f successively to the sub-arrays in Y specified by k . If k is positive, it selects the k -cells of Y . If k is negative, it selects the $(r+k)$ -cells of Y where r is its rank. If k is -1 it selects the major cells of Y .

If X is specified, 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 and Y specified by k . k is a 2-element integer vector, or a scalar (which is implicitly extended), whose first element selects sub-arrays in X and whose second element selects sub-arrays of Y .

For further information, see *Programmer's Guide: Cells and Subarrays*.

Notice that it is necessary to prevent the right operand k binding to the right argument. This can be done using parentheses e.g. $(f \ddot{\circ} 1) Y$. The same can be achieved using \vdash because $\ddot{\circ}$ binds tighter to its right operand than \vdash does to its left argument, and \vdash 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
      ρY
2 3 4
```

¹The symbol $\ddot{\circ}$ is not available in Classic Edition, and the Rank operator is instead represented by `⊢U2364`

$$c \circ 2 \mapsto Y$$

36	99	20	5	66	72	27	74
63	50	26	10	44	1	46	62
64	90	68	98	48	9	81	22

$$c \circ 1 \mapsto Y$$

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 $\{ (c \circ \omega) \circ \omega \}$ sorts a vector.

$$\{ (c \circ \omega) \circ \omega \} \begin{matrix} 3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5 \\ 1 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 9 \end{matrix}$$

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

$$(\{ (c \circ \omega) \circ \omega \} \circ 1) Y$$

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 4p⌈12
10 11 12 13
24 25 26 27
38 39 40 41

```

Using the function $\{\alpha \ \omega\}$ as the left operand demonstrates how the dyadic case of the rank operator works.

```

      10 20 30 ({α ω}∘0 1)3 4p⌈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 ({α ω}∘^{-1})3 4p⌈12

```

10	0	1	2	3
20	4	5	6	7
30	8	9	10	11

Reduce **$R \leftarrow f / [K] Y$**

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 $R \leftarrow f \neq Y$ 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 .

Table 11: Identity Elements

Function		Identity
Add	+	0
Subtract	-	0
Multiply	×	1
Divide	÷	1
Residue		0
Minimum	⌊	$M^{(1)}$
Maximum	⌈	$-M^{(1)}$
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	τ	0
Union	\cup	Θ
Replicate	$/\neq$	1
Expand	$\backslash\neq$	1
Rotate	$\phi\Theta$	0

Notes:

M represents the largest representable value: typically this is 1.7E308, unless $\square FR$ is 1287, when the value is 1E6145.

For a typical vector Y , the result is: $c(1\triangleright Y)f(2\triangleright Y)f\ldots\ldots f(n\triangleright Y)$

The shape of R is the shape of Y excluding the K th axis. If Y is a scalar then R is a scalar. If the length of the K th axis is 1, then R is the same as Y . If the length of the K th axis is 0, then **DOMAIN ERROR** is reported unless function f occurs in Table 1, in which case its identity element is returned in each element of the result.

Examples

```

      v/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

      +/ι0
0

      ,/''
DOMAIN ERROR

      ,/''
^
```

Reduce First **$R \leftarrow f \nearrow Y$**

The form $R \leftarrow f \nearrow Y$ implies reduction along the first axis of Y . See [Reduce above](#).

Reduce N-Wise **$R \leftarrow X f / [K] Y$**

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 $R \leftarrow X f \nearrow Y$ 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 .

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 $(\rho Y) + (\rho \rho Y) = \iota \rho \rho Y$ array of identity elements for the function f . See [Identity Elements on page 149](#).

If X is negative, each sub-vector is reversed before being reduced.

Examples

```

      1 4
1 2 3 4
      3+/14# (1+2+3) (2+3+4)
6 9
      2+/14# (1+2) (2+3) (3+4)
3 5 7
      1+/14# (1) (2) (3) (4)
1 2 3 4
      0+/14# Identity element for +
0 0 0 0 0
      0×/14# Identity element for ×
1 1 1 1 1
      2,/14# (1,2) (2,3) (3,4)
1 2 2 3 3 4
      -2,/14# (2,1) (3,2) (4,3)
2 1 3 2 4 3
```

Scan

$$R \leftarrow f \backslash [K] Y$$

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 $R \leftarrow f \backslash Y$ 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 $f / I \uparrow V$.

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

```

      v\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

```

Scan First

$R \leftarrow f \downarrow Y$

The form $R \leftarrow f \downarrow Y$ implies scan along the first axis of Y . See [Scan above](#).

Spawn

 $\{R\} \leftarrow \{X\} f \& Y$

$\&$ 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

```

0.25      ÷&4          A Reciprocal in background

          □←÷&4       A Show thread number
1
0.25

          FOO&88      A Spawn monadic function.
          2 FOO&3     A dyadic
          {NIL}&0      A niladic
          ⚡&'NIL'      A ..
          X.GOO&99     A thread in remote space.
          ⚡&'□dl 2'    A Execute async expression.
          'NS'⚡&'FOO'  A .. remote .. .. .
          PRT&``↓□nl 9 A PRT spaces in parallel.
```

Variant

$$\{R\} \leftarrow \{X\} (f \text{ } \boxed{O} \text{ } B) Y$$

The Variant operator \boxed{O} 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**.

\boxed{O} and \boxed{OPT} are synonymous though only the latter is available in the Classic Edition.

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. 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** property 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
```


Chapter 3:

The I-Beam Operator

I-Beam

$$R \leftarrow \{X\} (A \mathbb{I}) Y$$

I-Beam is a monadic operator that provides a range of system related services.

WARNING: Although documentation is provided for I-Beam functions, any service provided using 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 as shown in the table below. Y is an array that supplies further information about what is to be done.

X may or may not be required depending on A .

R is the result of the derived function.

A	Derived Function
8	Inverted Table Index-of
181	Unsqueeze Type
200	Syntax Colouring
219	Compress/Decompress Vector of Short Integers
220	Serialise/Deserialise Array
1111	Number of Threads
1112	Parallel Execution Threshold
1113	Thread Synchronisation Mechanism

A	Derived Function
1159	Update Function Time and User Stamp
2000	Memory Manager Statistics
2002	Specify Workspace Available
2010	Update DataTable
2011	Read DataTable
2015	Create Data Source
2022	Flush Session Caption
2023	Close all Windows
2100	Export to Memory
2400	Set Workspace Save Options
2401	Expose Root Properties
3002	Disable Component Checksum Validation
4000	Fork New Task
4001	Change User
4002	Reap Forked Tasks
4007	Signal Counts
16807	Random Number Generator

Inverted Table Index Of

$R \leftarrow X(8\mathbb{I})Y$

This function computes X index-of Y (viz. $X\mathbb{I}Y$) 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, $1 = \rho\rho\omega$ and $(\neq\omega) = \neq''\omega$. 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:

$X \leftarrow (10\ 3\rho\Box a)\ (\mathbb{I}10)\ 'metalepsis'$
 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 :

$Y \leftarrow (c\leftarrow 3\ 1\ 4\ 1\ 5\ 9)\ \Box''X$
 Y

GHI	3	1	4	1	5	9	tmamli
ABC							
JKL							
ABC							
MNO							
YZA							

To compute the indices of Y in X using dyadic \mathbb{I} , it is necessary to first un-invert each of the tables in order to create nested matrices that \mathbb{I} can handle.

```
unvert ← {0 1 2 3 4 5 6 7 8 9}
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
```

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 [Data Representation \(Monadic\) on page 258](#).

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

```

11      ⌈dr 1⌈1 1000
163      (181⌈) 1⌈1 1000

```

Syntax Colouring

R←200IY

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**.

```
{(↑ω),↑ 200Iω} 'foo; local' 'global' 'local←pp' 'hello''
foo; local      21 21 21 19  3 31 31 31 31 31 0 0 0 0 0
global          7  7  7  7  7  7  0  0  0  0 0 0 0 0 0
local←pp'hello' 31 31 31 31 31 19 23 23  4  4 4 4 4 4 4
```

In this example:

- 21 is the syntax identifier for “function name”
- 19 is the syntax identifier for “primitive”
- 3 is the syntax identifier for “white space”
- 31 is the syntax identifier for “local name”
- 7 is the syntax identifier for “global name”
- 23 is the syntax identifier for “idiom”

Compress Vector of Short Integers

$R \leftarrow X(219\pm)Y$

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 i.e. they are type 83. For further information see [Data Representation \(Monadic\) on page 258](#).

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

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. 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←{ω-256×ω>127}
utf8←'UTF-8'∘ucs
str←'empty←θ'
~v←sint utf8 str
101 109 112 116 121 ~30 ~122 ~112 ~30 ~115 ~84
~comp←1 (219I) v
8 ~55 1 0 0 0 0 11 ~80 101 109 112 116 121 ~30 ~122 ~112
~30 ~115 ~84

utf8 256| 0(219I)comp
empty←θ
utf8 256| ~1(219I)2>comp
empty←θ

```

Serialise/Deserialise Array

$R \leftarrow X(220\text{I})Y$

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 i.e. they are type 83. For further information see [Data Representation \(Monadic\) on page 258](#).

It is expected that in many cases this I-Beam functionality will be used in conjunction with **219I** - 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(220I)** to serialise an array.

The result of **1(220I)** 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:

$$A \equiv 0(220\text{I})\ 0(219\text{I})\ 1(219\text{I})\ 1(220\text{I})\ A$$

Example

```

a←'ab'
b←1(220I)a
b
-33 -108 5 0 0 0 31 39 0 0 2 0 0 0 97 98 0 0
c←0(220I)b
c≡a
1

```

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 environment variable **APL_MAX_THREADS**. If this variable is not set, the default is the number of virtual processors that the machine is configured to have.

R is the previous value.

To reset the number of threads to be the same as the number of virtual processors run:

```
{ }1111⊖ 1111⊖0
```

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

Thread Synchronisation Mechanism

R←1113⊖Y

Y is Boolean and specifies whether or not the main thread does a busy wait for the others to complete or uses a semaphore when a function is executed in parallel.

The default and recommended value is 0 (use a semaphore). This function is provided only for Operating Systems that do not support semaphores.

A value of 1 **must** be set if you are running AIX Version 5.2 which does not support Posix semaphores. Later versions of AIX do not have this restriction.

R is the previous value

Update Function Time Stamp

$\{R\} \leftarrow X(1159\mathbb{I})Y$

Y is an array of function names in the same format as the right argument of $\square AT$. For further information, see [Attributes on page 220](#).

X is an array of function attributes in the same format as the output of $\square AT$.

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 time stamp must be set to 0 otherwise **1159 \mathbb{I}** will generate a **DOMAIN ERROR**. Additionally, the time stamp must be greater than **1970 1 1 0 0 0 0**.

Example

```
]disp  $\square AT$ 'Christmas'

0 0 0 | 2013 3 1 11 14 58 0 | 0 | Richard

x $\leftarrow$  $\square AT$  'Christmas'
x[2 4] $\leftarrow$ (2012 12 25 11 59 0 0)('Santa')
x (1159 $\mathbb{I}$ ) 'Christmas'

]disp  $\square AT$ 'Christmas'

0 0 0 | 2012 12 25 11 59 0 0 | 0 | Santa
```

3

Memory Manager Statistics

$$R \leftarrow \{X\} (2000 \pm) Y$$

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.

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" $\square WA$)
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
12	Sediment size
13	Current workspace allocation, i.e. the amount of memory that is actually being used
14	Workspace allocation high-water mark, i.e. the maximum amount of memory that has been used since the workspace was loaded or since this count was reset
15	Limit on minimum workspace allocation
16	Limit on maximum workspace allocation

Note that 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.

Examples

```

2000±0
55414796
2000±0 1 2 3 4 12 13 14 15 16
55414796 10121204 5 0 0 2120524 34489168 34489168 0 65536000

```

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 Θ . 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
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 .

Notes:

- Note that the workspace allocation high-water mark indicates a minimum value for **MAXWS**.
- Limiting the maximum workspace allocation can be used to prevent code which grabs 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

```

2000i2 3
6 0 33216252
0 (2000i)2 3 14 a Reset compaction count

2000i2 3
0 0
30000000 40000000(2000i)15 16 a Restrict min/max ws

(2000i)15 16
30000000 40000000

0 (2000i)15 16 a Reset min/max ws

(2000i)15 16
0 65536000
```

```
(2000I)13 14 A Current, peak WS allocation
4072532 4072532
```

```
a+10e6p'x' A Increase WS allocation
```

```
(2000I)13 14 A Current, peak WS allocation
15108580 15108580
```

```
ex 'a' { }wa A Decrease current WS allocation
```

```
(2000I)13 14 A Current, peak WS allocation
1962856 15108580
```

```
0 (2000I) 14 A Reset High-water mark
```

```
(2000I)13 14 A Current, peak WS allocation
1962856 1962856
```


Specify Workspace Available

R←2002IY

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.

Like **□WA**, **2002I** 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*.

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, **2002I** 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 **2000I**) or larger than the maximum value (which defaults to **MAXWS**), a **DOMAIN ERROR** is signalled.

See also [Memory Manager Statistics on page 170](#).

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.

¹The term *memory* here means virtual memory which includes memory mapped to disk.

Update DataTable

$$R \leftarrow \{X\} 2010 \mp Y$$

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 `dtRef`, `Data`, `NullValues` and `Rows` as described in the table below.

The optional argument `X` is the Boolean vector `ParseFlags` as described in the table below.

Argument	Description
<code>dtRef</code>	A reference to an instance of <code>System.Data.DataTable</code> .
<code>Data</code>	A matrix with the same number of columns as the table.
<code>NullValues</code>	An optional vector with one element per column, containing the value which should be mapped to <code>DBNull</code> when this column is written to the <code>DataTable</code> .
<code>Rows</code>	Row indices (zero origin) of the rows to be updated. If not provided, data will be appended to the <code>DataTable</code> .
<code>ParseFlags</code>	A Boolean vector, where a 1 indicates that the corresponding element of <code>Data</code> is a string which needs to be passed to the <code>Parse</code> method of the data type of column in question.

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 α ω}
    'S1' 'S2' 'I1' 'D1' ac←String String Int32 DateTime
S1 S2 I1 D1

    NextYear←DateTime.Now+{NEW TimeSpan (4↑ω)}↑n←365
    data←(f↑n),(np'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←ω ◇ 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.

2010I does all the looping in compiled code:

```

    dt.Rows.Clear A Delete the rows inserted above
    SetDT←2010I
    t←3>AI ◇ SetDT dt data ◇ (3>AI)-t
4

```

So in this case, using **2010I** achieves over 90,000 rows per second.

Using ParseFlags

Sometimes it is more convenient to handle .NET datatypes in the workspace as strings rather than as the appropriate APL array equivalent. The System.DateTime datatype (which by default is represented in the workspace as a 6-element numeric vector) is one such example. **2010I** will accept such character data and convert it to the appropriate .NET datatype internally.

If specified, the optional left argument **X(ParseFlags)** instructs the system to pass the corresponding columns of Data to the Parse() method of the data type in question prior to performing the update.

```

5      NextYear←⊖⊖`DateTime.Now+{⊖NEW TimeSpan (4↑ω)}⊖⊖`⊖⊖←36
      data←(⊖⊖⊖⊖), (np'odd' 'even'), (10|⊖⊖), NextYear
      ~2 4↑data
364  even   4   18-01-2011 14:03:29
365  odd    5   19-01-2011 14:03:29

      SetDT←2010⊖      0 0 0 1 SetDT dt data

```

Handling Nulls

If applicable, `NullValues` 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 declares 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 `NullValues`, 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 4p'one' 'odd' 1 DateTime.Now) ⊖ 0

```

Note

- the values must be provided as a matrix, even if you only want to update a single row,
- row indices are 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

$$R \leftarrow \{X\} 2011 \mp Y$$

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 datatypes to the appropriate internal APL form 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 scalar or a 2-item array containing `dtRef`, and `NullValues` as described in the table below.

The optional argument `X` is the Boolean vector `ParseFlags` as described in the table below.

The result `R` is the array `Data` as described in the table below.

Argument	Description
<code>dtRef</code>	A reference to an instance of <code>System.Data.DataTable</code> .
<code>Data</code>	A matrix with the same number of columns as the table.
<code>NullValues</code>	An optional vector with one element per column, containing the value to which a <code>DBNull</code> in the corresponding column of the <code>DataTable</code> should be mapped in the result array <code>Data</code> .
<code>ParseFlags</code>	A Boolean vector, where a 1 indicates that the corresponding element of <code>Data</code> should be converted to a string using the <code>ToString()</code> method of the data type of column in question. It is envisaged that this argument may be extended in the future, to allow other conversions – for example converting Dates to a floating-point format.

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.

`2011I` does the looping entirely in compiled code and is significantly faster:

```
GetDT←2011I
t←3>[]AI ♦ data2←GetDT dt ♦ (3>[]AI)-t
25
```

ParseFlags Example

In the example shown above, `2011I` 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 `2011I` 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.

Handling Nulls

Using the DataTable produced by the corresponding example shown for `2010I` it can be shown that by default null values will be read back into the APL workspace as instances of `System.DBNull`.

```
GetDT←2011I>
```

```
z←z←GetDT dt
```

```
      odd  1  21-01-2010 14:50:19
two      2  22-01-2010 14:50:19
three odd   23-01-2010 14:50:19
```

```
(1 1Qz).GetType
```

```
System.DBNull System.DBNull System.DBNull
```

However, by supplying a `NullValues` argument to `2011I`, we can request that nulls in each column are mapped to a corresponding value of our choice; in this case, '`<null>`', '`even`', and `99` respectively.

```
      GetDT dt ('<null>' 'even' 99 '')
<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
```

Data Binding

R←{X}2015±Y

Creates an object that may be used as a data source for WPF data binding.¹

Data binding 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 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.

¹It is beyond the scope of this document to fully explain the concepts of WPF data binding. See Microsoft Developer Network, Data Binding Overview.

`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 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`. It is likely that in a future release a rank-2 array will be bound as a `DataTable`.

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 (not currently supported)

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←2015I'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(2015I)'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(2015I) 'valSource'
```

Binding a Scalar .NET Object

This is not supported in the first release of Version 14.0. It is intended that it will be added in due course.

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←2015I '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, i.e. as a `System.Collection`, suitable for binding to the `ItemSource` property of an `ItemsControl`.

```
□EX 'yearsSource'
  yearsSource←2000+120
  bindSource←2015I '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 (e.g. `Int32`, `Double`) for the array as the left argument. For example:

```
□EX 'arraySource'
  arraySource←42 24
  bindSource←Int32 (2015I) 'arraySource'
```

Binding a Vector of .NET Objects

A vector of .NET objects is bound in the same way as a vector of character vectors, i.e. 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 ω}''Easter
      bindSource←2015⍲'dt'
```

Binding a Matrix

Currently, the system allows a bind variable to contain a matrix (simple or nested) but the default binding type is `System.Object`. This is unlikely to be of any use. It is intended that in a future release of Dyalog APL a matrix will be bound as a `DataTable` or similar.

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←2015I'num'A (default) binding type System.Object

□EX 'num'
num←42
bs←Int32(2015I)'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 2p'txtSource'String'sizeSource'Int32
bindSource←options(2015I)'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"(pWines)p<' '
winelist.Name<Wines
winelist.Price<0.01×10000+?(pWines)p10000

bindSource<2015I'winelist'
```

Flush Session Caption**R←2022IY**

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

```
2022I0
```

Close All Windows**R←2023IY**

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; it is not possible to save a workspace if any editor or tracer windows are open.

Under UNIX, this is the only mechanism for closing all such windows.

Example

```
2023I0
```

Export To Memory**R←2100IY**

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.

Set Workspace Save Options:**R←2400IY**

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

```
(2400I)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
^

```


Disable Component Checksum Validation

{R}←3002±Y

Checksums allow component files to be validated and repaired using **±FCHK**.

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.

Fork New Task (UNIX only)

R←4000⍲Y

Y must be 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 execution stack.

Following the execution of this function, there will be two identical APL processes running on the machine, each with the same execution stack 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 (UNIX only)

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 current user name is *root* (*uid*=0).

This call is intended to be made in the child process after a fork (**4000⊖θ**) in a process with an effective user id of *root*. It can however be used in any APL process with an effective user id of *root*.

If the operation is successful, **R** is the user name specified in **Y**.

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 (UNIX only)

R←4002IY

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. **4002I** 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 `timeval` structs 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 */
};
```

4002I 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;rpids
[1]   rpids←fpids←0
[2]   :For i :In 15
[3]       fpid←4000I'' A fork() a process
[4]   A if the child, hang around for a while
[5]       :If fpid=0
[6]           □DL 2×i
[7]           □OFF
[8]       :Else
[9]   A if the parent, save child's pid
[10]       +fpids,←fpid
[11]       :EndIf
[12]   :EndFor
[13]
[14]   :For i :In 120
[15]       □DL 3
[16]   A get list of newly terminated child processes
[17]       rpids←4002I''
[18]   A and if not empty, make note of their pids
[19]       :If 0≠⇒prpids
[20]           +rpids,←rpids[;1]
[21]       :EndIf
[22]   A if all fork()'d child processes accounted for
[23]       :If fpids≡fpids∪rpids
[24]           :Leave A quit
[25]       :EndIf
[26]   :EndFor
▽

```

Signal Counts (UNIX only) **$R \leftarrow 4007 \pm Y$**

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.

Random Number Generator

R←16807⊖Y

Specifies the random number generator that is to be used by Roll and Deal.

Y is an integer that specifies which random number generator is to be enabled and must be one of the numbers listed in the first column of the table below.

R is an integer that identifies the previous random number generator in use.

The 3 random number generators are as follows :

Id	Algorithm
0	Lehmer linear congruential generator.
1	Mersenne Twister.
2	Operating System random number generator.

Under Windows, the Operating System random number generator uses the `CryptGenRandom()` function. Under UNIX/Linux it uses `/dev/urandom[3]`.

The default random number generator in a **CLEAR WS** is 1 (Mersenne Twister).

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.

The Mersenne Twister algorithm *RNG1* produces 64-bit values with good distribution.

The Operating System algorithm *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.

For further information, see [Random Link on page 416](#).

Chapter 4:

System Functions

Dyalog includes a collection of built-in facilities which 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.

<code>⎕</code>	<code>⎕</code>	<code>⎕Á</code>	<code>⎕A</code>	<code>⎕AI</code>
<code>⎕AN</code>	<code>⎕ARBIN</code>	<code>⎕ARBOUT</code>	<code>⎕AT</code>	<code>⎕AV</code>
<code>⎕AVU</code>	<code>⎕BASE</code>	<code>⎕CLASS</code>	<code>⎕CLEAR</code>	<code>⎕CMD</code>
<code>⎕CR</code>	<code>⎕CS</code>	<code>⎕CT</code>	<code>⎕CY</code>	<code>⎕D</code>
<code>⎕DCT</code>	<code>⎕DF</code>	<code>⎕DIV</code>	<code>⎕DL</code>	<code>⎕DM</code>
<code>⎕DMX</code>	<code>⎕DQ</code>	<code>⎕DR</code>	<code>⎕ED</code>	<code>⎕EM</code>
<code>⎕EN</code>	<code>⎕EX</code>	<code>⎕EXCEPTION</code>	<code>⎕EXPORT</code>	<code>⎕FAPPEND</code>
<code>⎕FAVAIL</code>	<code>⎕FCHK</code>	<code>⎕FCOPY</code>	<code>⎕FCREATE</code>	<code>⎕FDROP</code>
<code>⎕FERASE</code>	<code>⎕FHIST</code>	<code>⎕FHOLD</code>	<code>⎕FIX</code>	<code>⎕FLIB</code>
<code>⎕FMT</code>	<code>⎕FNAMES</code>	<code>⎕FNUMS</code>	<code>⎕FPROPS</code>	<code>⎕FR</code>
<code>⎕FRDAC</code>	<code>⎕FRDCI</code>	<code>⎕FREAD</code>	<code>⎕FRENAME</code>	<code>⎕FREPLACE</code>
<code>⎕FRESIZE</code>	<code>⎕FSIZE</code>	<code>⎕FSTAC</code>	<code>⎕FSTIE</code>	<code>⎕FTIE</code>
<code>⎕FUNTIE</code>	<code>⎕FX</code>	<code>⎕INSTANCES</code>	<code>⎕IO</code>	<code>⎕KL</code>
<code>⎕LC</code>	<code>⎕LOAD</code>	<code>⎕LOCK</code>	<code>⎕LX</code>	<code>⎕MAP</code>
<code>⎕ML</code>	<code>⎕MONITOR</code>	<code>⎕NA</code>	<code>⎕NAPPEND</code>	<code>⎕NC</code>
<code>⎕NCREATE</code>	<code>⎕NERASE</code>	<code>⎕NEW</code>	<code>⎕NL</code>	<code>⎕NLOCK</code>
<code>⎕NNAMES</code>	<code>⎕NNUMS</code>	<code>⎕NQ</code>	<code>⎕NR</code>	<code>⎕NREAD</code>
<code>⎕NRENAME</code>	<code>⎕NREPLACE</code>	<code>⎕NRESIZE</code>	<code>⎕NS</code>	<code>⎕NSI</code>

□NSIZE	□NTIE	□NULL	□NUNTIE	□NXLATE
□OFF	□OPT	□OR	□PATH	□PFKEY
□PP	□PROFILE	□PW	□R	□REFS
□RL	□RSI	□RTL	□S	□SAVE
□SD	□SE	□SH	□SHADOW	□SI
□SIGNAL	□SIZE	□SM	□SR	□SRC
□STACK	□STATE	□STOP	□SVC	□SVO
□SVQ	□SVR	□SVS	□TC	□TCNUMS
□TGET	□THIS	□TID	□TKILL	□TNAME
□TNUMS	□TPOOL	□TPUT	□TRACE	□TRAP
□TREQ	□TS	□TSYNC	□UCS	□USING
□VFI	□VR	□WA	□WC	□WG
□WN	□WS	□WSID	□WX	□XML
□XSI	□XT			

System Constants

System constants, which can be regarded as niladic system functions, return information from the system. They have distinguished names, beginning with the quad symbol, `□`. A system constant may **not** be assigned a value. System constants may not be localised or erased. System constants are summarised in the following table:

Name	Description
<code>□Á</code>	Underscored Alphabetic upper case characters
<code>□A</code>	Alphabetic upper case characters
<code>□AI</code>	Account Information
<code>□AN</code>	Account Name
<code>□AV</code>	Atomic Vector
<code>□D</code>	Digits
<code>□DM</code>	Diagnostic Message
<code>□DMX</code>	Extended Diagnostic Message
<code>□EN</code>	Event Number
<code>□EXCEPTION</code>	Reports the most recent Microsoft .NET Exception
<code>□LC</code>	Line Count
<code>□NULL</code>	Null Item
<code>□SD</code>	Screen (or window) Dimensions
<code>□TC</code>	Terminal Control (backspace, linefeed, newline)
<code>□TS</code>	Time Stamp
<code>□WA</code>	Workspace Available

System Variables

System variables retain information used by the system in some way, usually as implicit arguments to functions.

The characteristics of an array assigned to a system variable must be appropriate; otherwise an error will be reported immediately.

Example

```

      IO←3
DOMAIN ERROR
      IO←3
      ^

```

System variables may 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
<code>IO</code>	Character Input/Output	Session
<code>IOE</code>	Evaluated Input/Output	Session
<code>AVU</code>	Atomic Vector – Unicode	Namespace
<code>CT</code>	Comparison Tolerance	Namespace
<code>DCT</code>	Decimal Comp Tolerance	Namespace
<code>DIV</code>	Division Method	Namespace
<code>FR</code>	Floating-Point Representation	Workspace
<code>IO</code>	Index Origin	Namespace
<code>LX</code>	Latent Expression	Workspace
<code>ML</code>	Migration Level	Namespace
<code>PATH</code>	Search Path	Session

Name	Description	Scope
<code>□PP</code>	Print Precision	Namespace
<code>□PW</code>	Print Width	Session
<code>□RL</code>	Random Link	Namespace
<code>□RTL</code>	Response Time Limit	Namespace
<code>□SM</code>	Screen Map	Workspace
<code>□TRAP</code>	Event Trap	Workspace
<code>□USING</code>	Microsoft .NET Search Path	Namespace
<code>□WSID</code>	Workspace Identification	Workspace
<code>□WX</code>	Window Expose	Namespace

In other words, `□`, `□SE`, `□PATH` and `□PW` relate to the session. `□LX`, `□SM`, `□TRAP` and `□WSID` relate to the active workspace. All the other system variables relate to the current namespace.

Session	Workspace	Namespace
<code>□</code>	<code>□FR</code>	<code>□AVU</code>
<code>□</code>	<code>□LX</code>	<code>□CT</code>
<code>□PATH</code>	<code>□SM</code>	<code>□DCT</code>
<code>□PW</code>	<code>□TRAP</code>	<code>□DIV</code>
	<code>□WSID</code>	<code>□IO</code>
		<code>□ML</code>
		<code>□PP</code>
		<code>□RL</code>
		<code>□RTL</code>
		<code>□USING</code>
		<code>□WX</code>

System Operators

The following system facilities are for convenience implemented as operators rather than as functions:

Name	Description
<code>□R</code>	Replace
<code>□S</code>	Search
<code>□OPT</code>	Variant (Classic Edition only)

System Namespaces

`□SE` is currently the only system namespace.

System Functions Categorised

Dyalog includes a collection of built-in facilities which 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.

The following tables list the system functions divided into appropriate categories. Each is then described in detail in alphabetical order.

Settings Affecting Behaviour of Primitive Functions

Name	Description
<code>⎕CT</code>	Comparison Tolerance
<code>⎕DCT</code>	Decimal Comp Tolerance
<code>⎕DIV</code>	Division Method
<code>⎕FR</code>	Floating-Point Representation
<code>⎕IO</code>	Index Origin
<code>⎕ML</code>	Migration Level
<code>⎕PP</code>	Print Precision
<code>⎕RL</code>	Random Link

Session Information/Management

Name	Description
<code>AI</code>	Account Information
<code>AN</code>	Account Name
<code>CLEAR</code>	Clear workspace (WS)
<code>CY</code>	Copy objects into active WS
<code>DL</code>	Delay execution
<code>LOAD</code>	Load a saved WS
<code>OFF</code>	End the session
<code>PATH</code>	Search Path
<code>SAVE</code>	Save the active WS
<code>TS</code>	Time Stamp

Constants

Name	Description
<code>A</code>	Alphabetic upper case characters
<code>D</code>	Digits
<code>NULL</code>	Null Item

Tools and Access to External Utilities

Name	Description
<code>□CMD</code>	Execute the Windows Command Processor or another program
<code>□CMD</code>	Start a Windows AP
<code>□DR</code>	Data Representation (Monadic)
<code>□DR</code>	Data Representation (Dyadic)
<code>□FMT</code>	Resolve display
<code>□FMT</code>	Format array
<code>□MAP</code>	Map a file
<code>□NA</code>	Declare a DLL function
<code>□R</code>	Replace
<code>□S</code>	Search
<code>□SH</code>	Execute a UNIX command or another program
<code>□SH</code>	Start a UNIX AP
<code>□UCS</code>	Unicode Convert
<code>□USING</code>	Microsoft .NET Search Path
<code>□VFI</code>	Verify and Fix numeric
<code>□XML</code>	XML Convert

Manipulating Functions and Operators

Name	Description
<code>□AT</code>	Object Attributes
<code>□CR</code>	Canonical Representation
<code>□ED</code>	Edit one or more objects
<code>□EX</code>	Expunge objects
<code>□FX</code>	Fix definition
<code>□LOCK</code>	Lock a function
<code>□NR</code>	Nested Representation
<code>□PROFILE</code>	Profile Application
<code>□REFS</code>	Local References
<code>□STOP</code>	Set Stop vector
<code>□STOP</code>	Query Stop vector
<code>□TRACE</code>	Set Trace vector
<code>□TRACE</code>	Query Trace vector
<code>□VR</code>	Vector Representation

Namespaces and Objects

Name	Description
<code>□BASE</code>	Base Class
<code>□CLASS</code>	Class
<code>□CS</code>	Change Space
<code>□DF</code>	Display Format
<code>□FIX</code>	Fix
<code>□INSTANCES</code>	Instances
<code>□NEW</code>	New Instance
<code>□NS</code>	Create Namespace
<code>□SRC</code>	Source
<code>□THIS</code>	This

Input/Output

Name	Description
<code>□</code>	Evaluated Input/Output
<code>□</code>	Character Input/Output

Built-in GUI and COM Support

Name	Description
<code>□DQ</code>	Await and process events
<code>□EXPORT</code>	Export objects
<code>□NQ</code>	Place an event on the Queue
<code>□WC</code>	Create GUI object
<code>□WG</code>	Get GUI object properties
<code>□WN</code>	Query GUI object Names
<code>□WS</code>	Set GUI object properties

Component Files

Name	Description
<code>□FAPPEND</code>	Append a component to File
<code>□FAVAIL</code>	File system Availability
<code>□FCHK</code>	File Check and Repair
<code>□FCOPY</code>	Copy a File
<code>□FCREATE</code>	Create a File
<code>□FDROP</code>	Drop a block of components
<code>□FERASE</code>	Erase a File
<code>□FHIST</code>	File History
<code>□FHOLD</code>	File Hold
<code>□FLIB</code>	List File Library
<code>□FNAMES</code>	Names of tied Files
<code>□FNUMS</code>	Tie Numbers of tied Files
<code>□FPROPS</code>	File Properties
<code>□FRDAC</code>	Read File Access matrix
<code>□FRDCI</code>	Read Component Information
<code>□FREAD</code>	Read a component from File
<code>□FRENAME</code>	Rename a File
<code>□FREPLACE</code>	Replace a component on File
<code>□FRESIZE</code>	File Resize
<code>□FSIZE</code>	File Size
<code>□FSTAC</code>	Set File Access matrix
<code>□FSTIE</code>	Share-Tie a File
<code>□FTIE</code>	Tie a File exclusively
<code>□FUNTIE</code>	Untie Files

Native Files

Name	Description
<code>□NAPPEND</code>	Append to File
<code>□NCREATE</code>	Create a File
<code>□NERASE</code>	Erase a File
<code>□NLOCK</code>	Lock a region of a file
<code>□NNames</code>	Names of tied Files
<code>□NNums</code>	Tie Numbers of tied Files
<code>□NREAD</code>	Read from File
<code>□NRENAME</code>	Rename a File
<code>□NREPLACE</code>	Replace data on File
<code>□NRESIZE</code>	File Resize
<code>□NSIZE</code>	File Size
<code>□NTIE</code>	Tie a File exclusively
<code>□NUNTIE</code>	Untie Files
<code>□NXLATE</code>	Specify Translation Table

Threads

Name	Description
□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
□TGET	Get Tokens
□TKILL	Kill Threads
□TPOOL	Token Pool
□TPUT	Put Tokens
□TREQ	Token Requests

Error Handling

Name	Description
□DMX	Extended Diagnostic Message
□EM	Event Messages
□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
□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

Various Other

Name	Description
<code>□Á</code>	Underscored Alphabetic Characters
<code>□ARBIN</code>	Arbitrary Input
<code>□ARBOUT</code>	Arbitrary Output
<code>□AV</code>	Atomic Vector
<code>□AVU</code>	Atomic Vector - Unicode
<code>□DM</code>	Diagnostic Message
<code>□EN</code>	Event Number
<code>□KL</code>	Key Labels
<code>□PFKEY</code>	Programmable Function Keys
<code>□SD</code>	Screen Dimensions
<code>□SM</code>	Screen Map
<code>□SR</code>	Screen Read
<code>□MONITOR</code>	Monitor set
<code>□MONITOR</code>	Monitor query
<code>□NXLATE</code>	Specify Translation Table
<code>□OPT</code>	Variant Operator
<code>□OR</code>	Object Representation
<code>□RTL</code>	Response Time Limit
<code>□TC</code>	Terminal Control
<code>□XT</code>	Associate External variable
<code>□XT</code>	Query External variable
<code>□WX</code>	Expose GUI property names

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 *Programmer's Guide: 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
```

```
      ρR
14
```

The output of simple arrays of rank greater than 1 through `PRINT` includes a new-line character at the end of each line. Input through `INPUT` includes the preceding output through `PRINT` since the last new-line character. The result from `INPUT`, including the prior output, is limited to 256 characters.

A soft interrupt causes an **INPUT INTERRUPT** error if entered while `INPUT` is awaiting input, and execution is then suspended (unless the interrupt is trapped):

```
R←A
```

(Interrupt)

INPUT INTERRUPT

A time limit is imposed on input through `INPUT` if `RTL` is set to a non-zero value:

```
RTL←5 ♦ A←'PASSWORD ? ' ♦ R←A
PASSWORD ?
TIMEOUT
RTL←5 ♦ A←'PASSWORD : ' ♦ R←A
      ^
```

The **TIMEOUT** interrupt is a trappable event.

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 *Programmer's Guide: Display of Arrays*).

Example

```
⎕←2+⍳5
3 4 5 6 7

⎕←2 4ρ'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):

```

p3,
:
)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.

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
ABCDEFGHIJKLMNOPQRSTUVWXYZ

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
 ÁÂÃÇÈÉÊËÌÍÎÏÐÒÓÔÕÖÙÚÛÜÝÞàíðòó

Alphabetic Characters

R←⎕A

This is a simple character vector, composed of the letters of the alphabet.

Example

⎕A
 ABCDEFGHIJKLMNOPQRSTUVWXYZ

Account Information

R←AI

This is a simple integer vector, whose four elements are:

AI[1]	user identification. ¹
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
```

¹Under Windows, this is the `apluid` (network ID from configuration dialog box). Under UNIX and Linux this is the *effective* UID of the account.

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.

Example

```
AN
Pete

ρAN
4
```

Arbitrary Output

{X}▯ARBOU Y

This transmits **Y** to an output device specified by **X**.

Under Windows, the use of **▯ARBOU** to the screen or to RS232 ports is not supported.

Y may be a scalar, a simple vector, or a vector of simple scalars or vectors. The items of the simple arrays of **Y** must each be a character or a number in the range 0 to 255. Numbers are sent to the output device without translation. Characters undergo the standard **▯AV** to ASCII translation. If **Y** is an empty vector, no codes are sent to the output device.

X defines the output device. If **X** is omitted, output is sent to standard output (usually the screen). If **X** is supplied, it must be a simple numeric scalar or a simple text vector.

If it is a numeric scalar, it must correspond to a Windows device handle or UNIX stream number.

If it is a text vector, it must correspond to a valid device or file name.

You must have permission to write to the chosen device.

Examples

Write ASCII digits '123' to UNIX stream 9:

```
9 ▯ARBOU 49 50 51
```

Write ASCII characters 'ABC' to MYFILE:

```
'MYFILE' ▯ARBOU 'ABC'
```

Beep 3 times:

```
▯ARBOU 7 7 7
```

Prompt for input:

```
▯← 'Prompt: ' ♦ ▯arbout 12 ♦ ans←▯
```

Attributes

 $R \leftarrow \{X\} \square AT Y$

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 (\square 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	0
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) which both report the **SIZE** of the object.

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.

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 52p12+⎕av
%'αω_abcdefghijklmnopqrstuvwxyz_-.0123456789_π¥$£¢
ΔABCDEFHIJKLMNOPQRSTUVWXYZ_γ·ΔΔÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÓÔÕÖÙÚ
ÝÞàîððö{€}~"ÀÄÅÆ~ÉÑÖÜßàáâãäåæçèéêëìíîï[/\<=>≠v^
-+÷×?ερ~†‡ο*[ \∇ο( cοnυ⊥τ| ; , ~ñψΔQφθ⊞!⊞±∇∇≡óôöø"#_&'
-----@ùúû^ü`⎕⌥:εζι◊↔A) ]ΔΔS⎕⎕*%'αω_abcdefghijklmnopqrstuvwxyz
```

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**.

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 13p⎕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 13p⎕AVU[97+⌈26]
ÁÃÇÈÊËÌÍÎÏÐÒ
ÓÔÕÙÚÝþǣǿǾ
```

`⎕AVU` has namespace scope and 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 sa
ved Thu Dec 06 11:24:32 2007

      2 13p⎕AVU[97+⌈26]
9398 9399 9400 9401 9402 9403 9404 9405 9406 9407 9408 94
09 9410
9411 9412 9413 9414 9415 9416 9417 9418 9419 9420 9421 94
22 9423
      ⎕UCS 2 13p⎕AVU[97+⌈26]
ABCDEFGHIJKLM
NOPQRSTUVWXYZ
```

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 (i.e. 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.

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 A Parrot

:Class DomesticParrot: Parrot
  ▽ R←Speak
  :Access Public
  R←BASE.Speak, ' Who''s a pretty boy, then!'
  ▽
:EndClass A DomesticParrot

Maccaw←NEW Parrot
Maccaw.Speak
Squark!

Polly←NEW DomesticParrot
Polly.Speak
Squark! Who's a pretty boy, then!
```

Class **$R \leftarrow \{X\} \square \text{CLASS } Y$** **Monadic Case**

Monadic $\square \text{CLASS}$ 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 or 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

\square CLASS Eeyore\square NEW Animal
#.Animal
\square CLASS Robin\square NEW Bird
#.Bird      #.Animal
\square CLASS Polly\square NEW Parrot
#.Parrot    #.Bird    #.Animal

\square CLASS'' Parrot Animal
#.Parrot    #.Bird    #.Animal    #.Animal
```


Example 2

The Penguin Class example (see *Programmer's Guide: Penguin Class 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 *Programmer's Guide: Penguin Class 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!
```

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.

The contents of the session namespace `⌈SE` are not affected.

Example

```
⌈CLEAR
⌈WSID
CLEAR WS
```

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 [Execute \(UNIX\) Command on page 421](#) for a discussion of the behaviour of these system functions under UNIX.

The system commands **SH** and **CMD** provide similar facilities. For further information, see [Execute \(UNIX\) Command on page 518](#) and [Windows Command Processor on page 502](#).

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'
ρZ
8
      ↑Z
Volume in drive C has no label
Directory of C:\DYALOG

.                <DIR>      5-07-89   3.02p
..               <DIR>      5-07-89   3.02p
SALES    DWS      110092 5-07-89   3.29p
EXPENSES DWS      154207 5-07-89   3.29p
```

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 <code>⎕CMD</code> 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.

<code>'Normal'</code> <code>''</code>	Application is started in a normal window, which is given the input focus
<code>'Unfocused'</code>	Application is started in a normal window, which is NOT given the input focus
<code>'Hidden'</code>	Application is run in an invisible window
<code>'Minimized'</code> <code>'Minimised'</code>	Application is started as an icon which is NOT given the input focus
<code>'Maximized'</code> <code>'Maximised'</code>	Application is started maximized (full screen) and is given the input focus

An application started by `□CMD` may ONLY be terminated by itself or by the user. There is no way to close it from APL. 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).

Start Windows Auxiliary Processor

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 [Start UNIX Auxiliary Processor on page 422](#) 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). See *User Guide* for details of how to write an AP.

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 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.

Canonical Representation

$$R \leftarrow \square CR \quad Y$$

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      A Arithmetic mean
[1]  R←(+/X)÷ρX
[2]  ▽
      +F←□CR'MEAN'
R←MEAN X      A Arithmetic mean
R←(+/X)÷ρX

      ρF
2 30

```

The definition of $\square CR$ has been extended to names assigned to functions by specification (\leftarrow), 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'
+
      ρF
1
      C←□CR
      C'C'
□CR
      ρC'C'
1

```



```

       $\nabla R \leftarrow \text{CONDITION (FN1 ELSE FN2) } X$ 
[1]    $\rightarrow \text{CONDITION/L1}$ 
[2]    $R \leftarrow \text{FN2 } X \diamond \rightarrow 0$ 
[3]    $L1: R \leftarrow \text{FN1 } X$ 
[4]    $\nabla$ 

```

```

      2  $\square$ STOP 'ELSE'
      ( $X \geq 0$ )  $\square$  ELSE  $\square$   $X \leftarrow -2.5$ 

```

```

ELSE[2]
      X
-2.5
       $\square$ CR 'FN2'
 $\square$ 
       $\rightarrow \square$ 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 \square CRs, so in this respect the definition of \square CR is recursive. Primitive operators are treated like primitive functions, and are represented by their corresponding symbols. Arrays are represented by themselves.

Example

```

      BOX  $\leftarrow$  2 2  $\circ$   $\rho$ 
      +F  $\leftarrow \square$ CR 'BOX'
      2 2  $\circ$   $\rho$ 
       $\rho$ F
      3
      ]display F
      .----->
      | .-----> |
      | | 2 2 |  $\circ$   $\rho$  |
      | |-----| |
      |  $\epsilon$ -----> |

```

If Y is a name assigned to a defined function, R is the \square 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  $\leftarrow$  MEAN
       $\square$ CR 'AVERAGE'
      R  $\leftarrow$  MEAN X       $\rho$  Arithmetic mean
      R  $\leftarrow$  (+/X)  $\div$   $\rho$ X

```

Change Space

$$\{R\} \leftarrow \{X\} \square CS \ Y$$

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 ' $\square SE$ ', ' $\#$ ', and ' $\#\#$ '.

The result R is the full name (starting $\#$.) of the space in which the function or operator was executing prior to the $\square CS$.

$\square CS$ 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 $\square CS$** , 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 (i.e. those local to the current function or operator) are, with the exception of those with name class 9, unaffected. Local names with name class 9 are however no longer 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 $\square CS$ 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 $\square CS$ with an empty argument Y obtains the namespace in which a function is currently executing.

Example

This simple example illustrates how $\square CS$ 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 $\square CS$ 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
  A Note that both instances reference
  A the same callback function
  'B1'□WS'Event' 'Select' 'PRESS'
  'B2'□WS'Event' 'Select' 'PRESS'

  A Initialise total and instance counts.
  TOTAL ← B1.COUNT + B2.COUNT ← 0

▽ PRESS MSG
[1]  'FMT' 'TOTAL'□CS>MSG A      Switch to instance space
[2]  (TOTAL COUNT)++1  A      Incr total & instance count
[3]  □WS'Caption'(COUNT FMT TOTAL)A Set instance caption
▽

▽ CAPT←INST FMT TOTL      A Format button caption.
[1]  CAPT←(⌘INST),'/',⌘TOTL  A E.g. 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      A Display namespace tree
[1]   tabs,space
[2]   'tree'CS space
[3]   →(psubs←↓NL 9)↓0
[4]   (tabs,'.'')*tree"subs
    ▽

    )ns x.y
#.x.y
    )ns z
#.z
    ''tree '#
#
.   x
.   .   y
.   z

```

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:

$$(|X - Y| \leq \square CT \times (|X| \vee |Y|) \quad \text{where } \leq \text{ is applied without tolerance.}$$

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 2×10^{-32} (about 2.3×10^{-10}). A value of 0 ensures exact comparison. The value in a clear workspace is 1×10^{-14} .

□CT is an implicit argument of the monadic primitive functions Ceiling (⌈), Floor (⌊) and Unique (⋈), and of the dyadic functions Equal (=), Excluding (⊖), Find (⋈), Greater (>), Greater or Equal (≥), Index of (⋈), Intersection (⋈), Less (<), Less or Equal (≤), Match (≡), Membership (∈), Not Match (≠), Not Equal (≠), Residue (⌊) and Union (⋈), as well as □FMT O-format.

If □FR is 1287, the system uses □DCT.

Examples

```
□CT←1E-10
1.000000000001 1.0000001 = 1
1 0
```

Copy Workspace

{X}□CY Y

Y must be a simple character scalar or vector identifying a saved workspace. **X** is optional. If present, it must be a simple character scalar, vector or matrix. A scalar or vector is treated as a single row matrix. Each (implied) row of **X** is interpreted as an APL name.

Each (implied) row of **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.

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 [Copy Workspace on page 504](#) for further information and, in particular, the manner in which dependant objects are copied.

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 [Atomic Vector - Unicode on page 224](#) 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

      (2 3p'□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

```

Digits**R←D**

This is a simple character vector of the digits from 0 to 9.

Example

```

D
0123456789

```

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 $2.3283064365386962890625 \times 10^{-10}$). A value of 0 ensures exact comparison. The value in a clear workspace is 1×10^{-28} .

For further information, see [Comparison Tolerance on page 241](#).

Examples

```

DCT←1E-10
1.00000000001 1.0000001 = 1
1 0

```


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 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*).

The result **R** is the previous value of the Display Form which initially is **NULL**.

```

      'F' WC 'Form'
      F
#.F
      F
3
      FMT F
#.F
      FMT F
1 3
      F A default display uses F
#.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 5pA
      F
ABCDE
FGHIJ

KLMNO
PQRST
      F
2 2 5

```

```

      PD←⊖FMT F
ABCD E
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)A set display form to current captio
n
      F
This is the Caption

      F.Caption←'New Caption' A changing caption does not
      A 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 A 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

```

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 (**÷**).

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
```

Delay **$\{R\} \leftarrow \square DL \ Y$**

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 an scalar numeric value indicating the length of the pause in seconds.

The pause may be interrupted by a strong interrupt.

Diagnostic Message **$R \leftarrow \square DM$**

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      ^

```

Extended Diagnostic Message

R←□DMX

□DMX is a system object that provides information about the last reported APL error. □DMX has *thread scope*, i.e. 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 <i>thread safe</i>
EM	character vector	Event message; this is the same as □EM □EN
EN	integer	Error number. This is the same as □EN, but <i>thread safe</i>
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 <i>http:</i> , <i>https:</i> , <i>mailto:</i> and <i>www</i> . 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
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 the operating system error was retrieved. 0 = by the C-library <code>errno()</code> function 1 = by the Windows <code>GetLastError()</code> function
2	integer	Error code. The error number returned by the operating system using <code>errno()</code> or <code>GetLastError()</code> 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
HelpURL http://help.dyalog.com/dmx/13.1/General/1

□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
ρ⌈FMT ⌈DMX
0 0

```

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
HelpURL  http://help.dyalog.com/dmx/13.1/Componentfilesystem/9
1
```

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 tying 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.
HelpURL  http://help.dyalog.com/dmx/13.1/Componentfilesystem/11
```

Once again, the local value of `DMX` is discarded on exit from the `:Trap`, revealing the shadowed value as before.

```

      pFMT 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' 'o'
[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

```

Dequeue Events

{R}←DQ Y

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 `'.'`, 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, or 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, i.e. 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 {larg}

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& {larg}

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 \square DQ

\square DQ terminates, returning the shy result **R**, in one of four instances.

Firstly, \square 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 numeric code specifying the event type.

\square 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 \square 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, \square 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 (e.g. by pressing an "OK" button). The return value of \square DQ in this case depends on the action code of the event.

Finally, \square DQ terminates with a **VALUE ERROR** if it attempts to execute a callback function that is undefined.

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 12: 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

Table 13: 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

Note that types **80**, **160** and **320** and **83** and **163** and **1287** are exclusive to Dyalog APL.

Data Representation (Dyadic)

$R \leftarrow X \square DR Y$

Dyadic $\square DR$ converts the data type of its argument Y according to the type specification X . See [Data Representation \(Monadic\) above](#) 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 seen as Boolean will have 8 times as many elements along the last axis.

Case 2:

X is a 2-element integer value. The bits in the right argument are interpreted as type $X[1]$. The system then attempts to convert the elements of the resulting array to type $X[2]$ 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.

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 $\square NAPPEND$ and $\square NREPLACE$ is recommended.

Conversion to and from character (data type 82) uses the translate vector given by $\square NXLATE\ 0$. By default this is the mapping defined by the current output translate table (usually WIN.DOT).

Note. 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.

Edit Object

 $\{R\} \leftarrow \{X\} \text{ED } Y$

`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 with as many elements as there are names in `Y`. Each element of `X` specifies the type of the corresponding (new) object named in `Y`, where:

▽	function/operator
→	simple character vector
€	vector of character vectors
–	character matrix
⊗	Namespace script
○	Class script
◦	Interface

If an object named in `Y` already exists, the corresponding type specification in `X` is ignored.

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 (e.g. `a.b.c.foo`) are silently ignored if the namespace does not exist.

Event Message

$$R \leftarrow \square EM \ Y$$

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 ".

Example

```
 $\square EM \ 11$ 
DOMAIN ERROR
```

Event Number

$$R \leftarrow \square EN$$

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 $\square SIGNAL$ system function. Its value in a clear workspace is 0 .

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←DateTime.New 100000 0 0
EXCEPTION
DT←DateTime.New 100000 0 0

□EN
90
□EXCEPTION.Message
Specified argument was out of the range of valid values.

Parameter name: Year, Month, and Day parameters describe
an unrepresentable 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)

```

Expunge Object

 $\{R\} \leftarrow \square EX \ Y$

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 objects owned by the current thread **except** for the Root object itself. 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.

Export Object

$$\{R\} \leftarrow \{X\} \square \text{EXPORT } Y$$

$\square \text{EXPORT}$ is used to set or query the export type of a defined function (or operator) referenced by the $\square \text{PATH}$ 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 $\square \text{PATH}$ 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 $\square \text{PATH}$ 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 $\square \text{EXPORT}$ may be extended in future to include extra types 2, 3,... (for example, to change the behaviour of the function). This means that, while $\square \text{EXPORT}$ returns a Boolean result in the first version, this may not be the case in the future. If you need a Boolean result, use $0 \neq$ or an equivalent.

```
(0≠□EXPORT □nl 3 4)≠□nl 3 4  A list of exported
                                A functions and ops.
```

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 3p16) 'Geoff' (□OR'FOO') □FAPPEND 1
12

□←A B C □FAPPEND"1
13 14 15

Dump←{
  tie←α □FCREATE 0           A create file.
  (□FUNTIE tie){}ω □FAPPEND tie A append and untie.
}
```

File System Available

`R←□FAVAIL`

This niladic function returns the scalar value 1 unless the component file system is unavailable for some reason, in which case it returns scalar 0. If `□FAVAIL` does return 0, most of the component file system functions will generate the error message:

`FILE SYSTEM NOT AVAILABLE`

See *User Guide* for further details.

File Check and Repair

$$R \leftarrow \{X\} \text{ FCHK } Y$$

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. 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**, i.e. 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 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.

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, i.e. only those components that were written with the checksum option enabled (see [File Properties on page 289](#)).

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.

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.

□FCOPY creates a copy of the tied file specified by **Y**, named **X**. The new file **X** will be always be a large-span file, but will otherwise be identical to the original file. In particular all component level information, including the user number and update time, will be the same. The operating system file creation, modification and access times will be set to the time at which the copy occurred.

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).

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 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 64¹.

The *file tie number* must not be the tie number associated with another tied file.

X must be either

- a. a simple character scalar or vector which specifies the name of the file to be created. See *User Guide* for file naming conventions under UNIX and Windows.
- b. a vector of length 1 or 2 whose items are:
 - i. a simple character scalar or vector as above.
 - ii. 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, □FNUMS      A With next available number,
file □FCREATE tie      A ... create file.
```

to:

```
tie←file □FCREATE 0 A Create with first available..
```

Examples

```
'..\BUDGET\SALES'      □FCREATE 2      A Windows
'../budget/SALES.85'   □FCREATE 2      A UNIX

'COSTS' 200000 □FCREATE 4              A max size 20000
0
```

¹This 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.

The Principal Option is neither `'J'` nor `'C'` - but a combination 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

```

1      'newfile' (␣FCREATE␣3) 0
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:

```

1      'newfile' JFCREATE 0

```

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 A Append index to dictionary
cnos~←FDROP tie,-pvec A 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
```

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, and the file must be exclusively tied. The file named in **X** is erased and untied. See *User Guide* for file naming conventions under UNIX and Windows.

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
^
```

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)
3. Last update of the access matrix
4. (Undefined)
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).

Currently, the second and fourth rows of the result (undefined) contain (0 0).

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'ω}
[6] ToTS←{d t←1 1 0 0 0←0[0 24 60 60 60τω
[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←((c'),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

```

```

(ι10)□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

```

File Hold

$$\{R\} \leftarrow \square F\text{HOLD } Y$$

Access code 2048

This function holds component file(s) and/or external variable(s).

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 non-simple scalar or vector of character vectors, each of which is the name of an external variable. (NOT the file names associated with those variables).

If applied to component files **and** external variables, Y is a vector whose elements are either integer scalars representing tie numbers, or character vectors containing names of external variables.

The effect is as follows:

1. The user's preceding holds (if any) are released.
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 $\square F\text{HOLD}$ on any of the designated files will wait.

If Y is empty, the user's preceding hold (if any) is released, and execution continues.

A hold is released by any of the following:

- Another $\square F\text{HOLD}$
- 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.

Note that a hold is not released by a request for input through \square or \square .

Note also that point 5 above implies that $\square F\text{HOLD}$ is generally useful only when called from a defined function, as holds set in immediate execution (desk calculator) mode are released immediately.

The shy result of $\square F\text{HOLD}$ is a vector of tie numbers of the files held.

Examples:

```

❑F HOLD 1

❑F HOLD 0

❑F HOLD ← 'XTVAR'

❑F HOLD 1 2, [0.5] 0 16385

❑F HOLD 1 'XTVAR'

```

Fix Script

$$\{R\} \leftarrow \{X\} \text{❑F IX } Y$$

❑F IX fixes a Class from the script specified by *Y*.

Y must be a vector of character vectors or character scalars that contains a well-formed Class script. If so, the shy result *R* is a reference to the new Class fixed by ❑F IX.

The Class specified by *Y* may be named or unnamed.

If specified, *X* must be a numeric scalar. If *X* is omitted or non-zero, and the Class script *Y* specifies a name (for the Class), ❑F IX establishes that Class in the workspace.

If *X* is 0 or the Class specified by *Y* is unnamed, the Class is not established *per se*, although it will exist for as long as a reference to it exists.

In the first example, the Class specified by *Y* is named (**MyClass**) but the result of ❑F IX is discarded. The end-result is that **MyClass** is established in the workspace as a Class.

```

❑←❑F IX ':Class MyClass' ':EndClass'
#.MyClass

```

In the second example, the Class specified by *Y* is named (**MyClass**) and the result of ❑F IX 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←❑F IX ':Class MyClass' ':EndClass'
)CLASSES
MyClass MYREF
❑NC'MyClass' 'MYREF'
9.4 9.4
MYREF
#.MyClass

```

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

```

The final 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

```

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

```

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'

      ρA
1 1
A
A

      A←⌵VR 'FOO'

      A
▽ R←FOO
[1] R←10
▽

      ρA
31
B←⌵FMT A

      B
▽ R←FOO
[1] R←10
▽

      ρB
3 12

```

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 < >, c >, □ □ 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
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 \square t \square	prefixes negative results with the text t instead of the negative sign.
N \square t \square	post-fixes negative results with the text t
P \square t \square	prefixes positive or zero results with the text t .
Q \square t \square	post-fixes positive or zero results with the text t .
R \square t \square	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'  $\square$ FMT 10 20 30
10
20
30
```

The format specification is used cyclically to format the columns of the right argument:

```
'I3,F5.2'  $\square$ FMT 2 4p18
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'  $\square$ FMT (14)(2 2p 0.1 $\times$ 14)
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'  $\square$ FMT 1 6p'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 (13)(2 3p'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 6p16)('*')(1 3p'ABC')(22.2)
ABC*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
33333333333333333333__._
```

The Text Insertion format phrase inserts the given text repeatedly in all rows of the result:

```
MEN←3 5p'FRED BILL JAMES'
WOMEN←2 5p'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  3p-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  3pφ100|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 either side of it. Otherwise a blank appears. Similarly text between digit selectors appears only if digits appear on either 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 18 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
```

```
p←'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"
00.001<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
```

File Names

R←FNMAMES

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. 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

      FNMAMES
/usr/pete/SALESFILE
../budget/COSTFILE
PROFIT

      ρFNMAMES
3 19
      FNUMS,FNMAMES
16 /usr/pete/SALESFILE
2 ../budget/COSTFILE
5 PROFIT

```

File Numbers

R←FNUMS

The result is an integer vector of the **tie numbers** 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

```

      '/usr/pete/SALESFILE' FSTIE 16
      '../budget/COSTFILE' FSTIE 2
      'PROFIT' FCREATE 5

      FNUMS
16 2 5

      FNUMS,FNAMES
16 /usr/pete/SALESFILE
2 ../budget/COSTFILE
5 PROFIT

      FUNTIE FNUMS
ρFNUMS
0

```

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 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 <i>APL crash proof</i> Journaling 2 = Enable <i>System crash proof</i> Journaling; repair needed on recovery 3 = Enable full <i>System crash proof</i> 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 [File Create on page 271](#).

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 User Guide).

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 Journaling 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 (e.g. 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.

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*¹ 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*² 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 workspace scope, and 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

¹http://en.wikipedia.org/wiki/Double_precision_floating-point_format

²http://en.wikipedia.org/wiki/Decimal128_floating-point_format

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) 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 $-1\text{E}6145$ to $1\text{E}6145$. 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.

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.

See "File Access Control" in *User Guide* for further details.

Examples

```

⌈FRDAC 1
28 2105 16385
0 2073 16385
31 -1 0

```

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 (i.e. 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
```

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 ω 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

```
ρSALES←FREAD 1 241
3 2 12
```

GetFile←{io←0	A Extract contents.
tie←ω ifstie 0	A new tie number.
fm to←2↑fsize tie	A first and next component.
cnos←fm+1to-fm	A vector of component nos.
cvec←fread tie cnos	A vector of components.
cvec{α}funtie tie	A ... untie and return.
}	

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.

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←ω
  □FUNTIE to □FRENAME fm □FTIE 0
}
```

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 [File Read Component Information on page 296](#)) 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←α □FTIE 0
  ←{ω □FREPLACE tie α}/''ω
  □FUNTIE tie
}

'jms'Frep(3 'abc')(29 'xxx')(7 'yyy')
```

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. 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 1.844674407E19
(10 1000p1.1)□FAPPEND 1 ♦ □FSIZE 1
1 2 80288 1.844674407E19

100000 □FRESIZE 1 A Limit size to 100000 bytes

(10 1000p1.1)□FAPPEND 1
FILE FULL
(10 1000p1.1)□FAPPEND 1
^

□FRESIZE 1      A Force file compaction.
```


File Size

R←F SIZE 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, (i.e. the result of the next FAPPEND)
3	the current size of the file in bytes
4	the file size limit in bytes

Example

```
F SIZE 1
1 21 65271 4294967295
```

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, i.e. a 3 column integer matrix with any number of rows.

See "File Access Control" in *User Guide* for further details.

The shy result of FSTAC is the tie number of the file.

Examples

```
SALES FCREATE 1
(3 3p28 2105 16385 0 2073 16385 31 ^1 0) FSTAC 1
((F RDAC 1)^21 2105 16385) FSTAC 1

(1 3p0 ^1 0)FSTAC 2 n Allow everyone access
```

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.

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  A With next available number,
file □FSTIE tie    A ... share tie file.
```

to:

```
tie←file □FSTIE 0 A Tie with 1st available number.
```

Example

```
'SALES' □FSTIE 1
'../budget/COSTS' □FSTIE 2
```

Tieing Small-Span Component Files

To tie a small-span file you must specify the `ReadOnly` option via the Variant operator `⌈`.

Example

```
'old-32-bit-file' (⌈FTIE⌈'ReadOnly' 1)1
'SEUJC' ⌈FPROPS 1
32 0 0 1 1
```

Note that there is no Principle Option for this function; you must specify the `ReadOnly` option by name.

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.

The file must exist and be accessible by the user. 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 a With next available number,
file □FTIE tie  a ... tie file.
```

to:

```
tie←file □FTIE 0 a 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

File Untie

$\{R\} \leftarrow \square\text{FUNTIE } Y$

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 $\square\text{FUNTIE}$ is a vector of tie numbers of the files **actually untied**.

Example

```
 $\square\text{FUNTIE } \square\text{FNUMS } A$  Unties all tied files
 $\square\text{FUNTIE } \emptyset$       A Flushes all buffers to disk
```

Fix Definition

$\{R\} \leftarrow \square\text{FX } Y$

Y is the representation form of a function or operator which may be:

- its canonical representation form similar to that produced by $\square\text{CR}$ except that redundant blanks are permitted other than within names and constants.
- its nested representation form similar to that produced by $\square\text{NR}$ except that redundant blanks are permitted other than within names and constants.
- its object representation form produced by $\square\text{OR}$.
- its vector representation form similar to that produced by $\square\text{VR}$ except that additional blanks are permitted other than within names and constants.

$\square\text{FX}$ attempts to create (fix) a function or operator in the workspace or current namespace from the definition given by Y . $\square\text{IO}$ is an implicit argument of $\square\text{FX}$. Note that $\square\text{FX}$ 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 $\square\text{FIX}$.

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 ($\square\text{IO}$ 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**.

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.

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

```

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 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 (ι), Roll (?), and of the dyadic functions Deal (?), Grade Down (▽), Grade Up (▲), Index Of (ι), Indexed Assignment, Indexing, Pick (⇒) and Transpose (⊙).

Examples

```

      □IO←1
      ι5
1 2 3 4 5

      □IO←0
      ι5
0 1 2 3 4

      +/[0]2 3ρι6
3 5 7

      'ABC', [⌊.5] '=' = '
ABC
===

```

Key Label

R←□KL Y

Classic Edition only.

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

```

Line Count

R←□LC

This is a simple vector of line numbers drawn from the state indicator (See *Programmer's Guide: 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
0
ρ⊞LC

```

Load Workspace

⊞LOAD Y

Y must be a simple character scalar or vector containing the identification of a saved workspace.

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.

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.

Under UNIX, the interpreter attempts to open the file whose name matches the contents of **Y**. Under Windows, unless **Y** contains at least one ".", the interpreter will append the file extension ".DWS" to the name.

Lock Definition

{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** may be 1, 2 or 3 (the default) with the following meaning:

1. The object may not be displayed and you may not obtain its character form using **□CR**, **□VR** or **□NR**.
2. Execution cannot be suspended with the locked function or operator in the state indicator. On suspension of execution the state indicator is cut back to the statement containing the call to the locked function or operator.
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'
```

A **DOMAIN ERROR** is reported if **Y** is ill-formed.

Examples

```

□FX 'r←foo' 'r←10'
□NR 'foo'
r←foo r←10
ρ□NR 'foo'
2
□LOCK 'foo'
ρ□NR 'foo'
0
```

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 `' '`.

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
```

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 supplied utility function **ΔMPUT**. 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).
3. start byte offset (integer scalar/vector). Must be a multiple of 4 (default 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 [Data Representation \(Monadic\) on page 258](#).

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.

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 2p14
```

Map bytes 100-180 in raw file as a 5×2 read-only matrix of doubles:

```
dat←645 5 2 ⍵MAP'c:\myfile' 'R' 100
```

Put simple 4-byte integer array on disk ready for mapping:

```
(→83 323 ⍵DR 2 3 4p124)ΔMPUT'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:

```
42      +/163 -1 ⌈MAP'c:\shorts.dat'
```

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:

```
-86      83 ⌈MAP 'myfile'      A map FIRST BYTE of file.
```

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`.

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 0 changes the interpretation of certain symbols and language constructs.

<code>⌈ML←0</code>		Native Dyalog (Default)
<code>⌈ML←1</code>	<code>Z←∈R</code>	Monadic ' <code>∈</code> ' is interpreted as 'enlist' rather than 'type'.
<code>⌈ML←2</code>	<code>Z←↑R</code>	Monadic ' <code>↑</code> ' is interpreted as 'first' rather than 'mix'.
	<code>Z←↗R</code>	Monadic ' <code>↗</code> ' is interpreted as 'mix' rather than 'first'.
	<code>Z←≡R</code>	Monadic ' <code>≡</code> ' returns a positive rather than a negative value, if its argument has non-uniform depth.
<code>⌈ML←3</code>	<code>R←X<[K]Y</code>	Dyadic ' <code><</code> ' follows the APL2 (rather than the original Dyalog APL) convention.
	<code>⌈TC</code>	The order of the elements of <code>⌈TC</code> is the same as in APL2.

Subsequent versions of Dyalog APL may provide further migration levels.

Examples

$$X \leftarrow 2 \begin{pmatrix} 3 & 4 \end{pmatrix}$$

$$\begin{matrix} \square ML \leftarrow 0 \\ \in X \end{matrix}$$

$$\begin{matrix} 0 & 0 & 0 \end{matrix}$$

$$\uparrow X$$

$$\begin{matrix} 2 & 0 \end{matrix}$$

$$\begin{matrix} 3 & 4 \end{matrix}$$

$$\supset X$$

$$2$$

$$\equiv X$$

$$-2$$

$$\begin{matrix} \square ML \leftarrow 1 \\ \in X \end{matrix}$$

$$\begin{matrix} 2 & 3 & 4 \end{matrix}$$

$$\uparrow X$$

$$\begin{matrix} 2 & 0 \end{matrix}$$

$$\begin{matrix} 3 & 4 \end{matrix}$$

$$\supset X$$

$$2$$

$$\equiv X$$

$$-2$$

$$\begin{matrix} \square ML \leftarrow 2 \\ \in X \end{matrix}$$

$$\begin{matrix} 2 & 3 & 4 \end{matrix}$$

$$\uparrow X$$

$$2$$

$$\supset X$$

$$\begin{matrix} 2 & 0 \end{matrix}$$

$$\begin{matrix} 3 & 4 \end{matrix}$$

$$\equiv X$$

$$2$$

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. **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.

The effect of **□MONITOR** is to accumulate timing statistics for the lines for which the monitor has been set. See [Query Monitor on page 316](#) for details.

Examples

```
      +(0,110) □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.

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. 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 25p100
[2] B←⊖A
[3] C←⊖B
[4] R1←[0.5+A+.×B
[5] R2←A=C
▽

(0,15) MONITOR 'FOO' A Set monitor

FOO A Run function

MONITOR 'FOO' A 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
```


Name Association

 $\{R\} \leftarrow \{X\} \square NA \ Y$

$\square NA$ provides access from APL to compiled functions within a **Dynamic Link Library (DLL)**. A DLL is a collection of functions typically written in C (or C++) each of which may take arguments and return a result.

Instructional examples using $\square NA$ can be found in supplied workspace: `QUADNA.DWS`.

The DLL may be part of the standard operating system software, 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 $\square NA$ is successful, a function (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`. 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
```

Type Declaration

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  '
```

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.

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.

Pathname:

APL uses the `LoadLibrary()` system function under Windows and `dlopen()` under UNIX and Linux to load the DLL. If a full or relative pathname is omitted, these functions search standard operating system directories in a particular order. 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

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 ...'
```

Example

```
□NA'... mydll.exe.P32|foo ...'A 32 bit Pascal
```

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 ...'
```

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.

Data Type Coding Scheme

The type coding scheme introduced above is of the form:

[direction] [special] type [width] [array]

The options are summarised in the following table and their functions detailed below.

Description	Symbol	Meaning
Direction	<	Pointer to array <i>input</i> to DLL function.
	>	Pointer to array <i>output</i> from DLL function
	=	Pointer to input/output array.
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 ²
	A	APL array
	Z	APL array with header (as passed to a TCP/IP socket)

¹Classic Edition: - translated to/from ANSI

²equivalent to U4 on 32-bit versions and U8 on 64-bit versions

Description	Symbol	Meaning
Width	1	1-byte
	2	2-byte
	4	4-byte
	8	8-byte
	16	16-byte (128-bit)
Array	[n]	Array of length <i>n</i> elements
	[]	Array, length determined at call-time
Structure	{...}	Structure.

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).

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:

	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 <code>⎕AV</code> . This means that <code>⎕AV positions</code> map onto corresponding ANSI <i>positions</i> . For example, with <code>⎕IO=0</code> : <code>⎕AV[35] = 's'</code> , maps to <code>ANSI[35] = 's'</code>
	Type	Description
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 <i>translated</i> using standard Dyalog <code>⎕AV</code> to ANSI translation. This means that <code>⎕AV characters</code> map onto corresponding ANSI <i>characters</i> . For example, with <code>⎕IO=0</code> : <code>⎕AV[35] = 's'</code> , maps to <code>ANSI[115] = 's'</code>
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)
J	Complex	
P	uintptr-t	This is equivalent to U4 on 32-bit versions and U8 on 64-bit versions
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)
F	4, 8	8
D	16	16
J	16	16
P	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 or Linux.

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.

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 valu, 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.

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**. Notice that the corresponding associated APL function is niladic or monadic (never dyadic), and that it *always* returns a vector result - a null one if there is no output from the function. See *Result Vector* section below. Examples of the various combinations are:

DLL function Non-result-returning:

```
NA 'mydll|fn1'          A Niladic
NA 'mydll|fn2 <0T'      A Monadic - 1-element arg
NA 'mydll|fn3 =0T <0T' A Monadic - 2-element arg
```

DLL function Result-returning:

```
NA 'I4 mydll|fn4'          A Niladic
NA 'I4 mydll|fn5 F8'       A Monadic - 1-element arg
NA 'I4 mydll|fn6 >I4[] <0T' A 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 example functions defined above:

```
fn1          A Niladic Function.
fn2, c'Single String' A 1-element arg
fn3 'This' 'That'    A 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.

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 c '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'  A 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 (syserr 999 on Windows or SIGSEGV on UNIX).

= 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'
```

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.

ANA Declaration	Result	Output Arguments	Result Length
<code>mydll fn1</code>	0		0
<code>mydll fn2 <0T</code>	0	0	0
<code>mydll fn3 =0T <0T</code>	0	1 0	1
<code>I4 mydll fn4</code>	1		1
<code>I4 mydll fn5 F8</code>	1	0	1
<code>I4 mydll fn6 >I4[] <0T</code>	1	1 0	2

As a convenience, if the result would otherwise be a 1-item vector, it is disclosed. Using the third example above:

```
5      pfn3 '.....' 'abc'
```

`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.

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, i.e. `MessageBoxA()` for the Classic Edition, but `MessageBoxW()` for the Unicode Edition.

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 (typedefs)

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 [CNA](#) equivalents might prove useful.

Windows typedef	CNA 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

Windows typedef	ANA equivalent
COLORREF	{U1[4]}
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 =OT[] (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’ (<OT[]) or ‘output only’ (>OT[]). See *Direction* above.
2. LPCSTR is a pointer to a *constant* null-terminated string and therefore coding <OT[] 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.

Dyalog32.dll or Dyalog64.dll

Included with Dyalog APL are utility DLLs called dyalog32.dll and dyalog64.dll. These DLLs contain 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 'dyalog32|MEMCPY I4 <{P F4 T[20]} U4'
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`, null characters are 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 pointers to (max 20-char) null-terminated strings.

```
typedef struct { // null-terminated strings:
    char *first; // first name (max 19 chars + 1 null).
    char *last;  // last name. (max 19 chars + 1 null).
} name;
```

To copy the names *from* the structure:

```
'get' NA'dialog32|STRNCPY >OT[] P U4'
get 20 addr 20
Charlie
get 20 (addr+4) 20
Brown
```

Note that on a 64-bit version, **FR** will need to be 1287 for the addition to be reliable.

To copy data *from* the workspace *into* an already allocated (**new**) structure:

```
'put' NA'dialog32|STRNCPY I4 <OT[] 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.

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' 0NA'P dyalog32|STRLEN P'
'cpy' 0NA'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).

Example 1

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
```

Example 2

The 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
```

Example 3

The 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.

Example 4

The 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* I4 <OT'
```

To obtain the handle of the window entitled "CLEAR WS - Dyalog APL/W":

```
⌊←HNDL←FW 0 'CLEAR WS - Dyalog APL/W'
59245156
```

Example 5

The 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 ">OT" indicating that it is a pointer to a character output array.


```
⌊NA 'I user32|GetWindowText* P >OT 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
```



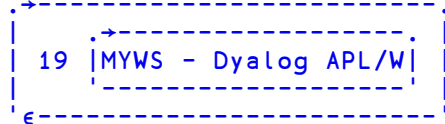
There are three points to note. Firstly, the number 255 is supplied as the second argument. This instructs APL to allocate a buffer large enough for a 255-element character vector into which the DLL routine will write. Secondly, the result of the APL function is a nested vector of 2 elements. The first element is the result of the DLL function. The second element is the output character array.

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.

e.g.

```
ⓘNA 'I User32|GetWindowText* P =0T I'
]display GetWindowText HNDL (255p' ') 255
```



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.

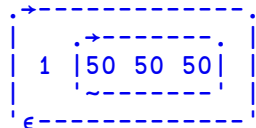
Example 6

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, int FAR*);
```

The following statements provide access to this routine through an APL function of the same name:

```
ⓘNA 'U4 gdi32|GetCharWidth* P U U >I[]'
'P'⊞WC'Printer'
]display GetCharWidth ('P' ⊞WG 'Handle') 65 67 3
```



Note: `'P'WG'Handle'` returns a handle. This is represented as a number. The number will be in the range $(0 - 2^{*32}]$ on a 32-bit version and $(0 - 2^{*64}]$ on a 64-bit version. These can be passed to a P type parameter. Older versions used a 32-bit signed integer.

Example 7

The following example from the supplied workspace: `QUADNA.DWS` 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 U4'
[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 >U
4'
[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 xsize↔Size file 0                      A Size of info
[14] :AndIf xhndl←Alloc 0 size                  A Alloc memory
[15]   :If xaddr←Lock hndl                      A Lock memory
[16]     :If xInfo file 0 size addr              A Version info
[17]       ok buff size←Value addr'\ ' 0 0 A Version valu
e
[18]       :If ok
[19]         buff←Copy(size÷4)buff size        A Copy info
[20]         version←(2/2*16)τ>2↓buff          A Split versio
n
[21]       :EndIf
[22]     :EndIf
[23]     ok←Unlock hndl                          A Unlock memor
y
[24]   :EndIf
[25]   ok←Free hndl                             A 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 <OT U4 <OT U4 U4 P >P >U4'
□NA' I4 advapi32 |RegEnumValue*       P U4 >OT =U4 =U4 >U4 >OT =U
4'
□NA' I4 advapi32 |RegOpenKey*          P <OT >P'
□NA' I4 advapi32 |RegOpenKeyEx*       P <OT U4 U4 >P'
□NA' I4 advapi32 |RegQueryValueEx*    P <OT =U4 >U4 >OT =U4'
□NA' I4 advapi32 |RegSetValueEx*      P <OT =U4 U4 <OT U4'
□NA' P dIALOG32 |STRNCOPY             P P P'
□NA' P dIALOG32 |STRNCPYA             P P P'
□NA' P dIALOG32 |STRNCPYW             P P P'
□NA' P dIALOG32 |MEMCPY               P P P'
□NA' I4 gdi32 |AddFontResource*        <OT'
□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*    <OT'
□NA' U4 gdi32 |SetPixel               P I4 I4 U4'
□NA' glu32 |gluPerspective             F8 F8 F8 F8'
□NA' I4 kernel32 |CopyFile*            <OT <OT I4'
□NA' P kernel32 |GetEnvironmentStrings'
□NA' U4 kernel32 |GetLastError'
□NA' U4 kernel32 |GetTempPath*         U4 >OT'
□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*             <OT <OT'
□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 <OT <OT 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 <OT U4 P'
□NA' I4 winmm |sndPlaySound             <OT U4'

```

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 file 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 file index result returned is the position within the file of the end of the record, which is also the start of the following one.

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

```

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**.

Name Classification

$R \leftarrow \square NC \ Y$

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	unused name
1	Label
2	Variable
3	Function
4	Operator
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, D-fns have name class 3.2, and so forth. The complete set of name classification is as follows:

	Array (2)	Functions (3)	Operators (4)	Namespaces (9)
n.1	Variable	Traditional	Traditional	Created by $\square NS,)NS$ or $:Namespace$
n.2	Field	dfns	dops	Instance
n.3	Property	Derived Primitive	Derived 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 6p'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
  :Access Public
    ▽ r←get
      r←pvtFld
    ▽
  :EndProperty

  ▽ r←NameClass x
    :Access Public
    r←NC x
  ▽

...
:EndClass A 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

```
¬2.3 0 [NC 'ncinst.pubProp' 'ncinst.pvtProp'
```

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:

```
¬2.3 0 ncinst.[NC 'pubProp' 'pvtProp'
```

Note that the names of Properties are reported as being *unused* if the argument to [NC is simple.

```
0 0 ncinst.[NC 2 6p'pubProppvtProp'
```

External Properties (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.

```
0 0 f.[NC 2 7p'CaptionSize '
```

Defined Functions (Name-Class 3.1)

Traditional APL defined functions have name-class 3.1.

```

▽ R←AVG X
[1]   R←(+/X)÷ρX
▽
    AVG ι100
50.5

    □NC'AVG'
3
    □NC='AVG'
3.1

    'MYSPACE'□NS 'AVG'
    MYSPACE.AVG ι100
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, i.e. 3.3.

```

    VMEAN←AVG°,
    □NC'AVG' 'VMEAN'
3.1 3.3

```

Dfns (Name-Class 3.2)

Dfns have name-class 3.2

```

    Avg←{(+/ω)÷ρω}

    □NC'Avg'
3
    □NC='Avg'
3.2

```

Derived Functions (Name-Class 3.3)

Derived Functions and functions created by naming a Primitive function have name-class 3.3.

```
PLUS←+
SUM←+ /
CUM←PLUS\
⊞NC'PLUS' 'SUM' 'CUM'
3.3 3.3 3.3
⊞NC 3 4p'PLUSSUM CUM '
3 3 3
```

Note that a function that is simply cloned from a defined function by assignment retains its name-class. Whereas, the name of a function that amalgamates a defined function with any other functions has the name-class of a Derived Function, i.e. 3.3.

```
▽ R←AVG X
[1] R←(+ /X)÷pX
▽

MEAN←AVG
VMEAN←AVG°,
⊞NC'AVG' 'MEAN' 'VMEAN'
3.1 3.1 3.3
```

External Functions (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      m
tol      ss      vtol
    NC 'hex' 'ss'
3.6 3.6

```

Operators (Name-Class 4.1)

Traditional Defined Operators have name-class 4.1.

```

    ∇FILTER∇
    ∇ VEC←(P FILTER)VEC  A Select from VEC those elts ..
[1]  VEC←(P**VEC)/VEC    A for which BOOL fn P is true.
    ∇
    NC 'FILTER'
4
    NC< 'FILTER'
4.1

```

Dops (Name-Class 4.2)

Dops have name-class 4.2.

```

    pred←{IO ML←1 3  A Partitioned reduction.
    =>αα/'(α/ιpα)<ω
    }
    2 3 3 2 +pred ι10
3 12 21 19
    NC 'pred'
4
    NC< 'pred'
4.2

```

External Events (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'
n')
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 ...

```

Namespaces (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

```

Instances (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 3p'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.Applicatio
n')
'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

Classes (Name-Class 9.4)

Classes created using the editor or `FIX` have name-class 9.4.

```
)ED oMyClass
```

```
:Class MyClass
  ▽ r←NameClass x
  :Access Public Shared
  r←NC x
  ▽
:EndClass a 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

Interfaces (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 A IGolfClub

    □NC 'IGolfClub'
9
    □NC ←'IGolfClub'
9.5
```

External Classes (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 Interfaces (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.

Native File Create

{R}←X □NCREATE Y

This function creates a new file. Under Windows the file is opened in compatibility mode. 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.

The shy result of **□NCREATE** 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,□NNUMS      a With next available number,
file □NCREATE tie      a ... create file.
```

to:

```
tie←file □NCREATE 0 a Create with first available n
o.
```

If the specified file already exists, **□NCREATE** fails with the error (22):

```
FILE NAME ERROR: Unable to create file
```

Native File Erase

{R}←X □NERASE Y

This function erases (deletes) 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 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
```

New Instance

R ← NEW Y

NEW creates a new instance of the Class 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*.

The result **R** is a reference to a new instance of Class or Type **Y**.

For further information, see *Interface Guide*.

Class Example

```
:Class Animal
  ▽ Name nm
    :Access Public
    :Implements Constructor
    □DF nm
  ▽
:EndClass A Animal

Donkey ← NEW Animal 'Eeyore'
Donkey
Eeyore
```

If **NEW** is called with just a Class reference (i.e. 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 A 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.

```

    F←NEW c'Form'
    F←NEW 'Form' (('Caption' 'Hello') ('Posn' (10 10)))
    NEW 'Form' (('Caption' 'Hello') ('Posn' (10 10)))
#. [Form]

```

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 [Name Classification on page 346](#).

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 DomesticPar
rot Eeyore FishBehaviour Nickel Parrot Penguin Polly Robin
      □NL -9.3 A Instances
Eeyore Nickel Polly Robin
      □NL -9.4 A Classes
Animal Bird Coin Cylinder DomesticParrot Parrot Pe
nguin
      □NL -9.5 A 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 A Properties
Accelerator AcceptFiles Active AlphaBlend AutoConf
Border BCol Caption ...

      F.□NL -3 A Methods
Animate ChooseFont Detach GetFocus GetTextSize Show
SIP Wait

      F.□NL -8 A Events
Close Create DragDrop Configure ContextMenu DropFile
s 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 Last
IndexOf Reverse Sort

⋄NL -9
Array DateTime
```

Another use for `□NL` is to examine .NET *enumerations*. For example:

```

    □USING←'System.Windows.Forms,system.windows.forms.d
ll'

    FormBorderStyle.□NL -2
Fixed3D   FixedDialog FixedSingle FixedToolWindow None
Sizable  SizableToolWindow

    FormBorderStyle.FixedDialog.value__
3

    FormBorderStyle.({ω,[1.5]±''ω,''c'.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.Applicatio
n')

    xl.□NL -2 # Properties
_Default ActiveCell ActiveChart ActiveDialog ActiveM
enuBar ActivePrinter ActiveSheet ActiveWindow ...

    xl.□NL -3 # Methods
_Evaluate _FindFile _Run2 _Wait _WSFunction Activat
eMicrosoftApp AddChartAutoFormat AddCustomList Browse
Calculate ...

    □NL -9
xl

```

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      A write-lock whole file
0 □NLOCK ^1      A unlock whole file.
1 □NLOCK ^1      A read (share) lock whole file.
2 □NLOCK''□NNUMS A write-lock all files.
0 □NLOCK''□NNUMS A unlock all files.

1 □NLOCK ^1 12 1  A read-lock byte 12.
1 □NLOCK ^1 0 10  A read-lock first 10 bytes.
2 □NLOCK ^1 20    A write-lock from byte 20 onwards.
2 □NLOCK ^1 10 2  A write-lock 2 bytes from byte 10
0 □NLOCK ^1 12 1  A 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 A write-lock from EOF.
-1 1000
... [NAPPEND -1           A append bytes to file
... [NAPPEND -1           A append bytes to file

0 [NLOCK region           A 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           A 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.

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.

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.

Enqueue Event

$$\{R\} \leftarrow \{X\} \square NQ \ Y$$

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 $\square DQ$. $\square NQ$ provides an "artificial" means to generate an event and is analogous to $\square SIGNAL$.

If the left argument X is omitted or is 0, $\square NQ$ adds the event specified by Y to the bottom of the event queue. The event will subsequently be processed by $\square DQ$ when it reaches the top of the queue.

If X is 1, the event is actioned **immediately** by $\square NQ$ itself and is processed in exactly the same way as it would be processed by $\square DQ$. For example, if the event has a callback function attached, $\square NQ$ will invoke it directly. See [Dequeue Events on page 255](#) for further details.

Note that it is not possible for one thread to use 1 $\square NQ$ to send an event to another thread.

If X is 2 and the name supplied is the name of an event, $\square NQ$ 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, $\square NQ$ invokes the method. Its (shy) result is the result produced by the method.

If X is 3, $\square NQ$ invokes a method in an OLE Control. The (shy) result of $\square NQ$ is the result produced by the method.

If X is 4, $\square NQ$ signals an event from an ActiveXControl object to its host application. The (shy) result of $\square NQ$ 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 `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

```
A Send a keystroke ("A") to an Edit Field
□NQ TEST.ED 'KeyPress' 'A'
```

```
A Iconify all top-level Forms
{□NQ ω 'StateChange' 1}''Form'□WN'.'
```

```
A Set the focus to a particular field
□NQ TEST.ED3 40
```

```
A Throw a new page on a printer
1 □NQ PR1 'NewPage'
```

```
A Terminate □DQ under program control
```

```
'TEST'□WC 'Form' ... ('Event' 1001 1)
```

```
□DQ 'TEST'
```

```
□NQ TEST 1001 A From a callback
```

```
A Call GetItemState method for a TreeView F.TV
+2 □NQF.TV 'GetItemState' 6
```

96

```
+2 □NQ'.' 'GetEnvironment' 'Dyalog'
c:\Z\2\dyalog82
```

Nested Representation

$R \leftarrow \square NR \ Y$

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

```
[1]  ▽R←MEAN X      A Average
      R←(+/X)÷ρX
      ▽

      +F←□NR 'MEAN'
      R←MEAN X      AAverage      R←(+/X)÷ρX

      ρF
2    ]display F
```

The definition of $\square NR$ has been extended to names assigned to functions by specification (\leftarrow), and to local names of functions used as operands to defined operators. In these cases, the result of $\square NR$ is identical to that of $\square CR$ except that the representation of defined functions and operators is as described above.

Example

```

      AVG←MEAN∘,
      +F←⊞NR'AVG'
      R←MEAN X      A Average      R←(+/X)÷ρX      ∘,
      ρF
3      ]display F

```

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,
- [4] start byte, counting from 0.

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.

Table 14: Unicode Edition : Conversion Codes

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	$8 \times \text{count}$
80	count	8 bits character	count
82 ¹	count	8 bits character	count
83	count	8 bits integer	count
160	$2 \times \text{count}$	16-bits character	count
163	$2 \times \text{count}$	16 bits integer	count
320	$4 \times \text{count}$	32-bits character	count
323	$4 \times \text{count}$	32 bits integer	count
645	$8 \times \text{count}$	64bits floating	count

Table 15: Classic Edition : Conversion Codes

Value	Number of bytes read	Result Type	Result shape
11	count	1 bit Boolean	$8 \times \text{count}$
82	count	8 bits character	count
83	count	8 bits integer	count
163	$2 \times \text{count}$	16 bits integer	count
323	$4 \times \text{count}$	32 bits integer	count
645	$8 \times \text{count}$	64bits floating	count

Note that types **80**, **160** and **320** and **83** and **163** are exclusive to Dyalog APL.

If **Y[4]** is omitted, data is read starting from the current position in the file (initially, 0).

Example

```
DATA←⊞NREAD ¯1 160 (0.5×⊞NSIZE ¯1) 0 ⍺ Unicode
DATA←⊞NREAD ¯1 82 (⊞NSIZE ¯1) 0 ⍺ Classic
```

¹Conversion 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.

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.

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,
- [3] conversion code (optional).

See [Native File Read on page 369](#) for a list of valid conversion codes.

The shy result is the position within the file of the end of the record, or, equivalently, the start of the following one. Used, for example, in:

```
⌞ Replace sequentially from indx.
{α □NREPLACE tie ω}/vec,indx
```

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 ʀ See Example on page 345
␣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`.

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.

Create Namespace

$$\{R\} \leftarrow \{X\} \square NS \ Y$$

If specified, X must be a simple character scalar or vector identifying the name of a namespace.

Y is either a character array which represents a list of names of objects to be copied into the namespace, or is an array produced by the $\square OR$ of a namespace.

In the first case, 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 the namespace X . The identifiers in X and Y may be simple names or compound names separated by $'.'$ and including the names of the special namespaces $'\#'$, $'\#\#'$ and $'\square SE'$.

The namespace X is created if it doesn't already exist. If the name is already in use for an object other than a namespace, APL issues a **DOMAIN ERROR**.

If X is omitted, an unnamed namespace is created.

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 $\square SE.$) of the namespace X . If X is omitted, the result R is a namespace reference, or *ref*, to an unnamed namespace.

Examples

```

# .X      + 'X' NS ' '      A Create namespace X.
# .X      + 'X' NS 'VEC' 'UTIL.DISP' A Copy VEC and DISP to X.
# .X      )CS X            A Change to namespace X.
# .X      + 'Y' NS '#.MAT' '##.VEC' A Create #.X.Y &copy in
# .X.Y    + '#.UTIL' NS 'Y.MAT'     A Copy MAT from Y to UTIL #
# .UTIL.
# .UTIL   + '# ' NS 'Y'            A Copy namespace Y to root.
#         + ' ' NS '#.MAT'         A Copy MAT to currentspace.
# .X      + ' ' NS ' '            A Display current space.
# .X      + 'Z' NS OR 'Y'         A Create nspace from OR.
# .X.Z
          NONAME+ NS ' '          A Create unnamed nspace
          NONAME
# .[Namespace]
          DATA+ NS ``3p< ' '      A Create 3-element vector of
                                   A distinct unnamed nspaces
          DATA
# .[Namespace] # .[Namespace] # .[Namespace]

```

The second case is where *Y* is the `OR` of a namespace.

If *Y* is the `OR` of a GUI object, `#.Z` must be a valid parent for the GUI object represented by *Y*, or the operation will fail with a **DOMAIN ERROR**.

Otherwise, the result of the operation depends upon the existence of *Z*.

- If *Z* does not currently exist (name class is 0), *Z* is created as a complete copy (clone) of the original namespace represented by *Y*. If *Y* is the `OR` of a GUI object or of a namespace containing GUI objects, the corresponding GUI components of *Y* will be instantiated in *Z*.
- If *Z* is the name of an existing namespace (name class 9), the contents of *Y*, including any GUI components, are merged into *Z*. Any items in *Z* with corresponding names in *Y* (names with the same path in both *Y* and *Z*) will be replaced by the names in *Y*, unless they have a conflicting name class in which case the existing items in *Z* will remain unchanged. However, all GUI spaces in *Z* will be stripped of their GUI components prior to the merge operation.

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 ($\rho\Box\text{NSI} \leftrightarrow \rho\Box\text{RSI} \leftrightarrow \rho\Box\text{SI}$).

$\Box\text{RSI}$ and $\Box\text{NSI}$ are identical except that $\Box\text{RSI}$ returns refs to the spaces whereas $\Box\text{NSI}$ returns their names. Put another way: $\Box\text{NSI} \leftrightarrow \text{" "}\Box\text{RSI}$.

Note that $\Box\text{NSI}$ 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

```

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.

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.

Needed from existing users		Granted to subsequent users	
0	read access	0	compatibility mode
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 first code (**16|mode**) is passed to the `open(2)` call as the access parameter. See include file `fcntl.h` for details.

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,⌈NNUMS      A With next available number,
file ⌈NTIE tie         A ... tie file.
```

to:

```
tie←file ⌈NTIE 0      A Tie with first available no.
```

Example

```
ntie←{                  A tie file and return tie no.
  α←2+64                A default all access.
  ω ⌈ntie 0 α           A return new tie no.
}
```

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. An example might be 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: =, ≠, ≡, ≠, ,, ρ, ⊃, ⊆

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
```

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 [File Untie on page 305](#) for more explanation.

The shy result of **□NUNTIE** is a vector of tie numbers of the files **actually untied**.

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, maps **□AV** characters to their corresponding ANSI positions and is derived from the mapping defined in the current output translation table (normally WIN.DOT)

Between them, ANSI and RAW translations should cater for most uses.

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.

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.

Variant

`{R}←{X}(f ⎕OPT B)Y`

`⎕OPT` is synonymous with the Variant Operator symbol `⌈` and is the only form available in the Classic Edition.

See [Variant on page 155](#).

Object Representation

$$R \leftarrow \text{OR } Y$$

`OR` converts a function, 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. Classes and Instances are however outside the domain of `OR`.

Taking the `OR` of a 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 `OR`s 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 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, $R \leftarrow \text{OR } Y$ is identical to $R \leftarrow Y$.

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 Depth (\equiv), Disclose (\supset), Enclose (\Leftarrow), Rotate (Φ), Transpose (Φ), Indexing ($[]$), Format (Φ), Identity ($+$), Shape (ρ), Type (ϵ) and Unique (\cup), of the dyadic functions Assignment (\leftarrow), Without (\sim), Index Of (ι), Intersection (\cap), Match (\equiv), Membership (ϵ), Not Match (\neq) and Union (\cup), and of the monadic system functions Canonical Representation ($\square CR$), Cross-Reference ($\square REFS$), Fix ($\square FX$), Format ($\square FMT$), Nested Representation ($\square NR$) and Vector Representation ($\square VR$).

Nested arrays which include the object representations of functions and operators are in the domain of many mixed functions which do not use the values of items of the arrays.

Note that a $\square OR$ 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 $\square OR$ may not be used as arguments to any primitive functions. The only operations permitted for such objects (or arrays containing such objects) are $\square EX$, $\square FAPPEND$, $\square FREPLACE$, $\square NS$, and $\square WC$.

Example

```

F ← □OR □FX 'R ← FOO' 'R ← 10'

ρF

ρρF
0
≡F
1
F ≡ ∈ F
1

```

The display of the $\square OR$ form of a function or operator is a listing of the function or operator. If the $\square OR$ form of a function or operator has been enclosed, then the result will display as the operator name preceded by the symbol ∇ . It is permitted to apply $\square OR$ 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
      ▽

      cF
      ▽FOO

      □LOCK 'FOO'

      □OR 'FOO'
      ▽FOO

      A←ι5

      A[3]←cF

      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.

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 [Export Object on page 265](#)) can then be used directly from anywhere in the workspace without giving their full path names.

Example

To make the **DISPLAY** function available directly from within any namespace.

```
A Create and reference utility namespace.
▯PATH←'▯se.util'▯ns'
A Copy DISPLAY function from UTIL into it.
'DISPLAY'▯se.util.▯cy'UTIL'
A (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 [Export Object on page 265](#)), 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.

Program Function Key

$$R \leftarrow \{X\} \square \text{PFKEY } Y$$

$\square \text{PFKEY}$ is a system function that sets or queries the programmable function keys.

$\square \text{PFKEY}$ 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.

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 Input Translate Table codes.

Programmable function keys are recognised in any of the three types of window (SESSION, EDIT and TRACE) provided by the Dyalog APL development environment. $\square \text{SR}$ operates with the 'raw' function keys and ignores programmed settings.

Note that key definitions can reference other function keys.

The size of the buffer associated with $\square \text{PFKEY}$ is specified by the *pfkey_size* parameter.

Examples

```
(')FNS',⊖'ER') □PFKEY 1
)FNS ER
```

```
]display □PFKEY 1
```



```
(')VARS',⊖'ER') □PFKEY 2
)VARS ER
'F1' 'F2' □PFKEY 3  ␣ Does )FNS and )VARS
F1 F2
```

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 17. The value in a clear workspace is 10. Note that in all Versions of Dyalog APL prior to Version 11.0, the maximum value for **⎕PP** was 16.

⎕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 its maximum value of 17, floating-point numbers may be converted between binary and character representation without loss of precision. In particular, if **⎕PP** is 17 and **⎕CT** is 0 (to ensure exact comparison), for any floating-point number **N** the expression **N=⌞⌞N** is true. Note however that *denormal* numbers are an exception to this rule.

Numbers, very close to zero, in the range **2.2250738585072009E⁻³⁰⁸** to **4.9406564584124654E⁻³²⁴** are called *denormal* numbers. Such numbers can occur as the result of calculations and are displayed correctly.

Numbers below the lower end of this range (**4.94E⁻³²⁴**) are indistinguishable from zero in IEEE double floating point format.

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.

Y specifies the action to perform and any options for that action, if applicable. **Y** is case-insensitive.

Use	Description
<code>state←⊞PROFILE 'start' {timer}</code>	Turn profiling on using the specified timer or resume if profiling was stopped
<code>state←⊞PROFILE 'stop'</code>	Suspend the collection of profiling data
<code>state←⊞PROFILE 'clear'</code>	Turn profiling off, if active, and discard any collected profiling data
<code>state←⊞PROFILE 'calibrate'</code>	Calibrate the profiling timer
<code>state←⊞PROFILE 'state'</code>	Query profiling state
<code>data←⊞PROFILE 'data'</code>	Retrieve profiling data in flat form
<code>data←⊞PROFILE 'tree'</code>	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' {timer}`

Turn profiling on; `timer` is an optional case-independent character vector containing `'CPU'` or `'elapsed'` or `'none'`. If omitted, it defaults to `'CPU'`. If `timer` is `'none'`, `□PROFILE` can be used to record which lines of code are executed without incurring the timing 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 0.000103749999 0.000103749999
```

state←`□PROFILE 'stop'`

Suspends the collection of profiling data.

```
□PROFILE 'stop'
inactive CPU 0.000103749999 0.000103749999
```

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 ws
)clear
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')[;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 39347.64945 503 4080803
#.foo 1 1 0.12488 0.12488 1 1
#.foo 2 100 0.58851 39347.19390 200 4080500
#.foo 3 100 0.21340 0.21340 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 `PROFILER` and provides reporting and analysis tools that act upon the profiling data. For further information, see *Tuning Applications using the Profile User Command*.

Using `PROFILER` Directly

If you choose to use `PROFILER` directly, the following guidelines and information may be of use to you.

Note: Running your application with `PROFILER` turned on incurs a significant processing overhead and will slow your application down.

Decide which timer to use

`PROFILER` 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 $(4000 \times 4 \times \text{PROFILER 'state'})$ milliseconds.
- Turn profiling on with `PROFILER 'start' CPU`
- Run your application.
- Pause the profiler with `PROFILER 'stop'`
- Examine the profiling data from `PROFILER 'data'` or `PROFILER '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
```

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.

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
```

```
⌵←3p÷3
```

```
0.3333333333 0.3333333333 0.3333333333
0.3333333333
```

Cross References

R ← REF S 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 [Object Representation on page 380](#)). **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]  R REQUESTS AND EXECUTES AN OPTION
[2]  OPTS ← 'INPUT' 'REPORT' 'END'
[3]  IN:INP←ASK 'OPTION: '
[4]  →EXP~(CINP)∈OPTS
[5]  'INVALID OPTION. SELECT FROM',OPTS ♦ →IN
[6]  EX:→EX+OPTS└CINP
[7]  INPUT ♦ →IN
[8]  REPORT ♦ →IN
[9]  END:
    ▽

    REF S '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' ♦ REF S 'OPTIONS'
OPTIONS

```

If **Y** is the name of a primitive, external or derived function, **R** is an empty matrix with shape 0 0.

Replace

$$R \leftarrow \{X\} (A \ \square R \ B) \ Y$$

$\square R$ (Replace) and $\square S$ (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. $\square R$ and $\square S$ utilise the open-source regular-expression search engine PCRE, which is built into Dyalog APL and distributed according to the PCRE license which is published separately.

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, $\square R$ for replace and $\square S$ for search, are syntactically identical. With $\square R$, the input document is examined; text which matches the search pattern is amended and the remainder is left unchanged. With $\square S$, 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 {φω.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 {ω.((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 `IR` or `IS` it is possible to reset the file position to the start so that it may be read again using:

```
{ } INREAD t ienum 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 there is no result - instead 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.

□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 there is no result - instead 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. 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

A summary of the syntax of the search pattern is reproduced from the PCRE documentation. See [Appendix A - PCRE Syntax Summary on page 526](#).

A full description is provided in [Appendix B - PCRE Regular Expression Details on page 533](#).

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:

%	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; <i>n</i> may be any single digit value from 1 to 9
\(n)	acts as a placeholder for the text which matched the numbered subpattern; <i>n</i> 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

The above may be qualified to fold matching text to upper- or lower-case by using the **u** and **l** modifiers respectively. 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 than 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, origin zero.
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 index-zero 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.
Lengths	A vector of one or more lengths, corresponding to each value in Offset.
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 QR (value 1) or QS (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`.

The locations of the match within Block and subpatterns within Match are given as offsets rather than positions, i.e. the values are the number of characters preceding the data, and are not affected by the Index Origin.

There may be only one transformation function, regardless of the number of search patterns.

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 New Line
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  Dx F
( '.' ␣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' E 'Greedy' 1 ) 'ABC123 DEF456'
X
( '[A-Z].*[0-9]' R 'X' E '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 <i>start</i> of the match, thus permitting overlapping matches.
0	Searching continues from the character following the <i>end</i> 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' E 'OM' 0 ) 'A 1234 5678 B'
1234 5678
( '[0-9]+' S '\0' E '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.

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.

UTF8	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.
UTF16LE	The stream is processed as UTF16 little-endian data.
UTF16BE	The stream is processed as UTF16 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.
ANSI	The stream is processed as ANSI (Windows-1252) data.

For compatibility with the **OutEnc** option, the above UTF formats may be qualified with -BOM (e.g. UTF-BOM). For input streams, the qualified and unqualified options are equivalent.

OutEnc Option

When the output is written to a stream, the data may be encoded on one of the following forms:

Implied	If input came from a stream then the encoding format is the same as the input stream, otherwise UTF-8
UTF8	The data is written in UTF-8 format.
UTF16LE	The data is written in UTF-16 little-endian format.
UTF16BE	The data is written in UTF-16 big-endian format.
ASCII	The data is written in ASCII format.
ANSI	The data is written in ANSI (Windows-1252) format.

The above UTF formats may be qualified with -BOM (e.g. UTF8-BOM) to specify that a Byte Order Mark should be written at the start of the stream. For files, this is ignored if the file already contains any data.

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

```

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 license 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' B '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' B ('IC' 1)('ML' 2)) text
To b\VOWEL or not to be- that is the question:
Wheth\VOWELr 'tis nobler in the mind to suffer
The sl\VOWELngs 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<re
mainder>') 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 '%' B ('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 B ('IC' 1)('ML' 1)) text
0 3
```

Note the difference between the characters 'or' (which appear in 'fortune') and the word 'or'.

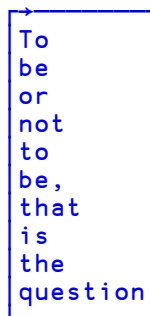
Place every non-space sequence of characters in brackets:

```
(['^\\s]+' R '(&)' ) 'To be or not to be, that is t
he question'
(To) (be) (or) (not) (to) (be,) (that) (is) (the) (questi
on)
```

Replace all sequences of one or more spaces by newline. Note that the effect of this is dependent on the input format:

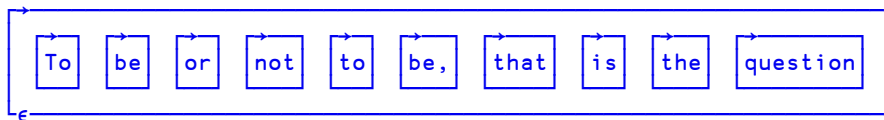
Character vector input results in a single character vector output with embedded newlines:

```
]display ('\\s+' R '\\r') 'To be or not to be, that
is the question'
```



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'
```



Change numerals to their expanded names, using a function:

```
∇r←f a
```

```
[1] r←' ',>(⊥a.Match)↓'zero' 'one' 'two' 'three' 'four'
'five' 'six' 'seven' 'eight' 'nine'
```

```
∇
```

```
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[w+1]+1a.Lengths[w+
1]]}⌵3
[5]       ret←y,'-',m,'-',d,', '
[6]   :Else
[7]       v←a.Block[a.Offsets[1]+1a.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
```

Random Link

⍱RL

⍱RL establishes a base or *seed* for generating random numbers using Roll and Deal, and returns the current state of such generation.

Three different random number generators are provided, which are referred to here as *RNG0*, *RNG1* and *RNG2*. These are selected using (16807⍵). See [Random Number Generator on page 195](#). **⍱RL** is relevant only to *RNG0* and *RNG1* for which repeatable pseudo-random series can be obtained by setting **⍱RL** to a particular value first.

Using *RNG0* or *RNG1*, you can set **⍱RL** to any integer in the range 0 to $2^{31}-1$. The value 0 has a special meaning (see below).

In a **clear ws**, **⍱RL** is initialised to the value defined by the **default_rl** parameter which itself defaults to 16807 if it is not defined.

Using *RNG0*, **⍱RL** returns 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** returns 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.

Internally, APL maintains the current state separately for *RNG0* and *RNG1*. When you switch from one Random Number Generator to the other, the appropriate state is loaded into **⍱RL**.

If `RL` is assigned the value 0, `RL` is initialised with a random seed generated by the operating system. This provides the means to initiate a non-repeatable series of pseudo-random numbers when using *RNG0* or *RNG1*.

RNG2 does not permit access to the *seed*, so in this case `RL` is not relevant and is not used by Roll and Deal. It will accept any value but will always return zilch.

Examples

```

16807I1 A Select RNG1
0
  RL←16807
  10?10
4 1 6 5 2 9 7 10 3 8
  5↑RL
10 0 16807 1819658750 -355441828
  X←?1000p1000
  5↑RL
100 -465541037 -1790786136 -205462449 996695303

  RL←16807
  10?10
4 1 6 5 2 9 7 10 3 8
  Y←?1000p1000
  X≡Y
1
  5↑RL
100 -465541037 -1790786136 -205462449 996695303

16807I0 A Select RNG0
1
  RL
16807
  ?9 9 9
2 7 5
  ?9
7
  RL
984943658

  RL←16807
  ?9 9 9
2 7 5
  ?9
7
  RL
984943658

```

```

16807I1 A Select RNG1
0
5↑RL
100 -465541037 -1790786136 -205462449 996695303

```

Space Indicator

R←R SI

R is a vector of refs to the spaces from which functions in the state indicator were called ($\rho R SI \leftrightarrow \rho NSI \leftrightarrow \rho SI$).

R SI and **NSI** are identical except that **R SI** returns refs to the spaces whereas **NSI** returns their names. Put another way: $NSI \leftrightarrow \text{⌈} R SI$.

Note that **R SI** returns refs to the spaces *from which* functions were called not those *in which* they are currently running.

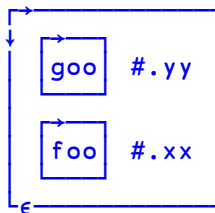
Example

```

)OBJECTS
xx      yy
      VR 'yy.foo'
      ▽ r←foo
[1]     r←SE.goo
      ▽
      VR 'SE.goo'
      ▽ r←goo
[1]     r←SI,[1.5]R SI
      ▽

)CS xx
#.xx    calling←#.yy.foo
        ]display calling

```



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.

Example

```
RTL←5 ♦ 'FUEL QUANTITY?' ♦ R←
FUEL QUANTITY?
TIMEOUT
RTL←5 ♦ 'FUEL QUANTITY?' ♦ R←
```

Search

R←{X}(A S B) Y

See [Replace on page 397](#).

Save Workspace

{R}←{X}SAVE Y

Y must be a simple character scalar or vector, identifying a workspace name. Note that the name must represent a valid file name for the current Operating System. **R** is a simple logical scalar. The active workspace is saved with the given name in **Y**. In the active workspace, the value 1 is returned. The result is suppressed if not used or assigned.

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.

A **DOMAIN ERROR** is reported if the name in **Y** is not a valid workspace name or file name, or the reference is to an unauthorised directory.

As is the case for `)SAVE` (see [Save Workspace on page 516](#)), 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
```

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.

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 Guide* for further details.

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 [Execute Windows Command on page 231](#) for a discussion of the behaviour of these system functions under Windows.

The system commands **)SH** and **)CMD** provide similar facilities. For further information, see [Execute \(UNIX\) Command on page 518](#) and [Windows Command Processor on page 502](#).

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. It could cause another process to be entered, such as **sed** or **vi**. If the command does not return a result, **R** is **␣'** 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.

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 &'
```

Start UNIX Auxiliary Processor

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. □SH is probably more natural for the UNIX user. This section describes the behaviour of □SH and □CMD under UNIX. See [Start Windows Auxiliary Processor on page 235](#) for a discussion of the behaviour of these system functions under Windows.

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 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 **pipelined** 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 `exec(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'
```

See *User Guide* for further information.

Examples

```
'xutils' □SH 'xutils' 'ss' 'dbr'
'/bin/sh' □SH 'sh' '-c' 'adb test'
```

State Indicator

$R \leftarrow \square SI$

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]*
␣

PLUS  ␣SI
      MATDIV  FOO

(ρ␣LC)=ρ␣SI
1

```

If execution stops in a callback function, $\square DQ$ 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 [Extended State Indicator](#) on page 495.

Shadow Name

␣SHADOW Y

Y must be a simple character scalar, vector or matrix identifying one or more APL names. For a 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.

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

VALUE FOO
      ERROR
      FOO
      ^

```

Signal Event

{X}□SIGNAL Y

Y must be a scalar or vector.

If **Y** is an empty vector nothing is signalled.

If **Y** is a vector of more than one element, all but the first element are ignored.

If the first element of **Y** is a simple integer in the range 1-999 it is taken to be an event number. **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 *Programmer's Guide: APL Error Messages*. 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 range 1-999 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**, and 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
    ▽

```

Further examples

Example 1

```

      'Hello' □ SIGNAL 200
Hello  'Hello' □ SIGNAL 200
      ^
      □ DMX
EM      Hello
Message
HelpURL

      □ DM
Hello      'Hello' □ SIGNAL 200      ^
      □ SIGNAL <= ('EN' 200)
ERROR 200
      □ SIGNAL <= ('EN' 200)
      ^

      □ DMX
EM      ERROR 200
Message
HelpURL

      □ 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
HelpURL

      τ □ 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)
      ^

```

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 **Y** identifies an object with an active referent, the workspace required in bytes by that object is returned in the corresponding element of **R**. Otherwise, 0 is returned in that element of **R**.

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←ι10

'EXT/ARRAY' ⊞XT'E' ♦ E←ι20

⊞SIZE 'A' 'FOO' 'E' 'UND'
28 76 120 0
```


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.

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 (Async terminals) 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`.

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. e.g.

```
⌈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)

Screen Read

$R \leftarrow \{X\} \square SR \ Y$

$\square SR$ is a system function that allows the user to edit or otherwise interact with the form defined by $\square SM$.

In versions of Dyalog APL that support asynchronous terminals, if the current screen is the SESSION screen, $\square SR$ immediately switches to the USER SCREEN and displays the form defined by $\square SM$.

In Dyalog APL/X, $\square SR$ causes the input cursor to be positioned in the USER window. During execution of $\square SR$, only the USER Window defined by $\square SM$ 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 $\square SM$ window, and allows $\square SM$ to be used in conjunctions with other GUI components. In these versions, $\square SR$ acts as a superset of $\square DQ$ (see [Dequeue Events on page 255](#)) but additionally controls the character-based user interface defined by $\square SM$.

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 $\square SR$ are:

```
EXIT_CONTEXT ←  $\square SR$  FIELDS
```

```
EXIT_CONTEXT ← (=EXIT_KEYS)  $\square SR$  FIELDS
```

```
EXIT_CONTEXT ← (EXIT_KEYS) (INITIAL_CONTEXT)  $\square SR$  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, i.e. 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 <Enter>, <Esc> and <Shift+Esc>.

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.

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 oMyClass

:Class MyClass
▽ r←foo arg
:Access public shared
r←1+arg
▽
:EndClass

        z←SRC MyClass
        z
6
        p..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 object_reference**.

State Indicator StackR←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.
Null 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
<code>⌵</code>	execute level
<code>□</code>	evaluated input
*	desk calculator level
<code>□DQ</code>	in callback function
other	primitive operator

Example

```

)SI
#.PLUS[2]*
.
#.MATDIV[4]
#.FOO[1]*
⌵

      □STACK
      *
▽PLUS
      .
▽MATDIV
      *
▽FOO
      ⌵
      *

ρ□STACK
8 2

(ρ□LC)=1↑ρ□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]    A Divide matrix A by matrix B
[2]    C←A⌈B
[3]    A Check accuracy
[4]    D←[0.5+A PLUS.TIMES B
      ▽
```

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. 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: (ρ1→R)=ρLC
2→R	Numeric vector giving the stack state of this name as it entered this level. Note: (ρ2→R)=ρLC 0=not on stack 1=suspended 2=pending (may also be suspended) 3=active (may also be pending or suspended)
3→R	Numeric vector giving the name classification of Y as it entered this level. Note: (ρ3→R)=+/1→R
4→R	Vector giving the contents of Y before it was shadowed at this level. Note: (ρ4→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

      R←STATE 'TRAP'

```

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 1      ←(0,110) □STOP 'FOO'
```

Existing stop controls in the function or operator named by **Y** are cancelled before new stop controls are set:

```
1      ←1 □STOP 'FOO'
```

All stop controls may be cancelled by giving **X** an empty vector:

```
0      ρ' ' □STOP 'FOO'
```

```
0      ρθ □STOP 'FOO'
```

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 [Trap Event](#) on page 458.

Example

```

      FX 'R←FOO' 'R←10'
      0 1  STOP 'FOO'

      FOO
FOO[1]

      R
VALUE ERROR
      R
      ^

      →1
FOO[0]

      R
10

      →LC
10

```

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

```

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 is 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 **4p0**.

Examples

```

      1 0 0 1 □SVC 'X'
1 0 0 1

      1 □SVC 'X EXTNAME'
1 1 1 1

      (2 4p1 0 0 1 0 1 1 0) □SVC ↑'ONE' 'TWO'
1 1 1 1
0 1 1 0

```

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

```

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 contain '**DDE:**'.

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

```

1      'DDE:' □SVO 'X'

1      'DDE:' □SVO 'X SALES_92'

1 1    'DDE:' □SVO ↑'X SALES_92' 'COSTS_92'

2      'DDE:DIALOG|SERV_WS' □SVO 'X'

2      'DDE:EXCEL|SHEET1' □SVO 'DATA R1C1:R10C12'

```

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:

```

2      'DDE:EXCEL|SYSTEM' □SVO 'X ⚡'

X←'[OPEN("c:\mydir\mysheet.xls")]'
X

1

X←'[SELECT("R1C1:R5C10")]'
X

1

```

Query Degree of Coupling

$R \leftarrow \square SVO \ Y$

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

```

2      □SVO 'X'
2      □SVO ↑'X SALES' 'Y' 'JUNK'
2 1 0
```

Shared Variable Query

 $R \leftarrow \square \text{SVQ } Y$

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 $\square \text{SVQ}$ effectively.

Shared Variable Retract Offer

 $R \leftarrow \square \text{SVR } Y$

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 or 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 [Shared Variable State on page 448](#) for further information on the possible states of a shared variable.

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:

0 0 1 1	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.
1 0 1 0	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.
0 1 0 1	means that your partner has reset the shared variable but that you have not yet used it.
0 0 0 0	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
```

Terminal Control

(**ML**)

R←TC

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[IO+13]** for **ML < 3**.

If **ML ≥ 3** the 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

Thread Child Numbers

$$R \leftarrow \square \text{TCNUMS } Y$$

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

```

Get Tokens

$$\{R\} \leftarrow \{X\} \square \text{TGET } Y$$

Y must be a simple integer 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.

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 ($\square \text{TGET}$) 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. *Negatively* 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 timeout (as specified by `X`) occurs.

For example, if the pool contains only tokens of type 2:

```
□TGET 2 4      A 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 3×2 -tokens in the pool:

```
□TGET 3/2      A 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      A empty pool.
'ABCDE'□TPUT''2 2 3 2 3  A pool some tokens.
```

```
AC
↳□TGET 2 3
```

```
BE
↳□TGET 2 3
```

Timeout is signalled by the return of an empty numeric vector \emptyset (zilde). By default, the value of a token is the same as its type. This means that, unless you have explicitly set the value of a token to \emptyset , a `□TGET` result of \emptyset unambiguously identifies a timeout.

Beware - the following statement will wait forever and can only be terminated by an interrupt.

```
□TGET 0      A wait forever ...
```

Note too that if a thread waiting to `□TGET` tokens is `□TKILL`d, the thread disappears without removing any tokens from the pool. Conversely, if a thread that has removed tokens from the pools is `□TKILL`d, the tokens are not returned to the pool.

This Space

R←THIS

THIS returns a reference to the current namespace, i.e. 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.(ρINSTANCES⇒CLASS THIS)
1
```


$$R \leftarrow \square TID$$

Examples

```
0      tid      A Base thread number
      &tid      A Thread number of async &.
1
```

$$\{R\} \leftarrow \{X\} \square \text{TKILL } Y$$

The **base thread 0** is always excluded from the cull.

Examples

```

□TKILL 0          a Kill background threads.
□TKILL □TID       a Kill self and descendants.
0 □TKILL □TID     a Kill self only.
□TKILL □TCNUMS □TID a Kill descendants.

```

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.

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
```

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
```

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 ⍵){(⍋⍵ ⍵){⍵}⍵ ⌈TPOOL''⍵}⍵}
snap ⌈TPOOL
1      hello world
2                      2
3                      2
2 three-type token
2                      2
```

Put Tokens

 $\{R\} \leftarrow \{X\} \quad \text{TPUT } Y$

Y must be a simple integer scalar or vector of non-zero token types.

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      A put a 2-token, a 3-token and another
r              2-token into the pool.
```

```
88 TPUT 2      A put another 2-token into the pool
              this token has the value 88.
```

```
'Hello' TPUT -4 A put a -4-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 .

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 3p'ABCDEF')A
[3]  →NEXT×1SKIP
[4]  'SKIPPED LINE'
[5]  NEXT:C←'one' ♦ D←'two'
[6]  END:R←C D
▽

(0,16) □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

```

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

```

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. 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:

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.

Table 16: Trappable Event Codes

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

Code	Event
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE
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

Code	Event
1000	Any event in range 1001-1008
1001	Stop vector
1002	Weak interrupt
1003	INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE (Dyalog APL/X, Dyalog APL/W)
1008	DEADLOCK

See *Programmer's Guide: Trap Statement* for an alternative 'control structured' error trapping mechanism.

Examples

```

      □TRAP←c(3 4 5) 'E' 'ERROR' ♦ ρ□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
      ρ□TRAP
1

```

The value of `□TRAP` in a clear workspace is an empty vector whose prototype is `0ρ(⊖ ' ' '')`. A convenient way of cancelling a `□TRAP` definition is:

```

□TRAP←0ρ□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]  A Option selection and execution
[2]  A A general cut-back trap
[3]  □TRAP←(0 1000)'C' '→ERR'
[4]  INP:□←'OPTION : ' ♦ OPT←(OPT≠' ')/OPT←9↓□
[5]  →EXP~(cOPT)∈Options ♦ 'INVALID OPTION' ♦ →INP
[6]  EX:±OPT ♦ →INP
[7]  ERR:ERRORΔACTION ♦ →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

```

Token Requests

$R \leftarrow \square \text{TREQ } Y$

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      A tokens required by all threads.

```

Time Stamp

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.

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 ← {ω ω}           A Duplicate
      R ← dup&88           A Show thread number
11
88 88

      TSYNC dup&88        A 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           A 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          A thread (3) runs and terminates.
3
0.25
    ⌈TSYNC 3        A no result required: no prob
    ⌈←⌈tsync 3      A context requires result
VALUE ERROR

    ⌈←⌈tsync {}&0  A non-result-returning fn: no resul
t.
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**.

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 11p72 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 Ελλάδα'
954 945 955 951 956 941 961 945 32 949 955 955 940 948
```

Unicode also contains the APL character set. For example:

```
⎕UCS 123 40 43 47 9077 41 247 9076 9077 125
{(+/ω)÷ρω}
```

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 13
3
'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 'ABCÆØÅΨΦ'
65 66 67 198 216 197 9042 9035
←unihan←UCS (2×2×16)+13  x20001-x20003
𐤁𐤂𐤃
'UTF-16' UCS unihan
55360 56321 55360 56322 55360 56323
```

Translation Error

`UCS` will generate **TRANSLATION ERROR** (event number 92) if the argument cannot be converted. In the Classic Edition, a **TRANSLATION ERROR** is generated if the result is not in `AV` or the numeric argument is not in `AVU`.

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 is a vector of character vectors, each element of which specifies the name of a .NET Namespace followed optionally by a comma (,) and the Assembly in which it is to be found.

If a pathname is specified, the file is loaded from that location. Otherwise the system will attempt to load the assembly first from the directory in which the Dyalog program (or host application) is located, and then from the .NET installation directory.

If the Microsoft .NET Framework is installed, the System namespace in `mscorlib.dll` is automatically loaded when Dyalog APL starts. To access this namespace, it is not necessary to specify the name of the Assembly.

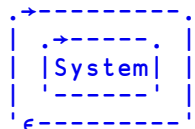
⚡USING has *namespace scope*. If the local value of ⚡USING is anything other than empty, and you reference a name that would otherwise generate a **VALUE ERROR**, APL searches the list of .NET Namespaces and Assemblies specified by ⚡USING for a class of that name. If it is found, an entry for the class is added to the symbol table in the current space and the class is used as specified. Note that subsequent references to that class in the current space will be identified immediately.

If ⚡USING is empty (its default value in a **CLEAR WS**) no such search is performed.

Note that when you assign a value to ⚡USING, you may specify a simple character vector or a vector of character vectors. If you specify a simple character vector (including an empty vector ''), this is equivalent to specifying a 1-element enclosed vector containing the specified characters. Thus to clear ⚡USING, you must set it to `0ρ<''` and not `''`.

Examples:

```
⚡USING←'System'
]display ⚡USING
```



```
⚡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←,c',.\LoanService.dll'
```

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

```
⌈V←⌈VR 'PLUS'
128
      V
      ▽ R←{A}PLUS B
[1]   ⌈ MONADIC OR DYADIC +
[2]   →DYADIC⌈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      A Arithmetic mean
[1]   R←(+/X)÷ρX
      ∇ °,

      ρF
3

      ]display F

```

°,
-,

Verify & Fix Input **$R \leftarrow \{X\} \sqcup VFI \ Y$**

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
      (Θ Θ)≡□VFI''
1

```

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. Workspace above a small amount (1/16 of the configured maximum workspace size) of working memory is returned to the Operating System. On a Windows system, you can see the process size changing by using Task Manager.

Example

```

      □WA
261412

```

See also: [Specify Workspace Available on page 173](#)

Windows Create Object

$$\{R\} \leftarrow \{X\} \square WC \ Y$$

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 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 result of **□OR**, the new object is a complete copy of the one from which the **□OR** was made, 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 *Chapter 1*.

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

A Create a default Form called F1

```
'F1' □WC 'Form'
```

A Create a Form with specified properties (by position)

```
A Caption = "My Application" (Title)
A Posn    = 10 30 (10% down, 30% across)
A Size    = 80 60 (80% high, 60% wide)
```

```
'F1' □WC 'Form' 'My Application' (10 30)(80 60)
```

```

A Create a Form with specified properties (by keyword)
A   Caption = "My Application" (Title)
A   Posn    = 10 30 (10% down, 30% across)
A   Size     = 80 60 (80% high, 60% wide)

```

```

PROPS←c'Type' 'Form'
PROPS,←c'Caption' 'My Application'
PROPS,←c'Posn' 10 30
PROPS,←c'Size' 80 60
'F1' □WC PROPS

```

```

A Create a default Button (a pushbutton) in the Form F1

```

```

'F1.BTN' □WC 'Button'

```

```

A Create a pushbutton labelled "Ôk"
A 10% down and 10% across from the start of the FORM
A with callback function FOO associated with EVENT 30
A (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' A Attach a Form to this namespace
)CS
#
  F.□WC'Form' A Attach a Form to namespace F

```

Windows Get Property

$R \leftarrow \{X\} \square WG \ Y$

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│
│ └───┘ └───┘ └───┘
└───┘
```

Windows Child Names

$R \leftarrow \{X\} \square WN \ Y$

This system function reports the Windows objects whose parent is Y .

If Y is a name (i.e. is a character vector) then the result R is a vector of character vectors containing the names of the named direct Windows children of Y .

If Y is a reference then the result R is a vector of references to the direct Windows children of Y , named or otherwise.

The optional left argument X is a character vector which specifies the **Type** of Windows 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 $\square WN$.

If Y refers to a namespace with no GUI element, a **VALUE ERROR** is reported.

Note that $\square WN$ reports **only** those child objects visible from the current thread.

GUI objects are named **relative** to the current namespace. The following examples are equivalent:

```
 $\square WN \ 'F1.B1'$ 
 $F1.\square WN \ 'B1'$ 
 $F1.B1.\square WN \ ''$ 
```

Example

```
f ←  $\square NEW$  c 'Form'
f.n ←  $\square ns$  ''
                                     A A non-windows object

f.l ← f. $\square NEW$  c 'Label'
'f.b1'  $\square wc$  'Button'
f.(b2 ←  $\square new$  c 'Button')
                                     A A reference to a Label
                                     A A named Button
                                     A A reference to a Butto

n
   $\square wn \ 'f'$ 
  [Form].b1
   $\square wn \ f$ 
  #.[Form].[Label]  #.[Form].b1  #.[Form].[Button]
  'Button'  $\square wn \ f$ 
  #.[Form].b1  #.[Form].[Button]
```


Windows Set Property

$\{R\} \leftarrow \{X\} \square WS \ Y$

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 $Y[2 \ 3 \ 4 \ \dots]$ 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 $\square WS$.

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 $\square WS$, 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 X (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 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
```

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 *Programmer's Guide: 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'.

Example

```
WSID  
CLEAR WS
```

```
WSID←'WS/MYWORK'      (UNIX)
```

```
WSID←'B:\WS\MYWORK'   (Windows)
```

Window Expose

WX

WX is a system variable that determines:

- a. whether or not the names of properties, methods and events provided by a Dyalog APL GUI object are exposed.
- b. certain aspects of behaviour of .NET and COM objects.

The permitted values of **WX** are 0, 1, or 3. Considered as a sum of bit flags, the first bit in **WX** specifies (a), and the second bit specifies (b).

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 Version 11 behaviour, as opposed to the behaviour in previous versions of Dyalog APL.

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.

The value of **WX** in a clear workspace is defined by the default_wx parameter (see User Guide) 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.

XML Convert

$$R \leftarrow \{X\} \text{ XML } Y$$

`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, (<code>0 2p<' '</code>) when empty
5	A numeric value which indicates what the row contains

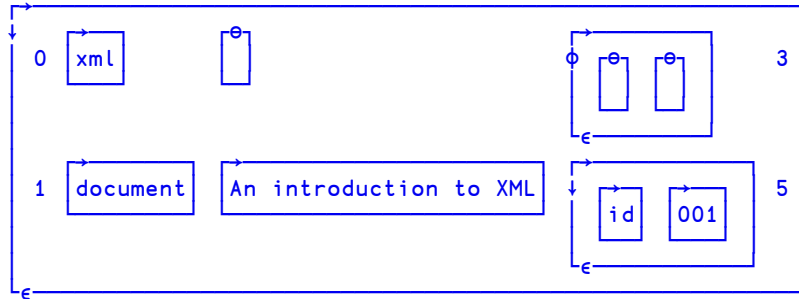
The values in column 5 have the following meanings:

Value	Description
1	Element
2	Child element
4	Character data
8	Markup not otherwise defined
16	Comment markup
32	Processing instruction markup

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 specification¹ 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.

`⎕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.

¹<http://www.w3.org/TR/2008/REC-xml-20081126/>

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 ‘-->’ 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 are delimited by ‘<![CDATA[‘ and ‘]]>’. 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
- `<xml><document id="001">An introduction to XML </document></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 $(n \times 2)$ 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 there are no attributes in a particular row then the `(0 2p<'')` may be abbreviated as `␣` (zilde). If the fifth column is omitted then the fourth column may also be omitted altogether.
- 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 $\leq +1$.
- 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 is 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; e.g. **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` may 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 `⌈`. i.e.

```
]display (⌈XML⌈'Whitespace' 'Strip')eg
A      '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

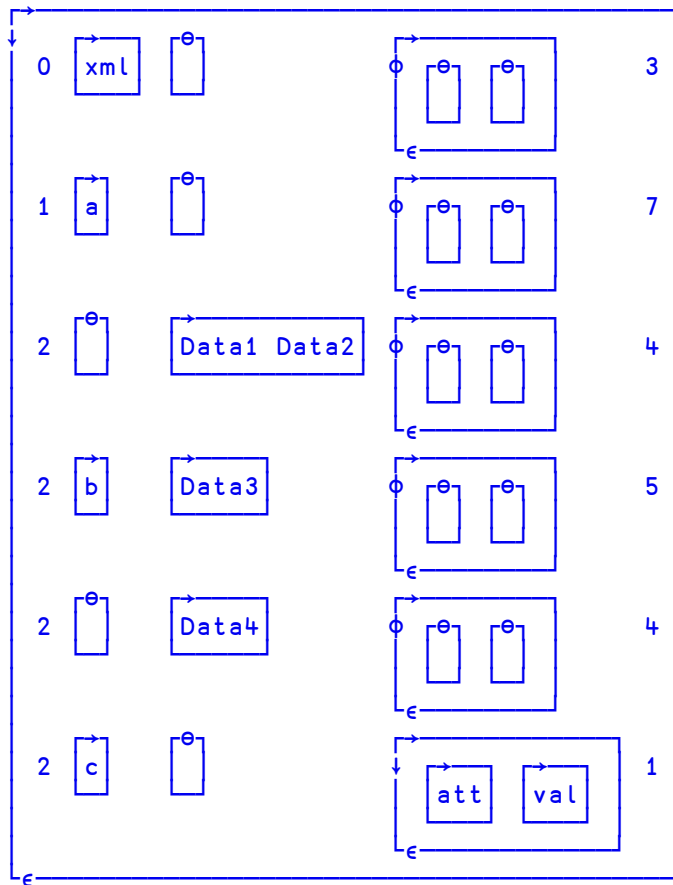
Whitespace specifies the default formatting of the XML. Note that attribute values are not comprised of character data so white space in attribute values is always preserved.

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>
```

```
]display (["XML" 'Whitespace' 'Strip'])eg
A      'whitespace' 'strip' ["XML"] eg
```



Index	Left Node	Right Node	Value
0			7
1			4
1			7
2			4
2			5
2			4
2			1
2			4
1			4

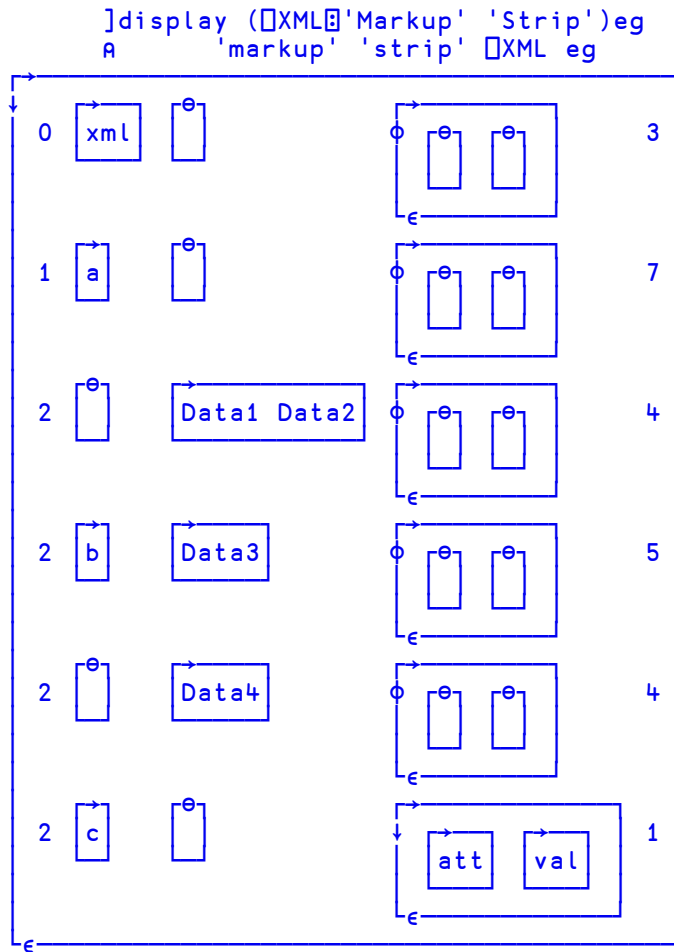
Markup (markup)

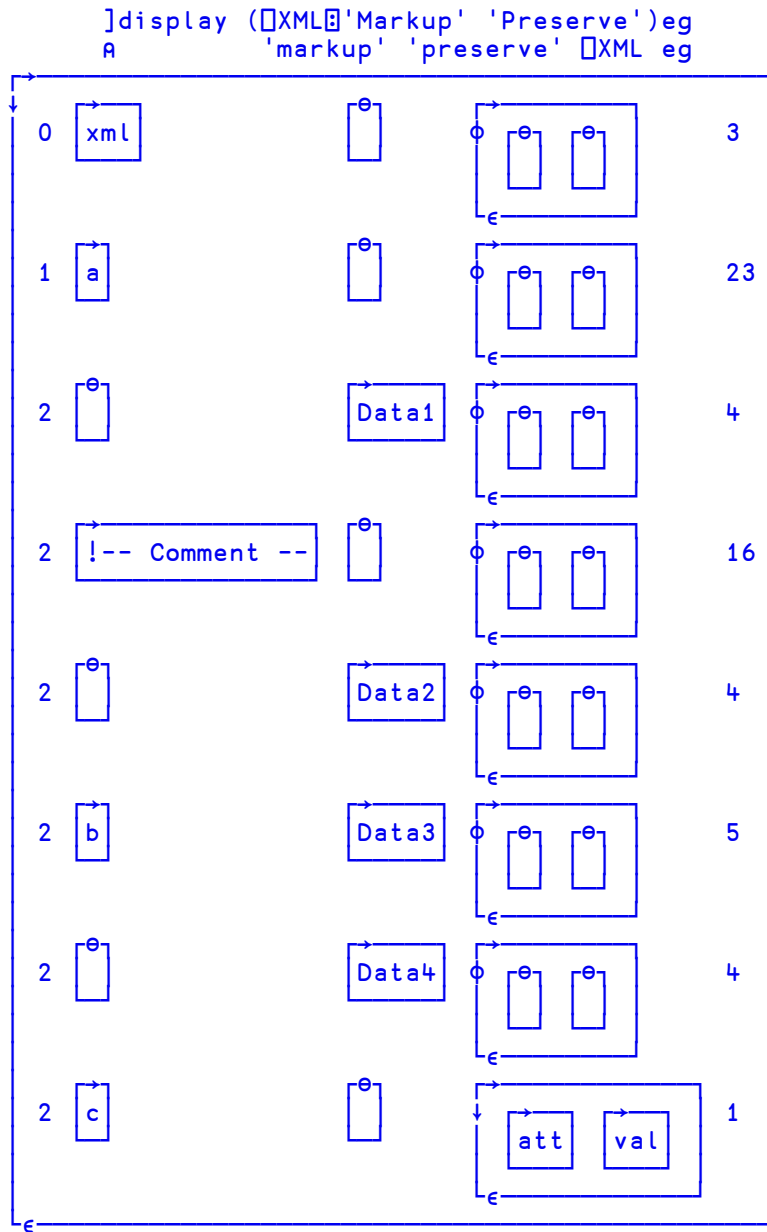
When converting from XML, **Markup** determines whether markup (other than entity tags) appears in the output array or not. When converting to XML **Markup** has no effect.

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>
```





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 ‘&’

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 [State Indicator](#) on page 423.

Set External Variable

X **EXT** **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**. See *User Guide* for file naming conventions under Windows and UNIX.

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.

Example

```
'EXT\ARRAY' EXT 'V'
```

If the file reference does not exist, the external variable has no value until a value is assigned:

```
VALUE V
      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←15
      +/V
15

      V[3]←c'ABC'

      V
1 2 ABC 4 5

      ρ∘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 4ρ18

      ρV
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←110
+ /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.

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
0
pXT 'G'
```

Chapter 5:

System Commands

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.

Each command is described in alphabetical order in this chapter.

Table 17: System Commands

Command	Description
<code>)CLASSES</code>	List classes
<code>)CLEAR</code>	Clear the workspace
<code>)CMD Y</code>	Execute a Windows Command
<code>)CONTINUE</code>	Save a Continue workspace and terminate APL
<code>)COPY {Y}</code>	Copy objects from another workspace
<code>)CS {Y}</code>	Change current namespace
<code>)DROP {Y}</code>	Drop named workspace
<code>)ED Y</code>	Edit object(s)
<code>)ERASE Y</code>	Erase object(s)
<code>)EVENTS</code>	List events of GUI namespace or object

Command	Description
)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
)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 <code>⎕LX</code>
{ } indicates that the parameter(s) denoted by Y are optional.	

List Classes

)CLASSES

This command lists the names of APL Classes in the active workspace.

Example:

```

)CLEAR
clear ws
)ED oMyClass

:Class MyClass
  ▽ Make Name
    :Implements Constructor
    □DF Name
  ▽
:EndClass R MyClass

)CLASSES
MyClass
)COPY OO YourClass
.\OO saved Sun Jan 29 18:32:03 2006
)CLASSES
MyClass YourClass
  □NC 'MyClass' 'YourClass'
9.4 9.4

```

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 as described in [System Variables on page 200](#).

In GUI implementations of Dyalog APL, **)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.

Example

```

)CLEAR
clear ws

```

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 [Execute \(UNIX\) Command on page 518](#) for a discussion of the behaviour of these commands under UNIX.

The system functions **□SH** and **□CMD** provide similar facilities but may be executed from within APL code. For further information, see [Execute \(UNIX\) Command on page 421](#) and [Execute Windows Command on page 231](#).

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 has no label
Directory of C:\PETE\WS
```

```
.                <DIR>      5-07-94   3.02p
..              <DIR>      5-07-94   3.02p
SALES      DWS      110092 5-07-94   3.29p
EXPENSES   DWS      154207 5-07-94   3.29p
```

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.

Save Continuation

)CONTINUE

This command saves the active workspace under the name `CONTINUE` and ends the Dyalog APL session.

When you subsequently start another Dyalog APL session, the `CONTINUE` workspace is loaded automatically. When a `CONTINUE` workspace is loaded, the latent expression (if any) is NOT executed.

Note that the values of all system variables (including `□SM`) and GUI objects are also saved in `CONTINUE`.

Copy Workspace

)COPY {ws {nms}}

This command brings all or selected global objects **nms** from a stored workspace 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 *Programmer's Guide: 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 `UAV` indices, whether or not that object is specified by `nms`. See [Atomic Vector - Unicode on page 224](#) 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.

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 dependant 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).

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
```

Drop Workspace

)DROP {ws}

This command removes the specified workspace from disk storage. See *Programmer's Guide: Workspaces* for information regarding 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
```

Edit Object

)ED nms

)ED invokes the Dyalog APL editor and opens an Edit window for each of the objects specified in **nms**.

If a name specifies a new symbol it is taken to be a function/operator. However, if a name is localised in a suspended function/operator but is otherwise undefined, it is assumed to be a vector of character vectors.

The type of a new object may be specified explicitly by preceding its name with an appropriate symbol as follows:

▽	function/operator
→	simple character vector
€	vector of character vectors
–	character matrix
⊗	Namespace script
○	Class script
◦	Interface

The first object named becomes the top window on the stack. See *User Guide* for details.

Examples

```
)ED MYFUNCTION
```

```
)ED ▽FOO –MAT €VECVEC
```

Objects specified in **nms** that cannot be edited are silently ignored. Objects qualified with a namespace path are (e.g. **a.b.c.foo**) are silently ignored if the namespace does not exist.

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 [Expunge Object on page 263](#) for details.

Example

```
)ERASE FOO A []IO
not found []IO
```

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
```


List Global Defined Functions

)FNS {nm}

This command displays the names of global defined functions in the active work-space 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
```

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.
```

List Workspace Library

)LIB {dir}

This command lists the names of Dyalog APL 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
```

Load Workspace

)LOAD {ws}

This command causes the named stored workspace to be loaded. The current active workspace is lost.

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 workspace name "**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 latent expression (**⎕LX**) is executed unless APL was invoked with the **-x** option.

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 now 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
/opt/mdyalog/14.0/64/unicode/ws/dfns saved Thu Jan 16 00:
09:55 2014
```

An assortment of D Functions and Operators.

```
tree #           A Workspace map.
↑~10↑↓attrib ⎕nl 3 4 A What's new?
notes find 'Word'  A Apropos "Word".
```

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

```

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

```

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
```

List Global Namespaces

)OBS {nm}

This command is the same as the **)OBJECTS** command. See [List Global Namespaces on page 514](#)

Sign Off APL

)OFF

This command terminates the APL session, returning to the Operating System command processor or shell.

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
```

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 *Programmer's Guide: 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 [Copy Workspace on page 242](#).

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
```

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
```

Reset State Indicator

)RESET

This command cancels all suspensions recorded in the state indicator and discards any unprocessed events in the event queue.

)RESET also performs an internal re-organisation of the workspace and process memory. See [Workspace Available on page 471](#) for details.

Example

```
)SI
#.FOO[1]*
⚡
#.FOO[1]*

)RESET

)SI
```

Save Workspace

)SAVE {ws}

This command compacts (see [Workspace Available on page 471](#) for details) and saves the active workspace

The workspace is saved with its state of execution intact. 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**.

This command may fail with one of the following error messages:

<code>unacceptable char</code>	The given workspace name was ill-formed
<code>not saved this ws is WSID</code>	An attempt was made to change the name of the workspace for the save, and that workspace already existed.
<code>not saved this ws is CLEAR WS</code>	The active workspace was CLEAR WS and no attempt was made to change the name.
<code>Can't save - file could not be created.</code>	The workspace name supplied did not represent a valid file name for the current Operating System.
<code>cannot create</code>	The user does not have access to create the file OR the workspace name conflicts with an existing non-workspace file.
<code>cannot save with windows open</code>	A workspace may not be saved if trace or edit windows are open.

An existing stored workspace with the same name will be replaced. The active workspace may be renamed by the system command `)WSID` or the system function `QWSID`.

After a successful save, the system reports the workspace name, "**saved**", followed by the time and date.

Example

```
)SAVE MYWORK
./MYWORK saved Thu Sep 17 10:32:20 1998
```

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 [Windows Command Processor on page 502](#) for a discussion of their behaviour under WINDOWS.

The system functions **□SH** and **□CMD** provide similar facilities but may be executed from within APL code. For further information, see [Execute \(UNIX\) Command on page 421](#) and [Execute Windows Command on page 231](#).

)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 APL 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 *User 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 ls
EXT
FILES
```

State Indicator

)SI

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 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 multithreading 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].

Clear State Indicator

)SIC

This command is a synonym for) RESET. See [Reset State Indicator on page 516](#)

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

```

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

```

      A State indicator
    )si
.   #.print[1]
&3
.   .   #.sub_calc[2]*
.   &2
.   #.calc[1]
&1

      A Current thread
    )tid
is 2

      A Switch suspension to thread 3
    )tid 3
was 2

      A State indicator
    )si
.   #.print[1]*
&3
.   .   #.sub_calc[2]
.   &2
.   calc[1]
&1

      A Attempt to switch to pendent thread 1
    )tid 1
Can't switch, this thread is 3
```

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
```

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 *Programmer's Guide: 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
```

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.

Appendices: PCRE Specifications

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.

There are two named sections: *pcrpattern*, which describes the full syntax and semantics; and *preresyntax*, a quick reference summary.

Appendix A - PCRE Syntax Summary

The following is a summary of search pattern syntax.

PCRESYNTAX (3)

PCRESYNTAX (3)

NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION SYNTAX SUMMARY

The full syntax and semantics of the regular expressions that are supported by PCRE are described in the `pcrepattern` documentation. This document contains just a quick-reference summary of the syntax.

QUOTING

<code>\x</code>	where x is non-alphanumeric is a literal x
<code>\Q...\E</code>	treat enclosed characters as literal

CHARACTERS

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any ASCII character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	newline (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or backreference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh..

CHARACTER TYPES

<code>.</code>	any character except newline; in dotall mode, any character whatsoever
<code>\C</code>	one byte, even in UTF-8 mode (best avoided)
<code>\d</code>	a decimal digit
<code>\D</code>	a character that is not a decimal digit
<code>\h</code>	a horizontal whitespace character
<code>\H</code>	a character that is not a horizontal whitespace character
<code>\N</code>	a character that is not a newline
<code>\p{xx}</code>	a character with the xx property
<code>\P{xx}</code>	a character without the xx property
<code>\R</code>	a newline sequence
<code>\s</code>	a whitespace character
<code>\S</code>	a character that is not a whitespace character
<code>\v</code>	a vertical whitespace character
<code>\V</code>	a character that is not a vertical whitespace character
<code>\w</code>	a "word" character
<code>\W</code>	a "non-word" character
<code>\X</code>	an extended Unicode sequence

In PCRE, by default, `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` recognize only ASCII

characters, even in UTF-8 mode. However, this can be changed by setting the PCRE_UCP option.

GENERAL CATEGORY PROPERTIES FOR \p and \P

C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
L&	Ll, Lu, or Lt
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

PCRE SPECIAL CATEGORY PROPERTIES FOR \p and \P

Xan	Alphanumeric: union of properties L and N
Xps	POSIX space: property Z or tab, NL, VT, FF, CR
Xsp	Perl space: property Z or tab, NL, FF, CR
Xwd	Perl word: property Xan or underscore

SCRIPT NAMES FOR \p AND \P

Arabic, Armenian, Avestan, Balinese, Bamum, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Egyptian_Hieroglyphs, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hira-gana, Imperial_Aramaic, Inherited, Inscriptional_Pahlavi, Inscriptional_Parthian, Javanese, Kaithi, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lisu, Lycian, Lydian, Malayalam, Meetei_Mayek, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Old_South_Arabian, Old_Turkic, Ol_Chiki, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Samaritan, Saurashtra, Shavian, Sinhala, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tai_Tham, Tai_Viet, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, Yi.

CHARACTER CLASSES

[...]	positive character class
[^...]	negative character class
[x-y]	range (can be used for hex characters)
[:xxx:]	positive POSIX named set
[!^xxx:]	negative POSIX named set

alnum	alphanumeric
alpha	alphabetic
ascii	0-127
blank	space or tab
cntrl	control character
digit	decimal digit
graph	printing, excluding space
lower	lower case letter
print	printing, including space
punct	printing, excluding alphanumeric
space	whitespace
upper	upper case letter
word	same as \w
xdigit	hexadecimal digit

In PCRE, POSIX character set names recognize only ASCII characters by default, but some of them use Unicode properties if PCRE_UCP is set. You can use \Q...\E inside a character class.

QUANTIFIERS

?	0 or 1, greedy
?+	0 or 1, possessive
??	0 or 1, lazy
*	0 or more, greedy
*+	0 or more, possessive
*?	0 or more, lazy
+	1 or more, greedy
++	1 or more, possessive
+	1 or more, lazy
{n}	exactly n
{n,m}	at least n, no more than m, greedy
{n,m}+	at least n, no more than m, possessive

<code>{n,m}?</code>	at least <i>n</i> , no more than <i>m</i> , lazy
<code>{n,}</code>	<i>n</i> or more, greedy
<code>{n,}+</code>	<i>n</i> or more, possessive
<code>{n,}? </code>	<i>n</i> or more, lazy

ANCHORS AND SIMPLE ASSERTIONS

<code>\b</code>	word boundary
<code>\B</code>	not a word boundary
<code>^</code>	start of subject also after internal newline in multiline mode
<code>\A</code>	start of subject
<code>\$</code>	end of subject also before newline at end of subject also before internal newline in multiline mode
<code>\Z</code>	end of subject also before newline at end of subject
<code>\z</code>	end of subject
<code>\G</code>	first matching position in subject

MATCH POINT RESET

<code>\K</code>	reset start of match
-----------------	----------------------

ALTERNATION

`expr|expr|expr...`

CAPTURING

<code>(...)</code>	capturing group
<code>(?<name>...)</code>	named capturing group (Perl)
<code>(?'name'...)</code>	named capturing group (Perl)
<code>(?P<name>...)</code>	named capturing group (Python)
<code>(?:...)</code>	non-capturing group
<code>(? ...)</code>	non-capturing group; reset group numbers for capturing groups in each alternative

ATOMIC GROUPS

<code>(?>...)</code>	atomic, non-capturing group
-------------------------	-----------------------------

COMMENT

<code>(?#....)</code>	comment (not nestable)
-----------------------	------------------------

OPTION SETTING

<code>(?i)</code>	caseless
<code>(?J)</code>	allow duplicate names
<code>(?m)</code>	multiline
<code>(?s)</code>	single line (dotall)
<code>(?U)</code>	default ungreedy (lazy)

(?x)	extended (ignore white space)
(?-...)	unset option(s)

The following are recognized only at the start of a pattern or after one of the newline-setting options with similar syntax:

(*NO_START_OPT)	no start-match optimization (PCRE_NO_START_OPTIMIZE)
(*UTF8)	set UTF-8 mode (PCRE_UTF8)
(*UCP)	set PCRE_UCP (use Unicode properties for \d etc)

LOOKAHEAD AND LOOKBEHIND ASSERTIONS

(?=...)	positive look ahead
(?!...)	negative look ahead
(?<=...)	positive look behind
(?<!=...)	negative look behind

Each top-level branch of a look behind must be of a fixed length.

BACKREFERENCES

\n	reference by number (can be ambiguous)
\gn	reference by number
\g{n}	reference by number
\g{-n}	relative reference by number
\k<name>	reference by name (Perl)
\k'name'	reference by name (Perl)
\g{name}	reference by name (Perl)
\k{name}	reference by name (.NET)
(?P=name)	reference by name (Python)

SUBROUTINE REFERENCES (POSSIBLY RECURSIVE)

(?R)	recurse whole pattern
(?n)	call subpattern by absolute number
(?+n)	call subpattern by relative number
(?-n)	call subpattern by relative number
(?&name)	call subpattern by name (Perl)
(?P>name)	call subpattern by name (Python)
\g<name>	call subpattern by name (Oniguruma)
\g'name'	call subpattern by name (Oniguruma)
\g<n>	call subpattern by absolute number (Oniguruma)
\g'n'	call subpattern by absolute number (Oniguruma)
\g<+n>	call subpattern by relative number (PCRE extension)
\g'+n'	call subpattern by relative number (PCRE extension)
\g<-n>	call subpattern by relative number (PCRE extension)
\g'-n'	call subpattern by relative number (PCRE extension)

CONDITIONAL PATTERNS

(?(condition)yes-pattern)	
(?(condition)yes-pattern no-pattern)	
(?(n)...	absolute reference condition
(?(+n)...	relative reference condition
(?(-n)...	relative reference condition

(?(<name>)...	named reference condition (Perl)
(?('name')...	named reference condition (Perl)
(?(name)...	named reference condition (PCRE)
(?(R)...	overall recursion condition
(?(Rn)...	specific group recursion condition
(?(R&name)...	specific recursion condition
(?(DEFINE)...	define subpattern for reference
(?(assert)...	assertion condition

BACKTRACKING CONTROL

The following act immediately they are reached:

(*ACCEPT)	force successful match
(*FAIL)	force backtrack; synonym (*F)

The following act only when a subsequent match failure causes a backtrack to reach them. They all force a match failure, but they differ in what happens afterwards. Those that advance the start-of-match point do so only if the pattern is not anchored.

(*COMMIT)	overall failure, no advance of starting point
(*PRUNE)	advance to next starting character
(*SKIP)	advance start to current matching position
(*THEN)	local failure, backtrack to next alternation

NEWLINE CONVENTIONS

These are recognized only at the very start of the pattern or after a (*BSR_...) or (*UTF8) or (*UCP) option.

(*CR)	carriage return only
(*LF)	linefeed only
(*CRLF)	carriage return followed by linefeed
(*ANYCRLF)	all three of the above
(*ANY)	any Unicode newline sequence

WHAT \R MATCHES

These are recognized only at the very start of the pattern or after a (*...) option that sets the newline convention or UTF-8 or UCP mode.

(*BSR_ANYCRLF)	CR, LF, or CRLF
(*BSR_UNICODE)	any Unicode newline sequence

CALLOUTS

(?C)	callout
(?Cn)	callout with data n

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 21 November 2010

Copyright (c) 1997-2010 University of Cambridge.

Appendix B - PCRE Regular Expression Details

PCREPATTERN(3)

PCREPATTERN(3)

NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions that are supported by PCRE are described in detail below. There is a quick-reference syntax summary in the `pcresyntax` page. PCRE tries to match Perl syntax and semantics as closely as it can. PCRE also supports some alternative regular expression syntax (which does not conflict with the Perl syntax) in order to provide some compatibility with regular expressions in Python, .NET, and Oniguruma.

Perl's regular expressions are described in its own documentation, and regular expressions in general are covered in a number of books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

The original operation of PCRE was on strings of one-byte characters. However, there is now also support for UTF-8 character strings. To use this, PCRE must be built to include UTF-8 support, and you must call `pcre_compile()` or `pcre_compile2()` with the `PCRE_UTF8` option. There is also a special sequence that can be given at the start of a pattern:

(*UTF8)

Starting a pattern with this sequence is equivalent to setting the `PCRE_UTF8` option. This feature is not Perl-compatible. How setting UTF-8 mode affects pattern matching is mentioned in several places below. There is also a summary of UTF-8 features in the section on UTF-8 support in the main `pcre` page.

The remainder of this document discusses the patterns that are supported by PCRE when its main matching function, `pcre_exec()`, is used. From release 6.0, PCRE offers a second matching function, `pcre_dfa_exec()`, which matches using a different algorithm that is not Perl-compatible. Some of the features discussed below are not available when `pcre_dfa_exec()` is used. The advantages and disadvantages of the alternative function, and how it differs from the normal function, are discussed in the `pcrematching` page.

NEWLINE CONVENTIONS

PCRE supports five different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (line-feed) character, the two-character sequence CRLF, any of the three preceding, or any Unicode newline sequence. The `pcreapi` page has further discussion about newlines, and shows how to set the newline convention in the options arguments for the compiling and matching functions.

It is also possible to specify a newline convention by starting a pat-

tern string with one of the following five sequences:

```
(*CR)      carriage return
(*LF)      linefeed
(*CRLF)    carriage return, followed by linefeed
(*ANYCRLF) any of the three above
(*ANY)     all Unicode newline sequences
```

These override the default and the options given to `pcre_compile()` or `pcre_compile2()`. For example, on a Unix system where LF is the default newline sequence, the pattern

```
(*CR)a.b
```

changes the convention to CR. That pattern matches "a\nb" because LF is no longer a newline. Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used.

The newline convention does not affect what the `\R` escape sequence matches. By default, this is any Unicode newline sequence, for Perl compatibility. However, this can be changed; see the description of `\R` in the section entitled "Newline sequences" below. A change of `\R` setting can be combined with a change of newline convention.

CHARACTERS AND METACHARACTERS

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the `PCRE_CASELESS` option), letters are matched independently of case. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of metacharacters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

```
\      general escape character with several uses
^      assert start of string (or line, in multiline mode)
$      assert end of string (or line, in multiline mode)
.      match any character except newline (by default)
```

```
[      start character class definition
|      start of alternative branch
(      start subpattern
)      end subpattern
?      extends the meaning of (
        also 0 or 1 quantifier
        also quantifier minimizer
*      0 or more quantifier
+      1 or more quantifier
        also "possessive quantifier"
{      start min/max quantifier
```

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

```
\      general escape character
^      negate the class, but only if the first character
-      indicates character range
[      POSIX character class (only if followed by POSIX
        syntax)
]      terminates the character class
```

The following sections describe the use of each of the metacharacters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a `*` character, you write `*` in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

If a pattern is compiled with the `PCRE_EXTENDED` option, whitespace in the pattern (other than in a character class) and characters between a `#` outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or `#` character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q...\E` sequences in PCRE, whereas in Perl, `$` and `@` cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
<code>\Qabc\$xyz\E</code>	<code>abc\$xyz</code>	<code>abc</code> followed by the contents of <code>\$xyz</code>
<code>\Qabc\ \$xyz\E</code>	<code>abc\ \$xyz</code>	<code>abc\ \$xyz</code>
<code>\Qabc\E\ \$Qxyz\E</code>	<code>abc\$xyz</code>	<code>abc\$xyz</code>

The `\Q...\E` sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences than the binary character it represents:

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	linefeed (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or back reference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh..

The precise effect of `\cx` is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

After `\x`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between `\x{` and `}`, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2^{31} in UTF-8 mode. That is, the maximum value in hexadecimal is 7FFFFFFF. Note that this is bigger than the largest Unicode code point, which is 10FFFF.

If characters other than hexadecimal digits appear between `\x{` and `}`, or if there is no terminating `}`, this form of escape is not recognized. Instead, the initial `\x` will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for `\x`. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}`.

After `\0` up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x\07` specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to gen-

erate a data character. Any subsequent digits stand for themselves. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

```
\040  is another way of writing a space
\40   is the same, provided there are fewer than 40
       previous capturing subpatterns
\7    is always a back reference
\11   might be a back reference, or another way of
       writing a tab
\011  is always a tab
\0113 is a tab followed by the character "3"
\113  might be a back reference, otherwise the
       character with octal code 113
\377  might be a back reference, otherwise
       the byte consisting entirely of 1 bits
\81   is either a back reference, or a binary zero
       followed by the two characters "8" and "1"
```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by an unsigned or a negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed later, following the discussion of parenthesized subpatterns.

Absolute and relative subroutine calls

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a "subroutine". Details are discussed later. Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

```
\d    any decimal digit
\D    any character that is not a decimal digit
\h    any horizontal whitespace character
\H    any character that is not a horizontal whitespace character
\s    any whitespace character
\S    any character that is not a whitespace character
\v    any vertical whitespace character
\V    any character that is not a vertical whitespace character
\w    any "word" character
```

`\W` any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32). If "use locale;" is included in a Perl script, `\s` may match the VT character. In PCRE, it never does.

In UTF-8 mode, characters with values greater than 128 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`. This is true even when Unicode character property support is available. These sequences retain their original meanings from before UTF-8 support was available, mainly for efficiency reasons. Note that this also affects `\b`, because it is defined in terms of `\w` and `\W`.

The sequences `\h`, `\H`, `\v`, and `\V` are Perl 5.10 features. In contrast to the other sequences, these do match certain high-valued codepoints in UTF-8 mode. The horizontal space characters are:

U+0009	Horizontal tab
U+0020	Space
U+00A0	Non-break space
U+1680	Ogham space mark
U+180E	Mongolian vowel separator
U+2000	En quad
U+2001	Em quad
U+2002	En space
U+2003	Em space
U+2004	Three-per-em space
U+2005	Four-per-em space
U+2006	Six-per-em space
U+2007	Figure space
U+2008	Punctuation space
U+2009	Thin space
U+200A	Hair space
U+202F	Narrow no-break space
U+205F	Medium mathematical space
U+3000	Ideographic space

The vertical space characters are:

U+000A	Linefeed
U+000B	Vertical tab
U+000C	Formfeed
U+000D	Carriage return
U+0085	Next line
U+2028	Line separator
U+2029	Paragraph separator

A "word" character is an underscore or any character less than 256 that is a letter or digit. The definition of letters and digits is con-

trolled by PCRE's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the pcreapi page). For example, in a French locale such as "fr_FR" in Unix-like systems, or "french" in Windows, some character codes greater than 128 are used for accented letters, and these are matched by \w. The use of locales with Unicode is discouraged.

Newline sequences

Outside a character class, by default, the escape sequence \R matches any Unicode newline sequence. This is a Perl 5.10 feature. In non-UTF-8 mode \R is equivalent to the following:

```
(?>\r\n|\n|\x0b|\f|\r|\x85)
```

This is an example of an "atomic group", details of which are given below. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

In UTF-8 mode, two additional characters whose codepoints are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

It is possible to restrict \R to match only CR, LF, or CRLF (instead of the complete set of Unicode line endings) by setting the option PCRE_BSR_ANYCRLF either at compile time or when the pattern is matched. (BSR is an abbreviation for "backslash R".) This can be made the default when PCRE is built; if this is the case, the other behaviour can be requested via the PCRE_BSR_UNICODE option. It is also possible to specify these settings by starting a pattern string with one of the following sequences:

```
(*BSR_ANYCRLF)  CR, LF, or CRLF only
(*BSR_UNICODE)   any Unicode newline sequence
```

These override the default and the options given to pcre_compile() or pcre_compile2(), but they can be overridden by options given to pcre_exec() or pcre_dfa_exec(). Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used. They can be combined with a change of newline convention, for example, a pattern can start with:

```
(*ANY) (*BSR_ANYCRLF)
```

Inside a character class, \R matches the letter "R".

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences that match characters with specific properties are available. When not in UTF-8 mode, these sequences are of course limited to testing characters whose codepoints are less than 256, but they do work in this mode. The extra escape sequences are:

```
\p{xx}  a character with the xx property
```

```
\P{xx}  a character without the xx property
\X      an extended Unicode sequence
```

The property names represented by xx above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Other properties such as "InMusical-Symbols" are not currently supported by PCRE. Note that \P{Any} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

```
\p{Greek}
\P{Han}
```

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian Aboriginal, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hira-gana, Inherited, Kannada, Katakana, Kharoshthi, Khmer, Lao, Latin, Limbu, Linear_B, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Runic, Shavian, Sinhala, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Yi.

Each character has exactly one general category property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \p{^Lu} is the same as \P{Lu}.

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

```
\p{L}
\pL
```

The following general category property codes are supported:

C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
M	Mark
Mc	Spacing mark

Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

The special property `L&` is also supported: it matches a character that has the `Lu`, `Ll`, or `Lt` property, in other words, a letter that is not classified as a modifier or "other".

The `Cs` (Surrogate) property applies only to characters in the range U+D800 to U+DFFF. Such characters are not valid in UTF-8 strings (see RFC 3629) and so cannot be tested by PCRE, unless UTF-8 validity checking has been turned off (see the discussion of `PCRE_NO_UTF8_CHECK` in the `pcreapi` page). Perl does not support the `Cs` property.

The long synonyms for property names that Perl supports (such as `\p{Letter}`) are not supported by PCRE, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the `Cn` (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, `\p{Lu}` always matches only upper case letters.

The `\X` escape matches any number of Unicode characters that form an extended Unicode sequence. `\X` is equivalent to

```
(?>\PM\pM*)
```

That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character. None of them have codepoints less than 256, so in non-UTF-8 mode `\X` matches any one character.

Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as `\d` and `\w` do not use Unicode properties in PCRE.

Resetting the match start

The escape sequence `\K`, which is a Perl 5.10 feature, causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:

```
foo\Kbar
```

matches "foobar", but reports that it has matched "bar". This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of `\K` does not interfere with the setting of captured substrings. For example, when the pattern

```
(foo)\Kbar
```

matches "foobar", the first substring is still set to "foo".

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

```
\b    matches at a word boundary
\B    matches when not at a word boundary
\A    matches at the start of the subject
\Z    matches at the end of the subject
      also matches before a newline at the end of the subject
\z    matches only at the end of the subject
\G    matches at the first matching position in the subject
```

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively. Neither PCRE nor Perl has a separate "start of word" or "end of word" metasequence. However, whatever follows `\b` normally determines which it is. For example, the fragment `\ba` matches "a" at the start of a word.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the `PCRE_NOTBOL` or `PCRE_NOTEOL` options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the `startoffset` argument of `pcre_exec()` is non-zero, indi-

cating that matching is to start at a point other than the beginning of the subject, `\A` can never match. The difference between `\Z` and `\z` is that `\Z` matches before a newline at the end of the string as well as at the very end, whereas `\z` matches only at the end.

The `\G` assertion is true only when the current matching position is at the start point of the match, as specified by the `startoffset` argument of `pcre_exec()`. It differs from `\A` when the value of `startoffset` is non-zero. By calling `pcre_exec()` multiple times with appropriate arguments, you can mimic Perl's `/g` option, and it is in this kind of implementation where `\G` can be useful.

Note, however, that PCRE's interpretation of `\G`, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with `\G`, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the `startoffset` argument of `pcre_exec()` is non-zero, circumflex can never match if the `PCRE_MULTILINE` option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the `PCRE_DOLLAR_ENDONLY` option at compile time. This does not affect the `\Z` assertion.

The meanings of the circumflex and dollar characters are changed if the `PCRE_MULTILINE` option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when `PCRE_MULTILINE` is set. When newline is specified as the two-character sequence `CRLF`, isolated `CR` and `LF` characters do not indicate newlines.

For example, the pattern `/^abc$` matches the subject string `"def\nabc"` (where `\n` represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode, and a match for circumflex is possible when the `startoffset` argument of `pcre_exec()` is non-zero. The `PCRE_DOLLAR_ENDONLY` option is ignored if `PCRE_MULTILINE` is set.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether or not `PCRE_MULTILINE` is set.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line. In UTF-8 mode, the matched character may be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

The behaviour of dot with regard to newlines can be changed. If the `PCRE_DOTALL` option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence `\C` matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the `\C` escape sequence is best avoided.

PCRE does not allow `\C` to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special by default. However, if the `PCRE_JAVASCRIPT_COMPAT` option is set, a lone closing square bracket causes a compile-time error. If a closing square bracket is required as a member of the class, it should be the

first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may be more than one byte long. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `[^aeiou]` matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion; it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the `\x{` escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `[^aeiou]` does not match "A", whereas a caseful version would. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching in UTF8-mode for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the `PCRE_DOTALL` and `PCRE_MULTILINE` options is used. A class such as `[^a]` always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as `[W-]46]` is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so `[W-\\]46]` is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example `[\000-\037]`. In UTF-8 mode, ranges can include characters whose values

are greater than 255, for example `[\x{100}-\x{2ff}]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[[\^_`wxyzabc]`, matched caselessly, and in non-UTF-8 mode, if character tables for a French locale are in use, `[\xc8-\xcb]` matches accented E characters in both cases. In UTF-8 mode, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character types `\d`, `\D`, `\p`, `\P`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `^[^W_]` matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]]%
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

<code>alnum</code>	letters and digits
<code>alpha</code>	letters
<code>ascii</code>	character codes 0 - 127
<code>blank</code>	space or tab only
<code>cntrl</code>	control characters
<code>digit</code>	decimal digits (same as <code>\d</code>)
<code>graph</code>	printing characters, excluding space
<code>lower</code>	lower case letters
<code>print</code>	printing characters, including space
<code>punct</code>	printing characters, excluding letters and digits
<code>space</code>	white space (not quite the same as <code>\s</code>)
<code>upper</code>	upper case letters
<code>word</code>	"word" characters (same as <code>\w</code>)
<code>xdigit</code>	hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to `\s`, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 128 do not match any of the POSIX character classes.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the `PCRE_CASELESS`, `PCRE_MULTILINE`, `PCRE_DOTALL`, and `PCRE_EXTENDED` options (which are Perl-compatible) can be changed from within the pattern by a sequence of Perl option letters enclosed between `"(?"` and `")"`. The option letters are

```
i for PCRE_CASELESS
m for PCRE_MULTILINE
s for PCRE_DOTALL
x for PCRE_EXTENDED
```

For example, `(?im)` sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as `(?im-sx)`, which sets `PCRE_CASELESS` and `PCRE_MULTILINE` while unsetting `PCRE_DOTALL` and `PCRE_EXTENDED`, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options `PCRE_DUPNAMES`, `PCRE_UNGREEDY`, and `PCRE_EXTRA` can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

When one of these option changes occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the `pcre_fullinfo()` function).

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches `abc` and `aBc` and no other strings (assuming `PCRE_CASELESS` is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches `"ab"`, `"aB"`, `"c"`, and `"C"`, even though when matching `"C"` the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

Note: There are other PCRE-specific options that can be set by the application when the `compile` or `match` functions are called. In some cases the pattern can contain special leading sequences such as `(*CRLF)` to override what the application has set or what has been defaulted. Details are given in the section entitled "Newline sequences" above. There is also the `(*UTF8)` leading sequence that can be used to set UTF-8 mode; this is equivalent to setting the `PCRE_UTF8` option.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words `"cat"`, `"cataract"`, or `"caterpillar"`. Without the parentheses, it would match `"cataract"`, `"erpillar"` or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the `ovector` argument of `pcre_exec()`. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string `"the red king"` is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are `"red king"`, `"red"`, and `"king"`, and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string `"the white queen"` is matched against the pattern

```
the (?:red|white) (king|queen)
```


the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

DUPLICATE SUBPATTERN NUMBERS

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (?| and is itself a non-capturing subpattern. For example, consider this pattern:

```
(?|(Sat)ur|(Sun))day
```

Because the two alternatives are inside a (?| group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (?| group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing buffers that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

```
# before -----branch-reset----- after
/ ( a ) (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1           2           2 3           2   3   4
```

A back reference to a numbered subpattern uses the most recent value that is set for that number by any subpattern. The following pattern matches "abcabc" or "defdef":

```
/(?|(abc)|(def))\1/
```

In contrast, a recursive or "subroutine" call to a numbered subpattern always refers to the first one in the pattern with the given number. The following pattern matches "abcabc" or "defabc":

```
/(?|(abc)|(def))(?1)/
```

If a condition test for a subpattern's having matched refers to a non-unique number, the test is true if any of the subpatterns of that number have matched.

An alternative approach to using this "branch reset" feature is to use duplicate named subpatterns, as described in the next section.

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax. Perl allows identically numbered subpatterns to have different names, but PCRE does not.

In PCRE, a subpattern can be named in one of three ways: `(?<name>...)` or `(?'name'...)` as in Perl, or `(?P<name>...)` as in Python. References to capturing parentheses from other parts of the pattern, such as back references, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. There is also a convenience function for extracting a captured substring by name.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the `PCRE_DUPNAMES` option at compile time. (Duplicate names are also always permitted for subpatterns with the same number, set up as described in the previous section.) Duplicate names can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a "branch reset" subpattern, as described in the previous section.)

The convenience function for extracting the data by name returns the substring for the first (and in this example, the only) subpattern of that name that matched. This saves searching to find which numbered subpattern it was.

If you make a back reference to a non-unique named subpattern from elsewhere in the pattern, the one that corresponds to the first occurrence of the name is used. In the absence of duplicate numbers (see the previous section) this is the one with the lowest number. If you use a named reference in a condition test (see the section about conditions below), either to check whether a subpattern has matched, or to check for recursion, all subpatterns with the same name are tested. If the condition is true for any one of them, the overall condition is true.

This is the same behaviour as testing by number. For further details of the interfaces for handling named subpatterns, see the `pcreapi` documentation.

Warning: You cannot use different names to distinguish between two subpatterns with the same number because PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are given to subpatterns with the same number. However, you can give the same name to subpatterns with the same number, even when `PCRE_DUPNAMES` is not set.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the `\C` escape sequence
- the `\X` escape sequence (in UTF-8 mode with Unicode properties)
- the `\R` escape sequence
- an escape such as `\d` that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)
- a recursive or "subroutine" call to a subpattern

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, `\x{100}{2}` matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, when Unicode property support is available, `\X{3}` matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier `{0}` is permitted, causing the expression to behave as if

the previous item and the quantifier were not present. This may be useful for subpatterns that are referenced as subroutines from elsewhere in the pattern. Items other than subpatterns that have a {0} quantifier are omitted from the compiled pattern.

For convenience, the three most common quantifiers have single-character abbreviations:

```
*   is equivalent to {0,}
+   is equivalent to {1,}
?   is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*..*\*/
```

to the string

```
/* first comment */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.??\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones

can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` or `{0,}` and the `PCRE_DOTALL` option (equivalent to Perl's `/s`) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by `\A`.

In cases where it is known that the subject string contains no newlines, it is worth setting `PCRE_DOTALL` in order to obtain this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When `.*` is inside capturing parentheses that are the subject of a back reference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d{+}` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

Note that a possessive quantifier can be used with an entire group, for example:

```
(abc|xyz){2,3}+
```

Possessive quantifiers are always greedy; the setting of the `PCRE_UNGREEDY` option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun's Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence `A+B` is treated as `A++B` because there is no point in backtracking into a sequence of A's when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled "Non-printing characters" above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. These examples are all identical:

```
(ring), \1
```

```
(ring), \g1
(ring), \g{1}
```

An unsigned number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence `\g{-1}` is a reference to the most recently started capturing subpattern before `\g`, that is, is it equivalent to `\2`. Similarly, `\g{-2}` would be equivalent to `\1`. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax `\k{name}` and the Perl syntax `\k<name>` or `\k'name'` are supported, as is the Python syntax `(?P=name)`. Perl 5.10's unified back reference syntax, in which `\g` can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

```
(?<p1>(?(i)rah)\s+\k<p1>
?'p1'(?(i)rah)\s+\k{p1}
(?P<p1>(?(i)rah)\s+(?P=p1)
(?<p1>(?(i)rah)\s+\g{p1}
```

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail by default. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". However, if the PCRE_JAVASCRIPT_COMPAT option is set at compile time, a back reference to an unset value matches an empty string.

Because there may be many capturing parentheses in a pattern, all digits following a backslash are taken as part of a potential back refer-

ence number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise, the `\g{` syntax or an empty comment (see "Comments" below) can be used.

Recursive back references

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Back references of this type cause the group that they reference to be treated as an atomic group. Once the whole group has been matched, a subsequent matching failure cannot cause backtracking into the middle of the group.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `\A`, `\G`, `\Z`, `\z`, `^` and `$` are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with `(?=` for positive assertions and `(?!` for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!), because an empty string always matches, so an assertion that requires there not to be an empty string must always fail. The Perl 5.10 backtracking control verb (*FAIL) or (*F) is essentially a synonym for (?!).

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<!= for negative assertions. For example,

```
(?<!=foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!=dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (5.8 and 5.10), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable to PCRE if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

In some cases, the Perl 5.10 escape sequence \K (see above) can be used instead of a lookbehind assertion to get round the fixed-length restriction.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the \C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes,

which can match different numbers of bytes, are also not permitted.

"Subroutine" calls (see below) such as (?2) or (?&X) are permitted in lookbehinds, as long as the subpattern matches a fixed-length string. Recursion, however, is not supported.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching of fixed-length strings at the end of subject strings. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^.*+(?<=abcd)
```

there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abc-foo". A pattern to do that is

```
(?<=\d{3}...)(?!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a specific capturing subpattern has already been matched. The two possible forms of conditional subpattern are:

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if a capturing subpattern of that number has previously matched. If there is more than one capturing subpattern with the same number (see the earlier section about duplicate subpattern numbers), the condition is true if any of them have been set. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by `(?-1)`, the next most recent by `(-2)`, and so on. In looping constructs it can also make sense to refer to subsequent groups with constructs such as `(+2)`.

Consider the following pattern, which contains non-significant white space to make it more readable (assume the `PCRE_EXTENDED` option) and to divide it into three parts for ease of discussion:

```
( \ ( ) ?    [ ^ ( ) ] +    ( ? ( 1 ) \ ) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

```
...other stuff... ( \ ( ) ?    [ ^ ( ) ] +    ( ? ( - 1 ) \ ) ) ...
```

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax `(?(<name>)...)` or `(?('name')...)` to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax `(?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.`

Rewriting the above example to use a named subpattern gives this:

```
(?(<OPEN> \ ( )?    [^ ( )]+    (?(<OPEN> \ ) )
```

If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them has matched.

Checking for pattern recursion

If the condition is the string `(R)`, and there is no subpattern with the name `R`, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter `R`, for example:

```
(?(R3)... ) or (?(R&name)...) )
```

the condition is true if the most recent recursion is into a subpattern whose number or name is given. This condition does not check the entire recursion stack. If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them is the most recent recursion.

At "top level", all these recursion test conditions are false. The syntax for recursive patterns is described below.

Defining subpatterns for use by reference only

If the condition is the string `(DEFINE)`, and there is no subpattern with the name `DEFINE`, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of `DEFINE` is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) )
\b (?&byte) (\.(?&byte)){3} \b
```

The first part of the pattern is a `DEFINE` group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because `DEFINE` acts like a false

condition. The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
 \d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence `(?#` marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the `PCRE_EXTENDED` option is set, an unescaped `#` character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr{\( (?> (?>^\() )+ ) | (?p{$re}) ) * \)}x;
```

The `(?p{...})` item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was subsequently introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

This PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
\( ( [^() ]++ | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis. Note the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \(( [^() ]++ | (?1) )* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references (a Perl 5.10 feature). Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always "subroutine" calls, as described in the next section.

An alternative approach is to use named parentheses instead. The Perl syntax for this is (?&name); PCRE's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

```
(?<pn> \(( [^() ]++ | (?&pn) )* \) )
```

If there is more than one subpattern with the same name, the earliest one is used.

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of a possessive quantifier for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa())
```

it yields "no match" quickly. However, if a possessive quantifier is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject,

and all have to be tested before failure can be reported.

At the end of a match, the values of capturing parentheses are those from the outermost level. If you want to obtain intermediate values, a callout function can be used (see below and the `pcrecallout` documentation). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the inner capturing parentheses (numbered 2) is "ef", which is the last value taken on at the top level. If a capturing subpattern is not matched at the top level, its final value is unset, even if it is (temporarily) set at a deeper level.

If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using `pcre_malloc`, freeing it via `pcre_free` afterwards. If no memory can be obtained, the match fails with the `PCRE_ERROR_NOMEMORY` error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?:(?R) \d++ | [^<>]*+) | (?R) * >
```

In this pattern, `(?R)` is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The `(?R)` item is the actual recursive call.

Recursion difference from Perl

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. This can be illustrated by the following pattern, which purports to match a palindromic string that contains an odd number of characters (for example, "a", "aba", "abcba", "abcdcba"):

```
^(.|.) (?1)\2$
```

The idea is that it either matches a single character, or two identical characters surrounding a sub-palindrome. In Perl, this pattern works; in PCRE it does not if the pattern is longer than three characters. Consider the subject string "abcba":

At the top level, the first character is matched, but as it is not at the end of the string, the first alternative fails; the second alternative is taken and the recursion kicks in. The recursive call to subpattern 1 successfully matches the next character ("b"). (Note that the beginning and end of line tests are not part of the recursion).

Back at the top level, the next character ("c") is compared with what subpattern 2 matched, which was "a". This fails. Because the recursion is treated as an atomic group, there are now no backtracking points, and so the entire match fails. (Perl is able, at this point, to re-enter the recursion and try the second alternative.) However, if the pattern is written with the alternatives in the other order, things are

different:

```
^((.) (?1)\2|.)$
```

This time, the recursing alternative is tried first, and continues to recurse until it runs out of characters, at which point the recursion fails. But this time we do have another alternative to try at the higher level. That is the big difference: in the previous case the remaining alternative is at a deeper recursion level, which PCRE cannot use.

To change the pattern so that matches all palindromic strings, not just those with an odd number of characters, it is tempting to change the pattern to this:

```
^((.) (?1)\2|.?)$
```

Again, this works in Perl, but not in PCRE, and for the same reason. When a deeper recursion has matched a single character, it cannot be entered again in order to match an empty string. The solution is to separate the two cases, and write out the odd and even cases as alternatives at the higher level:

```
^(?:((.) (?1)\2)|((.) (?3)\4|.))
```

If you want to match typical palindromic phrases, the pattern has to ignore all non-word characters, which can be done like this:

```
^\W*+(?:((.) \W*+(?1) \W*+\2)|((.) \W*+(?3) \W*+\4| \W*+.\W*+)) \W*+$
```

If run with the PCRE_CASELESS option, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works well in both PCRE and Perl. Note the use of the possessive quantifier `++` to avoid backtracking into sequences of non-word characters. Without this, PCRE takes a great deal longer (ten times or more) to match typical phrases, and Perl takes so long that you think it has gone into a loop.

WARNING: The palindrome-matching patterns above work only if the subject string does not start with a palindrome that is shorter than the entire string. For example, although "abcba" is correctly matched, if the subject is "ababa", PCRE finds the palindrome "aba" at the start, then fails at top level because the end of the string does not follow. Once again, it cannot jump back into the recursion to try other alternatives, so the entire match fails.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

```
(...(absolute)...)...(?2)...\n(...(relative)...)...(?-1)...\n(...(?+1)...(relative)...
```

An earlier example pointed out that the pattern

```
(sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?l)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a subroutine call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. Any capturing parentheses that are set during the subroutine call revert to their previous values afterwards.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

```
(abc)(?i:(? -1))
```

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

ONIGURUMA SUBROUTINE SYNTAX

For compatibility with Oniguruma, the non-Perl syntax `\g` followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a subroutine, possibly recursively. Here are two of the examples used above, rewritten using this syntax:

```
(?<pn> \ ( ( (?[^( )]+) | \g<pn> )* \ ) )
(sens|respons)e and \g'1'ibility
```

PCRE supports an extension to Oniguruma: if a number is preceded by a plus or a minus sign it is taken as a relative reference. For example:

```
(abc)(?i:\g<-1>)
```

Note that `\g{...}` (Perl syntax) and `\g<...>` (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

CALLOUTS

Perl has a feature whereby using the sequence `(?{...})` causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable

`pcre_callout`. By default, this variable contains `NULL`, which disables all calling out.

Within a regular expression, `(?C)` indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter `C`. The default value is zero. For example, this pattern has two callout points:

```
(?C1)abc(?C2)def
```

If the `PCRE_AUTO_CALLOUT` flag is passed to `pcre_compile()`, callouts are automatically installed before each item in the pattern. They are all numbered 255.

During matching, when PCRE reaches a callout point (and `pcre_callout` is set), the external function is called. It is provided with the number of the callout, the position in the pattern, and, optionally, one item of data originally supplied by the caller of `pcre_exec()`. The callout function may cause matching to proceed, to backtrack, or to fail altogether. A complete description of the interface to the callout function is given in the `precallout` documentation.

BACKTRACKING CONTROL

Perl 5.10 introduced a number of "Special Backtracking Control Verbs", which are described in the Perl documentation as "experimental and subject to change or removal in a future version of Perl". It goes on to say: "Their usage in production code should be noted to avoid problems during upgrades." The same remarks apply to the PCRE features described in this section.

Since these verbs are specifically related to backtracking, most of them can be used only when the pattern is to be matched using `pcre_exec()`, which uses a backtracking algorithm. With the exception of `(*FAIL)`, which behaves like a failing negative assertion, they cause an error if encountered by `pcre_dfa_exec()`.

If any of these verbs are used in an assertion or subroutine subpattern (including recursive subpatterns), their effect is confined to that subpattern; it does not extend to the surrounding pattern. Note that such subpatterns are processed as anchored at the point where they are tested.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. In Perl, they are generally of the form `(*VERB:ARG)` but PCRE does not support the use of arguments, so its general form is just `(*VERB)`. Any number of these verbs may occur in a pattern. There are two kinds:

Verbs that act immediately

The following verbs act as soon as they are encountered:

```
(*ACCEPT)
```

This verb causes the match to end successfully, skipping the remainder of the pattern. When inside a recursion, only the innermost pattern is ended immediately. If `(*ACCEPT)` is inside capturing parentheses, the

data so far is captured. (This feature was added to PCRE at release 8.00.) For example:

```
A(?:A|B(*ACCEPT)|C)D
```

This matches "AB", "AAD", or "ACD"; when it matches "AB", "B" is captured by the outer parentheses.

```
(*FAIL) or (*F)
```

This verb causes the match to fail, forcing backtracking to occur. It is equivalent to (?!), but easier to read. The Perl documentation notes that it is probably useful only when combined with (??) or (??{}). Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

```
a+(?C) (*FAIL)
```

A match with the string "aaaa" always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

Verbs that act after backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is no subsequent match, a failure is forced. The verbs differ in exactly what kind of failure occurs.

```
(*COMMIT)
```

This verb causes the whole match to fail outright if the rest of the pattern does not match. Even if the pattern is unanchored, no further attempts to find a match by advancing the starting point take place. Once (*COMMIT) has been passed, `pcre_exec()` is committed to finding a match at the current starting point, or not at all. For example:

```
a+(*COMMIT)b
```

This matches "xxaab" but not "aacaab". It can be thought of as a kind of dynamic anchor, or "I've started, so I must finish."

```
(*PRUNE)
```

This verb causes the match to fail at the current position if the rest of the pattern does not match. If the pattern is unanchored, the normal "bumpalong" advance to the next starting character then happens. Backtracking can occur as usual to the left of (*PRUNE), or when matching to the right of (*PRUNE), but if there is no match to the right, backtracking cannot cross (*PRUNE). In simple cases, the use of (*PRUNE) is just an alternative to an atomic group or possessive quantifier, but there are some uses of (*PRUNE) that cannot be expressed in any other way.

```
(*SKIP)
```

This verb is like (*PRUNE), except that if the pattern is unanchored, the "bumpalong" advance is not to the next character, but to the position in the subject where (*SKIP) was encountered. (*SKIP) signifies that whatever text was matched leading up to it cannot be part of a

successful match. Consider:

```
a+(*SKIP)b
```

If the subject is "aaaac...", after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at "c". Note that a possessive quantifier does not have the same effect as this example; although it would suppress backtracking during the first match attempt, the second attempt would start at the second character instead of skipping on to "c".

```
(*THEN)
```

This verb causes a skip to the next alternation if the rest of the pattern does not match. That is, it cancels pending backtracking, but only within the current alternation. Its name comes from the observation that it can be used for a pattern-based if-then-else block:

```
( COND1 (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ ) ...
```

If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds); on failure the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If (*THEN) is used outside of any alternation, it acts exactly like (*PRUNE).

SEE ALSO

```
pcreapi(3), pcrecallout(3), pcrematching(3), pcresyntax(3), pcre(3).
```

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 11 January 2010
Copyright (c) 1997-2010 University of Cambridge.

Symbolic Index

$+$	See add, conjugate, plus	$=$	See equal
$-$	See minus, negate, subtract	\neq	See not equal
\times	See multiply, signum, times	\equiv	See depth, match
\div	See divide, reciprocal	\ncong	See not match, tally
\boxdiv	See matrix divide, matrix inverse	\sim	See excluding, not, without
$ $	See magnitude, residue	\wedge	See and, caret pointer
\lceil	See ceiling, maximum	\vee	See or
\lfloor	See floor, minimum	$\tilde{\wedge}$	See nand
$*$	See exponential, power	$\tilde{\vee}$	See nor
\otimes	See logarithm, natural logarithm	\cup	See union, unique
$<$	See less	\cap	See intersection
$>$	See greater	\subset	See enclose, partition, partitioned enclose
\leq	See less or equal	\supset	See disclose, mix, pick
\geq	See greater or equal	$?$	See deal, roll
		$!$	See binomial, factorial
		Δ	See grade up
		∇	See grade down
		\pm	See execute
		\mp	See format
		\perp	See decode
		\top	See encode
		\dashv	See same, left
		\vdash	See same, right
		\circ	See circular, pi times

Φ	See transpose	\backslash	See expand, scan
ϕ	See reverse, rotate	∇	See expand first, scan first
Θ	See reverse first, rotate first	$\ddot{\cdot}$	See each
\cdot	See catenate, laminare, ravel	$\ddot{\sim}$	See commute
$\bar{\cdot}$	See catenate first, table	$\&$	See spawn
τ	See index generator, index of	\ast	See power operator
ρ	See reshape, shape	\boxplus	See variant
ϵ	See enlist, membership, type	\boxminus	See key
$\underline{\epsilon}$	See find	\mathbf{i}	See i-beam
\uparrow	See disclose, mix, take	θ	See zilde
\downarrow	See drop, split	$-$	See negative sign
\leftarrow	See assignment	$-$	See underbar character
\rightarrow	See abort, branch	Δ	See delta character
\cdot	See name separator, decimal point, inner product	$\underline{\Delta}$	See delta- underbar character
\circ	See outer product	' '	See quotes
$\ddot{\circ}$	See rank	\square	See index, axis
\circ	See compose	$[]$	See indexing, axis
\cdot	See compress, replicate, reduce	$()$	See parentheses
∇	See replicate first, reduce first	$\{ \}$	See braces
		α	See left argument
		$\alpha\alpha$	See left operand
		ω	See right argument
		$\omega\omega$	See right operand
		$\#$	See Root object

##	See parent object	:EndNamespace	See endnamespace
◇	See statement separator	:EndProperty	See endproperty statement
Ⓐ	See comment symbol	:EndRepeat	See end-repeat control
▽	See function self, del editor	:EndSelect	See end-select control
▽▽	See operator self	:EndTrap	See end-trap control
;	See name separator, array separator	:EndWhile	See end-while control
:	See label colon	:EndWith	See end-with control
:AndIf	See and if condition	:Field	See field statement
:Access	See access statement	:For	See for statement
:Case	See case qualifier	:GoTo	See go-to branch
:CaseList	See caselist qualifier	:Hold	See hold statement
:Class	See class statement	:Include	See include statement
:Continue	See continue branch	:If	See if statement
:Else	See else qualifier	:Implements	See implements statement
:ElseIf	See else-if condition	:In	See in control
:End	See general end control	:InEach	See ineach control
:EndClass	See endclass statement	:Interface	See interface statement
:EndFor	See end-for control	:Leave	See leave branch
:EndHold	See end-hold control	:Namespace	See namespace statement
:EndIf	See end-if control	:OrIf	See or-if condition

:Property	See property statement	□CLASS	See class
:Repeat	See repeat statement	□CLEAR	See clear workspace
:Return	See return branch	□CMD	See execute Windows command, start AP
:Section	See section statement	□CR	See canonical representation
:Select	See select statement	□CS	See change space
:Trap	See trap statement	□CT	See comparison tolerance
:Until	See until condition	□CY	See copy workspace
:While	See while statement	□D	See digits
:With	See with statement	□DCT	See decimal comparison tolerance
□	See quote-quad, character I/O	□DF	See display form
□	See quad, evaluated I/O	□DIV	See division method
□Á	See underscored alphabet	□DL	See delay
□A	See alphabet	□DM	See diagnostic message
□AI	See account information	□DQ	See dequeue events
□AN	See account name	□DR	See data representation
□ARBIN	See arbitrary input	□ED	See edit object
□ARBOUT	See arbitrary output	□EM	See event message
□AT	See attributes	□EN	See event number
□AV	See atomic vector	□EX	See expunge object
□AVU	See atomic vector - unicode	□EXCEPTION	See exception
□BASE	See base class		

□EXPORT	See export object	□FRESIZE	See file resize
□FAPPEND	See file append component	□FSIZE	See file size
□FAVAIL	See file available	□FSTAC	See file set access matrix
□FCHK	See file check and repair	□FSTIE	See file share tie
□FCOPY	See file copy	□FTIE	See file tie
□FCREATE	See file create	□FUNTIE	See file untie
□FDROP	See file drop component	□FX	See fix definition
□FERASE	See file erase	□INSTANCES	See instances
□FHOLD	See file hold	□IO	See index origin
□FHIST	See file history	□KL	See key label
□FIX	See fix script	□LC	See line counter
□FLIB	See file library	□LOAD	See load workspace
□FMT	See format	□LOCK	See lock definition
□FNAMES	See file names	□LX	See latent expression
□FNUMS	See file numbers	□MAP	See map file
□FPROPS	See file properties	□ML	See migration level
□FR	See floating-point representation	□MONITOR	See monitor
□FRDAC	See file read access matrix	□NA	See name association
□FRDCI	See file read component information	□NAPPEND	See native file append
□FREAD	See file read component	□NC	See name class
□FRENAME	See file rename	□NCREATE	See native file create
□FREPLACE	See file replace component	□NERASE	See native file erase
		□NEW	See new instance
		□NL	See name list
		□NLOCK	See native file

	lock		precision
<code>□NNAMES</code>	See native file names	<code>□PROFILE</code>	See profile application
<code>□NNUMS</code>	See native file numbers	<code>□PW</code>	See print width
<code>□NQ</code>	See enqueue event	<code>□R</code>	See replace
<code>□NR</code>	See nested representation	<code>□REFS</code>	See cross references
<code>□NREAD</code>	See native file read	<code>□RL</code>	See random link
<code>□NRENAME</code>	See native file rename	<code>□RSI</code>	See space indicator
<code>□NREPLACE</code>	See native file replace	<code>□RTL</code>	See response time limit
<code>□NRESIZE</code>	See native file resize	<code>□S</code>	See search
<code>□NS</code>	See namespace	<code>□SAVE</code>	See save workspace
<code>□NSI</code>	See namespace indicator	<code>□SD</code>	See screen dimensions
<code>□NSIZE</code>	See native file size	<code>□SE</code>	See session namespace
<code>□NTIE</code>	See native file tie	<code>□SH</code>	See execute shell command, start AP
<code>□NULL</code>	See null item	<code>□SHADOW</code>	See shadow name
<code>□NUNTIE</code>	See native file untie	<code>□SI</code>	See state indicator
<code>□NXLATE</code>	See native file translate	<code>□SIGNAL</code>	See signal event
<code>□OFF</code>	See sign off APL	<code>□SIZE</code>	See size of object
<code>□OPT</code>	See variant	<code>□SM</code>	See screen map
<code>□OR</code>	See object representation	<code>□SR</code>	See screen read
<code>□PATH</code>	See search path	<code>□SRC</code>	See source
<code>□PFKEY</code>	See program function key	<code>□STACK</code>	See state indicator stack
<code>□PP</code>	See print		

<code>□STATE</code>	See state of object	<code>□TSYNC</code>	See threads synchronise
<code>□STOP</code>	See stop control	<code>□UCS</code>	See unicode convert
<code>□SVC</code>	See shared variable control	<code>□USING</code>	See using path
<code>□SVO</code>	See shared variable offer	<code>□VFI</code>	See verify and fix input
<code>□SVQ</code>	See shared variable query	<code>□VR</code>	See vector representation
<code>□SVR</code>	See shared variable retract	<code>□WA</code>	See workspace available
<code>□SVS</code>	See shared variable state	<code>□WC</code>	See window create object
<code>□TC</code>	See terminal control	<code>□WG</code>	See window get property
<code>□TCNUMS</code>	See thread child numbers	<code>□WN</code>	See window child names
<code>□TGET</code>	See get tokens	<code>□WS</code>	See window set property
<code>□THIS</code>	See this space	<code>□WSID</code>	See workspace identification
<code>□TID</code>	See thread identity	<code>□WX</code>	See window expose names
<code>□TKILL</code>	See thread kill	<code>□XML</code>	See xml convert
<code>□TNAME</code>	See thread name	<code>□XSI</code>	See extended state indicator
<code>□TNUMS</code>	See thread numbers	<code>□XT</code>	See external variable
<code>□TPOOL</code>	See token pool	<code>)CLASSES</code>	See list classes
<code>□TPUT</code>	See put tokens	<code>)CLEAR</code>	See clear workspace
<code>□TRACE</code>	See trace control	<code>)CMD</code>	See command
<code>□TRAP</code>	See trap event	<code>)CONTINUE</code>	See continue off
<code>□TREQ</code>	See token requests	<code>)COPY</code>	See copy workspace
<code>□TS</code>	See time stamp	<code>)CS</code>	See change

	space		See state indicator name
)DROP	See drop workspace)SINL	
)ED	See edit object)TID	See thread identity
)ERASE	See erase object)VARS	See list variables
)EVENTS	See list events)WSID	See workspace identity
)FNS	See list functions)XLOAD	See quiet-load workspace
)HOLDS	See held tokens		
)LIB	See workspace library		
)LOAD	See load workspace		
)METHODS	See list methods		
)NS	See namespace		
)OBJECTS	See list objects		
)OBS	See list objects		
)OFF	See sign off APL		
)OPS	See list operators		
)PCOPY	See protected copy		
)PROPS	See list properties		
)RESET	See reset state indicator		
)SAVE	See save workspace		
)SH	See shell command		
)SI	See state indicator		

Index

A

- abort function 10
- absolute value 80
- access codes 297-301, 304
- Account Information 218
- Account Name 218
- add arithmetic function 11
- alphabetic characters 217
- ancestors 384
- and boolean function 12
- APL
 - characters 224
- appending components to files 266
- appending to native file 345
- arbitrary output 219
- array separator 16, 73
- arrays
 - dimensions of 111
 - indexing 73
 - prototypes of 8
 - rank of 111
 - unit 4
- assignment 13
 - indexed 16
 - indexed modified 125
 - modified by functions 124
 - re-assignment 15
 - selective 21
 - selective modified 126
 - simple 13
- atomic vector 224
- atomic vector - unicode 224, 242, 313, 345, 372, 379, 505
- attributes of operations 220
- auto_pw parameter 395
- auxiliary processors 235
- axis operator 9
 - with dyadic operands 128
 - with monadic operands 127

- axis specification 9, 123

B

- base class 227
- best fit approximation 83
- beta function 23
- binomial function 23
- Boolean functions
 - and (conjunction) 12
 - nand 91
 - nor 92
 - not 93
 - not-equal (exclusive disjunction) 93
 - or (inclusive disjunction) 95
- bracket indexing 73
- branch function 24
- byte order mark 399

C

- callback functions 257, 367
- canonical representation of operations 236
- caret pointer 249
- catenate function 26
- ceiling function 28
- change user 191
- changing namespaces 238, 506
- character input/output 213
- checksum 289, 291
- child names 476
- child threads 450
- choose indexed assignment 18
- choose indexing 75
- circular functions 29
- class (system function) 228
- classes
 - base class 227
 - casting 229
 - class system function 228
 - copying 505
 - display form 245
 - external interfaces 356
 - fields 347
 - fix script 277
 - instances 306
 - list classes 501

- name-class 355-356
- new instance 358
- properties 348
- source 436
- this space 452
- classic edition 155, 379, 407
- Classic Edition 57, 61, 224, 259, 308, 312, 370, 379, 449
- classification of names 346
- clear state indicator 516, 520
- clearing workspaces 230, 501
- close all windows 187
- CMD_POSTFIX parameter 503, 518
- CMD_PREFIX parameter 503, 518
- command operating system 502
- command processor 231, 502
- commute operator 131
- comparison tolerance 241
- complex numbers
 - circular functions 29
 - floating-point representation 295
- component files
 - checksum 289, 291
 - compression 292
 - file properties 289
 - journaling 290
 - unicode 289
- composition operator
 - form I 132
 - form II 133
 - form III 134
 - form IV 134
- compress operation 104
- compress/decompress vector of short integers 165
- compression 289, 292
- Compute Time 218
- conformability of arguments 8
- conjunction 12
- Connect Time 218
- continue off 503
- copying component files 270
- copying from other workspaces 242, 504
- CPU time 316
- creating component files 271
- creating GUI objects 472
- creating namespaces 373, 513
- creating native files 357

- cross references 396
- current thread identity 453
- currying 122
- cutback error trap 458

D

- data binding 180
- data representation
 - dyadic 259
 - monadic 258
- deal random function 30
- decimal comparison tolerance 244
- default property 67
- delay times 249
- denormal numbers 387
- deprecated features
 - atomic vector 224
 - terminal control 449
 - underscored alphabet 217
- dequeuing events 255
- derived functions 121
- diagnostic messages 249
- digits 0 to 9 244
- dimensions of arrays 111
- direction function 34
- disclose function 35
- disjunction 95
- display form 245
- displaying held tokens 510
- divide arithmetic function 36
- division methods 248
- dmx 250, 425
- DOMAIN ERROR 412
- DotAll option 407
- drop function 37
 - with axes 38
- dropping components from files 273
- dropping workspaces 506
- dyadic primitive functions
 - add 11
 - and 12
 - catenate 26
 - deal 30
 - divide 36
 - drop 37
 - encode 41

- execute 46
 - expand 47
 - expand-first 48
 - find 49
 - format 55
 - grade down 58
 - grade up 62
 - greater 63
 - greater or equal 64
 - greatest common divisor 95
 - index function 65
 - index of 70
 - intersection 77
 - left 78
 - less 79
 - less or equal 79
 - logarithm 80
 - match 81
 - matrix divide 82
 - maximum 85
 - member of 85
 - minimum 85
 - nand 91
 - nor 92
 - not equal 93
 - not match 94
 - or . 95
 - partition 96
 - partitioned enclose 98
 - pick 99
 - power 100
 - replicate 104
 - reshape 106
 - residue 106
 - right 107
 - rotate 109
 - subtract 112
 - take 114
 - transpose 117
 - unique 119
 - dyadic primitive operators
 - axis 127-128
 - compose 132-134
 - currying 122
 - each 136
 - inner product 138
 - key 139
 - outer product 143
 - rank 146
 - replace 397
 - search 397
 - variant 155, 379, 397, 405
 - dyadic scalar functions 4
 - dynamic data exchange 444
 - dynamic link libraries 317
- ## E
- each operator
 - with dyadic operands 136
 - with monadic operands 135
 - editing APL objects 260, 507
 - editor 260
 - empty vectors 119
 - Enc option 411
 - enclose function 39
 - with axes 40
 - encode function 41
 - enlist function 43
 - enqueueing an event 366
 - EOL option 407
 - equal relational function 44
 - erasing component files 274
 - erasing native files 357
 - erasing objects from workspaces 263, 508
 - error trapping system variable 458
 - evaluated input/output 215
 - event messages 261
 - event numbers 261
 - exception 262
 - excluding set function 45
 - exclusively tying files 304
 - execute error trap 458
 - execute operation
 - dyadic 46
 - monadic 46
 - executing commands
 - UNIX 421, 518
 - Windows 231, 502
 - exit code 379
 - exiting APL system 379, 514
 - expand-first operation 48
 - expand operation 47
 - with axis 47
 - exponential function 48

- exporting objects 265
- expose root properties 188
- exposing properties 479
- expunge objects 263
- extended diagnostic message 250, 425
- extended state indicator 495
- external arrays 496
- external functions 235
- external interfaces 356
- external variables
 - query 498
 - set 496

F

- factorial function 48
- fields 347
- file
 - append component 266
 - available 266
 - check and repair 267
 - copy 270
 - create 271
 - drop component 273
 - erase 274
 - history 274
 - hold 276
 - library 278
 - names 287
 - numbers 288
 - read access matrix 295
 - read component 297
 - read component information 296
 - rename 298
 - replace component 299
 - resize 300
 - set access matrix 301
 - share-tie 302
 - size 301
 - tie (number) 304
 - untie 305
- file history 274
- file properties 289
- file system availability 266
- files
 - APL component files 270-271
 - mapped 311

- operating system native files 357
- fill elements 8
- find function 49
- first function 50
- fix script 277
- fixing operation definitions 305
- floating-point representation 244, 293
 - complex numbers 295
- floor function 50
- flush session caption 186
- fork new task 190
- format function
 - dyadic 55
 - monadic 51
- format specification 280
- format system function
 - affixtures 282
 - digit selectors 284
 - G-format 284
 - O-format qualifier 285
 - qualifiers 281
 - text insertion 280
- formatting system function
 - dyadic 280
 - monadic 279
- function assignment 14
- function keys 386
- functions
 - mixed rank 5
 - pervasive 2
 - primitive 2
 - rank zero 2
 - scalar rank 2

G

- gamma function 48
- generating random numbers 416
- get tokens 450
- getting properties of GUI objects 475
- grade-down function
 - dyadic 58
 - monadic 57
- grade-up function
 - dyadic 62
 - monadic 60
- greater-or-equal function 64

greater-than relational function 63
greatest common divisor 95
Greedy option 409
GUI objects 255

H

held tokens 510
holding component files 276

I

i-beam 137, 159
 change user 191
 close all windows 187
 compress/decompress vector of short
 integers 165
 expose root properties 188
 flush session caption 186
 fork new task 190
 inverted table index of 161
 memory manager statistics 170
 number of threads 168
 parallel execution threshold 168
 read DataTable 177
 reap forked tasks 192
 serialise/deserialise arrays 167
 set workspace save options 187
 signal counts 194
 specify workspace available 173
 syntax colouring 164
 thread synchronisation mechanism 168
 unsqueezed type 163
 update DataTable 174
 update function time stamp 169
IC option 155, 405
identification of workspaces 522
identity 64
identity elements 149
identity function 30
identity matrix 84
index
 with axes 68
index-generator function 69
index-of function 70
index function 65
index of 161

index origin 307
indexed assignment 16
indexed modified assignment 125
indexing arrays 73
InEnc option 410
inner-product operator 138
instances 306, 354
interfaces 356
INTERRUPT 145
intersection set function 77
inverted table index of 161
iota 69

J

journaling 289-290

K

key labels 308
key operator 139
Keying Time 218
kill threads 453

L

labels 24
laminar function 26
latent expressions 311
least squares solution 83
left 78
legal names 472
less-or-equal function 79
less-than relational function 79
levels of migration towards APL2 1
libraries of component files 278
line number counter 308
list classes 501
list names in a class 359
listing global defined functions 509
listing global defined operators 514
listing global namespaces 514
listing global objects 514
listing global variables 522
listing GUI events 508
listing GUI methods 513

- listing GUI properties 516
- listing workspace libraries 511
- loading workspaces 309, 512
 - without latent expressions 523
- localisation 424
- lock native file 363
- locking defined operations 310
- logarithm function 80
- logical conjunction 12
- logical disjunction 95
- logical equivalence 81
- logical negation 93
- logical operations 12

M

- magnitude function 80
- major cell 71
- major cells 146
- map file 311
- markup 491
- match relational function 81
- matrix-divide function 82
- matrix-inverse function 84
- matrix product 82
- maximum function 85
- MAXWS parameter 171
- membership set function 85
- MEMCPY 334
- memory manager statistics 170
- migration levels 1, 35, 43, 86, 118, 313
- minimum function 85
- minus arithmetic function 85
- miscellaneous primitive functions 5
- mix function 86
 - with axis 86
- mixed rank functions 5
- ML option 408
- Mode option 155, 406, 411
- modified assignment 124
- monadic primitive functions
 - branch 24
 - ceiling 28
 - direction 34
 - disclose 35
 - enclose 39
 - enlist 43
 - execute 46
 - exponential 48
 - factorial 48
 - floor 50
 - format 51
 - grade down 57
 - grade up 60
 - identity 30, 64
 - index generator 69
 - magnitude 80
 - matrix inverse 84
 - mix 86
 - natural logarithm 92
 - negative 92
 - not 93
 - pi times 99
 - ravel 101
 - reciprocal 104
 - reverse 107
 - roll 108
 - same 110
 - shape 111
 - signum 34
 - split 112
 - table 113
 - tally 116
 - transpose 116
 - type 118
 - union 118
- monadic primitive operators
 - assignment 124-126
 - commute 131
 - each 135
 - reduce 149, 151
 - scan 152-153
 - spawn 154
- monadic scalar functions 3
- monitoring operation statistics
 - query 316
 - set 315
- MPUT utility 311
- multiply arithmetic function 91

N

- name association 317, 351
- name classifications 346

- name lists by classification 359
- name mangling 319
- name of thread 454
- name references in operations 396
- names
 - legal 472
- names of tied component files 287
- names of tied native files 365
- namespace indicator 375
- namespace reference 15, 238, 255, 475, 477
- namespace reference assignment 15
- namespace script 353
- namespaces
 - create 513
 - search path 384
 - this space 452
 - unnamed 373
- nand boolean function 91
- Naperian logarithm function 92
- natch 94
- native file
 - append 345
 - create 357
 - erase 357
 - lock 363
 - names 365
 - numbers 365
 - read 369
 - rename 371
 - replace 371
 - resize 372
 - size 375
 - tie (number) 376
 - translate 378
 - untie 378
- natural logarithm function 92
- negate 92
- negative function 92
- NEOL option 408
- nested representation of operations 368
- new instance 358
- next error trap 458
- niladic primitive functions
 - abort 10
 - zilde 119
- NONCE ERROR 67
- nor boolean function 92
- not-equal relational function 93

- not-match relational function 94
- not boolean function 93
- notation
 - keys 1
- nsi 375
- null 377
- number of each thread 454
- number of threads 168
- numbers
 - empty vectors 119
- numbers of tied component files 288
- numbers of tied native files 365

O

- object representation of operations 380
- OM option 409
- operands 121
- operator syntax 121
- operators
 - dyadic 121
 - monadic 121
 - syntax 121
- or boolean function 95
- OutEnc option 410
- outer-product operator 143

P

- parallel execution
 - number of threads 168
 - parallel execution threshold 168
 - thread synchronisation mechanism 168
- parallel execution threshold 168
- partition function 96
- partitioned enclose function 98
 - with axis 98
- pass-through values 124
- passnumbers of files 297
- PCRE 397
- PCRE Regular Expression Details 533
- pervasive functions 2
- pi-times function 99
- pick function 99
- plus arithmetic function 100
- power function 100
- primitive function classifications 5

- primitive functions 2
- primitive operators 121
 - axis 127-128
 - commute 131
 - compose 132-134
 - each 135-136
 - indexed modified assignment 125
 - inner product 138
 - key 139
 - modified assignment 124
 - outer product 143
 - power 144
 - rank 146
 - reduce 149
 - reduce-first 151
 - reduce n-wise 151
 - replace 397
 - scan 152
 - scan-first 153
 - search 397
 - selective modified assignment 126
 - spawn 154
 - variant 155, 379, 397
- Principal option 155-156, 405
- print precision in session 387
- print width in session 395
- product
 - inner 138
 - outer 143
- profile application 388
- profile user command 393
- programming function keys 386
- properties 348-349
 - propertyget Function 67
 - propertyset function 67
- protected copying from workspaces 515
- prototype 8, 135-136, 143
- put tokens 455

Q

- quad indexing 68
- quietly loading workspaces 523

R

- random link 416

- rank of arrays 111
- rank operator 146
- ravel function 101
 - with axes 101
- re-assignment 15
- reach indexed assignment 19
- reach indexing 76
- read DataTable 177
- reading components from files 297
- reading file access matrices 295
- reading file component information 296
- reading native files 369
- reading properties of GUI objects 475
- reading screen maps 432
- reap forked tasks 192
- reciprocal function 104
- reduce-first operator 151
- reduce operator 149
- reduction operator
 - n-wise 151
 - with axis 149
- regular expressions 397
- releasing component files 276
- renaming component files 298
- renaming native files 371
- replace operator 397
 - DotAll 407
 - Enc 411
 - EOL 407
 - Greedy 409
 - IC 155, 405
 - InEnc 410
 - ML 408
 - Mode 155, 406, 411
 - NEOL 408
 - OutEnc 410
- replacing components on files 299
- replacing data in native files 371
- replicate operation 104
 - with axis 104
- reset state indicator 516, 520
- reshape function 106
- residue function 106
- resizing component files 300
- resizing native files 372
- response time limit 419
- reverse-first function 107, 110

- reverse function 107
 - with axis 107
- right 107
- Right Parenthesis 499
- roll random function 108
- rotate function 109
 - with axis 109
- rsi 418

S

- same 110
- saving continuation workspaces 503
- saving workspaces 419, 516
- scalar extension 4
- scalar functions 2
- scan-first operator 153
- scan operator 152
 - with axis 152
- screen dimensions 420
- screen maps 429
- screen read 432
- search operator 397
 - DotAll 407
 - Enc 411
 - EOL 407
 - Greedy 409
 - IC 155, 405
 - InEnc 410
 - ML 408
 - Mode 155, 406, 411
 - NEOL 408
 - OM 409
 - OutEnc 410
- search path 384, 468
- selection primitive functions 5
- selective assignment 21
- selective modified assignment 126
- selector primitive functions 5
- serialise/deserialise arrays 167
- session namespace 420
- set difference 45
- set workspace save options 187
- setting properties of GUI objects 477
- shadowing names 424
- shape function 111
- share-tying files 302

- shared variables
 - offer couplings 444
 - query access control 443
 - query couplings 446
 - query outstanding offers 447
 - retract offers 447
 - set access control 442
 - states 448
- signal counts 194
- signal event 425, 576
- signing off APL 379, 514
- signum function 34
- simple assignment 13
- simple indexed assignment 16
- simple indexing 73
- size of objects 428
- sizes of component files 301
- sizes of native files 375
- source 436
- spawn thread operator 154
- special primitive functions 5
- specification
 - axis 9, 123
- specify workspace available 173
- split function 112
 - with axis 112
- squad indexing 65
- stack 437
- starting auxiliary processors
 - UNIX 422
 - Windows 235
- state indicator 423, 519
 - and name list 520
 - clear 516, 520
 - extension 495
 - reset 516, 520
 - stack 437
- states of objects 439
- stop control
 - query 441
 - set 440
- stop error trap 458
- STRLEN 336
- STRNCPY 335
- structural primitive functions 5
- subtract arithmetic function 112
- syntax colouring 164
- system commands 499

system constants 199
system functions 197, 203
 categorised 203
system namespaces 202
system operators 202

T

table function 113
take function 114
 with axes 115
tally 116
terminal control vector 449
this space 452
thread
 name 454
thread synchronisation mechanism 168
threads
 child numbers 450
 identity 453
 kill 453
 numbers 454, 462
 spawn 154
 synchronise 464
tie numbers 288, 365
time stamp 463
times arithmetic function 116
token pool 454
token requests 462
tokens
 get tokens 450
 put tokens 455
 time-out 450
 token pool 454
 token requests 462
tracing lines in defined operations
 query 457
 set 456
translating native files 378
TRANSLATION ERROR 226, 242, 313, 407,
505
transpose function
 dyadic 117
 monadic 116
transposition of axes 117
trapping error conditions 458
tying component files 302, 304

tying native files 376
type function 118

U

underscored alphabetic characters 217
unicode 289
unicode convert 224, 449, 465
Unicode Edition 57, 61, 224, 312-313, 369,
371, 379
union set function 118
unique set function 119
unit arrays 4
unknown-entity 494
unknownentity 494
unnamed copy 505
unsqueezed type 163
untying component files 305
untying native files 378
update DataTable 174
update function time stamp 169
User Identification 218
using 468
UTF-16 466
UTF-32 466
UTF-8 466

V

VALUE ERROR 465
variant operator 155, 379, 397, 405
vector representation of operations 469
vectors
 empty character 233
verify and fix input 470

W

waiting for threads to terminate 464
whitespace 488
wide character 325
window
 create object 472
 get property 475
 names of children 476
 set property 477

window expose names 479
without set function 119
workspace available 471
workspace identification 478, 522
writing file access matrices 301

X

xml convert 480
 markup 491
 unknown-entity 494
 unknownentity 494
 whitespace 488

Z

zilde constant 119

