

Dyalog Programming Reference Guide

Dyalog version 20.0



DYALOG

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

Dyalog Programming Reference Guide

Dyalog version: 20.0

Document Revision: 2025-10-30 main:e0843eae32

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

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

email: support@dyalog.com

<https://www.dyalog.com>

TRADEMARKS:

Raspberry Pi is a trademark of the Raspberry Pi Foundation.

Oracle®, MySQL, and Java™ are registered trademarks of Oracle and/or its affiliates.

JavaScript™ is a trademark of Oracle Corporation.

Unicode is a registered trademarks of Unicode, Inc. in the U.S. and other countries.

UNIX® is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Company Limited.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Windows® is a registered trademark of Microsoft Corporation in the U.S. and other countries.

macOS® and OS X® (operating system software) are registered trademarks of Apple Inc. in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Except where otherwise noted, this content is licensed under a [Creative Commons Attribution 4.0 International](https://creativecommons.org/licenses/by/4.0/) licence.

Contents

1 Introduction	14
1.1 Programmers Guide Introduction	14
1.1.1 Workspaces	14
1.1.2 Legal Names	15
1.1.3 Arrays	16
1.1.3.1 Introduction	16
1.1.3.2 Numbers	17
1.1.3.3 Characters	18
1.1.3.4 Enclosed Elements	19
1.1.3.5 Specification of Variables	19
1.1.3.6 Vector Notation	19
1.1.3.7 Array Notation	21
1.1.3.8 Structuring of Arrays	30
1.1.3.9 Display of Arrays	33
1.1.3.10 Prototypes and Fill Items	40
1.1.3.11 Cells and Subarrays	42
1.1.4 Expressions	44
1.1.5 Functions	45
1.1.6 Operators	47
1.1.7 Binding Strength	49
1.1.8 Function Trains	50
1.1.9 Search Functions and Hash Tables	55
1.1.10 Idiom Recognition	57
1.1.10.1 Introduction	57
1.1.10.2 Idiom List	57
1.1.11 Parallel Execution	65
1.1.12 Complex Numbers	65
1.1.13 128 Bit Decimal Floating-Point Support	68
1.1.14 Namespaces	73
1.1.14.1 Overview	73
1.1.14.2 Namespace Syntax	75

1.1.14.3 Namespace Reference Evaluation	77
1.1.14.4 Namespaces and Localisation	77
1.1.14.5 Namespace References	80
1.1.14.6 Unnamed Namespaces	82
1.1.14.7 Arrays of Namespace References	84
1.1.14.8 Distributed Assignment	86
1.1.14.9 Distributed Functions	88
1.1.14.10 Namespaces and Operators	90
1.1.14.11 Serialising Namespaces	91
1.1.15 External Variables	94
1.1.16 Component Files	95
1.1.17 Auxiliary Processors	95
2 Deprecated Functionality	96
3 Defined Functions & Operators	99
3.1 Traditional Functions and Operators	99
3.1.1 Introduction	99
3.1.2 Model Syntax	99
3.1.3 Statements	101
3.1.4 Global & Local Names	101
3.1.5 Locals Lines	104
3.1.6 Namelists	105
3.1.7 Locked Functions & Operators	106
3.1.8 Function Declaration Statements	106
3.1.8.1 Introduction	106
3.1.8.2 Access Statement	107
3.1.8.3 Attribute Statement	108
3.1.8.4 Implements Statement	108
3.1.8.5 Signature Statement	109
3.1.9 Control Structures	111
3.1.9.1 Introduction	111
3.1.9.2 If Statement	113
3.1.9.3 While Statement	115
3.1.9.4 Repeat Statement	116
3.1.9.5 For Statement	116
3.1.9.6 Select Statement	119
3.1.9.7 With Statement	120
3.1.9.8 Hold Statement	121
3.1.9.9 Trap Statement	124
3.1.9.10 GoTo Statement	127

3.1.9.11 Return Statement	127
3.1.9.12 Leave Statement :Leave	128
3.1.9.13 Continue Statement	128
3.1.9.14 Section Statement	128
3.1.9.15 Disposable Statement	128
3.1.10 APL Line Editor	131
3.2 Dfns and Dops	139
3.2.1 Introduction	139
3.2.2 MultiLine Dfns	140
3.2.3 Default Left Argument	141
3.2.4 Guards	142
3.2.5 Shy Result	143
3.2.6 Lexical Name Scope	143
3.2.7 Error Guards	144
3.2.8 Dops	146
3.2.9 Recursion	147
3.2.10 Tail Calls	152
3.2.11 Restrictions	153
4 Object Oriented Programming	155
4.1 Introducing Classes	155
4.1.1 Introducing Classes	155
4.1.2 Defining Classes	156
4.1.3 Editing Classes	156
4.1.4 Inheritance	157
4.1.5 Instances	159
4.2 Constructors	160
4.2.1 Introduction	160
4.2.2 Constructor Overloading	161
4.2.3 Niladic (Default) Constructors	163
4.2.4 Empty Arrays of Instances: Why ?	164
4.2.5 Empty Arrays of Instances: How?	165
4.2.6 Base Constructors	168
4.2.7 Niladic Example	169
4.2.8 Monadic Example	170
4.2.9 Destructors	172
4.3 Class Members	176
4.3.1 Introduction	176

4.3.2 Fields	176
4.3.2.1 Introduction	176
4.3.2.2 Public Fields	177
4.3.2.3 Initialising Fields	178
4.3.2.4 Private Fields	178
4.3.2.5 Shared Fields	179
4.3.2.6 Trigger Fields	180
4.3.3 Methods	182
4.3.3.1 Methods	182
4.3.3.2 Shared Methods	183
4.3.3.3 Instance Methods	183
4.3.3.4 Superseding Base Class Methods	184
4.3.4 Properties	186
4.3.4.1 Properties	186
4.3.4.2 Simple Instance Properties	187
4.3.4.3 Simple Shared Properties	189
4.3.4.4 Numbered Properties	190
4.3.4.4.1 Introduction	190
4.3.4.4.2 Example	191
4.3.4.5 The Default Property	193
4.3.4.6 ComponentFile Class	194
4.3.4.7 Keyed Properties	196
4.3.4.7.1 Introduction	196
4.3.4.7.2 Example 1	196
4.3.4.7.3 Example 2	199
4.4 Interfaces	202
4.4.1 Introduction	202
4.4.2 Example	202
4.5 Including Namespaces in Classes	204
4.5.1 Introduction	204
4.5.2 Example	205
4.6 Nested Classes	208
4.6.1 Nested Classes	208
4.6.2 GolfService Example	208
4.7 Namespace Scripts	211
4.7.1 Namespace Scripts	211
4.7.2 DiaryStuff Example	215

4.8 Including Script Files	217
4.9 Class Declaration Statements	217
4.9.1 Introduction	217
4.9.2 Interface Statement	217
4.9.3 Namespace Statement	218
4.9.4 Class Statement	218
4.9.5 Using Statement	219
4.9.6 Attribute Statement	220
4.9.7 Access Statement	221
4.9.8 Implements Statement	223
4.10 Field Statement	224
4.11 Property Section	225
4.11.1 Property Statement	225
4.11.2 PropertyArguments Class	227
4.11.3 PropertyGet Function Syntax	227
4.11.4 PropertySet Function Syntax	229
4.11.5 PropertyShape Function Syntax	230
5 Threads and Triggers	231
5.1 Threads	231
5.1.1 Language Elements	231
5.1.2 Thread Switching	232
5.1.3 Name Scope	233
5.1.4 Stack Considerations	233
5.1.5 Globals and the Order of Execution	234
5.1.6 Threads & Niladic Functions	236
5.1.7 Threads & External Functions	237
5.1.8 Synchronising Threads	238
5.1.9 Semaphore Example	239
5.1.10 Latch Example	240
5.1.11 Debugging	241
5.2 Triggers	244
5.2.1 Introduction	244
5.2.2 Example	245
5.2.3 Global Triggers	247
5.2.4 Trigger Fields (in Classes)	249

6 APL Files	251
6.1 Native Files	251
6.2 Component Files	257
6.2.1 Introduction	257
6.2.2 Component Files	257
6.2.3 Programming Techniques	266
6.2.4 File Design	269
6.2.5 Internal Structure	270
6.2.6 The Effect of Buffering	273
6.2.7 Integrity and Security	273
7 Error Trapping	276
7.1 Standard Error Action	276
7.2 Error Trapping	277
7.3 Error Trapping Examples	281
7.4 Signalling Events	288
7.5 Handling Unexpected Errors	290
8 Error Messages	292
8.1 Error Messages	292
8.1.1 Introduction	292
8.1.2 APL Error Messages Table	293
8.1.3 Operating System Error Messages Table	298
8.1.4 Windows Operating System Error Messages Table	301
8.1.5 APL Error Messages	301
8.1.6 APL Error Messages (A-Z)	302
8.1.6.1 bad ws	302
8.1.6.2 cannot create name	302
8.1.6.3 clear ws	302
8.1.6.4 copy incomplete	302
8.1.6.5 DEADLOCK	302
8.1.6.6 defn error	303
8.1.6.7 DOMAIN ERROR	303
8.1.6.8 EOF INTERRUPT	304
8.1.6.9 EXCEPTION	304
8.1.6.10 FIELD CONTENTS RANK ERROR	304
8.1.6.11 FIELD CONTENTS TOO MANY COLUMNS	304
8.1.6.12 FIELD POSITION ERROR	305
8.1.6.13 FIELD CONTENTS TYPE MISMATCH	305

8.1.6.14 FIELD TYPE BEHAVIOUR UNRECOGNISED	305
8.1.6.15 FIELD ATTRIBUTES RANK ERROR	305
8.1.6.16 FIELD ATTRIBUTES LENGTH ERROR	305
8.1.6.17 FULL SCREEN ERROR	305
8.1.6.18 KEY CODE UNRECOGNISED	305
8.1.6.19 KEY CODE RANK ERROR	305
8.1.6.20 KEY CODE TYPE ERROR	306
8.1.6.21 FORMAT FILE ACCESS ERROR	306
8.1.6.22 FORMAT FILE ERROR	306
8.1.6.23 FILE ACCESS ERROR	306
8.1.6.24 FILE ACCESS ERROR CONVERTING	307
8.1.6.25 FILE COMPONENT DAMAGED	307
8.1.6.26 FILE DAMAGED	307
8.1.6.27 FILE FULL	307
8.1.6.28 FILE INDEX ERROR	307
8.1.6.29 FILE NAME ERROR	308
8.1.6.30 FILE NAME QUOTA USED UP	308
8.1.6.31 FILE SYSTEM ERROR	308
8.1.6.32 FILE SYSTEM NO SPACE	309
8.1.6.33 FILE SYSTEM NOT AVAILABLE	309
8.1.6.34 FILE SYSTEM TIES USED UP	309
8.1.6.35 FILE TIE ERROR	309
8.1.6.36 FILE TIED	310
8.1.6.37 FILE TIED REMOTELY	310
8.1.6.38 FILE TIE QUOTA USED UP	310
8.1.6.39 FORMAT ERROR	311
8.1.6.40 HOLD ERROR	311
8.1.6.41 incorrect command	311
8.1.6.42 INDEX ERROR	312
8.1.6.43 INTERNAL ERROR	312
8.1.6.44 INTERRUPT	312
8.1.6.45 is name	313
8.1.6.46 LENGTH ERROR	313
8.1.6.47 LIMIT ERROR	313
8.1.6.48 NONCE ERROR	314
8.1.6.49 NO PIPES	314
8.1.6.50 name is not a ws	315
8.1.6.51 Name already exists	315
8.1.6.52 Namespace does not exist	315
8.1.6.53 not copied name	315
8.1.6.54 not found name	315
8.1.6.55 not saved this ws is name	316
8.1.6.56 PROCESSOR TABLE FULL	316

8.1.6.57 RANK ERROR	317
8.1.6.58 RESIZE	317
8.1.6.59 name saved date time	317
8.1.6.60 SYNTAX ERROR	318
8.1.6.61 sys error number	318
8.1.6.62 TIMEOUT	319
8.1.6.63 TRANSLATION ERROR	319
8.1.6.64 TRAP ERROR	319
8.1.6.65 too many names	319
8.1.6.66 VALUE ERROR	320
8.1.6.67 warning duplicate label	320
8.1.6.68 warning duplicate name	320
8.1.6.69 warning pendent operation	320
8.1.6.70 warning label name present	321
8.1.6.71 warning unmatched brackets	321
8.1.6.72 warning unmatched parentheses	322
8.1.6.73 was name	322
8.1.6.74 WS FULL	322
8.1.6.75 ws not found	323
8.1.6.76 ws too large	323
8.1.7 Operating System Error Messages	324
8.1.8 Operating System Error Messages (A-Z)	324
8.1.8.1 FILE ERROR 1 Not owner	324
8.1.8.2 FILE ERROR 2 No such file	324
8.1.8.3 FILE ERROR 5 I O error	324
8.1.8.4 FILE ERROR 6 No such device	324
8.1.8.5 FILE ERROR 13 Permission denied	324
8.1.8.6 FILE ERROR 20 Not a directory	324
8.1.8.7 FILE ERROR 21 Is a directory	325
8.1.8.8 FILE ERROR 23 File table overflow	325
8.1.8.9 FILE ERROR 24 Too many open	325
8.1.8.10 FILE ERROR 26 Text file busy	325
8.1.8.11 FILE ERROR 27 File too large	325
8.1.8.12 FILE ERROR 28 No space left	325
8.1.8.13 FILE ERROR 30 Read only file	325
8.1.9 System Errors	326
9 APLMon	333
9.1 Introduction	333
9.2 Using APLMON	333
9.3 APLMON output	334
9.4 Special syntax	338

9.5 Notes	340
-----------------	-----

1 Introduction

1.1 Programmers Guide Introduction

1.1.1 Workspaces

APL expressions are evaluated within a workspace. The workspace may contain objects, namely classes, namespaces, operators, functions and variables defined by the user. APL expressions may include references to primitive operators, functions and variables provided by APL. These objects do not reside in the workspace, but space is required for the actual process of evaluation to accommodate temporary data. During execution, APL records the state of execution through the STATE INDICATOR which is dynamically maintained until the process is complete. Space is also required to identify objects in the workspace in the SYMBOL TABLE. Maintenance of the symbol table is entirely dynamic. It grows and contracts according to the current workspace contents.

Workspaces may be explicitly saved with an identifying name. The workspace may subsequently be loaded, or objects may be selectively copied from a saved workspace into the current workspace.

Workspaces are stored in files whose names must conform to operating system conventions. When a workspace name is specified without a file suffix, these are added or implied. For further information, see *Installation/Configuration: Wsxt*.

If the name of the file in which the workspace is saved contains spaces, the ws argument for the system functions `)SAVE`, `)COPY`, `)PCOPY`, `)LOAD`, `)XLOAD` and `)DROP` should be surrounded by two double-quote (") characters. To include a " character in the file name, you must specify two adjoining double-quotes (that is, ""). Note however that Windows does not allow double-quotes in file names, so this effectively applies only to non-Windows systems.

EXAMPLES

```
)SAVE Pete's work
unacceptable char
```

The above statement fails because the presence of the space in the file name requires that it be surrounded by "s.

```

)SAVE "Pete's work"
Pete's work.dws saved Sun Jan 17 16:23:17 2016

)COPY "Pete's work" A B C
.\Pete's work.dws saved Sun Jan 17 16:23:17 2016

)DROP "Pete's work"
Sun Jan 17 16:24:16 2016

```

1.1.2 Legal Names

APL objects may be given names. A name may be any sequence of characters, starting with a non-numeric character, selected from the following:

ABCDEFGHIJKLMNOPQRSTUVWXYZ_
abcdefghijklmnopqrstuvwxyz
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝß
àáâãäåæçèéêëìíîïðñòóôõöøùúûüþ
0123456789
ΔΔ
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Note that using a standard Unicode font (rather than APL385 Unicode used in the table above), the last row above would appear as the circled alphabet, À to Ƶ.

Examples

Legal	Illegal
THISΔISΔAΔNAME	BAD NAME
X1233	3+21
SALES	S!H PRICE
pjb_1	1_pjb

1.1.3 Arrays

Arrays

A Dyalog APL data structure is called an array. An array is a rectangular arrangement of items, each of which may be a single number, a single character, a namespace reference (ref), another array, or the `⌵OR` of an object. An array which is part of another array is also known as a subarray.

An array has two properties; structure and data type. Structure is identified by rank, shape, and depth.

Rank

An array may have 0 or more axes or dimensions. The number of axes of an array is known as its rank. Dyalog APL supports arrays with a maximum of 15 axes.

- An array with 0 axes (rank 0) is called a scalar.
- An array with 1 axis (rank 1) is called a vector.
- An array with 2 axes (rank 2) is called a matrix.
- An array with more than 2 axes is called a multi-dimensional array.

Shape

Each axis of an array may contain zero or more items. The number of items along each axis of an array is called its shape. The shape of an array is itself a vector. Its first item is the length of the first axis, its second item the length of the second axis, and so on. An array, whose length along one or more axes is zero, is called an empty array.

Depth

An array whose items are all simple scalars (that is, single numbers, characters or refs) is called a simple array. If one or more items of an array is not a simple scalar (that is, is another array, or a `⌵OR`), the array is called a nested array. A nested array may contain items which are themselves nested arrays. The degree of nesting of an array is called its depth. A simple scalar has a depth of 0. A simple vector, matrix, or multi-dimensional

array has depth 1. An array whose items are all depth 1 subarrays has depth 2; one whose items are all depth 2 subarrays has depth 3, and so forth.

Type

An array, whose elements are all numeric, is called a numeric array; its TYPE is numeric. A character array is one in which all items are characters. An array whose items contain both numeric and character elements is of MIXED type.

Numbers

Dyalog APL supports both real numbers and complex numbers.

Real Numbers

Numbers are entered or displayed using conventional decimal notation (for example, 299792.458) or using a scaled form for example, 2.99792458E5).

On entry, a decimal point is optional if there is no fractional part. On output, a number with no fractional part (an integer) is displayed without a decimal point.

The scaled form consists of:

1. an integer or decimal number called the mantissa,
2. the letter E or e,
3. an integer called the scale, or exponent.

The scale specifies the power of 10 by which the mantissa is to be multiplied.

EXAMPLE

```
12 23.24 23.0 2.145E2
12 23.24 23 214.5
```

Negative numbers are preceded by the high minus (⁻) symbol, not to be confused with the minus (−) function. In scaled form, both the mantissa and the scale may be negative.

EXAMPLE

```

      22 2.145E-2 -10.25
22 0.02145 -10.25

```

Complex Numbers

Complex numbers use the J notation introduced in IBM APL2 and are written as aJb or ajb (without spaces) where the real and imaginary parts a and b are written as described above. The capital J is always used to display a value.

EXAMPLES

```

      2+-1*.5
2J1
      .3j.5
0.3J0.5
      1.2E5J-4E-4
120000J-0.0004

```

Zilde

The empty vector (ϵ) may be represented by the numeric constant 0 called ZILDE.

Characters

Characters are entered within a pair of APL quotes. The surrounding APL quotes are not displayed on output. The APL quote character itself must be entered as a pair of APL quotes.

EXAMPLES

```

      'DIALOG APL'
DIALOG APL

```

```

      'I DON'T KNOW'
I DON'T KNOW

```

```

      '*'
*

```

Enclosed Elements

An array may be enclosed to form a scalar element through any of the following means:

- by the enclose function ($\leftarrow$)
- by inclusion in vector notation
- as the result of certain functions when applied to arrays

EXAMPLES

```

      (←1 2 3),←'ABC'
1 2 3  ABC

      (1 2 3) 'ABC'
1 2 3  ABC

      1 2 3
1 1  1 2  1 3
2 1  2 2  2 3

```

Specification of Variables

A variable is a named array. An undefined name or an existing variable may be assigned an array by specification with the left arrow ($\leftarrow$).

EXAMPLES

```

      A←'CHIPS WITH EVERYTHING'
      A
CHIPS WITH EVERYTHING

      X Y←'ONE' 'TWO'
      X
ONE
      Y
TWO

```

Vector Notation

Vector notation complements [Section 1.1.3.7](#) with a clean syntax for vectors.

A series of two or more adjacent expressions results in a vector whose elements are the enclosed arrays resulting from each expression. This is known as vector (or strand) notation. Each expression in the series may consist of one of the following:

1. a single numeric value
2. the empty numeric vector symbol $\emptyset$
3. zero or more characters, within a pair of quotes
4. array notation
5. the name of a variable or system variable
6. a special namespace or dfn name like ##, ω , and so on
7. the evaluated input symbol $\square$
8. the character input symbol $\square$
9. the name of a niladic, defined function or system function yielding a result
10. any other APL expression which yields a result, within parentheses

EXAMPLES

```

3  ρA←2 4 10
   ρTEXT←'ONE' 'TWO'
2

```

Numbers and characters may be mixed:

```

2  ρX←'THE ANSWER IS ' 10
   X[1]
THE ANSWER IS
   X[2] + 32
42

```

Blanks, quotes or parentheses must separate adjacent items in vector notation. Redundant blanks and parentheses are permitted. In this manual, the symbol pair '↔' indicates the phrase 'is equivalent to'.

```

1 2 ↔ (1)(2) ↔ 1 (2) ↔ (1) 2
2 'X'3 ↔ 2 'X' 3 ↔ (2) ('X') (3)
1 (2+2) ↔ (1) ((2+2)) ↔ ((1)) (2+2)

```

Vector notation may be used to define an item in vector notation:

```

ρX ← 1 (2 3 4) ('THIS' 'AND' 'THAT')
3
  X[2]
2 3 4
  X[3]
THIS AND THAT

```

Expressions within parentheses are evaluated to produce an item in the vector:

```

Y ← (2+2) 'IS' 4
Y
4 IS 4

```

The following identity holds:

```
A B C ↔ (⊂A), (⊂B), ⊂C
```

Array Notation

Array notation extends [Section 1.1.3.6](#) to define arrays of higher rank, and namespaces, and lets these definitions span multiple lines:

- **Parentheses** embrace vector definitions and namespace name-value pairs
- **Square brackets** embrace higher-rank arrays
- **Diamonds** and **linebreaks** separate array elements and name-value pairs

Examples

Some examples of different kinds of arrays defined with array notation.

NESTED VECTOR

```

ρz← (0 6 1 8 ♦ 2*0 2 0 2 1  a four-element vector over two lines
2 7 1 8 2 8 ♦ 3 1 4 1 5)
4
z

```

0 6 1 8	1 4 1 4 2	2 7 1 8 2 8	3 1 4 1 5
---------	-----------	-------------	-----------

```

('Three'
'Blind'
'Mice')

```

Three	Blind	Mice
-------	-------	------

```

(,c'Three') ≡ ('Three' ♦)  a single-element vector
1

```

MATRIX

```

ρm←[0 6 1 8 ♦ 1 4 1 4
2 7 1 8 ♦ 3 1 4 2]
4 4
m

```

0 6 1 8
1 4 1 4
2 7 1 8
3 1 4 2

Short items are padded. (See *Dyalog APL Language: Mix* for details.)

```

pmice←[ 'Three'
        'Blind'
        'Mice' ]
3 5
mice, '|'
Three|
Blind|
Mice |

pRC←[0 'OK'
     1 'WS FULL'
     2 'SYNTAX ERROR'
     3 'INDEX ERROR'
     4 'RANK ERROR']
5 2
pexpenses←0/[      A typed template matrix
'Glasgow' 125.84
]
0 2

```

COLUMN MATRIX

```

[ 1 ♦ 2 ♦ 3 ]
1
2
3
p"cm3←[ ('Three' ♦)
        ('Blind' ♦)
        ('Mice' ♦)]
5
5
4
cm3
|Three|
|Blind|
|Mice|

```

RANK-3 ARRAY

```

ρblock←[
  [3 1 4 ♦ 1 5 ]
  [2 7 0 ♦ 2]
]
2 2 3
  block
3 1 4
1 5 0

2 7 0
2 0 0

```

Namespaces

Array notation allows you to write a namespace literal as zero or more name-value pairs, spanned by parentheses.

```

    ()                                A empty namespace
#. [Namespace]
    ()()                             A vector of empty namespaces
#. [Namespace] #. [Namespace]
    ( () ♦ () )                     A vector of empty namespaces
#. [Namespace] #. [Namespace]

    n←(x:'hello')
    n.x
hello

    m←(x:['hello'
          'world'])
    p≠m.x                            A matrix
2

    (y:(x:'hello'))                 A nested namespaces
#. [Namespace]

    (
      FirstName:'Wolfgang'
      LastName:'Mozart'
      Age:35
    )
#. [Namespace]

```

Scoping In Namespace Literals

Array and namespace literals can include value expressions.

```
LUE←(answer:7×6)    A Life, the Universe, and Everything
```

Any expressions are evaluated in the scope around the namespace.

```

    long←'bobby'
    short←'jack'
    ns←(short:'jill' ♦ inner:short∘.=short←3↑long)
    ns.inner
1 0 1
0 1 0
1 0 1
    short      A altered by inner assignment
bob
    ns.short   A unaffected by inner assignment
jill
    (y:(x:'hello')).y  A inner namespace's parent is NOT outer
namespace
#. [Namespace]

```

Specification

The new syntactic forms were previously errors in every mainstream APL implementation and therefore introduce no backward incompatibilities.

In the following:

- A *name-value pair* is an APL name followed by a colon and a value expression.
- A *separator* is a diamond or line break, and *separated* means separated by them.
- An *empty* value expression or name-value pair is two separators with nothing but white space between them.

Namespace

A namespace is defined by a parenthesised, separated list of zero or more name-value pairs.

Empty name-value pairs define no namespace members.

Vector

A vector is defined by a parenthesised, separated list of two or more value expressions.

Empty value expressions define no vector elements.

At least one value expression must be non-empty.

Matrices And Higher-rank Arrays

An array of rank 2 or higher is defined by a bracketed, separated list of two or more value expressions, which constitute the major cells of the array.

Short elements are padded to fill, and scalars are treated as length-1 vectors.

At least one value expression must be non-empty.

Information

Separators in a list of value expressions or name-value pairs make an enclosing parenthesis or bracket *broken*.

Separators encapsulated in a dfn or further contained in array notation do not break a parenthesis or bracket. For example, in

```
( {1=ω: 'y' ♦ 'n'}?2 )
```

the diamond is part of the dfn and does not break the surrounding parenthesis.

Unsupported

The following are not supported by array notation:

- scripted and external objects
- non-array namespace members
- reference loops
- class instances

- internal representations returned by `⌊OR`

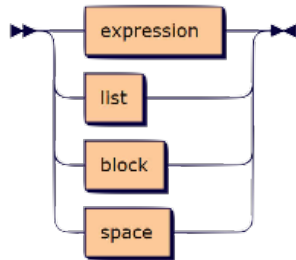
While one can include the last three items when writing in array notation, they cannot be displayed in the notation.

Formal Syntax

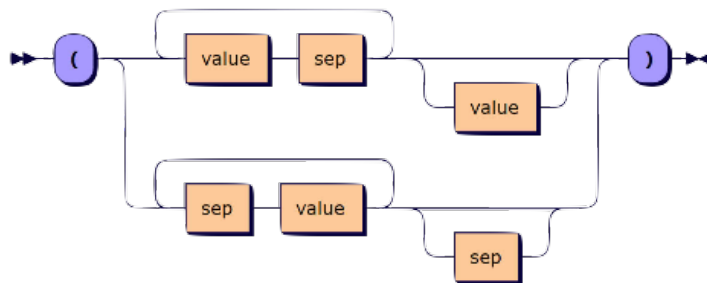
The array notation can be described in this form, where `expression` is any traditional APL expression:

```
value ::= expression | list | block | space
list  ::= '(' ( ( value sep )+ value? | ( sep value )+ sep? ) ')'
block ::= '[' ( ( value sep )+ value? | ( sep value )+ sep? ) ']'
space ::= '(' sep? ( name ':' value ( sep name ':' value )* )? sep? ')'
sep    ::= [⋄#x000A#x000D#x0085]+
```

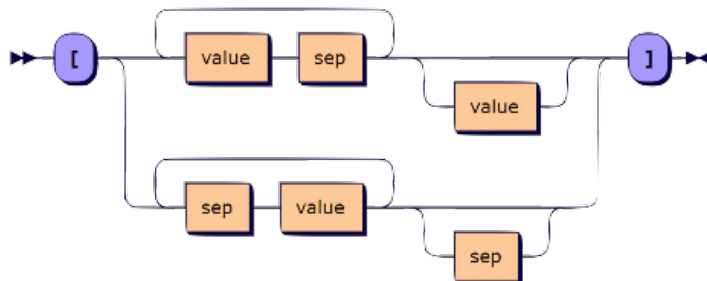
value:



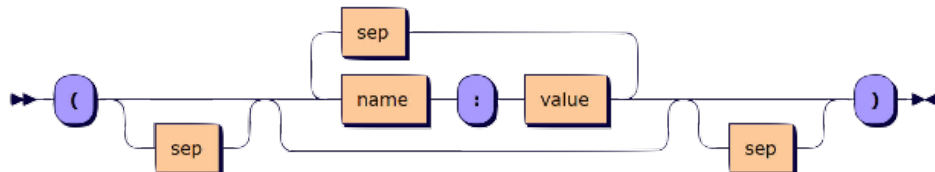
list:



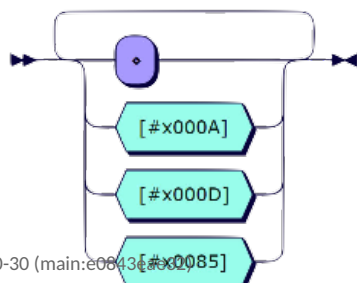
block:



space:



sep:



Structuring of Arrays

Primitive functions that restructure arrays:

symbol	monadic	dyadic
ρ		<i>Dyalog APL Language: Reshape</i>
$\cdot$	<i>Dyalog APL Language: Ravel</i>	<i>Dyalog APL Language: Catenate Laminate</i>
$\overline{\cdot}$	<i>Dyalog APL Language: Table</i>	<i>Dyalog APL Language: Catenate First</i>
ϕ°	<i>Dyalog APL Language: Reverse</i>	<i>Dyalog APL Language: Rotate</i>
Φ	<i>Dyalog APL Language: Transpose</i>	
$\uparrow$	<i>Dyalog APL Language: Mix</i>	<i>Dyalog APL Language: Take</i>
$\downarrow$	<i>Dyalog APL Language: Split</i>	<i>Dyalog APL Language: Drop</i>
ϵ	<i>Dyalog APL Language: Enlist</i>	
$\subset$	<i>Dyalog APL Language: Enclose</i>	<i>Dyalog APL Language: Partitioned Enclose</i>
$\subseteq$	<i>Dyalog APL Language: Nest</i>	<i>Dyalog APL Language: Partition</i>

EXAMPLES

```

      m←2 2p1 2 3 4
1 2
3 4

      2 2 4p'ABCDEFGHJKLMNOP'
ABCD
EFGH

IJKL
MNOP

      ,m
1 2 3 4
      1 2 3,4
1 2 3 4
      1ϕ1 2 3 4
2 3 4 1
      ϕm
2 1
4 3

      em
3 4
1 2
      ϕm
1 3
2 4
      †(1 2 3)(4)
1 2 3
4 0 0
      †2 4p'COWSHENS'

|COWS|HENS|

      [ 1 2 3 ♦ (4 5) 6 (7 8 9)]

|1|2|3|
|4 5|6|7 8 9|

      ∈[ 1 2 3 ♦ (4 5) 6 (7 8 9)]
1 2 3 4 5 6 7 8 9
      ≠□←1 2 3 4

|1 2 3 4|

```

`A reshape`

`A ravel`

`A catenate`

`A rotate`

`A reverse`

`A reverse first`

`A transpose`

`A mix`

`A split`

`A enlist`

`A enclose`

```

1
    2 0 1 3 0 2 0 1<'abcdefg'      A partitioned enclose

|ab|c||de|fg|
|---|
1 1 2 2 2≤1 2 3 4 5              A partition

|1 2|3 4 5|
|---|
    
```

Display of Arrays

Simple scalars and vectors are displayed in a single line beginning at the left margin. A number is separated from the next adjacent element by a single space.

```

    3
3
    1 4
1 2 3 4
    
```

Print Precision

The *Dyalog APL Language*: *Pp* system variable determines the number of significant digits to be printed. The fractional part of the number is rounded in the last digit if it cannot be represented within the print precision. Leading zeros, and trailing zeros after a decimal point, are omitted, as are decimal points for integers.

```

    0.1 1.0 1.12
0.1 1 1.12

    'A' 2 'B' 'C'
A 2 BC

    ÷3 2 6
0.3333333333 0.5 0.1666666667
    
```

If a number cannot be fully represented in □PP significant digits, or if the number requires more than five leading zeros after the decimal point, it is represented in scaled form. The mantissa displays up to □PP significant digits, omitting trailing zeros.

```

□PP←3

      123 1234 12345 0.12345 0.00012345 0.00000012345
123 1.23E3 1.23E4 0.123 0.000123 1.23E-7

```

Simple Matrices

Simple matrices are displayed in rectangular form, with one line for each matrix row.

```

      2 4p 'HANDFIST'
HAND
FIST

      1 2 3 0. x 6 2 5
6 2 5
12 4 10
18 6 15

```

All elements in a column are displayed in the same format, but the format and width for the column is determined independently of other columns.

A column is treated as numeric if it contains any numeric elements. Numeric columns are right-justified; others are left-justified. Numeric columns are separated from their neighbours by a single column of blanks.

A numeric column aligns any decimal points or E characters (for scaled formats), adding trailing zeros to the mantissae if necessary. Integers are right-adjusted one place to the left of any decimal points.

```

      2 3p2 4 6.1 8 10.24 12
2 4      6.1
8 10.24 12

      2 4p4 'A' 'B' 5 -0.000000003 'C' 'D' 123.56
4E0 AB 5
-3E-9 CD 123.56

```

Non-simple Arrays

In the display of non-simple arrays, each element is displayed within a rectangle such that the rows and columns of the array are aligned. Simple items within the array are displayed as above. For non-simple items, this rule is applied recursively, with one space added on each side of the enclosed element for each level of nesting.

```

      13
1 2 3

      c13
1 2 3

      cc13
1 2 3

      ('ONE' 1) ('TWO' 2) ('THREE' 3) ('FOUR' 4)
ONE  1  TWO  2  THREE  3  FOUR  4

      ['ONE' 1 'TWO' 2 ♦ 'THREE' 3 'FOUR' 4]
ONE   1  TWO   2
THREE 3  FOUR  4

```

Multi-dimensional Arrays

Multi-dimensional arrays are displayed in rectangular planes. Planes are separated by one blank line, and hyper-planes of higher dimensions are separated by increasing numbers of blank lines. In all other respects, multi-dimensional arrays are displayed in the same manner as matrices.

```

      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

      3 1 1 3p'THEREDFOX'
THE

RED

FOX
    
```

The power of this form of display is made apparent when formatting informal reports.

```

AREAS←'West' 'Central' 'East'
PRODUCTS←'Biscuits' 'Cakes' 'Buns' 'Rolls'

SALES←[
  50 5.25 75
  250 20.15 900
  500 80.98 650
  1000 90.03 1200
]

' ' PRODUCTS , AREAS SALES
      West Central East
Biscuits    50    5.25  75
Cakes       250   20.15 900
Buns        500   80.98 650
Rolls       1000   90.03 1200
    
```

Array Notation

Arrays (including namespaces) can be displayed in the session using [Section 1.1.3.7](#). This mode is enabled using the `JAPLAN.Output` user command. For example:

```

      SALES
    50  5.25  75
   250 20.15 900
   500 80.98 650
  1000 90.03 1200
      ]APLAN.Output ON
'Was OFF'
      SALES
[
    50  5.25  75
   250 20.15 900
   500 80.98 650
  1000 90.03 1200
]

```

In the *User Interface* it can also be toggled on and off using the



icon on the Session toolbar.

To enable the display of output using array notation when starting a Dyalog session, set the *Installation/Configuration: Aplan For Output* configuration parameter to 1.

Print Width

If the display of an array is wider than the print width, as defined by the *Dyalog APL Language: Pw* system variable, it will be folded at or before □PW and the folded portions indented six spaces. The display of a simple numeric or mixed array may be folded at a width less than □PW so that individual numbers are not split across a page boundary.

```

    PW←40

    ?3 20p100
54 22  5 68 68 94 39 52 84  4  6 53 68
85 53 10 66 42 71 92 77 27  5 74 33 64
66  8 64 89 28 44 77 48 24 28 36 17 49
    1  39  7 42 69 49 94
    76 100 37 25 99 73 76
    90 91  7 91 51 52 32
  
```

The]Disp User Command

The user command]Disp illustrates the specified array, with borders indicating sub-array shape and type. For example:

```

    ]Disp 'ABC' [1 2 3 4 ♦]

```

ABC	1	2	3	4
-----	---	---	---	---

```

    ]Disp (' ',PRODUCTS),AREAS,SALES p see above

```

	West	Central	East
Biscuits	50	5.25	75
Cakes	250	20.15	900
Buns	500	80.98	650
Rolls	1000	90.03	1200

This is similar to displaying array with]Boxing on -style=mid (see [The \]Boxing User Command](#) below).

An explanation of the symbols that appear in the borders can be seen by running]Disp -??

The]Display User Command

The user command]Display illustrates the specified array, with borders indicating array and sub-array shape and type. For example:

```
]Display 'ABC' [1 2 3 4 ♦]
```

```
]Display (' ',PRODUCTS),AREAS,SALES A see above
```

	West	Central	East
Biscuits	50	5.25	75
Cakes	250	20.15	900
Buns	500	80.98	650
Rolls	1000	90.03	1200

This is similar to displaying array with]Boxing on -style=max (see [The \]Boxing User Command](#) below).

An explanation of the symbols that appear in the borders can be seen by running]Display -??

The]Boxing User Command

The user command]Boxing changes how nested arrays are displayed in the Session. The following examples show different settings.

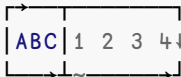
```
]Boxing on -style=min
Was OFF -style=min
```

```
'ABC' [1 2 3 4 ◊]
```

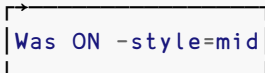


```
]Boxing on -style=mid
Was ON -style=min
```

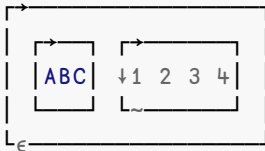
```
'ABC' [1 2 3 4 ◊]
```



```
]Boxing on -style=max
```



```
'ABC' [1 2 3 4 ◊]
```



```
]Boxing on -style=min
Was ON -style=max
]Boxing off
Was ON
```

```
'ABC' [1 2 3 4 ◊]
```

```
ABC 1 2 3 4
```

Information about all the options and explanation of the symbols that appear in the borders can be seen by running `]Display -??`

Prototypes and Fill Items

Every array has an associated *prototype* which is derived from the array's first item.

If the first item is a number, the prototype is 0. Otherwise, if the first item is a character, the prototype is ' ' (space). Otherwise, if the first item is a (ref to) an instance of a Class, the prototype is a ref to that Class.

Otherwise (in the nested case, when the first item is other than a simple scalar), the prototype is defined recursively as the prototype of each of the array's first item.

EXAMPLES

Array	Prototype
1 2 3.4	0
2 3 5p'hello'	' '
99 'b' 66	0
(1 2)(3 4 5)	0 0
((1 2)3)(4 5 6)	(0 0)0
'hello' 'world'	' '
□NEW MyClass	MyClass
(88(□NEW MyClass)'X')7	0 MyClass ' '

Fill Items

Fill items for an overtake operation, are derived from the argument's prototype. For each 0 or ' ' in the prototype, there is a corresponding 0 or ' ' in the fill item and for each class reference in the prototype, there is a ref to a (newly constructed and distinct) instance of that class that is initialised by the niladic (default) constructor for that class, if defined.

EXAMPLES

```

      4↑1 2
1 2 0 0
      4↑'ab'
ab
      4↑(1 2)(3 4 5)
1 2 3 4 5 0 0 0 0
      2↑□NEW MyClass
#. [Instance of MyClass] #. [Instance of MyClass]
```

In the last example, two distinct instances are constructed (the first by `NEW` and the second by the `overtake`).

Fill items are used in a number of operations including:

- First (`>` or `↑`) of an empty array
- Fill-elements for `overtake`
- For use with the `Each` operator on an empty array

Cells and Sub-arrays

Certain functions and operators operate on particular cells or sub-arrays of an array, which are identified and described as follows.

K-Cells

A *rank-k cell* or *k-cell* of an array are terms used to describe a sub-array on the last k axes of the array. Negative k is interpreted as $r+k$ where r is the rank of the array, and is used to describe a sub-array on the leading $|k|$ axes of an array.

If x is a 3-dimensional array of shape `2 3 4`, the 1-cells are its 6 rows each of 4 elements; and its 2-cells are its 2 matrices each of shape `3 4`. Its 3-cells is the array in its entirety. Its 0-cells are its individual elements.

Major Cells

The *major cells* of an array x is a term used to describe the sub-arrays on the leading dimension of the array x with shape `1 ↑ ρ x`. Using the k -cell terminology, the major cells are its -1 -cells.

The major cells of a vector are its elements (0-cells). The major cells of a matrix are its rows (1-cells), and the major cells of a 3-dimensional array are its matrices along the first dimension (2-cells).

EXAMPLES

In the following, the major cells of A are 1979, 1990, 1997, 2007, and 2010; those of B are 'Thatcher', 'Major', 'Blair', 'Brown', and 'Cameron'; and those of C are the four 2-by-3 matrices.

A
1979 1990 1997 2007 2010

B
Thatcher
Major
Blair
Brown
Cameron

ρB
5 8

$\square \leftarrow C \leftarrow 4 \quad 2 \quad 3 \rho 124$
0 1 2
3 4 5

6 7 8
9 10 11

12 13 14
15 16 17

18 19 20
21 22 23

Using the k-cell terminology, if r is the rank of the array, its major cells are its $r-1$ -cells.

Note that if the right operand k of the Rank Operator $\circ$ is negative, it is interpreted as $0[r+k]$. Therefore the value -1 selects the major cells of the array.

1.1.4 Expressions

An expression is a sequence of one or more syntactic tokens which may be symbols or constants or names representing arrays (variables) or functions. An expression which produces an array is called an ARRAY EXPRESSION. An expression which produces a function is called a FUNCTION EXPRESSION. Some expressions do not produce a result.

An expression may be enclosed within parentheses.

Evaluation of an expression proceeds from right to left, unless modified by parentheses. If an entire expression results in an array that is not assigned to a name, then that array value is displayed. (Some system functions and defined functions return an array result only if the result is assigned to a name or if the result is the argument of a function or operator.)

EXAMPLES

	$X \leftarrow 2 \times 3 - 1$
	$2 \times 3 - 1$
4	
	$(2 \times 3) - 1$
5	

Either blanks or parentheses are required to separate constants, the names of variables, and the names of defined functions which are adjacent. Excessive blanks or sets of parentheses are redundant, but permitted. If F is a function, then:

$F \ 2 \leftrightarrow F(2) \leftrightarrow (F)2 \leftrightarrow (F)(2) \leftrightarrow F \ (2) \leftrightarrow F \ ((2))$

Blanks or parentheses are not needed to separate primitive functions from names or constants, but they are permitted:

$-2 \leftrightarrow (-)(2) \leftrightarrow (-) \ 2$

Blanks or parentheses are not needed to separate operators from primitive functions, names or constants. They are permitted with the single exception that a dyadic operator must have its right argument available when encountered. The following syntactical forms are accepted:

$(+ \ . \ \times) \leftrightarrow (+) \ . \ \times \leftrightarrow + \ . \ (\times)$

The use of parentheses in the following examples is not accepted:

$+(.) \times$ or $(+.) \times$

1.1.5 Functions

A function is an operation which is performed on zero, one or two array arguments and may produce an array result. Three forms are permitted:

- NILADIC defined for no arguments
- MONADIC defined for a right but not a left argument
- DYADIC defined for a left and a right argument

The number of arguments is referred to as its VALENCE.

The name of a non-niladic function is AMBIVALENT; that is, it potentially represents both a monadic and a dyadic function, though it might not be defined for both. The usage in an expression is determined by syntactical context. If the usage is not defined an error results.

Functions have long SCOPE on the right; that is, the right argument of the function is the result of the entire expression to its right which must be an array. A dyadic function has short scope on the left; that is, the left argument of the function is the array immediately to its left. Left scope may be extended by enclosing an expression in parentheses whence the result must be an array.

For some functions, the explicit result is suppressed if it would otherwise be displayed on completion of evaluation of the expression. This applies on assignment to a variable name. It applies for certain system functions, and may also apply for defined functions.

EXAMPLES

	$10 \times 5 - 2 \times 4$
$^{-}30$	
	2×4
8	
	$5 - 8$
$^{-}3$	
	$10 \times ^{-}3$
$^{-}30$	
	$(10 \times 5) - 2 \times 4$
42	

Defined Functions

Functions may be defined with the system function `⌈FX`, or with the function editor. A function consists of a HEADER which identifies the syntax of the function, and a BODY in which one or more APL statements are specified.

The header syntax identifies the function name, its (optional) result and its (optional) arguments. If a function is ambivalent, it is defined with two arguments but with the left argument within braces (`{}`). If an ambivalent function is called monadically, the left argument has no value inside the function. If the explicit result is to be suppressed for display purposes, the result is shown within braces. A function need not produce an explicit result. Refer to *Chapter 2* for further details.

EXAMPLE

```

    ▽ R←{A} FOO B
[1]  R←'MONADIC' 'DYADIC' [⌈IO+0≠⌈NC'A']
[2]  ▽

      FOO 1
MONADIC

      'X' FOO 'Y'
DYADIC

```

Functions may also be created by using assignment (`←`).

Function Assignment & Display

The result of a function-expression may be given a name. This is known as FUNCTION ASSIGNMENT (see also [Section 3.2.1](#)). If the result of a function-expression is not given a name, its value is displayed. This is termed FUNCTION DISPLAY.

EXAMPLES

```

    PLUS←+
    PLUS
+
    SUM←+ /
    SUM
+ /

```

Function expressions may include defined functions and operators. These are displayed as a ▽ followed by their name.

EXAMPLE

```

    ▽ R←MEAN X      A Arithmetic mean
[1]    R←(+ /X)÷pX
    ▽

    MEAN
    ▽MEAN
    AVERAGE←MEAN
    AVERAGE
    ▽MEAN
    AVG←MEAN ° ,
    AVG
    ▽MEAN ° ,

```

1.1.6 Operators

An operator is an operation on one or two operands which produces a function called a DERIVED FUNCTION. An operand may be a function or an array. Operators are not ambivalent. They require either one or two operands as applicable to the particular operator. However, the derived function may be ambivalent. The derived function need not return a result. Operators have higher precedence than functions. Operators have long scope on the left. That is, the left operand is the longest function or array expression on its left. The left operand may be terminated by:

1. the end of the expression
2. a function with a function to its left
3. a function with an array to its left
4. an array with a function to its left
5. an array or function to the right of a monadic operator.
6. A dyadic operator has short scope on the right. That is, the right operand of an operator is the single function or array on its right. Right scope may be extended by enclosing an expression in parentheses.

EXAMPLES

```

      ρ∘X←'WILLIAM' 'MARY' 'BELLE'
7  4  5

      ρ∘ρ∘X
1  1  1

      (ρ∘ρ)∘X
1  1  1

```

```

      ⍺←⍺VR∘'PLUS' 'MINUS'
▽ R←A PLUS B
[1] R←A+B
▽
▽ R←A MINUS B
[1] R←A-B
▽

      PLUS/1 2 3 4
10

```

Defined Operators

Operators may be defined with the system function `⍺FX`, or with the function editor. A defined operator consists of a **HEADER** which identifies the syntax of the operator, and a **BODY** in which one or more APL statements are specified.

A defined operator may have one or two operands; and its derived function may have one or two arguments, and may or may not produce a result. The header syntax defines the operator name, its operand(s), the argument(s) to its derived function, and the result (if any) of its derived function. The names of the operator and its operand(s) are separated from the name(s) of the argument(s) to its derived function by parentheses.

EXAMPLE

```

      ▽ R←A(F AND G)B
[1] R←(A F B)(A G B)
      ▽

```

The above example shows a dyadic operator called **AND** with two operands (**F** and **G**). The operator produces a derived function which takes two arguments (**A** and **B**), and produces a result (**R**).

```
12 +AND÷ 4
16 3
```

Operands passed to an operator may be either functions or arrays.

```
12 (3 AND 5) 4
12 3 4 12 5 4

12 (× AND 5) 4
48 12 5 4
```

1.1.7 Binding Strength

For two entities x and y that are adjacent in an expression (that is, $x\ y$), the binding strength between them and the result of the bind is shown in this table:

		Y						
		A	F	H	MOP	DOP	DOT	IDX
X	A	6 A	3 AF	3 AF	4 F		7 REF	4 A
	F	2 A	1 F	4 F	4 F			4 F
	H		1 F	4 F	4 F			4 H
	AF	2 A	1 F					
	MOP			4 ERR				
	DOP	5 MOP	5 MOP	5 MOP				
	JOT	5 MOP	5 MOP	5 MOP	4 F			
	DOT	6 ERR	5 MOP	5 MOP		6 ERR		
	REF	7 A	7 F	7 H	7 MOP	7 DOP		
	IDX	3 ERR	3 ERR	3 ERR				

where

- **A:** Array¹, for example, 0 1 2 'hello' $\alpha\ \omega$
- **F:** Function¹{primitive | defined | derived | system}, for example, + - +.× myfn
□CR { $\alpha\ \omega$ }
- **H:** Hybrid¹ function/operator, that is, / ≠ \ ≠

¹ Indicates a "first-class" entity, which can be parenthesised or named.

- **AF**: Bound left argument, for example, 2+
- **MOP**: Monadic operator¹, for example, " ~ &
- **DOP**: Dyadic operator, for example, * ̢ ÷ ̢
- **JOT**: Jot, that is, compose/null operand °
- **DOT**: Dot, that is, reference/product .
- **IDX**: square-bracketed expression, for example, [α+ιω]
- **ERR**: Error

In this table:

- the higher the number, the stronger the binding
- an empty field indicates no binding for this combination; an error.

For example, in the expression `a b . c [ d ]`, where `a`, `b`, `c` and `d` are arrays, the binding proceeds:

```

a b . c [ d ]
 6 7 6 4      A binding strengths between entities
→ a ( b . ) c [ d ]
   0   7 4
→ a ( b . c ) [ d ]
   6     4
→ ( a ( b . c ) ) [ d ]
```

1.1.8 Function Trains

Introduction

A *Train* is a derived function constructed from a sequence of 2 or 3 functions, or from an array followed by two functions, which bind together to form a function.

Note that the right-most item of a function train (which is by definition a function) must be isolated from anything to its right, otherwise it will be bound to that rather than to the items to its left. This is done using parentheses.

For example, the following expression comprises a function train `- , ÷` that is separated from its argument 2 by parentheses:

```

( - , ÷ ) 2
-2 0.5
```

and means:

1. Calculate the reciprocal of 2
2. Calculate the negation of 2
3. Catenate these 2 results together

Whereas, without the parentheses to identify the function train, the expression means (as it did before):

1. Calculate the reciprocal of 2
2. Ravel the result of step 1
3. Negate the result of step 2

```

- , ÷ 2
-0.5

```

Forks and Atops

The following trains are currently supported where f , g and h are functions and A is an array:

```

f g h
A g h
g h

```

The 3-item trains $(f\ g\ h)$ and $(A\ g\ h)$ are termed *forks* while the 2-item train $(g\ h)$ is termed an *atop*. To distinguish the two styles of *fork*, we can use the terms *fgh-fork* or *Agh-fork*.

Trains as Functions

A train is syntactically equivalent to a function and so, in common with any other function, may be:

- named using assignment
- applied to or between arguments
- consumed by operators as an operand
- and so forth.

In particular, trains may be applied to a single array (monadic use) or between 2 arrays (dyadic use), providing six new constructs.

```

α(f g h)ω ↔ (α f ω) g (α h ω)  A dyadic (fgh) fork
α(A g h)ω ↔      A      g (α h ω)  A dyadic (Agh) fork
α(  g h)ω ↔           g (α h ω)  A dyadic      atop

(f g h)ω ↔ (  f ω) g (  h ω)  A monadic (fgh) fork
(A g h)ω ↔      A      g (  h ω)  A monadic (Agh) fork
(  g h)ω ↔           g (  h ω)  A monadic      atop

```

Identifying a Train

For a sequence to be interpreted as a train it must be separated from the argument to which it is applied. This can be done using parentheses or by naming the derived function.

Example - fork: negation of catenated with reciprocal

```

( -, ÷ ) 5
~5 0.2

```

Example - named fork

```

negrec ← -, ÷
negrec 5
~5 0.2

```

Whereas, without these means to identify the sequence as a train, the expression:

```

-, ÷ 5
~0.2

```

means the negation of the ravel of the reciprocal of 5.

Idiom Recognition

Function trains lend themselves to idiom recognition, a technique used to optimise the performance of certain expressions.

EXAMPLE

An expression to find the first position in a random integer vector *x* of a number greater than 999000 is:

```
X←?1e6p1e6
(X≥999000)⌈1
1704
```

A function train is not only more concise, it is faster too.

```
X (⌈1 ≥) 999000
1704
```

Trains of Trains

As a train resolves to a function, a sequences of more than 3 functions represents a train of trains. Function sequences longer than 3 are bound in threes, starting from the right:

```
... fu fv fw fx fy fz → ... fu (fv fw (fx fy fz))
```

This means that, in the absence of parentheses, a sequence of an odd number of functions resolves to a 3-train (fork) and an even-numbered sequence resolves to a 2-train (atop):

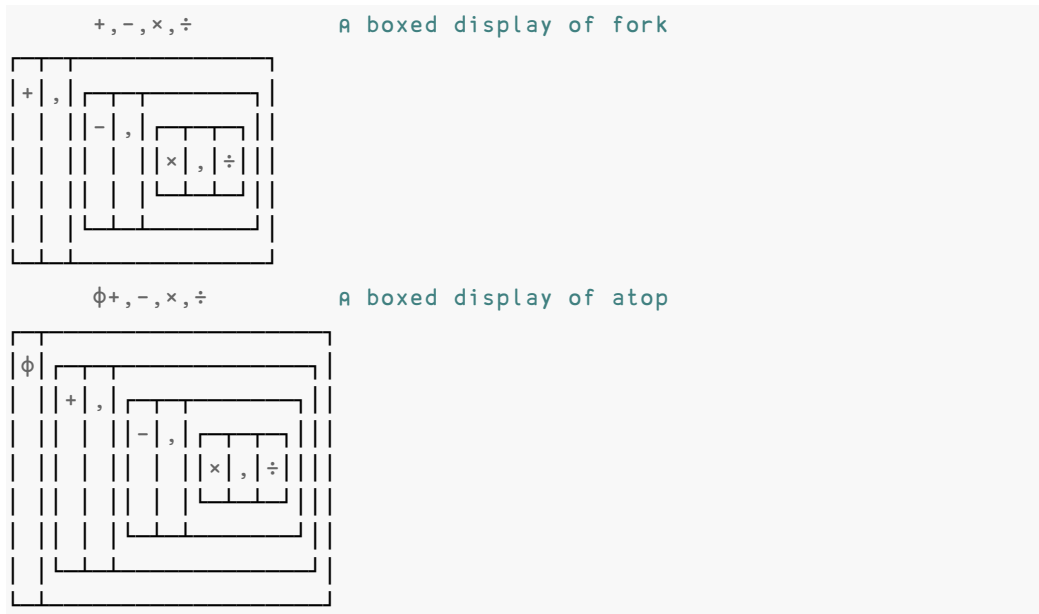
```
e f g h i j k → e f(g h(i j k))      A fork(fork(fork))
f g h i j k →  f(g h(i j k))        A atop(fork(fork))
```

EXAMPLES

```
6( +,-,×,÷)2      A fork:(6+2),((6-2),((6×2),(6÷2)))
8 4 12 3
```

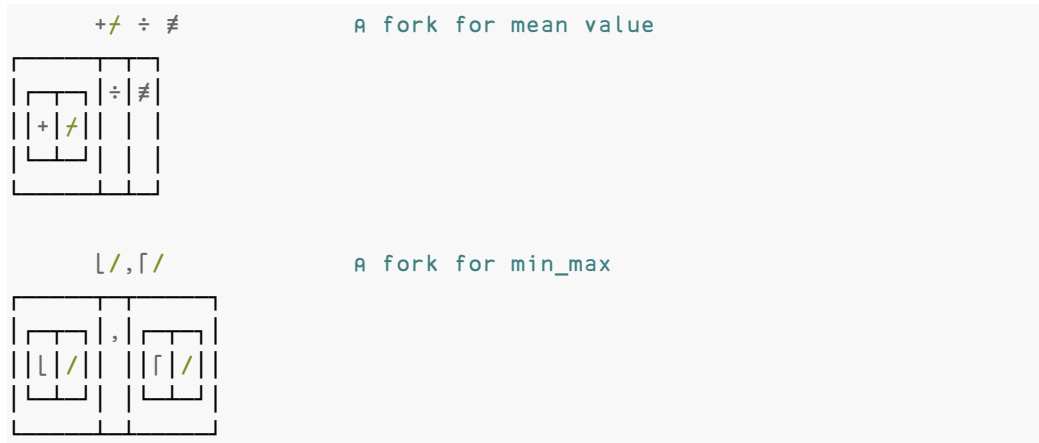
```
6(ϕ+,-,×,÷)2      A atop: ϕ (6+2), ...
3 12 4 8
```

```
]boxing on
Was OFF
```



Binding Strengths

The binding strength between the items of a train is less than that of operand-operator binding. In other words, operators bind first with their function (or array) operands to form derived functions, which may then participate as items in a train.

EXAMPLE

This means that any of the four hybrid tokens $/ \ \neq \ \backslash \ \nwarrow$ will not be interpreted as a function if there's a function to its left in the train. In order to fix one of these tokens as a replicate or expand function, it must be isolated from the function to its left:

$(\iota/\iota)3$	$A \rightarrow \iota/ \text{ atop } \iota3 \rightarrow \text{RANK ERROR}$
RANK ERROR	
$(\iota\{\alpha/\omega\}\iota)3$	$A \rightarrow (\iota3)\{\alpha/\omega\}(\iota3) \rightarrow (\iota3)/(\iota3)$
1 2 2 3 3 3	
$(\iota(/ \circ \vdash)\iota)3$	$A \rightarrow (\iota3)/\vdash(\iota3)$
1 2 2 3 3 3	
$(2/\iota)3$	A Agh-fork is OK
1 1 2 2 3 3	

1.1.9 Search Functions and Hash Tables

Primitive dyadic *search* functions, such as ι (index of) and ϵ (membership) have a *principal* argument in which items of the other *subject* argument are located.

In the case of ι , the principal argument is the one on the left and in the case of ϵ , it is the one on the right. The following table shows the principal (P) and subject (s) arguments for each of the functions.

$P \uparrow s$	Index of
$s \in P$	Membership
$s \cap P$	Intersection
$P \cup s$	Union
$s \sim P$	Without
$P \{ (\downarrow \alpha) \uparrow \downarrow \omega \} s$	Matrix Iota (idiom)
$P \circ \Phi$ and $P \circ \Psi$	Grade

The Dyalog APL implementation of these functions already uses a technique known as *hashing* to improve performance over a simple linear search. (Note that ϵ (find) does not employ the same hashing technique, and is excluded from this discussion.)

Building a *hash table* for the principal argument takes a significant time but is rewarded by a considerably quicker search for each item in the subject. Unfortunately, the hash table is discarded each time the function completes and must be reconstructed for a subsequent call (even if its principal argument is identical to that in the previous one).

For optimal performance of *repeated* search operations, the hash table may be retained between calls, by binding the function with its principal argument using the primitive $\circ$ (compose) operator. The retained hash table is then used directly whenever this monadic derived function is applied to a subject argument.

Notice that retaining the hash table pays off only on a second or subsequent application of the derived function. This usually occurs in one of two ways: either the derived function is named for later (and repeated) use, as in the first example below or it is applied repeatedly as the operand of a primitive or defined operator, as in the second example.

Example: naming a derived function.

```

words←'red' 'ylo' 'grn' 'brn' 'blu' 'pnk' 'blk'

find←words∘↑                A monadic find function
find'blk' 'blu' 'grn' 'ylo' A
7 5 3 2
find'grn' 'brn' 'ylo' 'red' A fast find
3 4 2 1

```

Example: repeated application by (¨) each operator.

```

    € ¢ □ A ¨ 'This' 'And' 'That'
1 0 0 0 1 0 0 1 0 0 0

```

1.1.10 Idiom Recognition

Idiom Recognition

Idioms are commonly used expressions that are recognised and evaluated internally, providing a significant performance improvement.

For example, the idiom $BV \uparrow \rho A$ (where BV is a Boolean vector and A is an array) would (in earlier Versions of Dyalog APL) have been evaluated in 3 steps as follows:

1. Evaluate ρA and store result in temporary variable `temp1` (`temp1` is just an arbitrary name for the purposes of this explanation)
2. Evaluate $\uparrow \text{temp1}$ and store result in temporary variable `temp2`.
3. Evaluate $BV \uparrow \text{temp2}$
4. Discard temporary variables

In the current Version of Dyalog APL, the expression is recognised in its entirety and processed in a single step as if it were a single primitive function. In this case, the resultant improvement in performance is between 2 and 4.5.

Idiom recognition is precise; an expression that is almost identical but not exactly identical to an expression given in the [Section 1.1.10.2](#) table will not be recognised.

For example, $\square AV \uparrow$ will be recognised as an idiom, but $(\square AV) \uparrow$ will not. Similarly, $(,) /$ would not be recognized as the Join idiom.

Idiom List

In the following table, arguments to the idiom have types and ranks as follows:

Type	Description	Rank	Description
C	Character	S	Scalar or 1-item vector
B	Boolean	V	Vector
N	Numeric	M	Matrix
P	Nested	A	Array of any rank
X	any type		

For example: NV: numeric vector, CM: character matrix, PV: nested vector.

Idiom	Description
ρpXA	The rank of XA as a 1-element vector
$\neq pXA$	The rank of XA as a scalar
$BV/\iota NS$	The subset of NS corresponding to the 1s in BV
$BV/\iota pXV$	The positions in XV corresponding to the 1s in BV
$NA \triangleright \cdot \cdot \subset XV$	The subset of XV in the index positions defined by NA (equivalent to $XV[NA]$)
$XA1\{ \}XA2$	$XA1$ and $XA2$ are ignored (no result produced)
$XA1\{\alpha\}XA2$	$XA1$ ($XA2$ is ignored)
$XA1\{\omega\}XA2$	$XA2$ ($XA1$ is ignored)
$XA1\{\alpha \ \omega\}XA2$	$XA1$ and $XA2$ as a two item vector ($XA1 \ XA2$)
$\{0\}XA$	0 irrespective of XA
$\{0\} \cdot \cdot XA$	0 corresponding to each item of XA
$\cdot \cdot /PV$	The enclose of the items of PV (which must be of depth 2) catenated along their last axes
$\cdot /PV$	The enclose of the items of PV (which must be of depth 2) catenated along their first axes
$\triangleright \phi XA$	The item in the top right of XA ($\square ML < 2$)
$\uparrow \phi XA$	The item in the top right of XA ($\square ML \geq 2$)
$\triangleright \phi , XA$	The item in the bottom right of XA ($\square ML < 2$)
$\uparrow \phi , A$	The item in the bottom right of XA ($\square ML \geq 2$)
$0 = \rho XV$	1 if XV has a shape of zero, 0 otherwise
$0 = \rho pXA$	1 if XA has a rank of zero (scalar), 0 otherwise
$0 = \equiv XA$	1 if XA has a depth of zero (simple scalar), 0 otherwise
$XM1\{ (\downarrow \alpha) \iota \downarrow \omega \}$ $XM2$	A simple vector comprising as many items as there are rows in $XM2$, where each item is the number of the first row in $XM1$ that matches each row in $XM2$. See note below.
$\downarrow \Phi \uparrow PV$	A nested vector comprising vectors that each correspond to a position in the original vectors of PV – the first vector contains the first item from each vector in PV , padded to be the same length as the largest vector, and so on ($\square ML < 2$)

Idiom	Description
$\downarrow \Phi \triangleright PV$	A nested vector comprising vectors that each correspond to a position in the original vectors of PV – the first vector contains the first item from each vector in PV , padded to be the same length as the largest vector, and so on ($\square ML \geq 2$)
$\wedge \backslash ' ' = CA$	A Boolean mask indicating the leading blank spaces in each row of CA
$+ / \wedge \backslash ' ' = CA$	The number of leading blank spaces in each row of CA
$+ / \wedge \backslash BA$	The number of leading 1s in each row of BA
$\{ (\vee \backslash ' ' \neq \omega) / \omega \}$ CV	CV without any leading blank spaces
$\{ (+ / \wedge \backslash ' ' = \omega) \downarrow \omega \}$ CV	CV without any leading blank spaces
$\sim \circ ' ' \downarrow CA$	A nested vector comprising simple character vectors constructed from the rows of CA (which must be of depth 1) with all blank spaces removed
$\{ (+ / \vee \backslash ' ' \neq \Phi \omega) \uparrow \downarrow \omega \}$ CA	A nested vector comprising simple character vectors constructed from the rows of CA (which must be of depth 1) with trailing blank spaces removed
$\triangleright \circ \rho \backslash \backslash XA$	The length of the first axis of each item in XA ($\square ML < 2$)
$\uparrow \circ \rho \backslash \backslash XA$	The length of the first axis of each item in XA ($\square ML \geq 2$)
$XA1 , \leftarrow XA2$	XA1 redefined to be XA1 with XA2 catenated along its last axis
$XA1 \overline{\leftarrow} XA2$	XA1 redefined to be XA1 with XA2 catenated along its first axis
$\{ (\prec \Phi \omega) \square \omega \} XA$	XA sorted into ascending order
$\{ (\prec \Psi \omega) \square \omega \} XA$	XA sorted into descending order
$\{ \omega [\Phi \omega] \} XV$	XV sorted into ascending order
$\{ \omega [\Psi \omega] \} XV$	XV sorted into descending order
$\{ \omega [\Phi \omega ;] \} XM$	XM with the rows sorted into ascending
$\{ \omega [\Psi \omega ;] \} XM$	XM with the rows sorted into descending order
$1 \equiv XA$	1 if XA has a depth of 1 (simple array), 0 otherwise
$1 \equiv , XA$	1 if XA has a depth of 0 or 1 (simple scalar, vector, etc.), 0 otherwise

Idiom	Description
$0 \in \rho XA$	1 if XA is empty, 0 otherwise
$\sim 0 \in \rho XA$	1 if XA is not empty, 0 otherwise
$\rightarrow \neq XA$	The first sub-array along the first axis of XA
$\rightarrow / XA$	The first sub-array along the last axis of XA
$\leftarrow \neq XA$	The last sub-array along the first axis of XA
$\leftarrow / XA$	The last sub-array along the last axis of XA
$* \circ NA$	Euler's idiom (accurate when NA is a multiple of 0.5)
$0 \Rightarrow \rho XA$	1 if XA has an empty first dimension, 0 otherwise ($\square ML < 2$)
$0 \nRightarrow \rho XA$	1 if XA does not have an empty first dimension, 0 otherwise ($\square ML < 2$)
$\square AV \iota CA$	Classic version only: The character numbers (atomic vector index) corresponding to the characters in CA
$\lfloor 0.5 + NA$	Round to nearest integer
$XA \downarrow \rightsquigarrow \leftarrow NS$	This idiom applies only when NS is negative, when it removes the last $-NS$ items from XA along its leading axis. See note below.
$\{ (\leftarrow \Delta \omega) \square \omega \}$ $\{ (\leftarrow \Psi \omega) \square \omega \}$	These idioms provide the fastest way to sort arrays of any rank

Notes

$/ \iota$ and $/ \iota \rho$, as well as providing an execution time advantage, reduce intermediate workspace usage and, consequently, the incidence of memory compactions and the likelihood of a `WS FULL`.

$NA \triangleright \leftarrow XV$ is implemented as $XV[NA]$, which is significantly faster. The two are equivalent but the former now has no performance penalty.

$, /$ is special-cased only for vectors of vectors or scalars. Otherwise, the expression is evaluated as a series of concatenations. Recognition of this idiom turns **join** from an *n-squared* algorithm into a linear one. In other words, the improvement factor is proportional to the size of the argument vector.

$\Rightarrow\phi$ and $\Rightarrow\phi$, now take constant time. Without idiom recognition, the time taken depends linearly on the number of items in the argument.

$0==$ takes a small constant time. Without idiom recognition, the time taken would depend on the size and depth of the argument, which in the case of a deeply nested array could be significant.

$\downarrow\Phi\uparrow$ is special-cased only for a vector of nested vectors, each of whose items is of the same length.

$\{(\downarrow\alpha)\downarrow\omega\}$ can accommodate much larger matrices than its constituent primitives. It is particularly effective when bound with a left argument using the compose operator:

```
find←mat∘{(↓α)↓ω}      A find rows in mat table
```

In this case, the internal hash table for `mat` is retained so that it does not need to be generated each time the monadic derived function `find` is applied to a matrix argument.

$\{(\vee\backslash' \neq\omega)/\omega\}$ and $\{(+/\wedge\backslash' =\omega)\downarrow\omega\}$ are two codings of the same idiom. Both use the same C code for evaluation.

$\sim\circ' \text{ ''}\downarrow$ typically takes a character matrix argument and returns a vector of character vectors from which all blanks have been removed. An example might be the character matrix of names returned by the system function `□NL`. In general, this idiom accommodates character arrays of any rank.

$\{(+/\vee\backslash' \neq\phi\omega)\uparrow''\downarrow\omega\}$ typically takes a character matrix argument and returns a vector of character vectors. Any embedded blanks in each row are preserved but trailing blanks are removed. In general, this idiom accommodates character arrays of any rank.

$\Rightarrow\circ\rho''A$ ($\square ML < 2$) and $\uparrow\circ\rho''A$ ($\square ML > 2$) avoid having to create an intermediate nested array of shape vectors.

For an array of vectors, this idiom quickly returns a *simple array* of the length of each vector.

```
⇒∘ρ'' 'Hi' 'Pete' A Vector Lengths
2 4
```

For an array of matrices, it returns a simple array of the number of rows in each matrix.

```
⇒∘ρ''□CR''↓□NL 3      A Lines in functions
5 21...
```

$A, \leftarrow A$ and $A; \leftarrow A$ optimise the catenation of an array to another array along the last and first dimension respectively.

Among other examples, this idiom optimises repeated catenation of a scalar or vector to an existing vector.

```
props, ←c 'Posn' 0 0
props, ←c 'Size' 50 50
vector, ←2+4
```

Note that the idiom is not applied if the value of vector V is shared with another symbol in the workspace, as illustrated in the following examples:

Example 1: the idiom is used to perform the catenation to $V1$.

```
V1 ← 10
V1, ← 11
```

Example 2: the idiom is not used to perform the catenation to $V1$, because its value is at that point shared with $V2$.

```
V1 ← 10
V2 ← V1
V1, ← 11
```

Example 3: the idiom is not used to perform the catenation to V in `Join[1]` because its value is, at that point, shared with the array used to call the function.

```
▽ V ← V Join A
[1] V, ← A
▽
(10) Join 11
1 2 3 4 5 6 7 8 9 10 11
```

$\vdash \nrightarrow XA$, $\vdash /XA$, $\nrightarrow \nrightarrow XA$, and $\nrightarrow /XA$ return the first/last rank ($0 \vdash^{-1} + \rho \rho A$) sub-array along the first/last axis of XA . For example, if V is a vector, then:

$\nrightarrow /V$	First item of vector
$\vdash /V$	Last item of vector

Similarly, if M is a matrix, then:

$\rightarrow \neq M$	First row of matrix
$\rightarrow / M$	First column of matrix
$\leftarrow \neq M$	Last row of matrix
$\leftarrow / M$	Last column of matrix

The idiom generalises uniformly to higher-rank arrays.

Euler's idiom $\ast \circ N A$ produces accurate results for right argument values that are a multiple of $0J0.5$. This is so that Euler's famous identity $0=1+\ast \circ 0J1$ holds, despite π being represented as a floating point number.

For clarification; $X A \uparrow \sim \leftarrow N S$. If $N S$ is ~ 3 then the idiom removes the last ~ 3 (that is, 3) items.

The idiom $X M 1 \{ (\downarrow \alpha) \uparrow \downarrow \omega \} X M 2$ is still recognised, but since Version 14.0 is no faster than $X M 1 \uparrow X M 2$.

1.1.11 Parallel Execution

If your computer has more than one CPU or is a multi-core processor, then the scalar dyadic functions $\div$, $\geq$, $=$, $\leq$, $\otimes$, $|$, $!$, $\circ$, $\vee$ and $\wedge$ will, when applied to arrays with a sufficiently large number of elements, execute in parallel in separate system threads.

For example, if you have a computer with 4 cores (either real or virtual) and execute an expression such as $(A \div B)$ where A and/or B contain more than 32,768 elements, then Dyalog will start 4 separate threads, each performing the division on $\frac{1}{4}$ of the elements of the array(s) and simultaneously creating the corresponding $\frac{1}{4}$ of the result array. The threads are only started once, and are reused for subsequent multi-threaded operations.

The maximum number of threads to use can be controlled using `1111⌈`, and the parallel execution threshold is changed using `1112⌈`. These "tuning" I-beams should be considered experimental, and may be changed or replaced in a future release. (See *Dyalog APL Language: Number Of Threads* and *Dyalog APL Language: Parallel Execution Threshold*).

Note that these scalar dyadic functions are not multi-threaded when applied to arrays of Boolean or integer values, they are also not multi-threaded for $+$, $-$ or $\times$ when applied to arrays of 64 bits floating (type 645). Tests show that the overhead of preparing such arrays for multi-threaded operations outweigh the performance benefits.

1.1.12 Complex Numbers

A complex number is a number consisting of a real and an imaginary part which is usually written in the form $a + bi$, where a and b are real numbers, and i is the standard imaginary unit with the property $i^2 = -1$.

Dyalog APL adopts the J notation introduced in IBM APL2 to represent the value of a complex number which is written as `aJb` or `a jb` (without spaces). The former representation (with a capital J) is always used to display a value.

Notation

```
2+~1*.5
2J1

.3j.5
0.3J0.5

1.2E5J~4E~4
120000J~0.0004
```

Arithmetic

The arithmetic primitive functions handle complex numbers in the appropriate way.

```
2j3+.3j.5  A (a+bi)+(c+di) = (a+c)+(b+d)i
2.3J3.5

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

2j3*.3j.5  A (a+bi)(c+di) = ac+bc i+adi+bdi<sup>2</sup>
              A              = (ac-bd)+(bc+ad)i
~0.9J1.9
```

The absolute value, or magnitude of a complex number is naturally obtained using the Magnitude function

```
|3j4
5
```

Monadic + of a complex number (a+bi) returns its conjugate (a-bi) ...

```
+3j4
3J~4
```

... which when multiplied by the complex number itself, produces the square of its magnitude.

```
3j4*3j~4
25
```

Furthermore, adding a complex number and its conjugate produces a real number:

```
3j4+3j^-4
6
```

The famous Euler's Identity $e^{i\pi}+1=0$ may be expressed as follows:

```
1+*o0j1 a Euler Identity
0
```

Circular functions

The basic set of circular functions $X \circ Y$ cater for complex values in Y , while the following extended functions provide specific features for complex arguments. Note that a and b are the real and imaginary parts of Y respectively and θ is the phase of Y .

$(-X) \circ Y$	X	$X \circ Y$
$-8 \circ Y$	8	$(-1+Y*2)*0.5$
Y	9	a
$+Y$	10	$ Y $
$Y \times 0J1$	11	b
$*Y \times 0J1$	12	θ

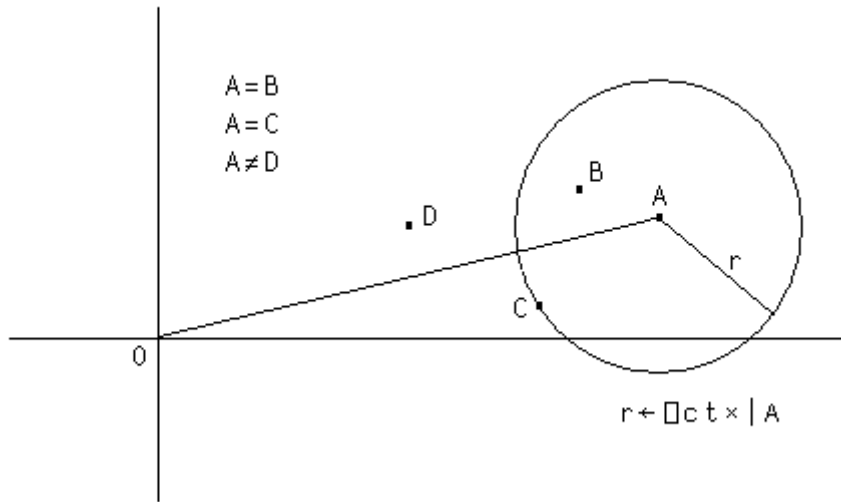
Note that $9 \circ Y$ and $11 \circ Y$ return the real and imaginary parts of Y respectively:

```
9 11o3.5J^-1.2
3.5 ^-1.2

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

Comparison

In comparing two complex numbers X and Y , $X=Y$ is 1 if the magnitude of $X-Y$ does not exceed ϵCT 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 ϵCT times the larger magnitude.



As with real values, complex values sufficiently close to Boolean or integral values are accepted by functions which require Boolean or integral values. For example:

```
2j1e-14 ρ 12
12 12
0 ~ 1j1e-15
0
```

Note that Dyalog APL always stores complex numbers as a pair of 64-bit binary floating-point numbers, regardless of the setting of $\square FR$. Comparisons between complex numbers and decimal floating-point numbers will require conversion of the decimal number to binary to allow the comparison. When $\square FR=1287$, comparisons are always subject to $\square DCT$, not $\square CT$ - regardless of the data type used to represent a number.

This only really comes into play when determining whether the imaginary part of a complex number is so small that it can be considered to be on the real line. However, Dyalog recommends that you do not mix the use of complex and decimal numbers in the same component of an application.

1.1.13 128 Bit Decimal Floating-Point Support

Introduction

The original IEEE-754 64-bit binary floating point (FP) data type (also known as type number 645), that is used internally by Dyalog APL to represent floating-point values,

does not have sufficient precision for certain financial computations – typically involving large currency amounts. The binary representation also causes errors to accumulate even when all values involved in a calculation are "exact" (rounded) decimal numbers, since many decimal numbers cannot be accurately represented regardless of the precision used to hold them. To reduce this problem, Dyalog APL includes support for the 128-bit decimal data type described by IEEE-754-2008 as an alternative representation for floating-point values.

System Variable: Floating-point Representation

Computations using 128-bit decimal numbers require twice as much space for storage, and run more than an order of magnitude more slowly on platforms which do not provide hardware support for the type. At this time, hardware support is only available from IBM (POWER 6 chips onwards, and recent System z mainframes). Even with hardware support, a slowdown of a factor of 4 can be expected. For this reason, Dyalog allows users to decide whether they need the higher-precision decimal representation, or prefer to stay with the faster and smaller binary representation.

The system variable `⎕FR` (for Floating-point Representation) can be set to the value 645 (the installed default) to indicate 64-bit binary FP, or 1287 for 128-bit decimal FP. The default value of `⎕FR` is configurable.

Simply put, the value of `⎕FR` decides the type of the result of any floating-point calculation that APL performs. In other words, when entered into the session:

```
⎕FR = ⎕DR 1.234  ⓐ Type of a floating-point constant
⎕FR = ⎕DR 3÷4    ⓐ Type of any floating-point result
```

`⎕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 `⎕FR` 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. Similarly, a constant typed into a line in the Session is evaluated using the value of `⎕FR` that pertained before the line is executed. Thus, it would be possible for the first line of code above to return 0, if it is in the body of a function. If the function was edited and while suspended and execution is resumed, the result would become 1. Also note:

```

□FR←1287
x←1÷3
    
```

```

□FR←645
x=1÷3
    
```

1

The decimal number has 17 more 3s. 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 128-bit 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 $-1E6145$ to $1E6145$. Loss of precision is accepted on conversion from 645 to 1287, but the magnitude of a number may make the conversion impossible, in which case a `DOMAIN ERROR` is issued:

```

□FR←1287
x←1E1000

□FR←645
x←0
DOMAIN ERROR

```

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 this is equivalent to 0 for complex numbers.

Conversion between Decimal and Binary

Conversion of data from Binary to Decimal is logically equivalent to formatting, and the reverse conversion is equivalent to evaluating input. These operations are performed according to the same rules that are used when formatting (and evaluating) numbers with □PP set to 17 (guaranteeing that the decimal value can be converted back to the same binary bit pattern). Because the precision of decimal floating-point numbers is much higher, there will always be a large number of potential decimal values which map to the same binary number: As with formatting, the rule is that the SHORTEST decimal number which maps to a particular binary value will be used as its decimal representation.

Data in component files will be stored without conversion, and only converted when a computation happens. It should be stored in decimal form if it will repeatedly be used by application code in which □FR has the value 1287. Even in applications which use decimal floating point everywhere, reading old component files containing arrays of type 645, or receiving data via □NA, the .NET interface or other external sources, will allow binary floating-point values to enter the system and require conversion.

Decimal Comparison Tolerance

When □FR has the value 1287, the system variable □DCT will be used to specify comparison tolerance. The default value of □DCT is $1E^{-28}$, and the maximum value is $2.3283064365386962890625E^{-10}$ (the value is chosen to avoid fuzzy comparison of 32-bit integers).

Name Association and Floating-point Values

APL supports the data type "D" to represent the Densely Packed Decimal (DPD) form of 128-bit decimal numbers, as specified by the IEEE-754 2008 standard. Dyalog has decided to use DPD, which is the format used by IBM for hardware support, on ALL platforms, although "Binary Integer Decimal" (BID) is the format that Intel libraries use to implement software libraries to do decimal arithmetic. Experiments have shown that the performance of 128-bit DPD and BID libraries are very similar on Intel platforms. In order to avoid the added complication of having two internal representations, Dyalog has elected to go with the hardware format, which is expected to be adopted by future hardware implementations.

The support libraries for writing APs and DLLs include new functions to extract the contents of a value of type D as a string or double-precision binary "float" – and convert data to D format.

Decimal Floats and Microsoft.NET

The Microsoft.NET framework contains a type named System.Decimal, which implements decimal floating-point numbers. However, it uses a different internal format from that defined by IEEE-754 2008.

Dyalog APL includes a Microsoft.NET class (called Dyalog.Dec128), which will perform arithmetic on data represented using the "Binary Integer Decimal" format. All computations performed by the Dyalog.Dec128 class will produce exactly the same results as if the computation was performed in APL. A "DCT" property allows setting the comparison tolerance to be used in comparisons, Ceiling/Floor, etc.).

The Dyalog class is modelled closely after the existing System.Decimal type, providing the same methods (Add, Ceiling, Compare, CompareTo, Divide, Equals, Finalize, Floor, FromOACurrency, GetBits, GetHashCode, GetType, GetTypeCode, MemberwiseClone, Multiply, Negate, Parse, Remainder, Round, Subtract, To*, Truncate, TryParse) and operators (Addition, Decrement, Division, Equality, Explicit, GreaterThan, GreaterThanOrEqual, Implicit, Increment, Inequality, LessThan, LessThanOrEqual, Modulus, Multiply, Subtraction, UnaryNegation, UnaryPlus).

The "bridge" between Dyalog and .NET is able to cast floating-point numbers to or from System.Double, System.Decimal and Dyalog.Dec128 (and perform all other reasonable casts to integer types etc.). Casting a Dyalog.Dec128 to or from strings will perform a "lossless" conversion.

Incoming .NET data types VT_DECIMAL (96-bit integer) and VT_CY (currency value represented by a 64-bit two's complement integer, scaled by 10,000) are converted to 126-bit decimal numbers (DECFs). This conversion is performed independently of the value of `⌈FR`.

If you want to perform arithmetic on values imported in this way, then you should set `⌈FR` to 1287, at least for the duration of the calculations.

Note that the .NET interface converts System.Decimal to DECFs but does not convert System.Int64 to DECFs.

1.1.14 Namespaces

Namespaces

Namespace is a (class 9) object in Dyalog APL. Namespaces are analogous to nested workspaces.

'Flat' APL Workspace	Workspace with Namespaces
<code>.OLD</code>	<code>.NEW</code>
	<code>FOO MAT VEC</code>
<code>DISPLAY</code>	<code>.Util</code>
<code>FOO MAT VEC</code>	<code>DISPLAY</code>
	<code>...</code>
<code>WsDoc_Init</code>	<code>.WsDoc</code>
<code>WsDoc_Xref</code>	<code>Init .prt .fmt</code>
<code>WsDoc_Tree</code>	<code>Init line</code>
<code>WsDoc_prt_init</code>	<code>Tree</code>
<code>WsDoc_current_page</code>	<code>Xref page</code>
<code>...</code>	<code>...</code>

They provide the same sort of facility for workspaces as directories do for filesystems. The analogy, based on DOS, might prove helpful:

Operation	Windows	Namespace
Create	mkdir	)NS or □NS
Change	cd	)CS or □CS
Relative name	dir1\dir\file	NS1.NS2.OBJ
Absolute name	\file\file	#.NS.OBJ
Name separator	\	.
Top (root) object	\	#
Parent object	..	##

Namespaces bring a number of major benefits

They provide lexical (as opposed to dynamic) local names. This means that a defined function can use local variables and functions which persist when it exits and which are available next time it is called.

Just as with the provision of directories in a filing system, namespaces allow us to organise the workspace in a tidy fashion. This helps to promote an object oriented programming style.

APL's traditional name-clash problem is ameliorated in several ways

- Workspaces can be arranged so that there are many fewer names at each namespace level. This means that when copying objects from saved workspaces there is a much reduced chance of a clash with existing names.
- Utility functions in a saved workspace may be coded as a single namespace and therefore on being copied into the active workspace consume only a single name. This avoids the complexity and expense of a solution which is sometimes used in 'flat' workspaces, where such utilities dynamically fix local functions on each call.
- In flat APL, workspace administration functions such as WSDOC must share names with their subject namespace. This leads to techniques for trying to avoid name clashes such as using obscure name prefixes like △△L1. This problem is now virtually eliminated because such a utility can operate exclusively in its own namespace.

The programming of GUI objects is considerably simplified.

- An object's callback functions may be localised in the namespace of the object itself.
- Static variables used by callback functions to maintain information between calls may be localised within the object.

This means that the object need use only a single name in its namespace.

Namespaces

Names within namespaces may be referenced *explicitly* or *implicitly*. An *explicit* reference requires that you identify the object by its full or relative pathname using a ' . ' syntax; for example:

```
X.NUMB ← 88
```

sets the variable NUMB in namespace X to 88.

```
88 UTIL.FOO 99
```

calls dyadic function FOO in namespace UTIL with left and right arguments of 88 and 99 respectively. The interpreter can distinguish between this use of ' . ' and its use as the inner product operator, because the leftmost name: UTIL is a (class 9) namespace, rather than a (class 3) function.

The general namespace reference syntax is:

```
SPACE . SPACE . (...) EXPR
```

Where SPACE is an *expression* which resolves to a namespace reference, and EXPR is any APL expression to be resolved in the resulting namespace.

There are two special space names:

is the top level or 'Root' namespace.

is the parent or space containing the current namespace.

□SE is a system namespace which is preserved across workspace load and clear.

EXAMPLES

```

WSDOC.PAGE.NO ←← 1      A Increment WSDOC page count

#.⍵NL 2                  A Variables in root space

UTIL.⍵FX 'Z←DUP A' 'Z←A A'  A Fix remote function

##.⍵ED'FOO'              A Edit function in parent space

⍵SE.RECORD ← PERS.RECORD  A Copy from PERS to ⍵SE

UTIL.(⍵EX ⍵NL 2)         A Expunge variables in UTIL

(=⍵SE #).(⍵→↑⍵NL 9).(⍵NL 2)  A Vars in first ⍵SE
                               A namespace.

UTIL.⍵STRING             A Execute STRING in UTIL space

```

You may also reference a function or operator in a namespace *implicitly* using the mechanism provided by `⍵EXPORT` (See *Dyalog APL Language: Export*) and `⍵PATH`. If you reference a name that is undefined in the current space, the system searches for it in the list of exported names defined for the namespaces specified by `⍵PATH`. See *Dyalog APL Language: Path* for further details.

Notice that the expression to the right of a dot may be arbitrarily complex and will be executed within the namespace or ref to the left of the dot.

```

X.(C←A×B)
X.C
10 12 14
16 18 20
NS1.C
10 12 14
16 18 20

```

Summary

Apart from its use as a decimal separator (3.14), '.' is interpreted by looking at the type or *class* of the expression to its left:

Template	Interpretation	Example
<code>◦.</code>	Outer product	<code>2 3 ◦.× 4 5</code>
<code>function.</code>	Inner product	<code>2 3 +.× 4 5</code>
<code>ref.</code>	Namespace reference	<code>2 3 x.foo 4 5</code>
<code>array.</code>	Reference array expansion	<code>(x y).⊖NC='foo'</code>

Namespace Reference Evaluation

When the interpreter encounters a namespace reference, it:

1. Switches to the namespace.
2. Evaluates the name.
3. Switches back to the original namespace.

If for example, in the following, the current namespace is `#.w`, the interpreter evaluates the line:

```
A ← X.Y.DUP MAT
```

in the following way:

1. Evaluate array `MAT` in current namespace `w` to produce argument for function.
2. Switch to namespace `X.Y` within `w`.
3. Evaluate function `DUP` in namespace `w.X.Y` with argument.
4. Switch back to namespace `w`.
5. Assign variable `A` in namespace `w`.

Namespaces and Localisation

The rules for name resolution have been generalised for namespaces.

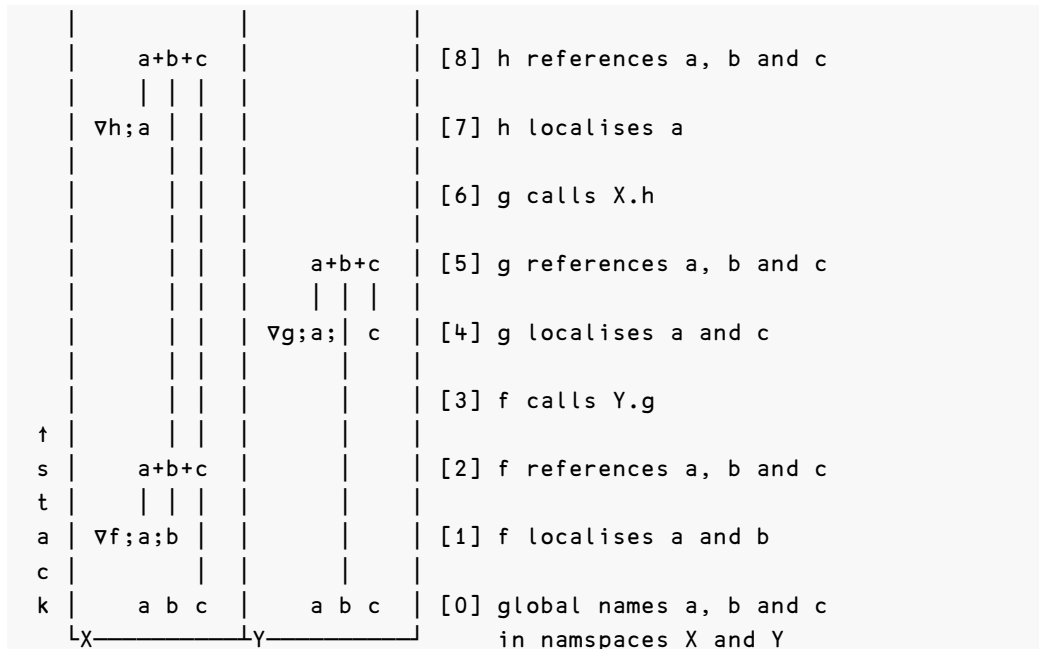
In flat APL, the interpreter searches the state indicator to resolve names referenced by a defined function or operator. If the name does not appear in the state indicator, then the workspace-global name is assumed.

With namespaces, a defined function or operator is evaluated in its 'home' namespace. When a name is referenced, the interpreter searches only those lines of the state indicator which belong to the home namespace. If the name does not appear in any of these lines, the home namespace-global value is assumed.

For example, if #.FN1 calls XX.FN2 calls #.FN3 calls XX.FN4, then:

- FN1:
 - is evaluated in #
 - can see its own dynamic local names
 - can see global names in #
- FN2:
 - is evaluated in XX
 - can see its own dynamic local names
 - can see global names in XX
- FN3:
 - is evaluated in #
 - can see its own dynamic local names
 - can see dynamic local names in FN1
 - can see global names in #
- FN4:
 - is evaluated in XX
 - can see its own dynamic local names
 - can see dynamic local names in FN2
 - can see global names in XX

The following picture illustrates how APL looks down the stack to find names:



The above diagram represents the SI stack, growing upwards from two namespaces X and Y, which each have three global names a, b and c.

1. Function f in X localises names a and b.
2. Function f references names a, b and c.

```

    ▽ f;a;b
  [1] a+b+c
  [2] Y.g
    
```

The interpreter looks down the stack and finds local names a and b in f's header and c in namespace X.

3. Function f calls function g in namespace Y.
4. Function g in Y localises names a and c.
5. Function g references names a, b and c.

```

    ▽ g;a;c
  [1] a+b+c
  [2] X.h
    
```

The interpreter looks down the stack and finds local names *a* and *c* in *g*'s header and *b* in namespaces *Y*.

6. Function *g* calls function *h* in namespace *X*.

7. Function *h* in *X* localises name *a*.

8. Function *h* references names *a*, *b* and *c*.

```

      ▽ h;a
[1]    a+b+c

```

The interpreter looks down the stack and finds local name *a* in *h*'s header; *b* in *f*'s header; and *c* in namespace *X*.

Namespace References

A namespace reference, or *ref* for short, is a unique data type that is distinct from and in addition to number and character.

Any expression may result in a *ref*, but the simplest one is the namespace itself:

```

)NS NS1      A Make a namespace called NS1
NS1.A←1      A and populate it with variables A
NS1.B←2 3p16  A and B

NS1          A expression results in a ref
#.NS1

```

You may assign a *ref*; for example:

```

X←NS1
X
#.NS1

```

In this case, the display of *x* informs you that *x* refers to the named namespace *#.NS1*.

You may also supply a *ref* as an argument to a defined function or a *dfn*:

```

      ▽ FOO ARG
[1]    ARG
      ▽

```

```

      FOO NS1
#.NS1

```

The name class of a ref is 9.

```

[NC 'X'
9

```

You may use a ref to a namespace anywhere that you would use the namespace itself. For example:

```

X.A
1
X.B
1 2 3
4 5 6

```

Notice that refs are references to namespaces, so that if you make a copy, it is the reference that is copied, not the namespace itself. This is sometimes referred to as a shallow as opposed to a deep copy. It means that if you change a ref, you actually change the namespace that it refers to.

```

X.A ← 1
X.A
2
NS1.A
2

```

Similarly, a ref passed to a defined function is call-by-reference, so that modifications to the content or properties of the argument namespace using the passed reference persist after the function exits. For example:

```

▽ FOO nsref
[1] nsref.B ← nsref.A
▽

FOO NS1
NS1.B
3 4 5
6 7 8
FOO X
NS1.B
5 6 7
8 9 10

```

Notice that the expression to the right of a dot may be arbitrarily complex and will be executed within the namespace or ref to the left of the dot.

```

      X.(C←A×B)
      X.C
10 12 14
16 18 20
      NS1.C
10 12 14
16 18 20

```

Unnamed Namespaces

The monadic form of `□NS` makes a new (and unique) unnamed namespace and returns a ref to it.

One use of unnamed namespaces is to represent hierarchical data structures; for example, a simple employee database:

The first record is represented by `JOHN` which is a ref to an unnamed namespace:

```

      JOHN←□NS ''
      JOHN
#. [Namespace]

      JOHN.FirstName←'John'
      JOHN.FirstName
John

      JOHN.LastName←'Smith'
      JOHN.Age←50

```

Data variables for the second record, `PAUL`, can be established using strand, or vector, assignment:

```

      PAUL←□NS ''
      PAUL.(FirstName LastName Age←'Paul' 'Brown' 44)

```

The function `SHOW` can be used to display the data in each record (the function is split into 2 lines only to fit on the printed page). Notice that its argument is a ref.

```

▼ R←SHOW PERSON
[1] R←PERSON.FirstName, ' ', PERSON.LastName
[2] R, ← ' is ', ⌘PERSON.Age
▼

    SHOW JOHN
John Smith is 50

    SHOW PAUL
Paul Brown is 44

```

An alternative version of the function illustrates the use of the `:With` `:EndWith` control structure to execute an expression, or block of expressions, within a namespace:

```

▼ R←SHOW1 PERSON
[1] :With PERSON
[2]   R←FirstName, ' ', LastName, ' is ', (⌘Age)
[3] :EndWith
▼

    SHOW1 JOHN
John Smith is 50

```

In this case, as only a single expression is involved, it can be expressed more simply using parentheses.

```

▼ R←SHOW2 PERSON
[1] R←PERSON.(FirstName, ' ', LastName, ' is ', (⌘Age))
▼

    SHOW2 PAUL
Paul Brown is 44

```

Dfns also accept refs as arguments:

```

SHOW3←{
  ω.(FirstName, ' ', LastName, ' is ', ⌘Age)
}

    SHOW3 JOHN
John Smith is 50

```

Arrays of Namespace References

You can construct arrays of refs using strand notation, array notation, catenate (,), reshape (ρ) and other structural primitives, as well as certain system functions.

```

EMP ← JOHN PAUL
ρEMP
2
EMP
#.[Namespace] #.[Namespace]
```

Like any other array, an array of refs has name class 2:

```

[NC 'EMP']
2
```

Expressions such as indexing and pick return refs that may in turn be used as follows:

```

EMP[1].FirstName
John
(2>EMP).Age
44
```

The each (``) operator may be used to apply a function to an array of refs:

```

SHOW``EMP
John Smith is 50 Paul Brown is 44
```

An *array* of namespace references (refs) to the left of a '.' is expanded according to the following rule, where x and y are refs, and exp is an arbitrary expression:

$$(x\ y).exp \rightarrow (x.exp)(y.exp)$$

If exp evaluates to a function, the items of its argument array(s) are *distributed* to each referenced function. In the dyadic case, there is a 3-way distribution among: left argument, referenced functions and right argument.

Monadic function f :

$$(x\ y).f\ d\ e \rightarrow (x.f\ d)(y.f\ e)$$

Dyadic function g :

$$a\ b\ (x\ y).g\ d\ e \rightarrow (a\ x.g\ d)(b\ y.g\ e)$$

An array of refs to the left of an assignment arrow is expanded thus:

```
(x y).a←c d → (x.a←c)(y.a←d)
```

Note that the array of refs can be of any rank. In the limiting case of a simple scalar array, the array construct: `refs.exp` is identical to the scalar construct: `ref.exp`.

Note that the expression to the right of the '.' pervades a nested array of refs to its left:

```
((u v)(x y)).exp → ((u.exp)(v.exp))((x.exp)(y.exp))
```

Note also that with successive expansions `(u v).(x y z). ...`, the final number of "leaf" terms is the product of the number of refs at each level.

EXAMPLES

```

(JOHN PAUL)←()()
JOHN.Children←()()
ρJOHN.Children
2
JOHN.Children[1].FirstName←'Andy'
JOHN.Children[1].Age←23

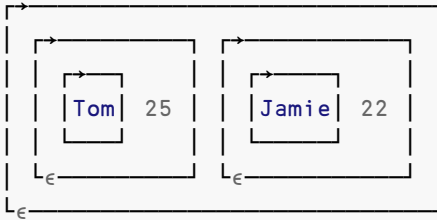
JOHN.Children[2].FirstName←'Katherine'
JOHN.Children[2].Age←19

PAUL.Children←()()
PAUL.Children[1].(FirstName Age←'Tom' 25)
PAUL.Children[2].(FirstName Age←'Jamie' 22)

EMP←JOHN PAUL
ρEMP
2
(⇒EMP).Children.(FirstName Age)
Andy 23 Katherine 19

]display (2⇒EMP).Children.(FirstName Age)

```



```

EMP.Children A Is an array of refs
#.[Namespace] #.[Namespace] #.[Namespace] ...

EMP.Children.(FirstName Age)
Andy 23 Katherine 19 Tom 25 Jamie 22

```

Distributed Assignment

Assignment pervades nested strands of names to the left of the arrow. The conformability rules are the same as for scalar (pervasive) dyadic primitive functions such as '+'. The mechanism can be viewed as a way of naming the parts of a structure.

EXAMPLES

```

    EMP.(FirstName Age)
JOHN 43    PAUL 44

    EMP.(FirstName Age)←('Jonathan' 21)('Pauline' 22)

    EMP.(FirstName Age)
Johnathan 21    Pauline 22

A Distributed assignment is pervasive
    JOHN.Children.(FirstName Age)
Andy 23    Katherine 19

    JOHN.Children.(FirstName Age)←('Andrew' 21)('Kate' 9)

    JOHN.Children.(FirstName Age)
Andrew 21    Kate 9

```

More Examples

```

((a b)(c d))←(1 2)(3 4)      A a←1 ♦ b←2 ♦ c←3 ♦ d←4

((IO ML)vec)←0 AV           A IO←0 ♦ ML←0 ♦ vec←AV

(i (j k))←+1 2               A i←+1 ♦ j←+2 ♦ k←+2

A Naming of parts:

((first last) sex (street city state))←n>pvec

A Distributed assignment in :For loop:

:For (i j)(k l) :In array

A Ref array expansion:

(x y).(first last)←('John' 'Doe')('Joe' 'Blow')
(f1 f2).(b1 b2).Caption←'OK' 'Cancel'

A Structure rearrangement:
rotate1←{                    A Simple binary tree rotation.
  (a b c)d e←ω
  a b(c d e)
}
rotate3←{                    A Compound binary tree rotation.
  (a b(c d e))f g←ω
  (a b c)d(e f g)
}

```

Distributed Functions

Namespace ref array expansion syntax applies to functions too.

[illegible]

```

    (x y).⌈NL 2 3           A x:vars, y:fns
varx  funy

    (x y).⌈NL<2 3          A x&y: vars&fns
funx  funy
varx  vary

    (x y).(⌈NL'')<2 3      A x&y: separate vars&fns
varx  funx    vary  funy

    'v'(x y).⌈NL 2 3       A x:v-vars, y:v-fns
varx

    'vf'(x y).⌈NL 2 3      A x:v-vars, y:f-fns
varx  funy

    'vf'(x y).⌈NL<2 3     A x:v-vars&fns,
                        A y:f-vars&fns
varx  funy

    x.⌈NL 2 3              A depth 0 ref
funx
varx

    (x y).⌈NL<2 3          A depth 1 refs
funx  funy
varx  vary

    ((u v)(x y)).⌈NL<2 3   A depth 2 refs
funu  funv    funx  funy
varu  varv    varx  vary

    (1 2)3 4(w(x y)z).+1 2(3 4) A arg distribution.
2 3  5 5  7 8

```

Namespaces and Operators

A function passed as operand to a primitive or defined operator, carries its namespace context with it. This means that if subsequently, the function operand is applied to an argument, it executes in its home namespace, irrespective of the namespace from which the operator was invoked or defined.

EXAMPLES

```

    VAR←99                                R #.VAR

    )NS X
#.X
    X.VAR←77                            R X.VAR
    X.FX'Z←FN R' 'Z←R,VAR'

    )NS Y
#.Y
    Y.VAR←88                            R Y.VAR
    Y.FX'Z←(F OP)R' 'Z←F R'

    X.FN"13
1 77  2 77  3 77

    X.FN 'VAR:'
VAR: 77

    X.FN Y.OP 'VAR:'
VAR: 77
    Y.OP'VAR'
99

```

Serialising Namespaces

The Serialisation of an array is its conversion from its internal representation, which may contain pointers to other structures in the workspace, into a self-contained series of bytes. This allows the array to be written to a file, transmitted over a socket or used in a variety of other ways. The de-serialisation of an array is the conversion back to an internal format whose content and structure is identical to the original array.

If an array contains a reference to a namespace or object that is within the same array, it can be serialised and de-serialised normally.

If an array contains a reference to a namespace or object that is not internal to the array itself, this presents a problem, which is resolved as follows:

1. If the reference is a direct reference to Root (#) or to `⌈SE`, it is serialised as a reference to that symbol, but the contents of # or `⌈SE` are not included. When the array is de-serialised, this results in a reference to the Root (#) or `⌈SE` in the current workspace. The newly reconstituted array is not strictly identical to the original because the contents of # or `⌈SE` may be different.

2. If the reference is to an arbitrary external namespace or object, a copy of that object is included but its path is discarded. When the array is de-serialised, the copy is reconstituted as a sibling (that is, as a child of the same parent as the de-serialised array). In this case the contents of the external namespace or object are preserved, but not its path. The newly reconstituted array is not strictly identical to the original because the path to the external reference has changed.
3. If however, the external namespace or object itself contains an external reference, the operation fails with `DOMAIN ERROR`.

The following example uses `220±` but applies equally to an array serialised by, for example `±FAPPEND`.

EXAMPLES

```
'A' ±NS ''
'B' ±NS ''
'C' ±NS ''
A.b←B
B.c←C
s←1 (220±)A
)ERASE A B C
)OBS

New←0(220±)s
New
#.A
New.b
#.B
New.b.c
#.C
```

```
)CLEAR
clear ws
'A' ±NS ''
'B' ±NS ''
'X'±NS ''
'X.C'±NS ''
A.b←B
B.c←X.C
s←1(220±)A
DOMAIN ERROR: Namespace is not self contained
s←1(220±)A
^
```

Note that a successful `0(2201)` does not mean that a `1(2201)` on the result will succeed. If the original reference was to, say, the MenuBar of `□SE` you cannot reconstitute that in `#`.

1.1.15 External Variables

An external variable is a variable whose contents (value) reside not in the workspace, but in a file. An external variable is associated with a file by the system function `⌈XT`. If at the time of association the file exists, the external variable assumes its value from the contents of the file. If the file does not exist, the external variable is defined but a `VALUE ERROR` occurs if it is referenced before assignment. Assignment of an array to the external variable or to an indexed element of the external variable has the effect of updating the file. The value of the external variable or the value of indexed elements of the external variable is made available in the workspace when the external variable occurs in an expression. No special restrictions are placed on the usage of external variables.

Normally, the files associated with external variables remain permanent in that they survive the APL session or the erasing of the external variable from the workspace. External variables may be accessed concurrently by several users, or by different nodes on a network, provided that the appropriate file access controls are established. Multi-user access to an external variable may be controlled with the system function `⌈FHOLD` between co-operating tasks.

Refer to the sections describing the system functions `⌈XT` and `⌈FHOLD` in *Chapter 6* for further details.

EXAMPLES

```

      'ARRAY' ⌈XT 'V'

      V←⍲10
      V[2] + 5
7

      ⌈EX 'V'

      'ARRAY' ⌈XT 'F'
      F
1 2 3 4 5 6 7 8 9 10

```

Information

Support for external variables has been deprecated, and is scheduled for removal in a future release. For information on how to identify calls to external variables in your existing codebase, see the *Release Notes*.

1.1.16 Component Files

A component file is a data file maintained by Dyalog APL. It contains a series of APL arrays known as components which are accessed by reference to their relative positions or component number within the file. A set of system functions is provided to perform a range of file operations. (See *Dyalog APL Language: System Functions By Category*.) These provide facilities to create or delete files, and to read and write components. Facilities are also provided for multi-user access including the capability to determine who may do what, and file locking for concurrent updates. (See the *Dyalog Programming Reference Guide*).

1.1.17 Auxiliary Processors

Auxiliary Processors (APs) are non-APL programs which provide Dyalog APL users with additional facilities. They run as separate tasks, and communicate with the Dyalog APL interpreter through pipes (UNIX) or via an area of memory (Windows). Typically, APs are used where speed of execution is critical, such as in screen management software, or for utility libraries. Auxiliary Processors may be written in any compiled language, although 'C' is preferred and is directly supported.

When an Auxiliary Processor is invoked from Dyalog APL, one or more *external functions* are fixed in the active workspace. Each external function behaves as if it was a locked defined function, but is in effect an entry point into the Auxiliary Processor. An external function occupies only a negligible amount of workspace.

Although Auxiliary Processors are still supported, Dyalog recommends that DLLs/shared libraries, called via the `⌈NA` interface should be used on all platforms in future, and that existing APs are converted to DLLs/shared libraries.

2 Deprecated Functionality

Over time, certain functionality (such as language elements, development environment features, or supplied samples or tools) can become obsolete or cease to be useful. There are many reasons why this might happen. For example:

- A superior alternative has been introduced.
For example, `⎕UCS` has superseded `⎕TC` (which generates only the newline, backspace, and linefeed characters).
- The feature was originally implemented as an I-beam but has since been superseded by a formal addition to Dyalog APL.
For example, `⎕JSON` replaced `7159⍤`.
- The feature is associated with hardware or technology that is itself becoming obsolete.
For example, 32-bit processes and address spaces limited to 4GB in size.

In these circumstances, the feature is classified as *deprecated*. This means that it is unlikely to be developed or extended further, and its use in new development work is discouraged. Some deprecated features remain for backwards compatibility reasons, but some are later removed in a pre-announced Dyalog version.

2.1 Deprecated Functionality Scheduled for Removal

If removing a deprecated feature is considered to be sufficiently significant, Dyalog Ltd will enable the ability to identify where this feature exists in a given codebase.

Deprecated functionality can be identified either when it is encountered in code that is called or by scanning a directory for deprecated file types.

2.1.1 Identifying Deprecated Functionality in Called Code

Dyalog can be configured to log use of a deprecated feature when it is encountered. Logging must be configured and enabled/disabled in each APL process – information about logging is not retained.

To enable logging of deprecated features:

1. Specify the name of the file into which the JSON5 log messages will be written (using *Dyalog APL Language: Log File For Deprecations* with a right argument of 0). By default, the file is created in the current working directory; if you do not have write permission for the current working directory, you will also need to include a path. If a filename is not set, then all logging information will be silently discarded.
2. Specify which deprecated features should be logged (using *Dyalog APL Language: Log Use Of Deprecated Features* with a right argument of the names identifying the deprecated features of interest). If no names are specified then all logging will be disabled.

The names are release-dependent; for a list of valid names see the *v20.0 Release Notes: Deprecated Functionality*.

In addition, there are two reserved names than can be used:

- ALL – enables logging for all valid names
- None – disable all logging

Each time `13⍤` is called, the new list of features replaces the existing list.

Example

```
A Specify the name of the log file
   'C:\Users\fiona\deprecated_log.txt'(109⍤)0

A Select the features to log
   13⍤ 'J0C0' 'S32'
```

After logging has been enabled, every subsequent use of the specified deprecated features is logged. Each line in the log file contains a complete JSON5 object, which includes a description of the deprecated feature and the SI Stack at the point it was called. The log file can be examined using any text editor or from within a Dyalog Session. For example:

```
log_entries←(⍵JSON⍵('Dialect' 'JSON5'))('Compact' 0))⌘2''>⍵NGET 'C:
\Users\fiona\deprecated_log.txt' 1
```

If an error occurs when writing to the log file, further logging is suspended. The log file status can be queried at any time by calling `109⍤` with a right argument of 1; the result is a numeric status code (0 indicates no error) and a character vector describing the error that was encountered (empty if no error). For example:

```
(109I)1
```

```
0
```

or:

```
(109I)1
```

```
3|The system cannot find the path specified.|
```

2.1.2 Identifying Files Pertaining to Deprecated Functionality

A directory can be scanned for deprecated functionality using *Dyalog APL Language: Scan For Deprecated Files* with a right argument of the directory to be scanned. If the left argument is set to 1, sub-directories will also be scanned. The names of any files that pertain to deprecated functionality are returned, with labels specifying the reason for identification.

The labels are release-dependent; for a list of valid labels see the *v20.0 Release Notes: Deprecated Functionality*.

Example

```
1(3535I)'.'
./J0C0.dcf      J0C0
./ws2000.dws    OLDWS
./subdir/S32J0C0.dcf  J0C0  S32
```

3 Defined Functions & Operators

3.1 Traditional Functions and Operators

3.1.1 Traditional Functions and Operators

Traditional Functions and Operators are the original user-defined functions and operators that are part of the APL standard. They are referred to herein as *Traditional* or *TradFns* to distinguish them from Dfns and Dops which are unique to Dyalog.

TradFns may be defined and edited using the Dyalog Editor or may be instantiated from an array containing source code using the system function `⌈FX`. The converse system functions `⌈CR`, `⌈VR`, `⌈NR` return the original source code.

A defined function or operators is composed of lines. The first line (line 0) is called the *header*. Remaining lines are APL statements, called the *body*.

The header consists of the following parts:

1. its model syntactical form,
2. an optional list of local names, each preceded by a semi-colon (;) character,
3. an optional comment, preceded by the symbol ⌈.

Only the model is required. If local names and comments are included, they must appear in the prescribed order.

3.1.2 Model Syntax

The model for the defined operation identifies the name of the operation, its valence, and whether or not an explicit result may be returned. Valence is the number of explicit arguments or operands, either 0, 1 or 2; whence the operation is termed NILADIC, MONADIC or DYADIC respectively. Only a defined function may be niladic. There is no relationship between the valence of a defined operator, and the valence of the derived function which it produces. Defined functions and derived functions produced by defined operators may be ambivalent, that is, may be executed monadically with one argument, or dyadically with two. An ambivalent operation is identified in its model by enclosing the left argument in braces.

The value of a result-returning function or derived function may be suppressed in execution if not explicitly used or assigned by enclosing the result in its model within braces. Such a suppressed result is termed SHY.

The tables below show all possible models for defined functions and operators respectively.

Defined Functions

Result	Niladic	Monadic	Dyadic	Ambivalent
None	f	$f \ Y$	$X \ f \ Y$	$\{X\} \ f \ Y$
Explicit	$R \leftarrow f$	$R \leftarrow f \ Y$	$R \leftarrow X \ f \ Y$	$R \leftarrow \{X\} \ f \ Y$
Suppressed	$\{R\} \leftarrow f$	$\{R\} \leftarrow f \ Y$	$\{R\} \leftarrow X \ f \ Y$	$\{R\} \leftarrow \{X\} \ f \ Y$

Note: the right argument Y and/or the result R may be represented by a single name, or as a blank-delimited list of names surrounded by parentheses. For further details, see [Section 3.1.6](#).

Derived Functions produced by Monadic Operator

Result	Monadic	Dyadic	Ambivalent
None	$(A \ op)Y$	$X(A \ op)Y$	$\{X\}(A \ op)Y$
Explicit	$R \leftarrow (A \ op)Y$	$R \leftarrow X(A \ op)Y$	$R \leftarrow \{X\}(A \ op)Y$
Suppressed	$\{R\} \leftarrow (A \ op)Y$	$\{R\} \leftarrow X(A \ op)Y$	$\{R\} \leftarrow \{X\}(A \ op)Y$

Derived Functions produced by Dyadic Operator

Result	Monadic	Dyadic	Ambivalent
None	$(A \ op \ B)Y$	$X(A \ op \ B)Y$	$\{X\}(A \ op \ B)Y$
Explicit	$R \leftarrow (A \ op \ B)Y$	$R \leftarrow X(A \ op \ B)Y$	$R \leftarrow \{X\}(A \ op \ B)Y$
Suppressed	$\{R\} \leftarrow (A \ op \ B)Y$	$\{R\} \leftarrow X(A \ op \ B)Y$	$\{R\} \leftarrow \{X\}(A \ op \ B)Y$

3.1.3 Statements

A statement is a line of characters understood by APL. It may be composed of:

1. a LABEL (which must be followed by a colon :), or a CONTROL STATEMENT (which is preceded by a colon), or both,
2. an EXPRESSION (see [Section 1.1.4](#)),
3. a SEPARATOR (consisting of the diamond character $\diamond$ which must separate adjacent expressions),
4. a COMMENT (which must start with the character ␣).

Each of the four parts is optional, but if present they must occur in the given order except that successive expressions must be separated by $\diamond$. Any characters occurring to the right of the first comment symbol (␣) that is not within quotes is a comment.

Comments are not executed by APL. Expressions in a line separated by $\diamond$ are taken in left-to-right order as they occur in the line. For output display purposes, each separated expression is treated as a separate statement.

EXAMPLES

```

50      5×10
50
50      MULT: 5×10
50
50      MULT: 5×10 ⋄ 2×4
50
50      MULT: 5×10 ⋄ 2×4 ␣ MULTIPLICATION
50
50      8
50      8
```

3.1.4 Global & Local Names

The following names, if present, are local to the defined operation:

1. the result,
2. the argument(s) and operand(s),
3. additional names in the header line following the model, each name preceded by a semi-colon character,

4. labels,
5. the argument list of the system function `⌈SHADOW` when executed,
6. a name assigned within a dfn.

All names in a defined operation must be valid APL names. The same name may be repeated in the header line, including the operation name (whence the name is localised). Normally, the operation name is not a local name.

The same name may not be given to both arguments or operands of a dyadic operation. The name of a label may be the same as a name in the header line. More than one label may have the same name. When the operation is executed, local names in the header line after the model are initially undefined; labels are assigned the values of line numbers on which they occur, taken in order from the last line to the first; the result (if any) is initially undefined.

In the case of a defined function, the left argument (if any) takes the value of the array to the left of the function when called; and the right argument (if any) takes the value of the array to the right of the function when called. In the case of a defined operator, the left operand takes the value of the function or array to the left of the operator when called; and the right operand (if any) takes the value of the function or array to the right of the operator when called.

During execution, a local name temporarily excludes from use an object of the same name with an active definition. This is known as LOCALISATION or SHADOWING. A value or meaning given to a local name will persist only for the duration of execution of the defined operation (including any time whilst the operation is halted). A name which is not local to the operation is said to be GLOBAL. A global name could itself be local to a pendent operation. A global name can be made local to a defined operation during execution by use of the system function `⌈SHADOW`. An object is said to be VISIBLE if there is a definition associated with its name in the active environment.

EXAMPLES

```

    A←1
    ▽ F
[1]    A←10
[2]    ▽

    F R <A> NOT LOCALISED IN <F>, GLOBAL VALUE REPLACED
    A
10
    A←1
    )ERASE F

    ▽ F;A
[1]    A←10
[2]    ▽

    F R <A> LOCALISED IN <F>, GLOBAL VALUE RETAINED
    A
1

```

Any statement line in the body of a defined operation may begin with a LABEL. A label is followed by a colon (:). A label is a constant whose value is the number of the line in the operation defined by system function □FX or on closing definition mode.

The value of a label is available on entering an operation when executed, and it may be used but not altered in any expression.

EXAMPLE

```

    [VR 'PLUS'
      ▽ R←{A} PLUS B
[1]   →DYADIC ρ←2=⊖NC'A' ♦ R←B ♦ →END
[2]   DYADIC: R←A+B
[3]   END:
      ▽

      1 [STOP 'PLUS'

      2 PLUS 2

PLUS[1]
    DYADIC
2

    END
3

```

3.1.5 Locals Lines

Locals Lines are lines in a defined function or operator that serve only to define local names.

A Locals Line may appear anywhere between line [0] and the first executable statement in the function or operator. Locals lines may be interspersed with blank lines and comments. A Locals Line is identified by starting with a semicolon, prefixed optionally by whitespace. It may contain a comment at the end.

A Locals Line must be of the form ;name;name;name where name is any valid APL name or localisable system variable. The names are localised on entry to the function exactly as if they were specified as locals on line [0].

EXAMPLE

```

    ▽ r←foo y;a;b      A some locals
      ;c;d            A some more locals
      (a b c d)←y
      r←a+b-c×d
    ▽

```

The function `foo` shown above localises names `a`, `b`, `c` and `d` (the indentation on line [1] in this example is entirely optional)

Syntactical errors on Locals Lines are detected when the user attempts to fix the function using the Editor or `FIX` and will causes the operation to fail.

3.1.6 Namelists

The right argument and the result of a function may be specified in the function header by a single name or by a *Namelist*. In this context, a Namelist is a blank-delimited list of names surrounded by a single set of parentheses.

Names specified in a Namelist are automatically local to the function; there is no need to localise them explicitly using semi-colons.

If the *right argument* of a function is declared as a Namelist, the function will only accept a right argument that is a vector whose length is the same as the number of names in the Namelist. Calling the function with any other argument will result in a `LENGTH ERROR` in the calling statement. Otherwise, the elements of the argument are assigned to the names in the Namelist in the specified order.

EXAMPLE

```

    ▽ IDN←Date2IDN(Year Month Day)
[1]   'Year is ',⌘Year
[2]   'Month is ',⌘Month
[3]   'Day is ',⌘Day
[4]   ...
    ▽

    Date2IDN 2004 4 30
Year is 2004
Month is 4
Day is 30

    Date2IDN 2004 4
LENGTH ERROR
    Date2IDN 2004 4
    ^

```

Note that if you specify a *single* name in the Namelist, the function may be called only with a 1-element vector right argument. If the *result* of a function is declared as a Namelist, the values of the names will automatically be stranded together in the specified order and returned as the result of the function when the function terminates.

EXAMPLE

```

    ▽ (Year Month Day)←Birthday age
[1]   Year←1949+age
[2]   Month←4
[3]   Day←30
    ▽
      Birthday 50
1999 4 30

```

3.1.7 Locked Functions & Operators

A defined operation may be locked by the system function □LOCK.

Once locked, an operation may not be displayed or edited and the system functions □CR, □NR and □VR return empty results.

Stop, trace and monitor settings are cancelled when an operation is locked.

A locked operation may not be suspended, nor may a locked operation remain pendent when execution is suspended. Instead, the state indicator is cut back to the point where the locked operation was invoked.

3.1.8 Function Declaration Statements**Function Declaration Statements**

Function Declaration statements are used to identify the characteristics of a function in some way.

The following declarative statements are provided.

- Section 3.1.8.2
- Section 3.1.8.3
- *:Implements*
- *:Signature*

With one exception, these statements are not executable statements and may theoretically appear anywhere in the body of the function. However, it is recommended that you place them at the beginning before any executable statements. The exception is:

```
:Implements Constructor <[:Base expr]>
```

In addition to being declarative (declaring the function to be a Constructor) this statement also executes the Constructor in the Base Class whether or not it includes :Base expr. Its position in the code is therefore significant.

Access Statement

:Access

```
:Access <Private|Public><Instance|Shared>
:Access <WebMethod>
```

The :Access statement is used to specify characteristics for functions that represent Methods in classes (see [Section 4.3.3.1](#)). It is also applicable to Classes and Properties.

Element	Description
Private Public	Specifies whether or not the method is accessible from outside the Class or an Instance of the Class. The default is Private .
Instance Shared	Specifies whether the method runs in the Class or Instance. The default is Instance .
WebMethod	Specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
Overridable	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
Override	Applies only to an Instance Method and specifies that the Method overrides the corresponding Overridable Method defined in the Base Class. See below

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being Overridable is replaced in-situ (that is, within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the Override keyword. For further information, see [Section 4.3.3.4](#).

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

Attribute Statement

:Attribute

```
:Attribute <Name> [ConstructorArgs]
```

The `:Attribute` statement is used to attach .NET Attributes to a Method (or Class).

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
Name	The name of a .NET attribute
ConstructorArgs	Optional arguments for the Attribute constructor

EXAMPLES

```
:Attribute ObsoleteAttribute
:Attribute ObsoleteAttribute 'Don't use' 1
```

Implements Statement

:Implements

The `:Implements` statement identifies the function to be one of the following types.

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><, name2, name3, ...>
:Implements Trigger *
```

Element	Description
Constructor	Specifies that the function is a <u>Class Constructor</u> .
:Base expr	Specifies that the <u>Base Constructor</u> be called with the result of the expression expr as its argument.
Destructor	Specifies that the function is a <u>Class Destructor</u> .
Method	Specifies that the function implements the Method MethodName whose syntax is specified by <u>Interface InterfaceName</u> .
Trigger	Identifies the function as a <u>Trigger Function</u> which is activated by changes to variable name1, name2, and so forth. Trigger * specifies a <u>Global Trigger</u> that is activated by the assignment of any global variable in the same namespace.

Signature Statement

:Signature

```
:Signature <rslttype*><name><arg1type arg1name>,...
```

This statement identifies the name and signature by which a function is exported as a method to be called from outside Dyalog APL. Several :Signature statements may be specified to allow the method to be called with different arguments and/or to specify a different result type.

Element	Description
rslttype	Specifies the data type for the result of the method
name	Specifies the name of the exported method.
argntype	Specifies the data type of the nth parameter
argnname	Specifies the name of the nth parameter

Argument and result data types are identified by the names of .NET Types which are defined in the .NET Assemblies specified by □USING or by a :USING statement.

EXAMPLES

In the following examples, it is assumed that the .NET Search Path (defined by :Using or □USING includes 'System'.

The following statement specifies that the function is exported as a method named `Format` which takes a single parameter of type `System.Object` named `Array`. The data type of the result of the method is an array (vector) of type `System.String`.

```
:Signature String[]←Format Object Array
```

The next statement specifies that the function is exported as a method named `Catenate` whose result is of type `System.Object` and which takes 3 parameters. The first parameter is of type `System.Double` and is named `Dimension`. The second is of type `System.Object` and is named `Arg1`. The third is of type `System.Object` and is named `Arg2`.

```
:Signature Object←Catenate Double Dimension,...  
...Object Arg1, Object Arg2
```

The next statement specifies that the function is exported as a method named `IndexGen` whose result is an array of type `System.Int32` and which takes 2 parameters. The first parameter is of type `System.Int32` and is named `N`. The second is of type `System.Int32` and is named `Origin`.

```
:Signature Int32[]←IndexGen Int32 N, Int32 Origin
```

The next block of statements specifies that the function is exported as a method named `Mix`. The method has 4 different signatures; that is, it may be called with 4 different parameter/result combinations.

```
:Signature Int32[,]←Mix Double Dimension, ...  
...Int32[] Vec1, Int32[] Vec2  
:Signature Int32[,]←Mix Double Dimension,...  
... Int32[] Vec1, Int32[] Vec2, Int32 Vec3  
:Signature Double[,]←Mix Double Dimension, ...  
... Double[] Vec1, Double[] Vec2  
:Signature Double[,]←Mix Double Dimension, ...  
... Double[] Vec1, Double[] Vec2, Double[] Vec3
```

3.1.9 Control Structures

Introduction

Control Structures are blocks of code in which the execution of APL statements follows certain rules and conditions. Control structures are implemented using a set of *control words* that all start with the colon symbol (:). Control Words are case-insensitive.

There are a number of different types of control structures defined by the control words, :If, :While, :Repeat, :For (with the supplementary control words :In and :InEach), :Select, :With, :Trap, :Hold and :Disposable. Each one of these control words may occur only at the beginning of an APL statement and indicates the start of a particular type of control structure.

Within a control structure, certain other control words are used as qualifiers. These are :Else, :ElseIf, :AndIf, :OrIf, :Until, :Case and :CaseList.

A third set of control words is used to identify the end of a particular control structure. These are

:EndIf, :EndWhile, :EndRepeat, :EndFor, :EndSelect, :EndWith, :EndTrap, :EndHold and :EndDisposable. Although formally distinct, these control words may all be abbreviated to :End.

Finally, the :GoTo, :Return, :Leave and :Continue control words may be used to conditionally alter the flow of execution within a control structure.

Control words, including qualifiers such as :Else and :ElseIf, may occur only at the beginning of a line or expression in a diamond-separated statement. The only exceptions are :In and :InEach which must appear on the same line within a :For expression.

Key to Notation

The following notation is used to describe Control Structures within this section:

aexp	an expression returning an array,
bexp	an expression returning a single Boolean value (0 or 1),
var	loop variable used by :For control structure,
code	0 or more lines of APL code, including other (nested) control structures,
andor	<i>either</i> one or more :AndIf statements, <i>or</i> one or more :OrIf statements. For further details, see below.

Notes

Code preceding :OrIf and :AndIf

Code that precedes a :OrIf control statement, e.g. code placed between a :If statement and a subsequent :OrIf, will be executed only if the outer condition is false. If instead the outer condition is true, there is no need to execute the :OrIf statement, so it and any preceding lines of code are skipped.

Code that precedes a :AndIf control statement, e.g. code placed between a :If statement and a subsequent :AndIf, will only be executed if the outer condition is true. If instead the outer condition is false, there is no need to execute the :AndIf statement, so it and any preceding lines of code are skipped.

The above behaviour may be examined using the Tracer.

A potential use for code before a `:OrIf` or `:AndIf` is to prepare for the conditional test. This preparatory work will only be done if required. For example:

```
:If x      A if x is false, skip everything up to the :EndIf
  y←..A set up stuff for the condition on the next line
  :AndIf y
    do stuff
:EndIf
```

Warning

With the exception of a diamondised statement, a control statement that should **not** be followed by an expression will generate an error if an expression is supplied.

A line in a function consisting of a control statement followed by a `◇` and subsequent expression(s) is not **currently** disallowed but may exhibit unexpected behaviour. In particular, the line will not honour `□STOP` and will not be metered by `□MONITOR`. This syntax is not recommended.

If Statement

`:If bexp`

The simplest `:If` control structure is a single condition of the form:

```
[1]  :If AGE<21
[2]      expr 1
[3]      expr 2
[5]  :EndIf
```

If the test condition (in this case `AGE<21`) is true, the statements between the `:If` and the `:EndIf` will be executed. If the condition is false, none of these statements will be run and execution resumes after the `:EndIf`. Note that the test condition to the right of `:If` must return a single element Boolean value 1 (true) or 0 (false).

`:If` control structures may be considerably more complex. For example, the following code will execute the statements on lines [2-3] if `AGE<21` is 1 (true), **or alternatively**, the statement on line [6] if `AGE<21` is 0 (false).

```
[1]  :If AGE<21
[2]      expr 1
[3]      expr 2
[5]  :Else
[6]      expr 3
[7]  :EndIf
```

Instead of a single condition, it is possible to have multiple conditions using the `:ElseIf` control word. For example:

```
[1]  :If WINEAGE<5
[2]      'Too young to drink'
[5]  :ElseIf WINEAGE<10
[6]      'Just Right'
[7]  :ElseIf WINEAGE<15
[8]      'A bit past its prime'
[9]  :Else
[10]      'Definitely over the hill'
[11] :EndIf
```

Notice that APL executes the expression(s) associated with the **first** condition that is true or those following the `:Else` if **none** of the conditions are true.

The `:AndIf` and `:OrIf` control words may be used to define a block of conditions and so refine the logic still further. You may qualify an `:If` or an `:ElseIf` with one or more `:AndIf` statements **or** with one or more `:OrIf` statements. You may not however mix `:AndIf` and `:OrIf` in the same conditional block. For example:

```
[1]  :If WINE.NAME='Chateau Lafitte'
[2]  :AndIf WINE.YEAR∈1962 1967 1970
[3]      'The greatest?'
[4]  :ElseIf WINE.NAME='Chateau Latour'
[5]  :Orif WINE.NAME='Chateau Margaux'
[6]  :Orif WINE.PRICE>100
[7]      'Almost as good'
[8]  :Else
[9]      'Everyday stuff'
[10] :EndIf
```

Please note that in a `:If` control structure, the conditions associated with each of the condition blocks are executed in order until an entire condition block evaluates to true. At that point, the APL statements following this condition block are executed. None of the conditions associated with any other condition block are executed. Furthermore, if an `:AndIf` condition yields 0 (false), it means that the entire block must evaluate to

false so the system moves immediately on to the next block without executing the other conditions following the failing `:AndIf`. Likewise, if an `:OrIf` condition yields 1 (true), the entire block is at that point deemed to yield true and none of the following `:OrIf` conditions in the same block are executed.

While Statement

`:While bexp`

The simplest `:While` loop is :

```
[1]  I←0
[2]  :While I<100
[3]      expr1
[4]      expr2
[5]      I←I+1
[6]  :EndWhile
```

Unless `expr1` or `expr2` alter the value of `I`, the above code will execute lines [3-4] 100 times. This loop has a single condition; the value of `I`. The purpose of the `:EndWhile` statement is solely to mark the end of the iteration. It acts the same as if it were a branch statement, branching back to the `:While` line.

An alternative way to terminate a `:While` structure is to use a `:Until` statement. This allows you to add a second condition. The following example reads a native file sequentially as 80-byte records until it finds one starting with the string 'Widget' or reaches the end of the file.

```
[1]  I←0
[2]  :While I<[NSIZE] -1
[3]      REC←[NREAD] -1 82 80
[4]      I←I+pREC
[5]  :Until 'Widget'≡6pREC
```

Instead of single conditions, the tests at the beginning and end of the loop may be defined by more complex ones using `:AndIf` and `:OrIf`. For example:

```
[1]  :While 100>i
[2]  :AndIf 100>j
[3]      i j←foo i j
[4]  :Until 100<i+j
[5]  :OrIf i<0
[6]  :OrIf j<0
```

In this example, there are complex conditions at both the start and the end of the iteration. Each time around the loop, the system tests that both `i` and `j` are less than

or equal to 100. If either test fails, the iteration stops. Then, after *i* and *j* have been recalculated by `foo`, the iteration stops if *i* + *j* is equal to or greater than 100, or if either *i* or *j* is negative.

Repeat Statement

:Repeat

The simplest type of `:Repeat` loop is as follows. This example executes lines [3-5] 100 times. Notice that as there is no conditional test at the beginning of a `:Repeat` structure, its code statements are executed at least once.

```
[1]  I←0
[2]  :Repeat
[3]      expr1
[4]      expr2
[5]      I←I+1
[6]  :Until I=100
```

You can have multiple conditional tests at the end of the loop by adding `:AndIf` or `:OrIf` expressions. The following example will read data from a native file as 80-character records until it reaches one beginning with the text string 'Widget' or reaches the end of the file.

```
[1]  :Repeat
[2]      REC←NREAD -1 82 80
[3]  :Until 'Widget'≡6pREC
[4]  :OrIf 0=pREC
```

A `:Repeat` structure may be terminated by an `:EndRepeat` (or `:End`) statement in place of a conditional expression. If so, your code must explicitly jump out of the loop using a `:Leave` statement or by branching. For example:

```
[1]  :Repeat
[2]      REC←NREAD -1 82 80
[3]      :If 0=pREC
[4]      :OrIf 'Widget'≡6pREC
[5]          :Leave
[6]      :EndIf
[7]  :EndRepeat
```

For Statement

:For var :In[Each] aexp

Single Control Variable

The `:For` loop is used to execute a block of code for a series of values of a particular control variable. For example, the following would execute lines [2-3] successively for values of `I` from 3 to 5 inclusive:

```
[1]   :For I :In 3 4 5
[2]       expr1 I
[3]       expr2 I
[4]   :EndFor
```

The way a `:For` loop operates is as follows. On encountering the `:For`, the expression to the right of `:In` is evaluated and the result stored. This is the *control array*. The *control variable*, named to the right of the `:For`, is then assigned the first value in the control array, and the code between `:For` and `:EndFor` is executed. On encountering the `:EndFor`, the control variable is assigned the next value of the control array and execution of the code is performed again, starting at the first line after the `:For`. This process is repeated for each value in the control array.

Note that if the control array is empty, the code in the `:For` structure is not executed. Note too that the control array may be any rank and shape, but that its elements are assigned to the control variable in ravel order.

The control array may contain any type of data. For example, the following code resizes (and compacts) all your component files

```
[1]   :For FILE :In (↓␣FLIB ' ' )~'' ' '
[2]       FILE ␣FTIE 1
[3]       ␣FRESIZE 1
[4]       ␣FUNTIE 1
[5]   :EndFor
```

You may also nest `:For` loops. For example, the following expression finds the timestamp of the most recently updated component in all your component files.

```

[1] TS←0
[2] :For FILE :In (↓□FLIB ' ' )~'' '
[3]     FILE □FTIE 1
[4]     START END←2p□FSIZE 1
[5]     :For COMP :In (START-1)↓END-1
[6]         TS↑←-1↑□FREAD FILE COMP
[7]     :EndFor
[8]     □FUNTIE 1
[9] :EndFor

```

Multiple Control Variables

The `:For` control structure can also take multiple variables. This has the effect of doing a strand assignment each time around the loop.

For example `:For a b c :in (1 2 3)(4 5 6)`, sets `a b c←1 2 3`, first time around the loop and `a b c←4 5 6`, the second time.

Another example is `:For i j :In 1pMatrix`, which sets `i` and `j` to each row and column index of `Matrix`.

:InEach Control Word

```
:For var ... :InEach value ...
```

In a `:For` control structure, the keyword `:InEach` is an alternative to `:In`.

For a single control variable, the effect of the keywords is identical but for multiple control variables the values vector is inverted.

The distinction is best illustrated by the following equivalent examples:

```

:For a b c :In (1 2 3)(3 4 5)(5 6 7)(7 8 9)
    □←a b c
:EndFor

:For a b c :InEach (1 3 5 7)(2 4 6 8)(3 5 7 9)
    □←a b c
:EndFor

```

In each case, the output from the loop is:

```
1 2 3
3 4 5
5 6 7
7 8 9
```

Notice that in the second case, the number of items in the values vector is the same as the number of control variables. A more typical example might be.

```
:For a b c :InEach avec bvec cvec
...
:EndFor
```

Here, each time around the loop, control variable *a* is set to the next item of *avec*, *b* to the next item of *bvec* and *c* to the next item of *cvec*.

Select Statement

:Select aexp

A **:Select** structure is used to execute alternative blocks of code depending upon the value of an array. For example, the following displays 'I is 1' if the variable *I* has the value 1, 'I is 2' if it is 2, or 'I is neither 1 nor 2' if it has some other value.

```
[1] :Select I
[2] :Case 1
[3]   'I is 1'
[4] :Case 2
[5]   'I is 2'
[6] :Else
[7]   'I is neither 1 nor 2'
[8] :EndSelect
```

In this case, the system compares the value of the array expression to the right of the **:Select** statement with each of the expressions to the right of the **:Case** statements and executes the block of code following the one that matches. If none match, it executes the code following the **:Else** (which is optional). Note that comparisons are performed using the **=** primitive function, so the arrays must match exactly. Note also that not all of the **:Case** expressions are necessarily evaluated because the process stops as soon as a matching expression is found.

Instead of a **:Case** statement, you may also use a **:CaseList** statement. If so, the *enclose* of the array expression to the right of **:Select** is tested for membership of the array expression to the right of the **:CaseList** using the **ε** primitive function.

Note also that any code placed between the `:Select` and the first `:Case` or `:CaseList` statements are unreachable; future versions of Dyalog APL may generate an error when attempting to fix functions which include such code.

EXAMPLE

```
[1] :Select ?6 6
[2] :Case 6 6
[3]   'Box Cars'
[4] :Case 1 1
[5]   'Snake Eyes'
[6] :CaseList 2p"16
[7]   'Pair'
[8] :CaseList (16),"ϕ16
[9]   'Seven'
[10] :Else
[11]   'Unlucky'
[12] :EndSelect
```

With Statement

`:With obj`

`:With` is a control structure that may be used to simplify a series of references to an object or namespace. `:With` changes into the specified namespace for the duration of the control structure, and is terminated by `:End[With]`. `obj` is either the name of or a reference to a namespace. For example, you could update several properties of a Grid object `F.G` as follows:

```
:With F.G
  Values←4 3p0
  RowTitles←'North' 'South' 'East' 'West'
  ColTitles←'Cakes' 'Buns' 'Biscuits'
:EndWith
```

`:With` is analogous to `□CS` in the following senses:

- The namespace argument to `:With` is interpreted relative to the current space.
- With the exception of those with name class 9, local names in the containing defined function continue to be visible in the new space.
- Global references from within the `:With` control structure are to names in the new space.
- Exiting the defined function from within a `:With` control structure causes the space to revert to the one from which the function was called.

On leaving the `:With` control structure, execution reverts to the original namespace. Notice however that the interpreter does not detect branches (`→`) out of the control structure. `:With` control structures can be nested in the normal fashion:

```
[1]   :With 'x'           A Change to #.x
[2]       :With 'y'       A Change to #.x.y
[3]           :With []SE  A Change to []SE
[4]               ...     A ... in []SE
[5]           :EndWith    A Back to #.x.y
[6]       :EndWith       A Back to #.x
[7]   :EndWith           A Back to #
```

Hold Statement

`:Hold tkns`

Whenever more than one thread tries to access the same piece of data or shared resource at the same time, you need some type of synchronisation to control access to that data. This is provided by `:Hold`.

`:Hold` provides a mechanism to control thread entry into a critical section of code. `tkns` must be a simple character vector or scalar, or a vector of character vectors. `tkns` represents a set of "tokens", all of which must be acquired before the thread can continue into the control structure. `:Hold` is analogous to the component file system `PFHOLD` which is used to synchronise access between **processes**. See also *Earlier Release Notes: Fhold*.

Within the whole active workspace, a token with a particular value may be held only once. If the hold succeeds, the current thread *acquires* the tokens and execution continues with the first phrase in the control structure. On exit from the structure, the tokens are released for use by other threads. If the hold fails, because one or more of the tokens is already in use:

1. If there is no `:Else` clause in the control structure, execution of the thread is blocked until the requested tokens become available.
2. Otherwise, acquisition of the tokens is abandoned and execution resumed immediately at the first phrase in the `:Else` clause.

`tkns` can be either a single token:

```
'a'
'Red'
'#.Util'
''
'Program Files'
```

... or a number of tokens:

```
'red' 'green' 'blue'
'doe' 'a' 'deer'
, '' 'abc'
↓␣NL 9
```

Pre-processing removes trailing blanks from each token before comparison, so that, for example, the following two statements are equivalent:

```
:Hold 'Red' 'Green'
:Hold ↓2 5p 'Red Green'
```

Unlike `␣HOLD`, a thread does not release all existing tokens before attempting to acquire new ones. This enables the nesting of holds, which can be useful when multiple threads are concurrently updating parts of a complex data structure.

In the following example, a thread updates a critical structure in a child namespace, and then updates a structure in its parent space. The holds will allow all "sibling" namespaces to update concurrently, but will constrain updates to the parent structure to be executed one at a time.

```
:Hold ␣CS''           A Hold child space
...                 A Update child space
:Hold ##.␣CS''       A Hold parent space
...                 A Update Parent space
:EndHold
...
:EndHold
```

However, with the nesting of holds comes the possibility of a "deadlock". For example, consider the two threads:

Thread 1	Thread 2
<pre>:Hold 'red' ... :Hold 'green' ... :EndHold :EndHold</pre>	<pre>:Hold 'green' ... :Hold 'red' ... :EndHold :EndHold</pre>

In this case if both threads succeed in acquiring their first hold, they will both block waiting for the other to release its token.

If this deadlock situation is detected acquisition of the tokens is abandoned. Then:

1. If there is an `:Else` clause in the control structure, execution jumps to the `:Else` clause.
2. Otherwise, APL issues an error (1008) DEADLOCK.

You can avoid deadlock by ensuring that threads always attempt to acquire tokens in the same chronological order, and that threads never attempt to acquire tokens that they already own.

Note that token acquisition for any particular `:Hold` is atomic, that is, either *all* of the tokens or *none* of them are acquired. The following example *cannot* deadlock:

Thread 1	Thread 2
<pre> :Hold 'red' ... :Hold 'green' ... :EndHold :EndHold </pre>	<pre> :Hold 'green' 'red' ... :EndHold </pre>

EXAMPLES

`:Hold` could be used for example, during the update of a complex data structure that might take several lines of code. In this case, an appropriate value for the token would be the name of the data structure variable itself, although this is just a programming convention: the interpreter does not associate the token value with the data variable.

```

:Hold 'Struct'
...           A Update Struct
Struct ← ...
:EndHold

```

The next example guarantees exclusive use of the current namespace:

```

:Hold □CS''   A Hold current space
...
:EndHold

```

The following example shows code that holds two positions in a vector while the contents are exchanged.

```
:Hold 0 to fm
  :If >/vec[fm to]
    vec[fm to]←vec[to fm]
  :End
:End
```

Between obtaining the next available file tie number and using it:

```
:Hold '[]FNUMS'
  tie←1+[/0,[]FNUMS
  fname []FSTIE tie
:End
```

The above hold is not necessary if the code is combined into a single line:

```
fname []FSTIE tie←1+[/0,[]FNUMS
```

or,

```
tie←fname []FSTIE 0
```

Note that `:Hold`, like its component file system counterpart `[]FHOLD`, is a device to enable *co-operating* threads to synchronise their operation.

`:Hold` does not *prevent* threads from updating the same data structures concurrently, it prevents threads only from `:Hold`ing the same tokens.

High-Priority Callbacks

`:Hold` with a non-zero number of tokens is not permitted in a high-priority callback and an attempt to use it will cause the error:

```
DOMAIN ERROR: Cannot :Hold within high priority callback
```

See *Interfaces: High Priority Callbacks*.

Trap Statement	:Trap ecode
-----------------------	--------------------

`:Trap` is an error trapping mechanism that can be used in conjunction with, or as an alternative to, the `⌈TRAP` system variable. It is equivalent to APL2's `⌈EA`, except that the code to be executed is not restricted to a single expression and is not contained within quotes (and so is slightly more efficient).

`ecode` is an integer scalar or vector containing the list of event codes which are to be *handled* during execution of the segment of code between the `:Trap` and `:End[Trap]` statements. Note that event codes 0 and 1000 are wild cards that means *any* event code in a given range. See [Section 8.1.2](#).

Operation

The segment of code immediately following the `:Trap` keyword is executed. On completion of this segment, if no error occurs, control passes to the code following `:End[Trap]`.

If an error occurs which is not specified by `ecode`, it is processed by outer `:Traps`, `⌈TRAPS`, or by the default system processing in the normal fashion.

If an error occurs, whose event code matches `ecode`:

- If the error occurred within a sub-function, the system cuts the state indicator back to the function containing the `:Trap` keyword. In this respect, `:Trap` behaves like `⌈TRAP` with a 'C' qualifier.
- If the `:Trap` segment contains a `:Case[List] ecode` statement whose `ecode` matches the event code of the error that has occurred, execution continues from the statement following that `:Case[List] ecode`.
- Otherwise, if the `:Trap` segment contains a `:Else` statement, execution continues from the first statement following the `:Else` statement.
- Otherwise, execution continues from the first statement following the `:End[Trap]` and no error processing occurs.

Note that the error trapping is in effect **only** during execution of the initial code segment. When a trapped error occurs, further error trapping is immediately disabled (or surrendered to outer level `:Traps` or `⌈TRAPS`). In particular, the error trap is no longer in effect during processing of `:Case[List]`'s argument or in the code following the `:Case[List]` or `:Else` statement. This avoids the situation sometimes encountered with `⌈TRAP` where an infinite "trap loop" occurs.

Note that the statement `:Trap 0` results in no errors being trapped.

EXAMPLES

```

    ▽ lx
[1]   :Trap 1000           A Cutback and exit on interrupt
[2]       Main ...
[3]   :EndTrap
    ▽

    ▽ ftie←Fcreate file      A Create null component file
[1]   :Trap 22             A Trap FILE NAME ERROR
[2]       ftie←file □FCREATE 0 A Try to create file.
[3]   :Else
[4]       ftie←file □FTIE 0   A Tie the file.
[5]       file □FERASE ftie   A Drop the file.
[6]       file □FCREATE ftie  A Create new file.
[7]   :EndTrap
    ▽

    ▽ lx A Distinguish various cases
[1]   :Trap 0 1000
[2]       Main ...
[3]   :Case 1002
[4]       'Interrupted ...'
[5]   :CaseList 1 10 72 76
[6]       'Not enough resources'
[7]   :CaseList 17+120
[8]       'File System Problem'
[9]   :Else
[10]      'Unexpected Error'
[11]   :EndTrap
    ▽

```

Note that :Traps can be nested:

```

    ▽ ntie←Ntie file                                A Tie native file
[1]   :Trap 22                                      A Trap FILE NAME ERROR
[2]       ntie←file 0NTIE 0                        A Try to tie file
[3]   :Else
[4]       :Trap 22                                A Trap FILE NAME ERROR
[5]       ntie←(file, '.txt')0NTIE 0              A Try with .txt extn
[6]   :Else
[7]       ntie←file 0NCREATE 0                    A Create null file.
[8]   :EndTrap
[9]   :EndTrap
    ▽
    
```

GoTo Statement

:GoTo aexp

A :GoTo statement is a direct alternative to → (branch) and causes execution to jump to the line specified by the first element of aexp.

The following are equivalent. See *Dyalog APL Language: Branch* for further details.

```

→Exit
:GoTo Exit

→(N<I←I+1)/End
:GoTo (N<I←I+1)/End

→1+⊂LC
:GoTo 1+⊂LC

→10
:GoTo 10
    
```

Return Statement

:Return

A :Return statement causes a function to terminate and has exactly the same effect as →0.

The :Return control word takes no argument.

A :Return statement may occur anywhere in a function or operator.

Leave Statement**:Leave**

A `:Leave` statement is used to explicitly terminate the execution of a block of statements within a `:For`, `:Repeat` or `:While` control structure.

The `:Leave` control word takes no argument.

Continue Statement**:Continue**

A `:Continue` statement starts the next iteration of the immediately surrounding `:For`, `:Repeat` or `:While` control loop.

When executed within a `:For` loop, the effect is to start the body of the loop with the next value of the iteration variable.

When executed within a `:Repeat` or `:While` loop, if there is a trailing test that test is executed and, if the result is true, the loop is terminated. Otherwise the leading test is executed in the normal fashion.

Section Statement**:Section**

Functions and scripted objects (classes, namespaces etc.) can be subdivided into Sections with `:Section` and `:EndSection` statements. Both statements may be followed by an optional and arbitrary name or description. The purpose is to split the function up into sections that you can open and close in the Editor, thereby aiding readability and code management. Sections have no effect on the execution of the code, but must follow the nesting rules of other control structures.

Disposable Statement**:Disposable**

The Dyalog interface to .NET involves the creation and removal of .NET objects. Many such objects are *managed* in that the .NET Common Language RunTime (CLR) automatically releases the memory allocated to the object when that object is no longer used. However, it is not possible to predict when the CLR garbage collection will occur. Furthermore, the garbage collector has no knowledge of *unmanaged* resources such as window handles, or open files and streams.

Typically, .NET classes implement a special interface called `IDisposable` which provides a standard way for applications to release memory and other resources when an instance is removed. Furthermore, the C# language has the `using` keyword, which "Provides a convenient syntax that ensures the correct use of `IDisposable` objects."

The `:Disposable` array statement in Dyalog APL provides a similar facility to C#'s `using`. `array` may be a scalar or vector of namespace references.

When the block is exited, any .NET objects in `array` that implement `IDisposable` will have `IDisposable.Dispose` called on them.

Note that exit includes normal exit as the code drops through `:EndDisposable`, or if an error occurs and is trapped, or if branch ($\rightarrow$) is used to exit the block, or anything else.

See also: `.Disposing of .NET Objects`.

Example (Normal Exit)

```
:Disposable f←NEW Font
.
.
:EndDisposable
```

In the above example, when the `:EndDisposable` statement is reached, the system disposes of the `Font` object `f` (and all the resources associated with it) by calling `(IDisposable)f.Dispose()`. A subsequent reference to `f` would generate `VALUE ERROR`.

Example (Normal Exit)

```
:Disposable fonts←NEW Font Font
.
.
:EndDisposable
```

In the above example, `Dispose()` is called on **each** of the `Font` objects in `fonts` during the processing of `:EndDisposable`.

Example (Branch Exit)

```

:Disposable fonts←[]NEW "Font Font
.
→0
.
:EndDisposable

```

In this example, Dispose() is called on the Font objects in `fonts` during the processing of the branch statement `→0`.

Example (TrapExit)

```

:trap 0

  :Disposable fonts←[]NEW "Font Font
  .
  ÷0
  .
  :EndDisposable

:else

  []←'failed'

:endif

```

Here, the objects are disposed of when the DOMAIN ERROR generated by the expression `÷0` causes the stack to be cut back to the `:Else` clause. At this point (just before the execution of the `:Else` clause) the name class of `fonts` becomes 0.

3.1.10 APL Line Editor

The APL Line Editor described herein is included for completeness and for adherence to the ISO APL standard. Dyalog recommends the use of the more powerful Editor and Tracer in preference to the APL Line Editor. Full details of these facilities can be found in the UI Guides for your version of Dyalog APL, as well as in the descriptions of `⌈ED` and `⌋ED` which appear in the *Dyalog APL Language Reference Guide*.

Using the APL Line Editor, functions and operators are defined by entering Definition Mode. This mode is opened and closed by the Del symbol, `▽`. Within this mode, all evaluation of input is deferred. The standard APL line editor (described below) is used to create and edit operations within definition mode.

Operations may also be defined using the system function `⌈FX` (implicit in a `⌈ED fix`) which acts upon the canonical (character), vector, nested or object representation form of an operation. (See *Dyalog APL Language: Fx* for details.)

Functions may also be created dynamically or by function assignment.

The line editor recognises three forms for the opening request.

Creating Defined Operation

The opening `▽` symbol is followed by the header line of a defined operation. Redundant blanks in the request are permitted except within names. If acceptable, the editor prompts for the first statement of the operation body with the line-number 1 enclosed in brackets. On successful completion of editing, the defined operation becomes the active definition in the workspace.

EXAMPLE

```

    ▽R←FOO
[1] R←10
[2] ▽

    FOO
10
```

The given operation name must not have an active referent in the workspace, otherwise the system reports `defn error` and the system editor is not invoked:

```

)VARs
SALES X Y

▽R←SALES Y
defn error

```

The header line of the operation must be syntactically correct, otherwise the system reports `defn error` and the system editor is not invoked:

```

▽R←A B C D:G
defn error

```

Listing Defined Operation

The ▽ symbol followed by the name of a defined operation and then by a closing ▽, causes the display of the named operation. Omitting the function name causes the suspended operation (that is, the one at the top of the state indicator) to be displayed and opened for editing.

EXAMPLE

```

▽FOO▽
▽ R←FOO
[1] R←10
▽

)SI
#.FOO[1] *

▽
▽ R←FOO
[1] R←10
[2]

```

Editing Active Defined Operation

Definition mode is entered by typing ▽ followed optionally by a name and editing directive.

The ▽ symbol on its own causes the suspended operation (that is, the one at the top of the state indicator) to be displayed. The editor then prompts for a statement or editing directive with a line-number one greater than the highest line-number in the function.

If the state indicator is empty, the system reports `defn error` and definition mode is not entered.

The `▽` symbol followed by the name of an active defined operation causes the display of the named operation. The editor then prompts for input as described above. If the name given is not the name of an active referent in the workspace, the opening request is taken to be the creation of a new operation as described in paragraph 1. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to displaying the operation. If the name refers to a locked operation, the system reports `defn error` and definition mode is not entered.

The `▽` symbol followed by the name of an active defined operation and an editing directive causes the operation to be opened for editing and the editing directive actioned. If the editing directive is invalid, it is ignored by the editor which then prompts with a line-number one greater than the highest line-number in the operation. If the name refers to a pendent operation, the editor issues the message `warning pendent operation` prior to actioning the editing directive. If the name refers to a locked operation, the system reports `defn error` and definition mode is not entered.

On successful completion of editing, the defined operation becomes the active definition in the workspace which may replace an existing version of the function. Monitors, and stop and trace vectors are removed.

EXAMPLE

```

    ▽FOO[2]
[2]  R←R*2
[3]  ▽

```

Editing Directives

Editing directives, summarised in Figure 2(iv) are permitted as the first non-blank characters either after the operation name on opening definition mode for an active defined function, or after a line-number prompt.

Syntax	Description
▽	Closes definition mode
[]	Displays the entire operation
[]n	Displays the operation starting at line n
[n]	Displays only line n
[Δn]	Deletes line n
[nΔm]	Deletes m lines starting at line n
[n]	Prompts for input at line n
[n]s	Replaces or inserts a statement at line n
[n□m]	Edits line n placing the cursor at character position m where an Edit Control Symbol performs a specific action.

Line Numbers

Line numbers are associated with lines in the operation. Initially, numbers are assigned as consecutive integers, beginning with [0] for the header line. The number associated with an operation line remains the same for the duration of the definition mode unless altered by editing directives. Additional lines may be inserted by decimal numbering. Up to three places of decimal are permitted. On closing definition mode, operation lines are re-numbered as consecutive integers.

The editor always prompts with a line number. The response may be a statement line or an editing directive. A statement line replaces the existing line (if there is one) or becomes an additional line in the operation:

```

    ▽R←A PLUS B
[ 1 ]  R←A+B
[ 2 ]

```

Position

The editing directive [n], where n is a line number, causes the editor to prompt for input at that line number. A statement or another editing directive may be entered. If a statement is entered, the next line number to be prompted is the previous number incremented by a unit of the display form of the last decimal digit. Trailing zeros are not displayed in the fractional part of a line number:

```
[2]    [0.8]
[0.8]  R MONADIC OR DYADIC +
[0.9]  R A ↔ OPTIONAL ARGUMENT
[1]
```

The editing directive `[n]s`, where `n` is a line number and `s` is a statement, causes the statement to replace the current contents of line `n`, or to insert line `n` if there is none:

```
[1] [0] R←{A} PLUS B
[1]
```

Delete

The editing directive `[Δn]`, where `n` is a line number, causes the statement line to be deleted. The form `[nΔm]`, where `n` is a line number and `m` is a positive integer, causes `m` consecutive statement lines starting from line number `n` to be deleted.

Edit

The editing directive `[n□m]`, where `n` is a line number and `m` is an integer number, causes line number `n` to be displayed and the cursor placed beneath the `m` character on a new line for editing. The response is taken to be edit control symbols selected from:

/	to delete the character immediately above the symbol.
1 to 9	to insert from 1 to 9 spaces immediately prior to the character above the digit.
A to Z	to insert multiples of 5 spaces immediately prior to the character above the letter, where A = 5, B = 10, C = 15 and so forth.
,	to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then re-display the line for further editing with the text inserted and any preceding deletions or space insertions also effected.
.	to insert the text after the comma, including explicitly entered trailing spaces, prior to the character above the comma, and then complete the edit of the line with the text inserted and any preceding deletions or space insertions also effected.

Invalid edit symbols are ignored. If there are no valid edit symbols entered, or if there are only deletion or space insertion symbols, the statement line is re-displayed with characters deleted and spaces inserted as specified. The cursor is placed at the first inserted space position or at the end of the line if none. Characters may be added to the line which is then interpreted as seen.

The line number may be edited.

EXAMPLES

```
[1] [1□7]
[1] R←A+B
    ,→(0=□NC'A')ρ1←□LC ◇
[1] →(0=□NC'A')ρ1←□LC ◇ R←A+B
                                .◇→END

[2] R←B
[3] END:
[4]
```

The form [n□0] causes the line number n to be displayed and the cursor to be positioned at the end of the displayed line, omitting the edit phase.

Display

The editing directive [□] causes the entire operation to be displayed. The form [□n] causes all lines from line number n to be displayed. The form [n□] causes only line number n to be displayed:

```
[4] [0□]
[0] R←{A} PLUS B
[0]
[0] [□]
[0] R←{A} PLUS B
[0.1] A MONADIC OR DYADIC +
[1] →(0=□NC'A')ρ1+□LC ◇ R←A+B ◇→END
[2] R←B
[3] 'END:
[4]
```

Close Definition Mode

The editing directive `▽` causes definition mode to be closed. The new definition of the operation becomes the active version in the workspace. If the name in the operation header (which may or may not be the name used to enter definition mode) refers to a pendent operation, the editor issues the message `warning pendent operation` before exiting. The new definition becomes the active version, but the original one will continue to be referenced until the operation completes or is cleared from the state indicator.

If the name in the operation header is the name of a visible variable or label, the editor reports `defn error` and remains in definition mode. It is then necessary to edit the header line or quit.

If the header line is changed such that it is syntactically incorrect, the system reports `defn error`, and re-displays the line leaving the cursor beyond the end of the text on the line. Backspace/linefeed editing may be used to alter or cancel the change:

```
[3]  [0] - display line 0
[0]  R←{A} PLUS B
[0]  R←{A} PLUS B;G;H - put syntax error in line 0
defn error
[0]  R←{A} PLUS B;G;H - line redisplayed
      ;G;H - backspace/linefeed editing
[1]
```

Local names may be repeated. However, the line editor reports warning messages as follows:

1. If a name is repeated in the header line, the system reports "warning duplicate name" immediately.
2. If a label has the same name as a name in the header line, the system reports "warning label name present in line 0" on closing definition mode.
3. If a label has the same name as another label, the system reports "warning duplicate label" on closing definition mode.
4. If a name is repeated in the header line, the system reports "warning duplicate name" immediately.
5. If a label has the same name as a name in the header line, the system reports "warning label name present in line 0" on closing definition mode.
6. If a label has the same name as another label, the system reports "warning duplicate label" on closing definition mode.

Improper syntax in expressions within statement lines of the function is not detected by the system editor with the following exceptions:

- If the number of opening parentheses in each entire expression does not equal the number of closing parentheses, the system reports "warning unmatched parentheses", but accepts the line.
- If the number of opening brackets in each entire expression does not equal the number of closing brackets, the system reports "warning unmatched brackets", but accepts the line.

These errors are not detected if they occur in a comment or within quotes. Other syntactical errors in statement lines will remain undetected until the operation is executed.

EXAMPLE

```
[4]    R←(A[;1)=2)/±EXP,'×2
warning unmatched parentheses
warning unmatched brackets
[5]
```

Note that there is an imbalance in the number of quotes. This will result in a SYNTAX ERROR when this operation is executed.

Quit Definition Mode

The user may quit definition mode by typing the INTERRUPT character. The active version of the operation (if any) remains unchanged.

3.2 Dfns and Dops

3.2.1 Dfns & Dops

A *dfn* (*dop*)² is an alternative function definition style suitable for defining small to medium sized functions. It bridges the gap between operator expressions: `rank←p∘p` and full "header style" definitions such as:

```
▽ rslt←larg func rarg;local...
```

In its simplest form, a dfn is an APL expression enclosed in curly braces {}, possibly including the special characters α and ω to represent the left and right arguments of the function respectively. For example:

```
{(+/ω)÷ρω} 1 2 3 4      ⍝ Arithmetic Mean (Average)
2.5
3 {ω*÷α} 64             ⍝ αth root
4
```

dfns can be named in the normal fashion:

```
mean←{(+/ω)÷ρω}
mean¨(2 3)(4 5)
2.5 4.5
```

dfns can be defined and used in any context where an APL function may be found, in particular:

- In immediate execution mode as in the examples above.
- Within a defined function or operator.
- As the operand of an operator such as each (¨).
- Within another dfn.
- The last point means that it is easy to define nested local functions.

² The terms dfn and dop refer to a special type of function (or operator) unique to Dyalog. They were originally named dynamic functions and dynamic operators, later abbreviated to Dfns and Dops or D-Fns and D-Ops, but all these terms have been dropped in favour of the current ones.

3.2.2 Multi-Line Dfns

The single expression which provides the result of the dfn may be preceded by any number of assignment statements. Each such statement introduces a name which is local to the function.

For example in the following, the expressions `sum←` and `num←` create **local** definitions `sum` and `num`.

```
mean←{           A Arithmetic mean
  sum←+/ω        A Sum of items
  num←pω         A Number of items
  sum÷num        A Mean
}
```

An assignment to ω is not allowed and will result in an error. For assignment to α , see [Section 3.2.3](#).

Note that dfns may be commented in the usual way using `A`.

When the interpreter encounters a local definition, a new local name is created. The name is shadowed dynamically exactly as if the assignment had been preceded by: `⌈SHADOW name ⋄`.

It is **important** to note the distinction between the two types of statement above. There can be **many** assignment statements, each introducing a new local definition, but only a **single** expression where the result is not assigned. As soon as the interpreter encounters such an expression, it is evaluated and the result returned immediately as the result of the function.

For example, in the following,

```
mean←{           A Arithmetic mean
  sum←+/ω        A Sum of items
  num←pω         A Number of items
  sum,num        A Attempt to show sum,num (wrong)!
  sum÷num        A ... and return result.
}
```

... as soon as the interpreter encounters the expression `sum,num`, the function terminates with the two item result `(sum,num)` and the following line is not evaluated.

To display arrays to the session from within a dfn, you can use the explicit display forms `⌈←` or `⌈←` as in:

```

mean←{           A Arithmetic mean
  sum←+/ $\omega$       A Sum of items
  num← $\rho\omega$        A Number of items
  □←sum,num      A show sum,num.
  sum÷num        A ... and return result.
}

```

Note that local definitions can be used to specify local nested dfns:

```

rms←{           A Root Mean Square
  root←{ $\omega*0.5$     A  $\nabla$  Square root
  mean←{ $(+/ $\omega$ )÷ $\rho\omega$  A  $\nabla$  Mean
  square←{ $\omega\times\omega$  A  $\nabla$  Square
  root mean square  $\omega$ 
}$ 
```

3.2.3 Default Left Argument

The special syntax: $\alpha\leftarrow\text{expr}$ is used to give a default value to the left argument if a dfn is called monadically. For example:

```

root←{           A  $\alpha$ th root
   $\alpha\leftarrow 2$    A default to sqrt
   $\omega*\div\alpha$ 
}

```

The expression to the right of $\alpha\leftarrow$ is evaluated *only* if its dfn is called with no left argument.

Note that the syntax must be exactly $\alpha\leftarrow$, that is, it cannot contain parentheses, and so on.

$\alpha\leftarrow$ must be the first tokens at the beginning of an expression.

Ambivalence

The assignment $\alpha\leftarrow\vdash$ allows an ambivalent function to call an ambivalent sub-function. For example in:

```

foo←{
   $\alpha\leftarrow\vdash$ 
   $\alpha$  goo  $\omega$ 
}

```

If $f\circ o$ is given a left argument, this is passed to $g\circ o$. Otherwise, α is assigned $\vdash$ and the last line is $\vdash\ g\circ o\ \omega$, which is a monadic call on $g\circ o$ followed by the $\vdash$ (Right) of the result of $g\circ o$, which is the same value.

The assignment $\alpha \leftarrow *0$ allows an ambivalent operator to skip the application of an operand to a missing argument. For example in:

```
over←{
  α←*0
  (αα α)ωω(αα ω)
}
```

If the function derived from *over* is given a left argument, this argument is preprocessed by the left operand $\alpha\alpha$ and the result is passed to the right operand $\omega\omega$. Otherwise, α is assigned $*0$ and the last line is $(\alpha\alpha *0)\omega\omega(\alpha\alpha\ \omega)$, which is a monadic call on $\omega\omega$ followed by not applying $\alpha\alpha$ to the result of $\omega\omega$, returning it unmodified.

The assignment $\alpha \leftarrow \omega$ allows a function to act as if the *commute* operator ($\sim$) was applied to it twice:

```
sort←{
  α←ω
  ω[⌵α]
}
```

If *sort* is given a left argument, the right argument is sorted according to the left argument. Otherwise, α is assigned ω and the last line is $\omega[\lceil\omega\rceil]$, which has the right argument sorted according to itself. This is, therefore, equivalent to $\text{sort} \leftarrow \{\omega[\lceil\omega\rceil]\} \sim \sim$ or $\text{sort} \leftarrow \{\alpha[\lceil\omega\rceil]\} \sim$.

3.2.4 Guards

A Guard is a Boolean-single valued expression followed on the right by a `:`. For example:

```
0≡ω:      A Right arg simple scalar
α<0:      A Left arg negative
```

The guard is followed by a single APL expression: the result of the function.

```
ω≥0: ω*0.5    A Square root if non-negative.
```

A dfn may contain any number of guarded expressions each on a separate line (or collected on the same line separated by diamonds). Guards are evaluated in turn until

one of them yields a 1. The corresponding expression to the right of the guard is then evaluated as the result of the function.

If an expression occurs without a guard, it is evaluated immediately as the default result of the function. For example:

```
sign←{
  ω>0: '+ve'      A Positive
  ω=0: 'zero'     A zero
  '-ve'          A Negative (Default)
}
```

Local definitions and guards can be interleaved in any order.

Note again that any code following the first unguarded expression (which terminates the function) could never be executed and would therefore be redundant.

See also [Section 3.2.7](#).

3.2.5 Shy Result

Dfns are usually 'pure' functions that take arguments and return explicit results. Occasionally, however, the main purpose of the function might be a side-effect such as the display of information in the session, or the updating of a file, and the value of a result, a secondary consideration. In such circumstances, you might want to make the result 'shy', so that it is discarded unless the calling context requires it. This can be achieved by assigning a dummy variable after a (true) guard:

```
log←{
  tie←α □FSTIE 0      A Append ω to file α.
  cno←ω □FAPPEND tie  A tie number for file,
  tie←□FUNTIE tie     A new component number,
  1:rslt←cno          A untie file,
                     A comp number, shy result.
}
```

3.2.6 Lexical Name Scope

When an inner (nested) dfn refers to a name, the interpreter searches for it by looking outwards through enclosing dfns, rather than searching back along the state indicator. This regime, which is more appropriate for nested functions, is said to employ **lexical scope** instead of APL's usual **dynamic scope**. This distinction becomes apparent only if a call is made to a function defined at an outer level. For the more usual inward calls, the two systems are indistinguishable.

For example, in the following function, variable `type` is defined both within `which` itself and within the inner function `fn1`. When `fn1` calls outward to `fn2` and `fn2` refers to `type`, it finds the outer one (with value `'lexical'`) rather than the one defined in `fn1`:

```
which←{
    type←'lexical'

    fn1←{
        type←'dynamic'
        fn2 ω
    }

    fn2←{
        type ω
    }

    fn1 ω
}

which'scope'
lexical scope
```

3.2.7 Error-Guards

An **error-guard** is (an expression that evaluates to) a vector of error numbers (see [Section 8.1.2](#)), followed by the digraph: `::`, followed by an expression, the *body* of the guard, to be evaluated as the result of the function. For example:

```
11 5 :: ω×0 A Trap DOMAIN and LENGTH errors.
```

In common with `:Trap` and `□TRAP`, error numbers 0 and 1000 are catch-alls for synchronous errors and interrupts respectively.

When an error is generated, the system searches dynamically through the calling functions for an error-guard that matches the error. If one is found, the execution environment is unwound to its state immediately *prior* to the error-guard's execution and the body of the error-guard is evaluated as the result of the function. This means that, during evaluation of the body, the guard is no longer in effect and so the danger of a hang caused by an infinite "trap loop", is avoided.

Notice that you can provide "cascading" error trapping in the following way:

```

0::try_2nd
0::try_1st
expr

```

In this case, if `expr` generates an error, its immediately preceding: `0::` catches it and evaluates `try_1st` leaving the remaining error-guard in scope. If `try_1st` fails, the environment is unwound once again and `try_2nd` is evaluated, this time with no error-guards in scope.

See also [Section 3.2.4](#).

EXAMPLES

`Open` returns a handle for a component file. If the exclusive tie fails, it attempts a share-tie and if this fails, it creates a new file. Finally, if all else fails, a handle of 0 is returned.

```

open←{
    0::0
    22::ω □FCREATE 0
    24 25::ω □FSTIE 0
    ω □FTIE 0
}

```

A Handle for component file ω .
A Fails:: return 0 handle.
A FILE NAME:: create new one.
A FILE TIED:: try share tie.
A Attempt to open file.

An error in `div` causes it to be called recursively with *improved* arguments.

```

div←{
    α←1
    5::↑▽/↓↑α ω
    11::α ▽ ω+ω=0
    α÷ω
}

```

A Tolerant division:: $\alpha \div 0 \rightarrow \alpha$.
A default numerator.
A LENGTH:: stretch to fit.
A DOMAIN:: increase divisor.
A attempt division.

Notice that some arguments may cause `div` to recur twice:

```

6 4 2 div 3 2
→ 6 4 2 div 3 2 0
→ 6 4 2 div 3 2 1
→ 2 2 2

```

The final example shows the unwinding of the local environment before the error-guard's body is evaluated. Local name `trap` is set to describe the domain of its following error-guard. When an error occurs, the environment is unwound to expose `trap`'s statically correct value.

```

add←{
  trap←'domain' ♦ 11::trap
  trap←'length' ♦ 5::trap
  α+ω
}

2 add 3      A Addition succeeds
5

2 add 'three' A DOMAIN ERROR generated.
domain

2 3 add 4 5 6 A LENGTH ERROR generated.
length

```

Note

Following the setting of an error-guard, subsequent function calls will disable tail call optimisation:

```

{
  22::'Oops!'      A this error-guard means that ...
  tie←ω □FTIE 0
  subfn tie        A ... tail call not optimised
}

```

One way to maintain the tail call optimisation in the presence of an error-guard is to isolate it within an inner function:

```

{
  tie←{
    22::0      A error-guard local to inner fn
    ω □FTIE 0
  }ω
  tie=0:'Oops!'
  subfn tie    A ... so this is a tail call
}

```

3.2.8 Dops

The operator equivalent of a dfn is distinguished by the presence of either of the compound symbols $\alpha\alpha$ or $\omega\omega$ anywhere in its definition.

The syntax of a dop is:

- monadic – $\alpha \ (\alpha \alpha \ \text{op}) \ \omega$
- dyadic – $\alpha \ (\alpha \alpha \ \text{op} \ \omega \omega) \ \omega$

where $\alpha \alpha$ and $\omega \omega$ are the left and right operands (functions or arrays) respectively, and α and ω are the arguments of the derived function.

EXAMPLE

The following monadic `each` operator applies its function operand only to unique elements of its argument. It then distributes the result to match the original argument. This can deliver a performance improvement over the primitive `each` (``) operator if the operand function is costly and the argument contains a significant number of duplicate elements. Note however, that if the operand function causes side effects, the operation of `dop` and primitive versions will be different.

```
each←{
  shp←pω      A Fast each:
  vec←,ω      A Shape and ...
  nub←uvec    A ... ravel of arg.
  res←αα"nub  A Vector of unique elements.
  idx←nub⊔vec A Result for unique elts.
  shp⊔idx>"<res A Indices of arg in nub ...
}             A ... distribute result.
```

The dyadic `else` operator applies its left (else right) operand to its right argument depending on its left argument.

```
else←{
  α: αα ω      A True: apply Left operand
  ωω ω        A Else, .. Right ..
}
0 1 [else]" 2.5  A Try both false and true.
```

2 3

3.2.9 Recursion

A recursive dfn can refer to itself using its name explicitly, but because we allow unnamed functions, we also need a special symbol for implicit self-reference: '▽'. For example:

```
fact←{           A Factorial ω.
  ω≤1: 1         A Small ω, finished,
  ω×▽ ω-1       A Otherwise recur.
}
```

Implicit self-reference using '▽' has the further advantage that it incurs less interpretative overhead and is therefore quicker. Tail calls using '▽' are particularly efficient.

Recursive dops refer to their derived functions, that is the operator bound with its operand(s) using ▽ or the operator itself using the compound symbol: ▽▽. The first form of self reference is by far the more frequently used.

```
pow←{           A Function power.
  α=0:ω         A Apply function operand α times.
  (α-1)▽ αα ω A αα αα αα ... ω
}
```

EXAMPLE

The following example shows a rather contrived use of the second form of (operator) self reference. The `exp` operator composes its function operand with itself on each recursive call. This gives the effect of an exponential application of the original operand function:

```
exp←{           A Exponential fn application.
  α=0:αα ω      A Apply operand 2*α times.
  (α-1)αα◦αα ▽▽ ω A (αα◦αα)◦( ... ) ... ω
}
succ←{1+ω}      A Successor (increment).
10 succ exp 0
```

1024

EXAMPLE

```
▽pow←{ A Function power.
[1]   α=0:ω A Apply function operand α times.
[2]   (α-1)▽ αα ω A αα αα αα ... ω
[3]   }
▽
4 @ pow 5000
-0.2720968003
```

Example: Pythagorean triples

The following sequence shows an example of combining dfns and dops in an attempt to find Pythagorean triples: (3 4 5) (5 12 13) ...

```

    sqrt←{ω*0.5}           A Square root.

    sqrt 9 16 25
3 4 5

    hyp←{sqrt+ /> ω*2}      A Hypoteneuse of triangle.

    hyp(3 4)(4 5)(5 12)
5 6.403124237 13

    intg←{ω=[ω]}           A Whole number?

    intg 2.5 3 4.5
0 1 0

    pyth←{intg hyp ω}       A Pythagorean pair?

    pyth(3 4)(4 9)(5 12)
1 0 1

    pairs←{,ιω ω}           A Pairs of numbers 1..ω.

    pairs 3
1 1 1 2 1 3 2 1 2 2 2 3 3 1 3 2 3 3

    filter←{(αα ω)/ω}       A Op: ω filtered by αα.

    pyth filter pairs 12    A Pythagorean pairs 1..12
3 4 4 3 5 12 6 8 8 6 9 12 12 5 12 9

```

So far, so good, but we have some duplicates: (6 8) is just double (3 4).

```

rpm←{                                A Relatively prime?
    ω=0:α=1                          A C.f. Euclid's gcd.
    ω ∇ ω|α
}/''                                A Note the /''

rpm(2 4)(3 4)(6 8)(16 27)
0 1 0 1

rpm filter pyth filter pairs 20
3 4 4 3 5 12 8 15 12 5 15 8

```

We can use an operator to combine the tests:

```

and←{                                A Lazy parallel 'And'.
    mask←αα ω                        A Left predicate selects...
    mask\ωω mask/ω                  A args for right predicate.
}

pyth and rpm filter pairs 20
3 4 4 3 5 12 8 15 12 5 15 8

```

Better, but we still have some duplicates: (3 4) (4 3)

```

less←{</>ω}
less(3 4)(4 3)
1 0

less and pyth and rpm filter pairs 40
3 4 5 12 7 24 8 15 9 40 12 35 20 21

```

And finally, as promised, triples:

```

{ω,hyp ω}''less and pyth and rpm filter pairs 35
3 4 5 5 12 13 7 24 25 8 15 17 12 35 37 20 21 29

```

A Larger Example

Function `tokens` uses nested local dfns to split an APL expression into its constituent tokens. Note that all calls on the inner functions: `lex`, `acc`, and the unnamed dfn in each token case, are *tail calls*. In fact, the *only* stack calls are those on function: `all`, and the unnamed function: `{ω∇~1ϕω}`, within the "Char literal" case.

```

tokens←{
  alph←A,Á,'_ΔΔ',26↑17↓AV      A Lex of APL src line.
  all←{+/^α∈ω}                  A Alphabet for names.
  acc←{(α,↑/ω)lex>↓/ω}          A No. of leading α∈ω.
  lex←{                           A Accumulate tokens.
    0=ρω:α ⋄ hd←↑ω              A Next char else done.

    hd=' ':α{                    A White Space.
      size←ω all ' '
      α acc size ω
    }ω

    hd∈alph:α{                   A Name
      size←ω all alph,⍬
      α acc size ω
    }ω

    hd∈'⍬':α{                    A System Name/Keyword
      size←ω all hd,alph
      α acc size ω
    }ω

    hd='''':α{                   A Char literal
      size←+/^{\ωv~1ϕω}≠\hd=ω
      α acc size ω
    }ω

    hd∈⍬D,'-':α{                 A Numeric literal
      size←ω all ⍬D,'.-E'
      α acc size ω
    }ω

    hd='A':α acc(ρω)ω            A Comment
    α acc 1 ω                    A Single char token.
  }
  (0ρ<'')lex,ω
}

```

display tokens'xtok←size↑srce A Next token'



3.2.10 Tail Calls

A novel feature of the implementation of dfns is the way in which tail calls are optimised.

When a dfn calls a sub-function, the result of the call may or may not be modified by the calling function before being returned. A call where the result is passed back immediately without modification is termed a tail call.

For example in the following, the first call on function `fact` is a tail call because the result of `fact` is the result of the whole expression, whereas the second call isn't because the result is subsequently multiplied by ω .

<code>($\alpha \times \omega$) fact $\omega - 1$</code>	A Tail call on <code>fact</code> .
<code>$\omega \times$ fact $\omega - 1$</code>	A Embedded call on <code>fact</code> .

Tail calls occur frequently in dfns, and the interpreter optimises them by re-using the current stack frame instead of creating a new one. This gives a significant saving in both time and workspace usage. It is easy to check whether a call is a tail call by tracing it. An embedded call will pop up a new trace window for the called function, whereas a tail call will re-use the current one.

Using tail calls can improve code performance considerably, although at first the technique might appear obscure. A simple way to think of a tail call is as a **branch with arguments**. The tail call, in effect, branches to the first line of the function after installing new values for ω and α .

Iterative algorithms can almost always be coded using tail calls.

In general, when coding a loop, we use the following steps; possibly in a different order depending on whether we want to test at the "top" or the "bottom" of the loop.

1. Initialise loop control variable(s). `⍝ init`
2. Test loop control variable. `⍝ test`
3. Process body of loop. `⍝ proc`
4. Modify loop control variable for next iteration. `⍝ mod`
5. Branch to step 2. `⍝ jump`

For example, in classical APL you might find the following:

```

    ▽ value←limit loop value A init
[1] top:→(□CT>value-limit)/0 A test
[2] value←Next value         A proc, mod
[3] →top                     A jump
    ▽

```

Control structures help us to package these steps:

```

    ▽ value←limit loop value A init
[1] :While □CT<value-limit A test
[2] value←Next value         A proc, mod
[3] :EndWhile                A jump
    ▽

```

Using tail calls:

```

loop←{A init
    □CT>α-ω:ω A test
    α ▽ Next ω A proc, mod, jump
}

```

3.2.11 Restrictions

- Dfns need not return a result. However even a non-result-returning expression will terminate the function, so you can't, for example, call a non-result-returning function from the middle of a dfn.
- You can trace a dfn **only** if it is defined on more than one line. Otherwise it is executed atomically in the same way as an execute (⌘) expression. This deliberate restriction is intended to avoid the confusion caused by tracing a line and seeing nothing change on the screen.
- dfns do not currently support □CS which, if used, generates a NONCE ERROR.
- □SHADOW ignores dfns when looking down the stack for a traditional function (tradfn) in which to make a new local name.
- dfns do not support control structures or branch.
- dfns do not support modified assignment such as `X plus←10` where `X` is an array and `plus` is a function. In this example, both `X` and `plus` would be assigned the value 10.
- □MONITOR does not apply to dfns and dops.

Supplied Workspaces

You can find many samples of dfns and dops in utility workspace `dfns.dws` in the `ws` sub-directory.

Additional examples are in workspaces: `min.dws`, `max.dws`, `tube.dws` and `eval.dws`.

4 Object Oriented Programming

4.1 Introducing Classes

4.1.1 Introducing Classes

A Class is a blueprint from which one or more *Instances* of the Class can be created (instances are sometimes also referred to as *Objects*).

A Class may optionally derive from another Class, which is referred to as its Base Class.

A Class may contain *Methods*, *Properties* and *Fields* (commonly referred to together as *Members*) which are defined within the body of the class script or are inherited from other Classes. This version of Dyalog APL does not support *Events* although it is intended that these will be supported in a future release. However, Classes that are derived from .NET types may generate events using `⋄ ⌈NQ`.

A Class that is defined to derive from another Class automatically acquires the set of Properties, Methods and Fields that are defined by its Base Class. This mechanism is described as inheritance.

A Class may extend the functionality of its Base Class by adding new Properties, Methods and Fields or by substituting those in the Base Class by providing new versions with the same names as those in the Base Class.

Members may be defined to be Private or Public. A Public member may be used or accessed from outside the Class or an Instance of the Class. A Private member is internal to the Class and (in general) may not be referenced from outside.

Although Classes are generally used as blueprints for the creation of instances, a class can have Shared members which can be used without first creating an instance.

Class Names

Class names must adhere to the general rules for naming APL objects, and in addition should not conflict with the name of a .NET Type that is defined in any of the .NET Namespaces on the search path specified by `⌈USING`.

4.1.2 Defining Classes

A Class is defined by a script that may be entered and changed using the editor. A class script may also be constructed from a vector of character vectors, and fixed using `FIX`.

A class script begins with a [Section 4.9.4](#) and ends with a `:EndClass` statement.

For example, using the editor:

```
)CLEAR
clear ws
)ED oAnimal
```

[an edit window opens containing the following skeleton Class script ...]

```
:Class Animal
:EndClass
```

[the user edits and fixes the Class script]

```
)CLASSES
Animal
  NC<'Animal'
9.4
```

4.1.3 Editing Classes

Between the `:Class` and `:EndClass` statements, you may insert any number of function bodies, Property definitions, and other elements. When you fix the Class Script from the editor, these items will be fixed inside the Class namespace.

Note that the contents of the Class Script defines the Class *in its entirety*. You may not add or alter functions by editing them independently and you may not add variables by assignment or remove objects with `EX`.

When you *re-fix* a Class Script using the Editor or with `FIX`, the original Class is discarded and the new definition, as specified by the Script, replaces the old one in its entirety.

Note

Associated with a Class (or an instance of a class) there is a completely separate namespace which *surrounds* the class and can contain functions, variables and so forth that are created by actions external to the class.

For example, if *x* is *not* a public member of the class `MyClass`, then the following expression will insert a variable *x* into the namespace which surrounds the class:

```
MyClass.X←99
```

The namespace is analogous to the namespace associated with a GUI object and will be re-initialised (emptied) whenever the Class is re-fixed. Objects in this parallel namespace are not visible from inside the Class or an Instance of the Class.

For further information, see [Section 4.7.1](#).

4.1.4 Inheritance

If you want a Class to derive from another Class, you simply add the name of that Class to the `:Class` statement using colon+space as a separator.

The following example specifies that `CLASS2` derives from `CLASS1`.

```
:Class CLASS2: CLASS1
:EndClass
```

Note that `CLASS1` is referred to as the *Base Class* of `CLASS2`.

If a Class has a Base Class, it automatically acquires all of the **Public** Properties, Methods and Fields defined for its Base Class unless it replaces them with its own members of the same name. This principle of inheritance applies throughout the Class hierarchy. Note that **Private** members are **not** subject to inheritance.

Warning

When a class is fixed, it keeps a reference (a pointer) to its base class. If the global name of the base class is expunged, the derived class will still have the base class reference, and the base class will therefore be kept *alive* in the workspace. The derived class will be fully functional, but attempts to edit it will fail when it attempts to locate the base class as the new definition is fixed.

At this point, if a new class with the original base class name is created, the derived class has no way of detecting this, and it will continue to use the *old and invisible* version of the base class. Only when the derived class is re-fixed, will the new base class be detected.

If you edit, re-fix or copy an existing base class, APL will take care to patch up the references, but if the base class is expunged first and recreated later, APL is unable to detect the substitution. You can recover from this situation by editing or re-fixing the derived class(es) after the base class has been substituted.

Copying Classes

See *Dyalog APL Language: Copy* and *Dyalog APL Language: Copy*.

Classes that derive from .NET Types

You may define a Class that derives from any of the .NET Types by specifying the name of the .NET Type and including a `:USING` statement that provides a path to the .NET Assembly in which the .NET Type is located.

EXAMPLE

```
:Class APLGreg: GregorianCalendar
:Using System.Globalization
...
:EndClass
```

Classes that derive from the Dyalog GUI

You may define a Class that derives from any of the Dyalog APL GUI objects by specifying the *name* of the Dyalog APL GUI Class in quotes.

For example, to define a Class named `Duck` that derives from a `Poly` object, the Class specification would be:

```
:Class Duck: 'Poly'
:EndClass
```

The Base Constructor for such a Class is the `⌈WC` system function.

4.1.5 Instances

A Class is generally used as a blueprint or model from which one or more Instances of the Class are constructed. Note however that a class can have Shared members which can be used directly without first creating an instance.

You create an instance of a Class using the `⌈NEW` system function which is monadic.

The 1-or 2-item argument to `⌈NEW` contains a reference to the Class and, optionally, arguments for its Constructor function.

When `⌈NEW` executes, it creates a regular APL namespace to contain the Instance, and within that it creates an Instance space, which is populated with any Instance Fields defined by the class (with default values if specified), and pointers to the Instance Method and Property definitions specified by the Class.

If a monadic Constructor is defined, it is called with the arguments specified in the second item of the argument to `⌈NEW`. If `⌈NEW` was called without Constructor arguments, and the class has a niladic Constructor, this is called instead.

The Constructor function is typically used to initialise the instance and may establish variables in the instance namespace.

The result of `⌈NEW` is a reference to the instance namespace. Instances of Classes exhibit the same set of Properties, Methods and Fields that are defined for the Class.

4.2 Constructors

4.2.1 Constructors

A Constructor is a special function defined in the Class script that is to be run when an Instance of the Class is created by `□NEW`. Typically, the job of a Constructor is to initialise the new Instance in some way.

A Constructor is identified by a `:Implements Constructor` statement. This statement may appear anywhere in the body of the function after the function header. The significance of this is discussed below.

Note that it is also *essential* to define the Constructor to be *Public*, with a `:Access Public` statement, because like all Class members, Constructors default to being *Private*. Private Constructors currently have no use or purpose, but it is intended that they will be supported in a future release of Dyalog APL.

A Constructor function may be niladic or monadic and must not return a result.

A Class may specify any number of different Constructors of which one (and only one) may be niladic. This is also referred to as the *default* Constructor.

There may be any number of monadic Constructors, but each must have a differently defined argument list which specifies the number of items expected in the Constructor argument. See [Section 4.2.2](#) for details.

The only way a Constructor function should be invoked is by `□NEW`. See [Section 4.2.6](#) for further details. If you attempt to call a Constructor function from outside its Class, it will cause a `SYNTAX ERROR`. A Constructor function should not call another Constructor function within the same Class, although it will not generate an error. This would cause the Base Constructor to be called twice, with unpredictable consequences.

When `□NEW` is executed *with a 2-item argument*, the appropriate monadic Constructor is called with the second item of the `□NEW` argument.

The niladic (default) Constructor is called when `□NEW` is executed with a 1-item argument, a Class reference alone, or whenever APL needs to create a [Section 1.1.3.10](#) for the Class.

Note that `□NEW` first creates a new instance of the specified Class, and then executes the Constructor inside the instance.

EXAMPLE

The `DomesticParrot` Class defines a Constructor function `egg` that initialises the Instance by storing its name (supplied as the 2nd item of the argument to `NEW`) in a Public Field called `Name`.

```
:Class DomesticParrot: Parrot
  :Field Public Name

  ▽ egg name
    :Implements Constructor
    :Access Public
    Name←name
  ▽
  ...
:EndClass  # DomesticParrot

pol←NEW DomesticParrot 'Polly'
pol.Name
Polly
```

4.2.2 Constructor Overloading

NameList header syntax is used to define different versions of a Constructor each with a different number of parameters, referred to as its *signature*. See [Section 3.1.6](#) for details. The illustrates this principle.

In deciding which Constructor to call, APL matches the shape of the Constructor argument with the signature of each of the Constructors that are defined. If a constructor with the same number of arguments exists (remembering that 0 arguments will match a niladic Constructor), it is called. If there is no exact match, and there is a Constructor with a general signature (an un-parenthesised right argument), it is called. If no suitable constructor is found, a `LENGTH ERROR` is reported.

There may be one and only one constructor with a particular signature.

The only way a Constructor function should be invoked is by `NEW`. See [Section 4.2.6](#) for further details. If you attempt to call a Constructor function from outside its Class, it will cause a `SYNTAX ERROR`. A Constructor function should not call another Constructor function within the same Class, although it will not generate an error. This would cause the Base Constructor to be called twice, with unpredictable consequences.

In the example Class, the following Constructors are defined:

Constructor	Implied argument
Make1	1-item vector
Make2	2-item vector
Make3	3-item vector
Make0	No argument
MakeAny	Any array accepted

In the following examples, the Make function (see for details) displays:

```
<shape of argument> <name of Constructor called><argument>
(see function make)
```

Creating a new Instance of Clover with a 1-element vector as the Constructor argument, causes the system to choose the Make1 Constructor. Note that, although the argument to Make1 is a 1-element vector, this is disclosed as the list of arguments is unpacked into the (single) variable arg1.

```
(NEW Clover(,1)).Con
Make1 1
```

Creating a new Instance of Clover with a 2- or 3-element vector as the Constructor argument causes the system to choose Make2, or Make3 respectively.

```
(NEW Clover(1 2)).Con
2 Make2 1 2
(NEW Clover(1 2 3)).Con
3 Make3 1 2 3
```

Creating an Instance with any other Constructor argument causes the system to choose MakeAny.

```
(NEW Clover(110)).Con
10 MakeAny 1 2 3 4 5 6 7 8 9 10
(NEW Clover(2 2 14)).Con
2 2 MakeAny 1 2
3 4
```

Note that a scalar argument will call MakeAny and not Make1.

```
(NEW Clover 1).Con
MakeAny 1
```

and finally, creating an Instance without a Constructor argument causes the system to choose Make0.

```
(□NEW Clover).Con
Make0 0
```

4.2.3 Niladic (Default) Constructors

A Class may define a niladic Constructor and/or one or more Monadic Constructors. The niladic Constructor acts as the default Constructor that is used when □NEW is invoked without arguments and when APL needs a [Section 1.1.3.10](#).

```
:Class Bird
  :Field Public Species

  ▽ egg spec
    :Access Public Instance
    :Implements Constructor
    Species←spec
  ▽
  ▽ default
    :Access Public Instance
    :Implements Constructor
    Species←'Default Bird'
  ▽
  ▽ R←Speak
    :Access Public
    R←'Tweet, tweet!'
  ▽

:EndClass A Bird
```

The niladic Constructor (in this example, the function `default`) is invoked when □NEW is called without Constructor arguments. In this case, the Instance created is no different to one created by the monadic Constructor `egg`, except that the value of the `Species` Field is set to 'Default Bird'.

```
Birdy←□NEW Bird
Birdy.Species
Default Bird
```

The niladic Constructor is also used when APL needs to make a [Section 1.1.3.10](#) of the Class. For example, in the expression `(3↑Birdy)`, APL has to create two [Section 1.1.3.10](#)

of `Birdy` (one for each of the elements required to pad the array to length 3) and will in fact call the niladic Constructor twice.

In the following statement:

```
TweetyPie←3⇒10↑Birdy
```

The `10↑` (temporarily) creates a 10-element array comprising the single entity `Birdy` padded with 9 fill-elements of Class `Bird`. To obtain the 9 fill-elements, APL calls the niladic Constructor 9 times, one for each separate prototypical Instance that it is required to make.

```
TweetyPie.Species
Default Bird
```

4.2.4 Empty Arrays of Instances: Why?

In APL it is natural to use *arrays* of Instances. For example, consider the following example.

```
:Class Cheese
  :Field Public Name←''
  :Field Public Strength←0
  ▽ make2(name strength)
    :Access Public
    :Implements Constructor
    Name Strength←name strength
  ▽
  ▽ make1 name
    :Access Public
    :Implements Constructor
    Name Strength←name 1
  ▽
  ▽ make_excuse
    :Access Public
    :Implements Constructor
    ⍎←'The cat ate the last one!'
  ▽
:EndClass
```

We might create an array of Instances of the Cheese Class as follows:

```
cdata←('Camembert' 5)('Caephillly' 2) 'Mild Cheddar'
cheeses←{⍎NEW Cheese ω}¨cdata
```

Suppose we want a range of medium-strength cheese for our cheese board.

```
board←(cheeses.Strength<3)/cheeses
board.Name
Caephilly Mild Cheddar
```

But look what happens when we try to select really strong cheese:

```
board←(cheeses.Strength>5)/cheeses
board.Name
The cat ate the last one!
```

Note that this message is not the result of the expression, but was explicitly displayed by the `make_excuse` function. The clue to this behaviour is the shape of `board`; it is empty!

```
ρboard
0
```

When a reference is made to an empty array of Instances (strictly speaking, a reference that requires a *prototype*), APL creates a new Instance by calling the *niladic* (default) Constructor, uses the new Instance to satisfy the reference, and then discards it. Hence, in this example, the reference:

```
board.Name
```

caused APL to run the *niladic* Constructor `make_excuse`, which displayed:

```
The cat ate the last one!
```

Notice that the behaviour of empty arrays of Instances is modelled VERY closely after the behaviour of empty arrays in general. In particular, the Class designer is given the task of deciding what the types of the members of the prototype are.

The full explanation is given in the [Section 4.2.5](#).

4.2.5 Empty Arrays of Instances: How?

To cater for the need to handle empty arrays of Instances as easily as non-empty arrays, a reference to an empty array of Class Instances is handled in a special way.

Whenever a reference or an assignment is made to the content of an *empty array of Instances*, the following steps are performed:

1. APL creates a *new Instance* of the same Class of which the empty Instance belongs
2. the default (niladic) Constructor is run in the new Instance
3. the appropriate value is obtained or assigned:- if it is a reference is to a Field, the value of the Field is obtained
4. if it is a reference is to a Property, the PropertyGet function is run
5. if it is a reference is to a Method, the method is executed
6. if it is an assignment, the assignment is performed or the PropertySet function is run
7. if it is a reference, the result of step 3 is used to generate an empty result array with a suitable prototype by the application of the function $\{0\rho\omega\}$ to it
8. the Class Destructor (if any) is run in the new Instance
9. the New Instance is deleted

EXAMPLE

```

:Class Bird
  :Field Public Species

  ▽ egg spec
    :Access Public Instance
    :Implements Constructor
    □DF Species←spec
  ▽
  ▽ default
    :Access Public Instance
    :Implements Constructor
    □DF Species←'Default Bird'
    #.DISPLAY Species
  ▽
  ▽ R←Speak
    :Access Public
    #.DISPLAY R←'Tweet, Tweet, Tweet'
  ▽

:EndClass  ⌘ Bird

```

First, we can create an empty array of Instances of Bird using 0ρ .

4.2.6 Base Constructors

Constructors in a Class hierarchy are not inherited in the same way as other members. However, there is a mechanism for all the Classes in the Class inheritance tree to participate in the initialisation of an Instance.

Every Constructor function contains a `:Implements Constructor` statement which may appear anywhere in the function body. The statement may optionally be followed by the `:Base` control word and an arbitrary expression.

The statement:

```
:Implements Constructor :Base expr
```

calls a *monadic* Constructor in the Base Class. The choice of Constructor depends upon the rank and shape of the result of `expr` (see [Section 4.2.2](#) for details).

Whereas, the statement:

```
:Implements Constructor
```

or

```
:Implements Constructor :Base
```

calls the *niladic* Constructor in the Base Class.

Note that during the instantiation of an Instance, these calls potentially take place in every Class in the Class hierarchy.

If, anywhere down the hierarchy, there is a *monadic* call and there is no matching monadic Constructor, the operation fails with a `LENGTH ERROR`.

If there is a *niladic* call on a Class that defines **no Constructors**, the niladic call is simply repeated in the next Class along the hierarchy.

However, if a Class defines a monadic Constructor and no niladic Constructor it implies that that Class **cannot be instantiated without Constructor arguments**. Therefore, if there is a call to a niladic Constructor in such a Class, the operation fails with a `LENGTH ERROR`. Note that it is therefore impossible for APL to instantiate a [Section 1.1.3.10](#) or process a reference to an [Section 4.2.5](#) for such a Class or any Class that is based upon it.

A Constructor function may not call another Constructor function and a constructor function may not be called directly from outside the Class or Instance. The only way a

Constructor function may be invoked is by `NEW`. The fundamental reason for these restrictions is that there must be one and only one call on the Base Constructor when a new Instance is instantiated. If Constructor functions were allowed to call one another, there would be several calls on the Base Constructor. Similarly, if a Constructor could be called directly it would potentially duplicate the Base Constructor call.

4.2.7 Niladic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Desc` (inherited from `Bird`). Each of the 3 Classes has its own *niladic* Constructor called `egg0`.

```
:Class Bird
  :Field Public Desc
  ▽ egg0
    :Access Public
    :Implements Constructor
    Desc←'Bird'
  ▽
:EndClass A Bird

:Class Parrot: Bird
  ▽ egg0
    :Access Public
    :Implements Constructor
    Desc,←'→Parrot'
  ▽
:EndClass A Parrot

:Class DomesticParrot: Parrot
  ▽ egg0
    :Access Public
    :Implements Constructor
    Desc,←'→DomesticParrot'
  ▽
:EndClass A DomesticParrot

(NEW DomesticParrot).Desc
Bird→Parrot→DomesticParrot
```

Explanation

`NEW` creates the new instance and runs the niladic Constructor `DomesticParrot.egg0`. As soon as the line:

```
:Implements Constructor
```

is encountered, `NEW` calls the niladic constructor in the Base Class `Parrot.egg0`

`Parrot.egg0` starts to execute and as soon as the line:

```
:Implements Constructor
```

is encountered, `NEW` calls the niladic constructor in the Base Class `Bird.egg0`.

When the line:

```
:Implements Constructor
```

is encountered, `NEW` cannot call the niladic constructor in the Base Class (there is none) so the chain of Constructors ends. Then, as the state indicator unwinds ...

<code>Bird.egg0</code>	executes	<code>Desc ← 'Bird' '</code>
<code>Parrot.egg0</code>	executes	<code>Desc, ← '→Parrot' '</code>
<code>DomesticParrot.egg0</code>	execute	<code>Desc, ← '→DomesticParrot' '</code>

4.2.8 Monadic Example

In the following example, `DomesticParrot` is derived from `Parrot` which is derived from `Bird`. They all share the Field `Species` (inherited from `Bird`) but only a `DomesticParrot` has a Field `Name`. Each of the 3 Classes has its own Constructor called `egg`.

```

:Class Bird
  :Field Public Species
  ▽ egg spec
    :Access Public Instance
    :Implements Constructor
    Species←spec
  ▽
  ...
:EndClass A Bird

:Class Parrot: Bird
  ▽ egg species
    :Access Public Instance
    :Implements Constructor :Base 'Parrot: ',species
  ▽
  ...
:EndClass A Parrot

:Class DomesticParrot: Parrot
  :Field Public Name
  ▽ egg(name species)
    :Access Public Instance
    :Implements Constructor :Base species
    □DF Name←name
  ▽
  ...
:EndClass A DomesticParrot

```

```

    pol←□NEW DomesticParrot('Polly' 'Scarlet Macaw')
    pol.Name
Polly
    pol.Species
Parrot: Scarlet Macaw

```

Explanation

□NEW creates the new instance and runs the Constructor `DomesticParrot.egg`. The egg header splits the argument into two items `name` and `species`. As soon as the line:

```
:Implements Constructor :Base species
```

is encountered, □NEW calls the Base Class constructor `Parrot.egg`, passing it the result of the expression to the right, which in this case is simply the value in `species`.

`Parrot.egg` starts to execute and as soon as the line:

```
:Implements Constructor :Base 'Parrot: ',species
```

is encountered, `NEW` calls its Base Class constructor `Bird.egg`, passing it the result of the expression to the right, which in this case is the character vector `'Parrot: '` catenated with the value in `species`.

`Bird.egg` assigns its argument to the Public Field `Species`.

At this point, the state indicator would be:

```
)SI
[#.[Instance of DomesticParrot]] #.Bird.egg[3]*
[constructor]
:base
[#.[Instance of DomesticParrot]] #.Parrot.egg[2]
[constructor]
:base
[#.[Instance of DomesticParrot]] #.DomesticParrot.egg[2]
[constructor]
```

`Bird.egg` then returns to `Parrot.egg` which returns to `DomesticParrot.egg`.

Finally, `DomesticParrot.egg[3]` is executed, which establishes Field Name and the Display Format (`DF`) for the instance.

4.2.9 Destructors

A *Destructor* is a function that is called just before an Instance of a Class ceases to exist and is typically used to close files or release external resources associated with an Instance.

An Instance of a Class is destroyed when:

- The Instance is expunged using `EX` or `)ERASE`.
- A function, in which the Instance is localised, exits.

But be aware that a destructor will also be called if:

- The Instance is re-assigned (see below)
- The result of `NEW` is not assigned (the instance gets created then immediately destroyed).

- APL creates (and then destroys) a new Instance as a result of a reference to a member of an [Section 4.2.5](#). The destructor is called after APL has obtained the appropriate value from the instance and no longer needs it.
- The constructor function fails. Note that the Instance is actually created before the constructor is run (inside it), and if the constructor fails, the fledgling Instance is discarded. Note too that this means a destructor *may* need to deal with a partially constructed instance, so the code may need to check that resources were actually acquired, before releasing them.
- On the execution of `)CLEAR`, `)LOAD`, `⌈LOAD`, `)OFF` or `⌈OFF`.

Warning

A destructor may be executed on **any** thread.

Note that an Instance of a Class only disappears when the *last reference* to it disappears. For example, the sequence:

```
I1←⌈NEW MyClass
I2←I1
)ERASE I1
```

will not cause the Instance of `MyClass` to disappear because it is still referenced by `I2`.

A Destructor is identified by the statement `:Implements Destructor` which must appear immediately after the function header in the Class script.

```
:Class Parrot
...
▽ kill
  :Implements Destructor
  'This Parrot is dead'
▽
...
:EndClass ⌈ Parrot
```

```
pol←⌈NEW Parrot 'Scarlet Macaw'
)ERASE pol
This Parrot is dead
```

Note that reassignment to `pol` causes the Instance referenced by `pol` to be destroyed and the Destructor invoked:

```

pol←NEW Parrot 'Scarlet Macaw'
pol←NEW Parrot 'Scarlet Macaw'
This Parrot is dead

```

If a Class inherits from another Class, the Destructor in its Base Class is automatically called after the Destructor in the Class itself.

So, if we have a Class structure: DomesticParrot => Parrot => Bird containing the following Destructors:

```

:Class DomesticParrot: Parrot
...
▽ kill
  :Implements Destructor
  'This ',(THIS),' is dead'
▽
...
:EndClass A DomesticParrot

```

```

:Class Parrot: Bird
...
▽ kill
  :Implements Destructor
  'This Parrot is dead'
▽
...
:EndClass A Parrot

```

```

:Class Bird
...
▽ kill
  :Implements Destructor
  'This Bird is dead'
▽
...
:EndClass A Bird

```

Destroying an Instance of DomesticParrot will run the Destructors in DomesticParrot, Parrot and Bird and in that order.

```
pol ← NEW DomesticParrot  
  
    )CLEAR  
This Polly is dead  
This Parrot is dead  
This Bird is dead  
clear ws
```

4.3 Class Members

4.3.1 Class Members

A Class may contain *Methods*, *Fields* and *Properties* (commonly referred to together as *Members*) which are defined within the body of the Class script or are inherited from other Classes.

Methods are regular APL defined functions, but with some special characteristics that control how they are called and where they are executed. Dfns may not be used as Methods.

Fields are just like APL variables. To get the Field value, you reference its name; to set the Field value, you assign to its name, and the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

Properties are similar to APL variables. To get the Property value, you reference its name; to set the Property value, you assign to its name. However, Property values are actually accessed via *PropertyGet* and *PropertySet* functions that may perform all sorts of operations. In particular, the value of a Property is not stored *in* the Property and may be entirely dynamic.

All three types of member may be declared as *Public* or *Private* and as *Instance* or *Shared*.

Public members are visible from outside the Class and Instances of the Class, whereas Private members are only accessible from within.

Instance Members are unique to every Instance of the Class, whereas Shared Members are common to all Instances and Shared Members may be referenced directly on the Class itself.

4.3.2 Fields

Fields

A Field behaves just like an APL variable.

To get the value of a Field, you reference its name; to set the value of a Field, you assign to its name. Conceptually, the Field value is stored *in* the Field. However, Fields differ from variables in that they possess characteristics that control their accessibility.

A Field may be declared anywhere in a Class script by a `:Field` statement. This specifies:

- the name of the Field
- whether the Field is Public or Private
- whether the Field is Instance or Shared
- whether or not the Field is ReadOnly
- the .NET type for the Field when the Class is exported as a .NET Assembly.
- optionally, an initial value for the Field.

Note that [Section 5.2.1](#) may be associated with Fields. See [Section 5.2.4](#) for details.

Public Fields

A *Public* Field may be accessed from outside an Instance or a Class. Note that the default is *Private*.

Class `DomesticParrot` has a `Name` Field which is defined to be Public and Instance (by default).

```
:Class DomesticParrot: Parrot
  :Field Public Name

  ▽ egg nm
    :Access Public
    :Implements Constructor
    Name←nm
  ▽
  ...
:EndClass A DomesticParrot
```

The `Name` field is initialised by the Class constructor.

```
pet←NEW DomesticParrot'Polly'
pet.Name
Polly
```

The `Name` field may also be modified directly:

```

    pet.Name←pet.Name
    pet.Name
ylloP

```

Initialising Fields

A Field may be assigned an initial value. This can be specified by an arbitrary expression that is executed when the Class is fixed by the Editor or by `FIX`.

```

:Class DomesticParrot: Parrot
  :Field Public Name
    :Field Public Talks←1

  ▽ egg nm
    :Access Public
    :Implements Constructor
    Name←nm
  ▽
  ...
:EndClass A DomesticParrot

```

Field `Talks` will be initialised to 1 in every instance of the Class.

```

    pet←NEW DomesticParrot 'Dicky'

    pet.Talks
1
    pet.Name
Dicky

```

Note that if a Field is `ReadOnly`, this is the only way that it may be assigned a value.

See also: [Section 4.3.2.5](#)

Private Fields

A Private Field may only be referenced by code running inside the Class or an Instance of the Class. Furthermore, Private Fields are not inherited.

The [Section 4.3.4.6](#) has a Private Instance Field named `tie` that is used to store the file tie number in each Instance of the Class.

```

:Class ComponentFile
  :Field Private Instance tie

  ▽ Open filename
    :Implements Constructor
    :Access Public Instance
    :Trap 0
      tie←filename □FTIE 0
    :Else
      tie←filename □FCREATE 0
    :EndTrap
    □DF filename, '(Component File)'
  ▽

```

As the field is declared to be Private, it is not accessible from outside an Instance of the Class, but is only visible to code running inside.

```

F1←□NEW ComponentFile 'test1'
F1.tie
VALUE ERROR
F1.tie
^

```

Shared Fields

If a Field is declared to be *Shared*, it has the same value for every Instance of the Class. Moreover, the Field may be accessed from the Class itself; an Instance is not required.

The following example establishes a Shared Field called `Months` that contains abbreviated month names which are appropriate for the user's current International settings. It also shows that an arbitrarily complex statement may be used to initialise a Field.

```

:Class Example
  :Using System.Globalization
  :Field Public Shared ReadOnly Months←12↑(□NEW DateTimeFormatInfo).AbbreviatedMonthNames
:EndClass A Example

```

A Shared Field is not only accessible from an instance...

```

EG←□NEW Example
EG.Months
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov...

```

... but also, directly from the Class itself.

```
Example.Months
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov...
```

Notice that in this case it is necessary to insert a `:Using` statement (or the equivalent assignment to `USING`) in order to specify the .NET search path for the `DateTimeFormatInfo` type. Without this, the Class would fail to fix.

You can see how the assignment works by executing the same statements in the Session:

```
USING←'System.Globalizatiion'
12↑(NEW DateTimeFormatInfo).AbbreviatedMonthNames
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov...
```

Trigger Fields

A field may act as a *Trigger* so that a function may be invoked whenever the value of the Field is changed.

As an example, it is often useful for the Display Form of an Instance to reflect the value of a certain Field. Naturally, when the Field changes, it is desirable to change the Display Form. This can be achieved by making the Field a Trigger as illustrated by the following example.

Notice that the Trigger function is invoked both by assignments made within the Class (as in the assignment in `ctor`) and those made from outside the Instance.

```

:Class MyClass
  :Field Public Name
  :Field Public Country←'England'
  ▽ ctor nm
    :Access Public
    :Implements Constructor
    Name←nm
  ▽
  ▽ format
    :Implements Trigger Name, Country
    □DF 'My name is ', Name, ' and I live in ', Country
  ▽
:EndClass A MyClass

  me←□NEW MyClass 'Pete'
  me
My name is Pete and I live in England

  me.Country←'Greece'
  me
My name is Pete and I live in Greece

  me.Name←'Kostas'
  me
My name is Kostas and I live in Greece

```

4.3.3 Methods

Methods

Methods are implemented as regular defined functions, but with some special attributes that control how they are called and where they are executed.

A Method is defined by a contiguous block of statements in a Class Script. A Method begins with a line that contains a `∇`, followed by a valid APL defined function header. The method definition is terminated by a closing `∇`.

The behaviour of a Method is defined by an `:Access` control statement.

Public or Private

Methods may be defined to be Private (the default) or Public.

A Private method may only be invoked by another function that is running inside the Class namespace or inside an Instance namespace. The name of a Private method is not visible from outside the Class or an Instance of the Class.

A Public method may be called from outside the Class or an Instance of the Class.

Instance or Shared

Methods may be defined to be Instance (the default) or Shared.

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Shared methods are typically used to manipulate Shared Properties and Fields or to provide general services for all Instances that are not Instance specific.

Overridable Methods

Instance Methods may be declared with `:Access Overridable`.

A Method declared as being Overridable is replaced in situ (that is, within its own Class) by a Method of the same name that is defined in a higher Class which itself is declared with the `Override` keyword. See [Section 4.3.3.4](#).

Shared Methods

A Shared method runs in the Class namespace and may be called via an Instance or via the Class. However, a Shared method that is called via an Instance does not have direct access to the Fields and Properties of that Instance.

Class Parrot has a Speak method that does not require any information about the current Instance, so may be declared as Shared.

```
:Class Parrot:Bird
    ▽ R←Speak times
        :Access Public Shared
        R←⌞times⌞'Squark!'
    ▽
:EndClass

wild←NEW Parrot
wild.Speak 2
Squark! Squark!
```

Note that `Parrot.Speak` may be executed directly from the Class and does not in fact require an Instance.

```
Parrot.Speak 3
Squark! Squark! Squark!
```

Instance Methods

An Instance method runs in the Instance namespace and may only be called via the instance itself. An Instance method has direct access to Fields and Properties, both Private and Public, in the Instance in which it runs.

Class `DomesticParrot` has a `Speak` method defined to be Public and Instance. Where `Speak` refers to `Name`, it obtains the value of `Name` in the current Instance.

Note too that `DomesticParrot.Speak` supersedes the inherited `Parrot.Speak`.

```
:Class DomesticParrot: Parrot
  :Field Public Name

  ▽ egg nm
    :Access Public
    :Implements Constructor
    Name←nm
  ▽

  ▽ R←Speak times
    :Access Public Instance
    R←Name, ', ', Name
    R←↑R, timesp<' Who's a pretty boy, then!'
  ▽

:EndClass A DomesticParrot

pet←NEW DomesticParrot'Polly'
pet.Speak 3
Polly, Polly
Who's a pretty boy, then!
Who's a pretty boy, then!
Who's a pretty boy, then!

bil←NEW DomesticParrot'Billy'
bil.Speak 1
Billy, Billy
Who's a pretty boy, then!
```

Superseding Base Class Methods

Normally, a Method defined in a higher Class supersedes the Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available **in the Base Class** and is invoked by a reference to it *from within the Base Class*. This behaviour can be altered using the `Overridable` and `Override` key words in the `:Access` statement but only applies to Instance Methods.

If a Public Instance method in a Class is marked as *Overridable*, this allows a Class which derives from the Class with the Overridable method to supersede the Base Class method *in the Base Class*, by providing a method which is marked *Override*. The typical use of this is to replace code in the Base Class which handles an event, with a method provided by the derived Class.

For example, the base class might have a method which is called if any error occurs in the base class:

```

▼ ErrorHandler
[1]      :Access Public Overridable
[2]      □←↑□DM
▼

```

In your derived class, you might supersede this by a more sophisticated error handler, which logs the error to a file:

```

▼ ErrorHandler;TN
[1]      :Access Public Override
[2]      □←↑□DM
[3]      TN←'ErrorLog'□FSTIE 0
[4]      □DM □FAPPEND TN
[5]      □FUNTIE TN
▼

```

If the derived class had a function which was not marked *Override*, then function in the derived class which called `ErrorHandler` would call the function as defined in the derived class, but if a function in the base class called `ErrorHandler`, it would still see the base class version of this function. With *Override* specified, the new function supersedes the function as seen by code in the base class. Note that different derived classes can specify different *Overrides*.

In C#, Java and some other compiled languages, the term *Virtual* is used in place of *Overridable*, which is the term used by Visual Basic and Dyalog APL.

4.3.4 Properties

Properties

A Property behaves in a very similar way to an ordinary APL variable. To obtain the value of a Property, you simply reference its name. To change the value of a Property, you assign a new value to the name.

However, *under the covers*, a Property is accessed via a *PropertyGet* function and its value is changed via a *PropertySet* function. Furthermore, Properties may be defined to allow partial (indexed) retrieval and assignment to occur.

There are three types of Property, namely *Simple*, *Numbered* and *Keyed*.

- A *Simple Property* is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.
- A *Numbered Property* behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices. The Numbered Property is designed to allow APL to perform selections and structural operations on the Property.
- A *Keyed Property* is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

The following cases illustrate the difference between Simple and Numbered Properties.

If Instance `MyInst` has a Simple Property `SProp` and a Numbered Property `NProp`, the expressions

```
X←MyInst.SProp
X←MyInst.SProp[2]
```

both cause APL to call the *PropertyGet* function to retrieve the entire value of `SProp`. The second statement subsequently uses indexing to extract just the second element of the value.

Whereas, the expression:

```
X←MyInst.NProp[2]
```

causes APL to call the *PropertyGet* function with an additional argument which specifies that only the second element of the Property is required. Moreover, the expression:

```
X←MyInst.NProp
```

causes APL to call the PropertyGet function successively, for every element of the Property.

A Property is defined by a `:Property ... :EndProperty` section in a Class Script.

Within the body of a Property Section there may be:

- one or more `:Access` statements which **must appear first** in the body of the Property.
- a single PropertyGet function.
- a single PropertySet function
- a single PropertyShape function

Simple Instance Properties

A Simple Instance Property is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety. The following examples are taken from the .

The Simple Property Count returns the number of components on a file.

```

:Property Count
:Access Public Instance
  ▽ r←get
    r←-1+2÷⌊FSIZE tie
  ▽
:EndProperty a Count

F1←⌊NEW ComponentFile 'test1'
F1.Append'Hello World'
1
F1.Count
1
F1.Append 42
2
F1.Count
2

```

Because there is no `set` function defined, the Property is read-only and attempting to change it causes SYNTAX ERROR.

```

    F1.Count←99
SYNTAX ERROR
    F1.Count←99
    ^

```

The Access Property has both get and set functions which are used, in this simple example, to get and set the component file access matrix.

```

:Property Access
:Access Public Instance
  ▽ r←get
    r←⎕FRDAC tie
  ▽
  ▽ set am;mat;OK
    mat←am.NewValue
    :Trap 0
      OK←(2=ρρmat)^(3=2⇒ρmat)^^/,mat=⊔mat
    :Else
      OK←0
    :EndTrap
    'bad arg'⎕SIGNAL(~OK)/11
    mat ⎕FSTAC tie
  ▽
:EndProperty ⌞ Access

```

Note that the set function **must** be monadic. Its argument, supplied by APL, will be an Instance of PropertyArguments. This is an internal Class whose NewValue field contains the value that was assigned to the Property.

Note too that the set function does not have to accept the new value that has been assigned. The function may validate the value reject or accept it (as in this example), or perform whatever processing is appropriate.

```

    F1←NEW ComponentFile 'test1'
    ρF1.Access
0 3
    F1.Access←3 3ρ28 2105 16385 0 2073 16385 31 -1 0
    F1.Access
28 2105 16385
0 2073 16385
31 -1 0

    F1.Access←'junk'
bad arg
    F1.Access←'junk'
    ^

    F1.Access←1 2ρ10
bad arg
    F1.Access←1 2ρ10
    ^

```

Simple Shared Properties

The specifies a Simple Shared Property named `Files` which returns the names of all the Component Files in the current directory.

The previous examples have illustrated the use of Instance Properties. It is also possible to define *Shared* properties.

A Shared property may be used to handle information that is relevant to the Class as a whole, and which is not specific to any a particular Instance.

```

:Property Files
:Access Public Shared
  ▽ r←get
    r←FLIB ''
  ▽
:EndProperty

```

Note that `FLIB` (invoked by the `Files` get function) does not report the names of *tied* files.

```

F1←NEW ComponentFile 'test1'
EX'F1'
F2←NEW ComponentFile 'test2'
F2.Files A NB FLIB does not report tied files
test1
EX'F2'

```

Note that a Shared Property may be accessed from the Class itself. It is not necessary to create an Instance first.

```

ComponentFile.Files
test1
test2

```

Numbered Properties

Numbered Properties

A Numbered Property behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices.

To implement a Numbered Property, you **must** specify a PropertyShape function and either or both a PropertyGet and PropertySet function.

When an expression references or makes an assignment to a Numbered Property, APL first calls its PropertyShape function which returns the dimensions of the Property. Note that the shape of the result of this function determines the *rank* of the Property.

If the expression uses indexing, APL checks that the index or indices are within the bounds of these dimensions, and then calls the PropertyGet or PropertySet function. If the expression specifies a single index, APL calls the PropertyGet or PropertySet function once. If the expression specifies multiple indices, APL calls the function successively.

If the expression references or assigns the entire Property (without indexing) APL generates a set of indices for every element of the Property and calls the PropertyGet or PropertySet function successively for every element in the Property.

Note that APL generates a RANK ERROR if an index contains the wrong number of elements or an INDEX ERROR if an index is out of bounds.

When APL calls a monadic PropertyGet or PropertySet function, it supplies an argument of type PropertyArguments.

Example

The specifies a Numbered Property named Component which represents the contents of a specified component on the file.

```

:Property Numbered Component
:Access Public Instance
  ▽ r←shape
    r←-1+2>⊞FSIZE tie
  ▽
  ▽ r←get arg
    r←⊞FREAD tie arg.Indexers
  ▽
  ▽ set arg
    arg.NewValue ⊞FREPLACE tie,arg.Indexers
  ▽
:EndProperty

```

```

F1←⊞NEW ComponentFile 'test1'

F1.Append"(15)×⊞14
1 2 3 4 5

F1.Count
5

F1.Component[4]
4 8 12 16

4>F1.Component
4 8 12 16

(⊞4 3)⊞F1.Component
4 8 12 16 3 6 9 12

```

Referencing a Numbered Property in its entirety causes APL to call the get function successively for every element.

```

F1.Component
1 2 3 4  2 4 6 8  3 6 9 12  4 8 12 16  5 10 15 20

((←4 3)[]F1.Component)←'Hello' 'World'

F1.Component[3]
World

```

Attempting to access a Numbered Property with inappropriate indices generates an error:

```

F1.Component[6]
INDEX ERROR
F1.Component[6]
^
F1.Component[1;2]
RANK ERROR
F1.Component[1;2]
^

```

The Default Property

A single Numbered Property may be identified as the *Default* Property for the Class. If a Class has a Default Property, indexing with the `[]` primitive function and `[...]` indexing may be applied to the Property directly via a reference to the Class or Instance.

The Numbered Property example of the can be extended by adding the control word `Default` to the `:Property` statement for the Component Property.

Indexing may now be applied directly to the Instance `F1`. In essence, `F1[n]` is simply shorthand for `F1.Component[n]` and `n[]F1` is shorthand for `n[]F1.Component`

```

:Property Numbered Default Component
:Access Public Instance
  ▽ r←shape
    r←-1+2>[]FSIZE tie
  ▽
  ▽ r←get arg
    r←[]FREAD tie arg.Indexers
  ▽
  ▽ set arg
    arg.NewValue []FREPLACE tie,arg.Indexers
  ▽
:EndProperty

F1←[]NEW ComponentFile 'test1'
F1.Append''(15)×<14
1 2 3 4 5
F1.Count
5

F1[4]
4 8 12 16
(<4 3)[]F1
4 8 12 16 3 6 9 12
((<4 3)[]F1)←'Hello' 'World'
F1[3]
World

```

Note however that this feature applies only to indexing.

```

    ↳F1
DOMAIN ERROR
    ↳F1
    ^

```

ComponentFile Class

```

:Class ComponentFile
  :Field Private Instance tie

  ▽ Open filename
    :Implements Constructor
    :Access Public Instance
    :Trap 0
      tie←filename □FTIE 0
    :Else
      tie←filename □FCREATE 0
    :EndTrap
    □DF filename, '(Component File)'
  ▽

  ▽ Close
    :Access Public Instance
    □FUNTIE tie
  ▽

  ▽ r←Append data
    :Access Public Instance
    r←data □FAPPEND tie
  ▽

  ▽ Replace(comp data)
    :Access Public Instance
    data □FREPLACE tie,comp
  ▽

  :Property Count
  :Access Public Instance
    ▽ r←get
      r←-1+2□FSIZE tie
    ▽
  :EndProperty A Count

```

```

:Property Access
:Access Public Instance
  ▽ r←get arg
    r←[]FRDAC tie
  ▽
  ▽ set am;mat;OK
    mat←am.NewValue
    :Trap 0
      OK←(2=ppmat)^(3=2>pmat)^^/,mat=Lmat
    :Else
      OK←0
    :EndTrap
    'bad arg'[]SIGNAL(~OK)/11
    mat []FSTAC tie
  ▽
:EndProperty # Access

:Property Files
:Access Public Shared
  ▽ r←get
    r←[]FLIB''
  ▽
:EndProperty

:Property Numbered Default Component
:Access Public Instance
  ▽ r←shape args
    r←~1+2>[]FSIZE tie
  ▽
  ▽ r←get arg
    r←c[]FREAD tie,arg.Indexers
  ▽
  ▽ set arg
    (>arg.NewValue)[]FREPLACE tie,arg.Indexers
  ▽
:EndProperty

  ▽ Delete file;tie
    :Access Public Shared
    tie←file []FTIE 0
    file []FERASE tie
  ▽
:EndClass # Class ComponentFile

```

Keyed Properties

Keyed Properties

A Keyed Property is similar to a Numbered Property except that it may **only** be accessed by indexing (so-called square-bracket indexing) and indices are not restricted to integers but may be arbitrary arrays.

To implement a Keyed Property, only a `get` and/or a `set` function are required. APL does not attempt to validate or resolve the specified indices in any way, so does not require the presence of a `shape` function for the Property.

However, APL **does** check that the rank and lengths of the indices correspond to the rank and lengths of the array to the right of the assignment (for an indexed assignment) and the array returned by the `get` function (for an indexed reference). If the rank or shape of these arrays fails to conform to the rank or shape of the indices, APL will issue a `RANK ERROR` or `LENGTH ERROR`.

Note too that indices **may be elided**. If `KProp` is a Keyed Property of Instance `I1`, the following expressions are all valid.

```
I1.KProp
I1.KProp[]←10
I1.KProp[]←10
I1.KProp['One' 'Two']←10
I1.KProp['One' 'Two']←10
```

When APL calls a monadic `get` or a `set` function, it supplies an argument of type `PropertyArguments`, which identifies which dimensions and indices were specified. See [Section 4.11.2](#).

Keyed Properties Example 1

The `Sparse2` Class illustrates the implementation and use of a Keyed Property.

`Sparse2` represents a 2-dimensional sparse array each of whose dimensions are indexed by arbitrary character keys. The sparse array is implemented as a Keyed Property named `Values`. The following expressions show how it might be used.

```

SA1←NEW Sparse2
SA1.Values[<'Widgets';<'Jan']←100
SA1.Values[<'Widgets';<'Jan']
100
SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']←10×2 3p16
SA1.Values['Widgets' 'Grommets';'Jan' 'Mar' 'Oct']
10 20 30
40 50 60
SA1.Values[<'Widgets';'Jan' 'Oct']
10 30
SA1.Values['Grommets' 'Widgets';<'Oct']
60
30

```

Sparse2 Class Example

```

:Class Sparse2  A 2D Sparse Array
  :Field Private keys
  :Field Private values
  ▽ make
    :Access Public
    :Implements Constructor
    keys←0p<' ' ' '
    values←0
  ▽
  :Property Keyed Values
  :Access Public Instance
    ▽ v←get arg;k
      k←arg.Indexers
      [SIGNAL(2≠pk)/4
      k←fixkeys k
      v←(values,0)[keys:k]
    ▽
    ▽ set arg;new;k;v;n
      v←arg.NewValue
      k←arg.Indexers
      [SIGNAL(2≠pk)/4
      k←fixkeys k
      v←(pk)(p*(≧1=p,v))v
      [SIGNAL((pk)≠pv)/5
      k v←,"k v
      :If v/new←~k∈keys
        values,←new/v
        keys,←new/k
        k v/←~←~new
      :EndIf
      :If 0<pk
        values[keys:k]←v
      :EndIf
    ▽
  :EndProperty

  ▽ k←fixkeys k
    k←(2≠≡"k){,(c*α)ω}"k
    k←(◦.{>,/c"α ω})/k
  ▽
:EndClass  A 2D Sparse Array

```

Internally, `sparse2` maintains a list of keys and a list of values which are initialised to empty arrays by its constructor.

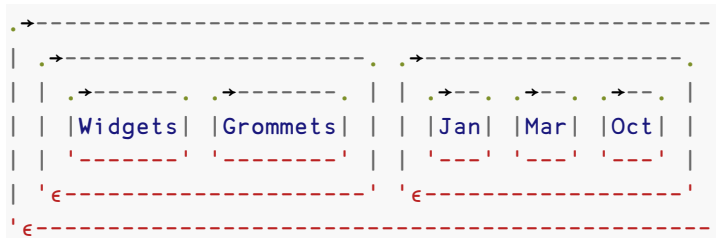
When an indexed assignment is made, the `set` function receives a list of keys (indices) in `arg.Indexer` and values in `arg.NewValue`. The function updates the values of existing keys, and adds new keys and their values to the internal lists.

When an indexed reference is made, the `get` function receives a list of keys (indices) in `arg.Indexer`. The function uses these keys to retrieve the corresponding values, inserting 0s for non-existent keys.

Note that in the expression:

```
SA1.Values['Widgets' 'Grommets'; 'Jan' 'Mar' 'Oct']
```

the structure of `arg.Indexer` is:



Example 2

A second example of a Keyed Property is provided by the `KeyedFile` Class which is based upon the used previously.

```

:Class KeyedFile: ComponentFile
  :Field Public Keys
  ML←0

  ▽ Open filename
    :Implements Constructor :Base filename
    :Access Public Instance
    :If Count>0
      Keys←{>ω⇒BASE.Component}Count
    :Else
      Keys←0p<' '
    :EndIf
  ▽

  :Property Keyed Component
  :Access Public Instance
  ▽ r←get arg;keys;sink
    keys←arg.Indexers
    [SIGNAL(~^/keys∈Keys)/3
    r←{2>ω⇒BASE.Component}KeysKeys
  ▽
  ▽ set arg;new;keys;vals
    vals←arg.NewValue
    keys←arg.Indexers
    [SIGNAL((p,keys)≠p,vals)/5
    :If v/new←~keys∈Keys
      sink←Append"↓Q↑(<new)/"keys vals
      Keys,←new/keys
      keys vals/"←~new
    :EndIf
    :If 0<p,keys
      Replace"↓Q↑(KeysKeys)(↓Q↑keys vals)
    :EndIf
  ▽
  :EndProperty

:EndClass A Class KeyedFile

```

```

K1←NEW KeyedFile 'ktest'
K1.Count
0
K1.Component[<'Pete']←42
K1.Count
1
K1.Component['John' 'Geoff']←(110)(3 4p112)
K1.Count
3
K1.Component['Geoff' 'Pete']
1 2 3 4 42
5 6 7 8
9 10 11 12
K1.Component['Pete' 'Morten']←(3 4p'o')(113)
K1.Count
4
K1.Component['Morten' 'Pete' 'John']
1 1 1 1 1 2 1 1 3 0000 1 2 3 4 5 6 7 8 9 10
1 2 1 1 2 2 1 2 3 0000
0000

```

4.4 Interfaces

4.4.1 Interfaces

An Interface is defined by a Script that contains skeleton declarations of Properties and/or Methods. These members are only *place-holders*; they have no specific implementation; this is provided by each of the Classes that support the Interface.

An Interface contains a collection of methods and properties that together represents a *protocol* that an application must follow in order to manipulate a Class in a particular way.

An example might be an Interface called `Icompare` that provides a single method (`Compare`) which compares two Instances of a Class, returning a value to indicate which of the two is greater than the other. A Class that implements `Icompare` must provide an appropriate `Compare` method, but every Class will have its own individual version of `Compare`. An application can then be written that sorts Instances of any Class that supports the `Icompare` Interface.

An Interface is implemented by a Class if it includes the name of the Interface in its `:Class` statement, and defines a corresponding set of the Methods and Properties that are declared in the Interface.

To implement a Method, a function defined in the Class must include a `:Implements Method` statement that maps it to the corresponding Method defined in the Interface:

```
:Implements Method <InterfaceName.MethodName>
```

Furthermore, the syntax of the function (whether it be result returning, monadic or niladic) must exactly match that of the method described in the Interface. The function name, however, need not be the same as that described in the Interface.

Similarly, to implement a Property the type (Simple, Numbered or Keyed) and syntax (defined by the presence or absence of a `PropertyGet` and `PropertySet` functions) must exactly match that of the property described in the Interface. The Property name, however, need not be the same as that described in the Interface.

4.4.2 Penguin Class Example

The Penguin Class example illustrates the use of Interfaces to implement *multiple inheritance*.

In this case, the Penguin Class derives from Animal but additionally supports the BirdBehaviour and FishBehaviour Interfaces, thereby inheriting members from both.

```
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!
```

4.5 Including Namespaces in Classes

4.5.1 Including Namespaces in Classes

A Class may import methods from one or more plain Namespaces. This allows several Classes to share a common set of methods, and provides a degree of multiple inheritance.

To import methods from a Namespace NS, the Class Script must include a statement:

```
:Include NS
```

When the Class is fixed by the editor or by `FIX`, all the defined functions and operators in Namespace NS are included as methods in the Class. The functions and operators which are brought in as methods from the namespace NS are treated exactly as if the source of each function/operator had been included in the class script at the point of the `:Include` statement. For example, if a function contains `:Signature` or `:Access` statements, these will be taken into account. Note that such declarations have no effect on a function/operator which is in an ordinary namespace.

Dfns and dops in NS are also included in the Class but as *Private members*, because dfns and dops may not contain `:Signature` or `:Access` statements. Variables and Sub-namespaces in NS are **not** included.

Note that objects imported in this way are not actually *copied*, so there is no penalty incurred in using this feature. Additions, deletions and changes to the functions in NS are immediately reflected in the Class.

If there is a member in the Class with the same name as a function in NS, the Class member takes precedence and supersedes the function in NS.

Conversely, functions in NS will supersede members of the same name that are inherited from the Base Class, so the precedence is:

Class supersedes

Included Namespace, supersedes

Base Class

Any number of Namespaces may be included in a Class and the `:Include` statements may occur anywhere in the Class script. However, for the sake of readability, it is

recommended that you have `:Include` statements at the top, given that any definitions in the script will supersede included functions and operators.

For information on copying classes that reference namespaces in this way, see *Dyalog APL Language: Copy*.

4.5.2 Example

In this example, Class `Penguin` inherits from `Animal` and includes functions from the plain Namespaces `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
  :Include BirdStuff
  :Include FishStuff
:EndClass A Penguin
```

Namespace `BirdStuff` contains 2 functions, both declared as Public methods.

```
:Namespace BirdStuff
  ▽ R←Fly
    :Access Public Instance
    R←'Fly, Fly ...'
  ▽
  ▽ R←Lay
    :Access Public Instance
    R←'Lay, Lay ...'
  ▽
:EndNamespace A BirdStuff
```

Namespace `FishStuff` contains a single function, also declared as a Public method.

```

:Namespace FishStuff
  ▽ R←Swim
    :Access Public Instance
    R←'Swim, Swim ...'
  ▽
:EndNamespace A FishStuff

  Pingo←NEW Penguin
  Pingo.Swim
Swim, Swim ...
  Pingo.Lay
Lay, Lay ...
  Pingo.Fly
Fly, Fly ...

```

This is getting silly - we all know that Penguin's can't fly. This problem is simply resolved by overriding the `BirdStuff.Fly` method with `Penguin.Fly`. We can hide `BirdStuff.Fly` with a Private method in `Penguin` that does nothing. For example:

```

:Class Penguin: Animal
  :Include BirdStuff
  :Include FishStuff
  ▽ Fly A Override BirdStuff.Fly
  ▽
:EndClass A Penguin

  Pingo←NEW Penguin
  Pingo.Fly
VALUE ERROR
  Pingo.Fly
  ^

```

or we can supersede it with a different Public method, as follows:

```

:Class Penguin: Animal
  :Include BirdStuff
  :Include FishStuff
  ▽ R←Fly A Override BirdStuff.Fly
    :Access Public Instance
    R←'Sadly, I cannot fly'
  ▽
:EndClass A Penguin

```

```
Pingo←NEW Penguin
Pingo.Fly
Sadly, I cannot fly
```

4.6 Nested Classes

4.6.1 Nested Classes

It is possible to define *Classes within Classes* (Nested Classes).

A Nested Class may be either `Private` or `Public`. This is specified by a `:Access` Statement, which must precede the definition of any Class contents. The default is `Private`.

A `Public` Nested Class is visible from outside its containing Class and may be used directly in its own right, whereas a `Private` Nested Class is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of `Private` Nested Classes and in that way expose them to the calling environment.

4.6.2 GolfService Example

The *GolfService Example Class* illustrates the use of nested classes. *GolfService* was originally developed as a Web Service for Dyalog.NET and is one of the samples distributed in `samples\asp.net\webservices`. This version has been reconstructed as a stand-alone APL Class.

GolfService contains the following nested classes, all of which are `Private`.

	A Class that represents a Golf Course, having Fields <code>Code</code> and <code>Name</code> .
	A Class that represents a tee-time or match, having Fields <code>Time</code> and <code>Players</code> . Up to 4 players may play together in a match.
	A Class that represents a reservation for a particular tee-time at a particular golf course. This has Fields <code>OK</code> , <code>Course</code> , <code>TeeTime</code> and <code>Message</code> . The value of <code>TeeTime</code> is an Instance of a <code>Slot</code> Class.
	A Class that represents a day's starting-sheet at a particular golf course. It has Fields <code>OK</code> , <code>Course</code> , <code>Date</code> , <code>Slots</code> , <code>Message</code> . <code>Slots</code> is an array of Instances of <code>Slot</code> Class.

The *GolfService* constructor takes the name of a file in which all the data is stored. This file is initialised by method `InitFile` if it doesn't already exist.

```
G←NEW GolfService 'F:\HELP11.0\GOLFDATA'
G
#. [Instance of GolfService]
```

The method returns an array of Instances of the internal (nested) Class . Notice how the display form of each Instance is established by the constructor, to obtain the output display shown below.

```
G.GetCourses
St Andrews(1) Hindhead(2) Basingstoke(3)
```

All of the dates and times employ instances of the .NET type System.DateTime, and the following statements just set up some temporary variables for convenience later.

```
←Tomorrow←(NEW DateTime(3↑TS)).AddDays 1
31/03/2006 00:00:00
←TomorrowAt7←Tomorrow.AddHours 7
31/03/2006 07:00:00
```

The method takes between 4 and 7 parameters viz.

- the code for the golf course at which the reservation is required
- the date and time of the reservation
- a flag to indicate whether or not the nearest available time will do
- a list of up to 4 players who wish to book that time.
- the code for the golf course at which the reservation is required
- the date and time of the reservation
- a flag to indicate whether or not the nearest available time will do
- a list of up to 4 players who wish to book that time.

The result is an Instance of the internal Class . Once again, DF is used to make the default display of these Instances meaningful. In this case, the reservation is successful.

```
G.MakeBooking 2 TomorrowAt7 1 'Pete' 'Tiger'
Hindhead(2) 31/03/2006 07:00:00 Pete Tiger OK
```

Bob, Arnie and Jack also ask to play at 7:00 but are given the 7:10 tee-time instead (4-player restriction).

```
G.MakeBooking 2 TomorrowAt7 1 'Bob' 'Arnie' 'Jack'
Hindhead(2) 31/03/2006 07:10:00 Bob Arnie Jack OK
```

However, Pete and Tiger are joined at 7:00 by Dave and Al.

```
G.MakeBooking 2 TomorrowAt7 1 'Dave' 'Al'
Hindhead(2) 31/03/2006 07:00:00 Pete Tiger Dave Al OK
```

Up to now, all bookings have been made with the tee-time flexibility flag set to 1. Inflexible Jim is only interested in playing at 7:00...

```
G.MakeBooking 2 TomorrowAt7 0 'Jim'
Hindhead(2) 31/03/2006 07:00:00 Not available
```

... so his reservation fails (4-player restriction).

Finally the method is used to obtain an Instance of the internal Class for the given course and day.

```
G.GetStartingSheet 2 Tomorrow
Hindhead(2) 31/03/2006 00:00:00
31/03/2006 07:00:00 Pete Tiger Dave Al
31/03/2006 07:10:00 Bob Arnie Jack
31/03/2006 07:20:00
....
```

4.7 Namespace Scripts

4.7.1 Namespace Scripts

A Namespace Script is a script that begins with a `:Namespace` statement and ends with a `:EndNamespace` statement. When a Namespace Script is fixed, it establishes an entire namespace that may contain other namespaces, functions, variables and classes.

The names of Classes defined within a Namespace Script which are parents, children, or siblings are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it. Names of Classes which are nieces or nephews and their descendants are however not visible.

For example:

```

:Namespace a

  d←[]NEW a1
  e←[]NEW bb2

  :Class a1
    ▽ r←foo
      :Access Shared Public
      r←[]NEW b1 b2
    ▽
  :EndClass a a1

  ▽ r←goo
    r←a1.foo
  ▽

  ▽ r←foo
    r←[]NEW b1 b2
  ▽

  :Namespace b
    :Class b1
    :EndClass a b1
    :Class b2
      :Class bb2
      :EndClass a bb2
    :EndClass a b2
  :EndNamespace a b

:EndNamespace a a
    
```

```

      a.d
# .a.[a1]
      a.e
# .a.[bb2]
      a.foo
# .a.[b1] # .a.[b2]
    
```

Note that the names of Classes b1 (a.b.b1) and b2 (a.b.b2) are not visible from their "uncle" a1 (a.a1).

```

      a.goo
VALUE ERROR
foo[2] r←[]NEW b1 b2
    
```

Notice that Classes in a Namespace Script are fixed before other objects (hence the assignments to `d` and `e` are evaluated *after* Classes `a1` and `bb2` are fixed), although the order in which Classes themselves are defined is still important if they reference one another during initialisation.

A practical example using `:Namespace` is given in the [Section 4.7.2](#).

Changing Scripted Objects Dynamically

The source of a scripted object can only be altered using the Editor, or by refixing it in its entirety using `FIX`. Dynamic changes to variables, fields and properties, and calling `FIX` to generate functions do not alter the source of a scripted object.

Furthermore, if you introduce new objects of any type (functions, variables, or classes) into a namespace or a class defined by a script by any means other than editing the script, then these objects will be lost the next time the script is edited and fixed.

If you fix a function using `FIX` with the same name as a function defined in the script, this new version will supercede the version defined from the script, although the version in the script will remain unchanged.

If you edit the function (as opposed to editing the script) the Editor will show the new version of the function.

If however you edit the script, the Editor will display the original version of the function embedded in the script.

If you were to edit both the script and the function, the Editor would show the two different versions of the function as illustrated in the example that follows.

When you fix the script, the version of the function in the script will replace the one created using `FIX`.

EXAMPLE

```

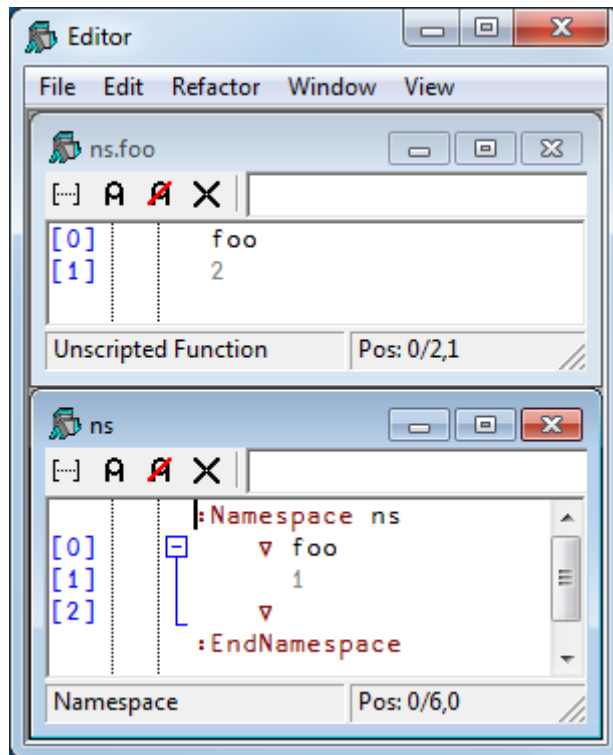
:Namespace ns
  ▾ foo
    1
  ▾
:EndNamespace

  ns.foo
1

  ns.⌊FX 'foo' '2'
  ns.foo
2

)ED ns.foo ns

```



Note that the Editor displays the description `Unscripted Function` in the status bar of the window showing the new version of `foo`.

Similarly, if you were to Trace the execution of `ns.foo`, the Tracer would display the current (⌊FX'ed) version of `foo`, with the same description in its status bar.

4.7.2 Namespace Script Example

The *DiaryStuff* example illustrates the manner in which classes may be defined and used in a Namespace script.

DiaryStuff defines two Classes named `Diary` and `DiaryEntry`.

`Diary` contains a (private) Field named `entries`, which is simply a vector of instances of `DiaryEntry`. These are 2-element vectors containing a .NET `DateTime` object and a description.

The `entries` Field is initialised to an empty vector of `DiaryEntry` instances which causes the invocation of the default constructor `DiaryEntry.Make0` when `Diary` is fixed. See [Section 4.2.4](#) for further explanation.

The `entries` Field is referenced through the `Entry` Property, which is defined as the [Section 4.3.4.5](#). This allows individual entries to be referenced and changed using indexing on a `Diary` Instance.

Note that `DiaryEntry` is defined in the script first (before `Diary`) because it is referenced by the initialisation of the `Diaries.entries` Field

Create a new instance of `Diary`.

```
D←NEW DiaryStuff.Diary
```

Add a new entry "meeting with John at 09:00 on April 30th"

```
D.Add(2006 4 30 9 0) 'Meeting with John'
30/04/2006 09:00:00 Meeting with John
```

Add another diary entry "Dentist at 10:00 on April 30th".

```
D.Add(2006 4 30 10 0) 'Dentist'
30/04/2006 10:00:00 Dentist
```

One of the benefits of the Namespace Script is that Classes defined within it (which are typically *related*) may be used *independently*, so we can create a stand-alone instance of `DiaryEntry`; "Doctor at 11:00"...

```
Doc←NEW DiaryStuff.DiaryEntry((2006 4 30 11 0) 'Doctor')
Doc
30/04/2006 11:00:00 Doctor
```

... and then use it to replace the second `Diary` entry with indexing:

```
D[2]←Doc
```

and just to confirm it is there...

```
D[2]
30/04/2006 11:00:00 Doctor
```

What am I doing on the 30th?

```
D.DoingOn 2006 4 30
30/04/2006 09:00:00 Meeting with John ...
... 30/04/2006 11:00:00 Doctor
```

Remove the 11:00 appointment...

```
D.Remove 2006 4 30 11 0
1
```

and the complete Diary is...

```
D
30/04/2006 09:00:00 Meeting with John
```

4.8 Including Script Files in Scripts

A Class or Namespace script in the workspace or in a script file may specify that other script files are to be loaded prior to the fixing of the script itself. To do so, it must begin with one or more `:Require` statements, with the following syntax:

```
:Require file://[path]/file
```

If no path is specified, the path is taken to be relative to the current script file or, if in a workspace script, the current working directory. Note that a leading `'./'` or `'.\'` in path is not allowed, to avoid any potential confusion with "current directory".

`:Require` is a directive to the Editor (more specifically, to the internal mechanism that fixes a script as an object in the workspace) and can appear in any script containing APL code, but **must** precede all code in the script. `:Require` is thus not valid within a function, class, namespace or any other definition.

The prefix `file://` allows for the possibility of a future extension of `https://` and `ftp://`.

In version 20.0 `⎕!require` is a synonym for `:Require`. This allows the user to create scripts which can be used in multiple versions of Dyalog; in 14.1 and earlier SALT parses `⎕!require` statements and loads the appropriate files, in 20.0 it is the interpreter loads the file named in `⎕!require` statements. Dyalog intends to remove support for the `⎕!require` statement from the interpreter in a future version. Note that unlike `:Require`, `⎕!require` can appear within code.

4.9 Class Declaration Statements

4.9.1 Class Declaration Statements

This section summarises the various declaration statements that may be included in a Class or Namespace Script. For information on other declaration statements, as they apply to functions and methods, see [Section 3.1.8.1](#).

4.9.2 :Interface Statement

```
:Interface <interface name>
...
:EndInterface
```

An Interface is defined by a Script containing skeleton declarations of Properties and/or Methods. The script must begin with a `:Interface Statement` and end with a `:EndInterface Statement`.

An Interface may not contain Fields.

Properties and Methods defined in an Interface, and the Class functions that implement the Interface, **may not** contain `:Access Statements`.

4.9.3 :Namespace Statement

```
:Namespace <namespace name>
...
:EndNamespace
```

A Namespace Script may be used to define an entire namespace containing other namespaces, functions, variables and Classes.

A Namespace script must begin with a `:Namespace` statement and end with a `:EndNamespace` statement.

Sub-namespaces, which may be nested, are defined by pairs of `:Namespace` and `:EndNamespace` statements within the Namespace script.

Classes are defined by pairs of `:Class` and `:EndClass` statements within the Namespace script, and these too may be nested.

The names of Classes defined within a Namespace Script are visible both to one another and to code and expressions defined in the same script, regardless of the namespace hierarchy within it.

A Namespace script is therefore particularly useful to group together Classes that refer to one another where the use of nested classes is inappropriate.

4.9.4 :Class Statement

```
:Class <class name><:base class name> <,interface name...>

:Include <namespace>
...
:EndClass
```

A class script begins with a `:Class` statement and ends with a `:EndClass` statement. The elements that comprise the `:Class` statement are as follows:

Element	Description
<code>class name</code>	Optionally, specifies the name of the Class, which must conform to the rules governing APL names.
<code>base class name</code>	Optionally specifies the name of a Class from which this Class is derived and whose members this Class inherits.
<code>interface name</code>	The names of one or more Interfaces which this Class supports.

A Class may import methods defined in separate plain Namespaces with one or more `:Include` statements. For further details, see [Section 4.5.1](#).

EXAMPLES

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and which supports two Interfaces named `BirdBehaviour` and `FishBehaviour`.

```
:Class Penguin: Animal,BirdBehaviour,FishBehaviour
...
:EndClass
```

The following statements define a Class named `Penguin` that derives from (is based upon) a Class named `Animal` and includes methods defined in two separate Namespaces named `BirdStuff` and `FishStuff`.

```
:Class Penguin: Animal
:Include BirdStuff
:Include FishStuff
...
:EndClass
```

4.9.5 :Using Statement

```
:Using <NameSpace[,Assembly]>
```

This statement specifies a .NET namespace that is to be searched to resolve unqualified names of .NET types referenced by expressions in the Class.

Element	Description
NameSpace	Specifies a .NET namespace.
Assembly	Specifies the Assembly in which NameSpace is located. If the Assembly is located in the Microsoft.NET installation directory, you need only specify its name. If not, you must specify a full or relative pathname.

If the Microsoft .NET Framework is installed, the System namespace `mscorlib.dll` is automatically loaded when Dyalog APL starts. To access this namespace, it is not necessary to specify the name of the Assembly.

When the class is fixed, `⎕USING` is inherited from the surrounding space. Each `:Using` statement appends an element to `⎕USING`, with the exception of `:Using` with no argument:

If you omit `<Namespace>`, this is equivalent to clearing `⎕USING`, which means that no .NET namespaces will be searched (unless you follow this statement with additional `:Using` statements, each of which will append to `⎕USING`).

To set `⎕USING` to a single empty character vector, which only allows references to fully qualified names of classes in `mscorlib.dll`, you must write:

```
:Using , (note the presence of the comma)
```

or

```
:Using ,mscorlib.dll
```

that is, specify an empty namespace name followed by no assembly, or followed by the default assembly, which is always loaded.

4.9.6 :Attribute Statement

```
:Attribute <Name> [ConstructorArgs]
```

The `:Attribute` statement is used to attach .NET Attributes to a Class or a Method.

Attributes are descriptive tags that provide additional information about programming elements. Attributes are not used by Dyalog APL but other applications can refer to the extra information in attributes to determine how these items can be used. Attributes are saved with the *metadata* of Dyalog APL .NET assemblies.

Element	Description
Name	The name of a .NET attribute
ConstructorArgs	Optional arguments for the Attribute constructor

EXAMPLE

The following Class has `SerializableAttribute` and `CLSCompliantAttribute` attributes attached to the Class as a whole, and `ObsoleteAttribute` attributes attached to Methods `foo` and `goo` within it.

```
:Class c1
:using System
  :attribute SerializableAttribute
  :attribute CLSCompliantAttribute 1

  ▽ foo(p1 p2)
    :Access public instance
    :Signature foo Object,Object
    :Attribute ObsoleteAttribute
  ▽

  ▽ goo(p1 p2)
    :Access public instance
    :Signature goo Object,Object
    :Attribute ObsoleteAttribute 'Don't use this' 1

  ▽

:EndClass # c1
```

When this Class is exported as a .NET Class, the attributes are saved in its metadata. For example, Visual Studio will warn developers if they make use of a member which has the `ObsoleteAttribute`.

4.9.7 :Access Statement

```
:Access <Private|Public><Instance|Shared><Overridable>
                                <Override>
:Access <WebMethod>
```

The `:Access` statement is used to specify characteristics for Classes, Properties and Methods.

Element	Description
<code>Private</code> <code>Public</code>	Specifies whether or not the (nested) Class, Property or Method is accessible from outside the Class or an Instance of the Class. The default is <code>Private</code> .
<code>Instance</code> <code>Shared</code>	For a Field, specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances. For a Property or Method, specifies whether the code associated with the Property or Method runs in the Class or Instance.
<code>WebMethod</code>	Applies only to a Method and specifies that the method is exported as a web method. This applies only to a Class that implements a Web Service.
<code>Overridable</code>	Applies only to an Instance Method and specifies that the Method may be overridden by a Method in a higher Class. See below.
<code>Override</code>	Applies only to an Instance Method and specifies that the Method overrides the corresponding <code>Overridable</code> Method defined in the Base Class. See below.

Overridable/Override

Normally, a Method defined in a higher Class replaces a Method of the same name that is defined in its Base Class, but only for calls made from above or within the higher Class itself (or an Instance of the higher Class). The base method remains available *in the Base Class* and is invoked by a reference to it *from within the Base Class*.

However, a Method declared as being `Overridable` is replaced in situ (that is, within its own Class) by a Method of the same name in a higher Class if that Method is itself declared with the `Override` keyword. For further information, see [Section 4.3.3.4](#).

Nested Classes

The `:Access` statement is also used to control the visibility of one Class that is defined within another (a nested Class). A Nested Class may be either `Private` or `Public`. Note that the `:Access` Statement must precede the definition of any Class contents.

A `Public Nested Class` is visible from outside its containing Class and may be used directly in its own right, whereas a `Private Nested Class` is not and may only be used by code inside the containing Class.

However, methods in the containing Class may return instances of Private Nested Classes and in that way expose them to the calling environment.

WebMethod

Note that `:Access WebMethod` is equivalent to:

```
:Access Public
:Attribute System.Web.Services.WebMethodAttribute
```

4.9.8 :Implements Statement

The `:Implements` statement identifies the function to be one of the following types.

```
:Implements Constructor <[:Base expr]>
:Implements Destructor
:Implements Method <InterfaceName.MethodName>
:Implements Trigger <name1><, name2, name3, ...>
:Implements Trigger *
```

Element	Description
Constructor	Specifies that the function is a <u><i>Class Constructor</i></u> .
:Base expr	Specifies that the <u><i>Base Constructor</i></u> be called with the result of the expression <code>expr</code> as its argument.
Destructor	Specifies that the function is a <u><i>Class Destructor</i></u> .
Method	Specifies that the function implements the Method <code>MethodName</code> whose syntax is specified by <u><i>Interface InterfaceName</i></u> .
Trigger	Identifies the function as a <u><i>Trigger Function</i></u> which is activated by changes to variable <code>name1</code> , <code>name2</code> , and so forth. <code>Trigger *</code> specifies a <u><i>Global Trigger</i></u> that is activated by the assignment of any global variable in the same namespace.

4.10 :Field Statement

```
:Field <Private|Public> <Instance|Shared> <ReadOnly>...  
... fieldName <← expr>
```

A `:Field` statement is a single statement whose elements are as follows:

Element	Description
<code>Private</code> <code>Public</code>	Specifies whether or not the Field is accessible from outside the Class or an Instance of the Class. The default is <code>Private</code> .
<code>Instance</code> <code>Shared</code>	Specifies if there is a separate value of the Field in each Instance of the Class, or if there is only a single value that is shared between all Instances.
<code>ReadOnly</code>	If specified, this keyword prevents the value in the Field from being changed after initialisation.
<code>Type</code>	If specified, this identifies a .Net type for the Field. This type applies only when the Class is exported as a .NET Assembly.
<code>FieldName</code>	Specifies the name of the Field (mandatory).
<code>← expr</code>	Specifies an initial value for the Field.

EXAMPLES

The following statement defines a Field called `Name`. It is (by default), an Instance Field so every Instance of the Class has a separate value. It is a Public Field and so may be accessed (set or retrieved) from outside an Instance.

```
:Field Public Name
```

The following statement defines a Field called `Months`.

```
:Field Shared ReadOnly Months←12↑(NEW  
DateTimeFormatInfo.AbbreviatedMonthNames
```

`Months` is a Shared Field so there is just a single value that is the same for every Instance of the Class. It is (by default), a Private Field and may only be referenced by code running in an Instance or in the Class itself. Furthermore, it is `ReadOnly` and may not be altered after initialisation. Its initial value is calculated by an expression that obtains the short month names that are appropriate for the current locale using the .NET Type `DateTimeFormatInfo`.

Notes

Note that Fields are initialised when a Class script is fixed by the editor or by `FIX`. If the evaluation of `expr` causes an error (for example, a `VALUE ERROR`), an appropriate message will be displayed in the Status Window and `FIX` will fail with a `DOMAIN ERROR`. Note that a `ReadOnly` Field may only be assigned a value by its `:Field` statement.

In the second example above, the expression will only succeed if `USING` is set to the appropriate path, in this case `System.Globalization`.

You may not define a Field with the name of one of the permissible keywords (such as `public`). In such cases the Class will not fix and an error message will be displayed in the Status Window. For example:

```
error AC0541: a field must have a name " :Field Public public"
```

4.11 Property Section

4.11.1 :Property Section

A Property is defined by a `:Property ... :EndProperty` section in a Class Script. The syntax of the `:Property` Statement, and its optional `:Access` statement is as follows:

```
:Property <Simple|Numbered|Keyed> <Default> Name<,Name>...
:Access <Private|Public><Instance|Shared>
...
:EndProperty
```

Element	Description
Name	Specifies the name of the Property by which it is accessed. Additional Properties, sharing the same PropertyGet and/or PropertySet functions, and the same access behaviour may be specified by a comma-separated list of names.
Simple Numbered Keyed	Specifies the type of Property (see below). The default is Simple .
Default	Specifies that this Property acts as the default property for the Class when indexing is applied directly to an Instance of the Class.
Private Public	Specifies whether or not the Property is accessible from outside the Class or an Instance of the Class. The default is Private .
Instance Shared	Specifies if there is a separate value of the Property in each Instance of the Class, or if there is only a single value that is shared between all Instances.

A [Section 4.3.4.2](#) is one whose value is accessed (by APL) in its entirety and re-assigned (by APL) in its entirety.

A [Section 4.3.4.4.1](#) behaves like an array (conceptually a vector) which is only ever *partially* accessed and set (one element at a time) via indices.

A [Section 4.3.4.7.1](#) is similar to a Numbered Property except that its elements are accessed via arbitrary keys instead of indices.

Numbered and Keyed Properties are designed to allow APL to perform selections and structural operations on the Property.

Within the body of a Property Section there may be:

- one or more `:Access` statements
- a single [Section 4.11.3](#) function.
- a single [Section 4.11.4](#) function
- a single [Section 4.11.5](#) function

The three functions are identified by case-independent names `Get`, `Set` and `Shape`.

Errors

When a Class is fixed by the Editor or by `FIX`, APL checks the validity of each Property section and the syntax of [Section 4.11.3](#), [Section 4.11.4](#) and [Section 4.11.5](#) functions within them.

- You may not specify a name which is the same as one of the keywords.
- There must be at least a [Section 4.11.3](#), or a [Section 4.11.4](#) or a [Section 4.11.5](#) function defined.
- You may only define a [Section 4.11.5](#) function if the Property is Numbered.

If anything is wrong, the Class is not fixed and an error message is displayed in the Status Window. For example:

```
error AC0545: invalid or empty property declaration
error AC0595: this property type should not implement a "shape" function
```

4.11.2 PropertyArguments Class

Where appropriate, APL supplies the `PropertyGet` and `PropertySet` functions with an argument that is an instance of the internal class `PropertyArguments`.

`PropertyArguments` has just 3 read-only Fields which are as follows:

Name	The name of the property. This is useful when one function is handling several properties.
NewValue	Array containing the new value for the Property or for selected element(s) of the property as specified by <code>Indexers</code> .
IndexersSpecified	A Boolean vector that identifies which dimensions of the Property are to be referenced or assigned.
Indexers	A vector that identifies the elements of the Property that are to be referenced or assigned.

4.11.3 PropertyGet Function

`R←Get {ipa}`

The name of the `PropertyGet` function must be `Get`, but is case-independent. For example, `get`, `Get`, `gEt` and `GET` are all valid names for the `PropertyGet` function.

The `PropertyGet` function must be result returning. For a Simple Property, it may be monadic or niladic. For a Numbered or Keyed Property it must be monadic.

The result `R` may be any array. However, for a Keyed Property, `R` must conform to the rank and shape specified by `ipa.Indexers` or be scalar.

If monadic, `ipa` is an instance of the internal class [Section 4.11.2](#).

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` is undefined (`VALUE ERROR`).

If the Property is *Simple*, `ipa.Indexers` is undefined (`VALUE ERROR`).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be obtained. In this case, `R` must be scalar.

If the Property is *Keyed*, `ipa.IndexersSpecified` is a Boolean vector with the same length as the rank of the property (as implied by the result of the `Shape` function). A value of 1 means that an indexing array for the corresponding dimension of the Property was specified, while a value of 0 means that the corresponding dimension was elided. `ipa.Indexers` is a vector of the same length containing the arrays that were specified within the square brackets in the reference expression. Specifically, `ipa.Indexers` will contain one fewer elements than, the number of semi-colon (;) separators. If any index was elided, the corresponding element of `ipa.Indexers` is `NULL`.

Note

It is not possible to predict the number of times that a `PropertyGet`, `PropertySet` or `PropertyShape` function will be called by a particular APL expression, as this depends upon how that expression is implemented internally. You should therefore not rely on the number of times that a `Get`, `Set` or `Shape` function is called, and none should have any side effects on any other APL object

4.11.4 PropertySet Function

Set *ipa*

The name of the PropertySet function must be `Set`, but is case-independent. For example, `set`, `Set`, `sEt` and `SET` are all valid names for the PropertySet function.

The PropertySet function must be monadic and may not return a result.

`ipa` is an instance of the internal class [Section 4.11.2](#).

In all cases, `ipa.Name` contains the name of the Property being referenced and `NewValue` contains the new value(s) for the element(s) of the Property being assigned.

If the Property is *Simple*, `ipa.Indexers` is undefined (VALUE ERROR).

If the Property is *Numbered*, `ipa.Indexers` is an integer vector of the same length as the rank of the property (as implied by the result of the `Shape` function) that identifies a single element of the Property whose value is to be set.

If the Property is *Keyed*, `ipa.IndexersSpecified` is a Boolean vector with the same length as the rank of the property (as implied by the result of the `Shape` function). A value of 1 means that an indexing array for the corresponding dimension of the Property was specified, while a value of 0 means that the corresponding dimension was elided. `ipa.Indexers` is a vector containing the arrays that were specified within the square brackets in the assignment expression. Specifically, `ipa.Indexers` will contain one fewer elements than, the number of semi-colon (;) separators. If any index was elided, the corresponding element of `ipa.Indexers` is `⍵NULL`. However, if the Keyed Property is being assigned in its entirety, without square-bracket indexing, `ipa.Indexers` is undefined (VALUE ERROR).

Note

It is not possible to predict the number of times that a `PropertyGet`, `PropertySet` or `PropertyShape` function will be called by a particular APL expression, as this depends upon how that expression is implemented internally. You should therefore not rely on the number of times that a `Get`, `Set` or `Shape` function is called, and none should have any side effects on any other APL object

4.11.5 PropertyShape Function

$R \leftarrow \text{Shape } \{ipa\}$

The name of the PropertyShape function must be `Shape`, but is case-independent. For example, `shape`, `Shape`, `sHape` and `SHAPE` are all valid names for the PropertyShape function.

A PropertyShape function is only called if the Property is a Numbered Property.

The PropertyShape function must be niladic or monadic and must return a result.

If monadic, `ipa` is an instance of the internal class [Section 4.11.2](#). `ipa.Name` contains the name of the Property being referenced and `NewValue` and `Indexers` are undefined (`VALUE ERROR`).

The result `R` must be an integer vector or scalar that specifies the `rank` of the Property. Each element of `R` specifies the length of the corresponding dimension of the Property. Otherwise, the reference or assignment to the Property will fail with `DOMAIN ERROR`.

Note that the result `R` is used by APL to check that the number of indices corresponds to the rank of the Property and that the indices are within the bounds of its dimensions. If not, the reference or assignment to the Property will fail with `RANK ERROR` or `LENGTH ERROR`.

Note

It is not possible to predict the number of times that a `PropertyGet`, `PropertySet` or `PropertyShape` function will be called by a particular APL expression, as this depends upon how that expression is implemented internally. You should therefore not rely on the number of times that a `Get`, `Set` or `Shape` function is called, and none should have any side effects on any other APL object

5 Threads and Triggers

5.1 Threads

5.1.1 Multi-Threading Language Elements

The following language elements are provided to support threads.

- Primitive operator, spawn: `&`.
- System functions: `□TID`, `□TCNUMS`, `□TNUMS`, `□TKILL`, `□TSYNC`.
- An extension to the GUI Event syntax to allow asynchronous callbacks.
- A control structure: `:Hold`.
- System commands: `)HOLDS`, `)TID`.
- Extended `)SI` and `)SINL` display.

Running Callback Functions as Threads

A callback function is associated with a particular event via the `Event` property of the object concerned. A callback function is executed by `□DQ` when the event occurs, or by `□NQ`.

If you append the character `&` to the name of the callback function in the `Event` specification, the callback function will be executed asynchronously as a thread when the event occurs. If not, it is executed synchronously as before.

For example, the event specification:

```
□WS'Event' 'Select' 'DoIt&'
```

tells `□DQ` to execute the callback function `DoIt` *asynchronously as a thread* when a `Select` event occurs on the object.

5.1.2 Thread Switching

Warning

Programming with threads requires care!

The interpreter may switch between running threads at the following points:

- Between any two lines of a defined function or operator.
- On entry to a dfn or dop.
- While waiting for a `□DL` to complete.
- While awaiting input from:
 - `□DQ`
 - `□SR`
 - `□ED`
- The session prompt or `□:` or `□`.
- While awaiting the completion of an external operation:
 - A call on an external (AP) function.
 - A call on a `□NA` (DLL) function.
 - A call on an OLE function.
 - A call on a .NET function.

At any of these points, the interpreter might execute code in other threads. If such threads change the global environment; for example by changing the value of, or expunging a name; then the changes will appear to have happened while the thread in question passes through the switch point. It is the task of the application programmer to organise and contain such behaviour.

You can prevent threads from interacting in critical sections of code by using the `:Hold` control structure.

High Priority Callback Functions

Note that the interpreter cannot perform thread-switching during the execution of a *high-priority callback*. This is a callback function that is invoked by a *high-priority* event which demands that the interpreter must return a result to Windows before it may process any other event. Such high-priority events include `Configure`, `ExitWindows`, `DateTimeChange`, `DockStart`, `DockCancel`, `DropDown`. It is therefore not permitted to use a `:Hold` control structure in a high-priority callback function.

5.1.3 Name Scope

APL's name scope rules apply whether a function call is synchronous or asynchronous. For example when a defined function is called, names in the calling environment are visible, unless explicitly shadowed in the function header.

Just as with a synchronous call, a function called asynchronously has its own local environment, but can communicate with its parent and "sibling" functions via local names in the parent.

This point is important. It means that siblings can run in parallel without danger of local name clashes. For example, a GUI application can accommodate multiple concurrent instances of its callback functions.

However, with an asynchronous call, as the calling function continues to execute, both child *and parent functions* may modify values in the calling environment. Both functions see such changes immediately they occur.

If a parent function terminates while any of its children are still running, those children will no longer have access to its local names, and references to such names will either generate `VALUE ERROR` or be replaced by values from the environment that called the parent function. If a child function references variables defined by its parent or relies in any other way on its parent's environment (such as a local value of `⌈IO`), the parent function should therefore execute a `⌈TSYNC` in order to wait for its children to complete before itself exiting.

If, on the other hand, after launching an asynchronous child, the parent function calls a *new* function (either synchronously or asynchronously); names in the new function are beyond the purview of the original child. In other words, a function can only ever see its calling stack decrease in size – never increase. This is in order that the parent may call new defined functions without affecting the environment of its asynchronous children.

5.1.4 Stack Considerations

When you start a thread, it begins with the SI stack of the calling function and sees all of the local variables defined in all the functions down the stack. However, unless the calling function specifically waits for the new thread to terminate (see *Dyalog APL Language: Tsync*), the calling functions will (bit by bit, in their turn) continue to execute. The new thread's view of its calling environment may then change. Consider the following example:

Suppose that you had the following functions: RUN[3] calls INIT which in turn calls GETDATA but as 3 separate threads with 3 different arguments:

```

    ▾ RUN;A;B
[1]  A←1
[2]  B←'Hello World'
[3]  INIT
[4]  CALC
[5]  REPORT
    ▾

```

```

    ▾ INIT;C;D
[1]  C←D←0
[2]  GETDATA&'Sales'
[3]  GETDATA&'Costs'
[4]  GETDATA&'Expenses'
    ▾

```

When each GETDATA thread starts, it immediately sees (via `ISI`) that it was called by INIT which was in turn called by RUN, and it sees local variables A, B, C and D. However, once INIT[4] has been executed, INIT terminates, and execution of the root thread continues by calling CALC. From then on, each GETDATA thread no longer sees INIT (it thinks that it was called directly from RUN) nor can it see the local variables C and D that INIT had defined. However, it *does* continue to see the locals A and B defined by RUN, until RUN itself terminates.

Note that if CALC were also to define locals A and B, the GETDATA threads would still see the values defined by RUN and not those defined by CALC. However, if CALC were to modify A and B (as globals) without localising them, the GETDATA threads would see the modified values of these variables, whatever they happened to be at the time.

5.1.5 Globals and the Order of Execution

It is important to recognise that any reference or assignment to a global or semi-global object (including GUI objects) is **inherently dangerous** (that is, a source of programming error) if more than one thread is running. Worse still, programming errors of this sort may not become apparent during testing because they are dependent upon random timing differences. Consider the following example:

```

    ▾ BUG; SEMI_GLOBAL
[1]   SEMI_GLOBAL ← 0
[2]   FOO & 1
[3]   GOO & 1
    ▾

    ▾ FOO
[1]   : If SEMI_GLOBAL = 0
[2]       DO_SOMETHING SEMI_GLOBAL
[3]   : Else
[4]       DO_SOMETHING_ELSE SEMI_GLOBAL
[5]   : EndIf
    ▾

    ▾ GOO
[1]   SEMI_GLOBAL ← 1
    ▾

```

In this example, it is formally impossible to predict in which order APL will execute statements in BUG, FOO or GOO from BUG[2] onwards. For example, the actual sequence of execution may be:

```

BUG[1] → BUG[2] → FOO[1] → FOO[2] →
        DO_SOMETHING[1]

```

or

```

BUG[1] → BUG[2] → BUG[3] → GOO[1] →
        FOO[1] → FOO[2] → FOO[3] →
        FOO[4] → DO_SOMETHING_ELSE[1]

```

This is because APL may switch from one thread to another between any two lines in a defined function. In practice, because APL gives each thread a significant time-slice, it is likely to execute many lines, maybe even hundreds of lines, in one thread before switching to another. However, you must not rely on this; **thread-switching may occur at any time between lines in a defined function.**

Secondly, consider the possibility that APL switches from the FOO thread to the GOO thread after FOO[1]. If this happens, the value of SEMI_GLOBAL passed to DO_SOMETHING will be 1 and not 0. Here is another source of error.

In this case, there are two ways to resolve the problem. To ensure that the value of SEMI_GLOBAL remains the same from FOO[1] to FOO[2], you can use diamonds instead of separate statements. For example:

```
:If SEMI_GLOBAL=0 ♦ DO_SOMETHING SEMI_GLOBAL
```

Even better, although less efficient, you can use `:Hold` to synchronise access to the variable. For example:

```

▽ FOO
[1]   :Hold 'SEMI_GLOBAL'
[2]       :If SEMI_GLOBAL=0
[3]           DO_SOMETHING SEMI_GLOBAL
[4]       :Else
[5]           DO_SOMETHING_ELSE SEMI_GLOBAL
[6]       :EndIf
[7]   :EndHold
▽

▽ GOO
[1]   :Hold 'SEMI_GLOBAL'
[2]       SEMI_GLOBAL←1
[3]   :EndHold
▽

```

Now, although you still cannot be sure which of `FOO` and `GOO` will run first, you can be sure that `SEMI_GLOBAL` will not change (because `GOO` cuts in) within `FOO`.

Note that the string used as the argument to `:Hold` is completely arbitrary, so long as threads competing for the same resource use the same string.

Warning

These types of problems are inherent in all multithreading programming languages, and not just with Dyalog APL. *If you want to take advantage of the additional power provided by multithreading, it is advisable to think carefully about the potential interaction between different threads.*

5.1.6 Threads & Niladic Functions

In common with other operators, the spawn operator `&` may accept monadic or dyadic functions as operands, but not niladic functions. This means that, using spawn, you cannot start a thread that consists only of a niladic function

If you wish to invoke a niladic function asynchronously, you have the following choices:

- Turn your niladic function into a monadic function by giving it a dummy argument which it ignores.
- Call your niladic function with a dfn to which you give an argument that is implicitly ignored. For example, if the function `NIL` is niladic, you can call it asynchronously using the expression: `{NIL}& 0`
- Call your function via a dummy monadic function, for example

```

    ▽ NIL_M DUMMY
[1]  NIL
    ▽
    NIL_M& ''

```

- Use `execute`, for example

```

⊞& 'NIL'

```

Note that niladic functions *can* be invoked asynchronously as callback functions. For example, the statement:

```

□WS'Event' 'Select' 'NIL&'

```

will execute correctly as a thread, even though `NIL` is niladic. This is because callback functions are invoked directly by `□DQ` rather than as an operand to the spawn operator.

5.1.7 Threads & External Functions

External functions in dynamic link libraries (DLLs) defined using the `□NA` interface may be run in separate C threads. Such threads:

- **take advantage of multiple processors** if the operating system permits.
- allow APL to **continue processing in parallel** during the execution of a `□NA` function.

When you define an external function using `□NA`, you may specify that the function be run in a separate C thread by appending an ampersand (&) to the function name, for example:

```

'beep'□NA'user32|MessageBeep& i'
A MessageBeep will run in a separate C thread

```

When APL first comes to execute a multi-threaded `⌈NA` function, it starts a new C-thread, executes the function within it, and waits for the result. Other APL threads may then run in parallel.

Note that when the `⌈NA` call finishes and returns its result, its new C-thread is retained to be re-used by any subsequent multithreaded `⌈NA` calls made within the same APL thread. Thus any APL thread that makes any multi-threaded `⌈NA` calls maintains a separate C-thread for their execution. This C-thread is discarded when its APL thread finishes.

Note that there is no point in specifying a `⌈NA` call to be multi-threaded, unless you wish to execute other APL threads at the same time.

In addition, if your `⌈NA` call needs to access an APL GUI object (strictly, a window or other handle) it should normally run within the same C-thread as APL itself, and not in a separate C-thread. This is because Windows associates objects with the C-thread that created them. Although you *can* use a multi-threaded `⌈NA` call to access (say) a Dyalog APL Form via its window handle, the effects may be different than if the `⌈NA` call was not multi-threaded. In general, `⌈NA` calls that access APL (GUI) objects should not be multi-threaded.

If you wish to run the same `⌈NA` call in separate APL threads at the same time, you must ensure that the DLL is *thread-safe*. Functions in DLLs which are not *thread-safe*, must be prevented from running concurrently by using the `:Hold` control structure. Note that all the standard Windows API DLLs **are** *thread safe*.

Notice that you may define two separate functions (with different names), one single-threaded and one multi-threaded, associated with the same function in the DLL. This allows you to call it in either way.

5.1.8 Synchronising Threads

Threads may be synchronised using *tokens* and a *token pool*.

An application can synchronise its threads by having one thread add tokens into the pool whilst other threads wait for tokens to become available and retrieve them from the pool.

Tokens possess two separate attributes, a *type* and a *value*.

The *type* of a token is a positive or negative numeric scalar. The *value* of a token is any arbitrary array that you might wish to associate with it.

The token pool may contain up to 2^{*31} tokens; they do not have to be unique neither in terms of their types nor of their values.

The following system functions are used to manage the token pool:

<code>□TALLOC</code>	Allocates ranges of tokens.
<code>□TPUT</code>	Puts tokens into the pool.
<code>□TGET</code>	If necessary waits for, and then retrieves some tokens from the pool.
<code>□TPOOL</code>	Reports the types of tokens in the pool
<code>□TREQ</code>	Reports the token requests from specific threads

A simple example of a thread synchronisation requirement occurs when you want one thread to reach a certain point in processing before a second thread can continue. Perhaps the first thread performs a calculation, and the second thread must wait until the result is available before it can be used.

This can be achieved by having the first thread put a specific type of token into the pool using `□TPUT`. The second thread waits (if necessary) for the new value to be available by calling `□TGET` with the same token type.

Notice that when `□TGET` returns, the specified tokens are *removed* from the pool. However, *negative* token types will satisfy an infinite number of requests for their positive equivalents.

The system is designed to cater for more complex forms of synchronisation. For example, a *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

A second complex example is that of a *latch* which holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads. A practical example of a latch is a ferry terminal.

5.1.9 Semaphore Example

A *semaphore* to control a number of resources can be implemented by keeping that number of tokens in the pool. Each thread will take a token while processing, and return it to the pool when it has finished.

For example, if we want to restrict the number of threads that can have sockets open at any one time.

```

    sock←99                                A socket-token
                                         any +ive number will do).
    □TPUT 5/sock                            A add 5 socket-tokens to pool.

    ▽ sock_open ...
[1]   :If sock=□TGET sock                A grab a socket token
[.]   ...                               A do stuff.
[.]   □TPUT sock                        A release socket token
[.]   :Else
[.]   error'sockets off'                A sockets switched off by
                                         retract (see below).
[.]   :EndIf
    ▽
    0 □TPUT □TREQ □TNUMS                A retract socket "service"
                                         with 0 value.

```

5.1.10 Latch Example

A *latch* holds back a number of threads until the coast is clear. At a signal from another thread, the latch is opened so that all of the threads are released. The latch may (or may not) then be closed again to hold up subsequently arriving threads.

A visual example of a latch might be a ferry terminal, where cars accumulate in the queue until the ferry arrives. The barrier is then opened and all (up to a maximum number) of the cars are allowed through it and on to the ferry. When the last car is through, the barrier is re-closed.

```

    tkt←6                                    A 6-token: ferry ticket.

    ▽ car ...
[1]   □TGET tkt                          A await ferry.
[2]   ...

    ▽ ferry ...
[1]   arrives in port
[2]   □TPUT(↑,□TREQ □TNUMS)ntkt        A ferry tickets for all.
[3]   ...

```

Note that it is easy to modify this example to provide a maximum number of ferry places per trip by inserting `max_places↑` between `□TPUT` and its argument. If fewer cars

than the ferry capacity are waiting, the `↑` will fill with trailing 0s. This will not cause problems because zero tokens are ignored.

Let us replace the car ferry with a new road bridge. Once the bridge is ready for traffic, the barrier could be opened permanently by putting a *negative* ticket in the pool.

```
□TPUT -tkt      A open ferry barrier permanently.
```

Cars could choose to take the last ferry if there are places:

```

▽ car ...
[1] :Select □TGET tkt
[2] :Case tkt ♦ take the last ferry.
[3] :Case -tkt ♦ ferry full: take the new bridge.
[4] :End

```

The above `:Select` works because by default, `□TPUT -tkt` puts a *value* of `-tkt` into the token.

5.1.11 Debugging Threads

If a thread sustains an untrapped error, its execution is *suspended* in the normal way. If the *Pause on Error* option is set, all other threads are *paused*. If *Pause on Error* option is not set, other threads will continue running and it is possible for another thread to encounter an error and suspend (see the *Dyalog for Microsoft Windows Installation and Configuration Guide*).

Using the facilities provided by the Tracer and the Threads Tool (see the *Dyalog for Microsoft Windows UI Guide*) it is possible to interrupt (suspend) and restart individual threads, and to pause and resume individual threads, so any thread may be in one of three states - *running*, *suspended* or *paused*.

The Tracer and the Session may be connected with any suspended thread and you can switch the attention of the Session and the Tracer between suspended threads using `)TID` or by clicking on the appropriate tab in the Tracer. At this point, you may:

- Examine and modify local variables for the currently suspended thread.
- Trace and edit functions in the current thread.
- Cut back the stack in the currently suspended thread.
- Restart execution.
- Start new threads

The error message from a thread other than the base is prefixed with its thread number:

```
260:DOMAIN ERROR
Div[2] rslt←num÷div
    ^
```

State indicator displays:)SI and)SINL have been extended to show threads' tree-like calling structure.

```
    )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, have been suspended at line [2].

Removing stack frames using Quit from the Tracer or → from the session affects only the current thread. When the final stack frame in a thread (other than the base thread) is removed, the thread is expunged.

)RESET removes all but the base thread.

Note the distinction between a *suspended* thread and a *paused* thread.

A *suspended* thread is stopped at the beginning of a line in a defined function or operator. It may be connected to the Session so that expressions executed in the Session do so in the context of that thread. It may be *restarted* by executing →line (typically, →LC).

A *paused* thread is an inactive thread that is currently being ignored by the thread scheduler. A paused thread may be paused within a call to □DQ, a call on an external function, at the beginning of a line, or indeed at any of the thread-switching points described earlier in this chapter.

A paused thread may be *resumed* only by the action of a menu item or button. A paused thread resumes only in the sense that it ceases to be ignored by the thread

scheduler and will therefore be switched back to at some point in the future. It does not actually continue executing until the switch occurs.

5.2 Triggers

5.2.1 Triggers

Triggers provide the ability to have a function called automatically whenever a variable or a Field is assigned. Triggers are actioned by all forms of assignment ($\leftarrow$), but only by assignment.

Triggers are designed to allow a class to perform some action when a field is modified – without having to turn the field into a property and use the property setter function to achieve this. Avoiding the use of a property allows the full use of the APL language to manipulate data in a field, without having to copy field data in and out of the class through get and set functions.

Triggers *can* also be applied to variables outside a class, and there will be situations where this is very useful. However, dynamically attaching and detaching a trigger from a variable is a little tricky at present.

The function that is called when a variable or Field changes is referred to as the *Trigger Function*. The name of a variable or Field which has an associated Trigger Function is termed a *Trigger*.

A function is declared as a Trigger function by including the statement:

```
:Implements Trigger Name1,Name2,Name3, ...
```

where Name1, Name2 etc. are the Triggers.

When a Trigger function is invoked, it is passed an Instance of the internal Class `TriggerArguments`. This Class has 3 Fields:

Member	Description
Name	Name of the Trigger whose change in value has caused the Trigger Function to be invoked.
NewValue	The newly assigned value of the Trigger
OldValue	The previous value of the Trigger. If the Trigger was not previously defined, a reference to this Field causes a <code>VALUE ERROR</code> .

A Trigger Function is called *as soon as possible* after the value of a Trigger was assigned; typically by the end of the currently executing line of APL code. The precise timing is

not guaranteed and may not be consistent because internal workspace management operations can occur at any time.

If the value of a Trigger is changed more than once by a line of code, the Trigger Function will be called at least once, but the number of times is not guaranteed.

A Trigger Function is not called when the Trigger is expunged.

Expunging a Trigger disconnects the name from the Trigger Function and the Trigger Function will not be invoked when the Trigger is reassigned. The connection may be re-established by re-fixing the Trigger Function.

A Trigger may have only a single Trigger Function. If the Trigger is named in more than one Trigger Function, the Trigger Function that was last fixed will apply.

In general, it is inadvisable for a Trigger function to modify its own Trigger, as this will potentially cause the Trigger to be invoked repeatedly and forever.

To associate a Trigger function with a *local* name, it is necessary to dynamically fix the Trigger function in the function in which the Trigger is localised; for example:

```

    ▾ TRIG arg
[1]   :Implements Trigger A
[2]   ...

    ▾ TEST;A
[1]   □FX □OR 'TRIG'
[2]   A←10

```

5.2.2 Trigger Example

The following function displays information when the value of variables A or B changes.

```

    ▾ TRIG arg
[1]   :Implements Trigger A,B
[2]   arg.Name'is now 'arg.NewValue
[3]   :Trap 6 A VALUE ERROR
[4]       arg.Name'was      'arg.OldValue
[5]   :Else
[6]       arg.Name' was      [undefined]'
[7]   :EndTrap
    ▾

```

Note that on the very first assignment to A, when the variable was previously undefined, `arg.OldValue` is a `VALUE ERROR`.

```

    A←10
A is now 10
A was [undefined]

    A←10
A is now 20
A was 10

    A←'Hello World'
A is now Hello World
A was 20

    A[1]←c2 3pi6
A is now 1 2 3 ello World
        4 5 6
A was Hello World

    B←φ**A
B is now 3 2 1 ello World
        6 5 4
B was [undefined]

    A←NEW MyClass
A is now #.[Instance of MyClass]
A was 1 2 3 ello World
        4 5 6

    'F'←WC'Form'
    A←F
A is now #.F
A was #.[Instance of MyClass]
    
```

Note that Trigger functions are actioned only by assignment, so changing A to a Form using `WC` does not invoke `TRIG`.

```
'A'←WC'FORM' A Note that Trigger Function is not invoked
```

However, the connection (between A and `TRIG`) remains and the Trigger Function will be invoked if and when the Trigger is re-assigned.

```

      A←99
A is now 99
A was    #.A

```

See [Section 5.2.4](#) for information on how a Field (in a Class) may be used as a Trigger.

5.2.3 Global Triggers

A global Trigger is a function that triggers on any assignment to a global variable in the same namespace. Global Triggers may be disabled and re-enabled using 2007⌘. See *Dyalog APL Language: Disable Global Triggers*.

This is implemented by the function declaration statement:

```
:Implements Trigger *
```

The argument to the trigger function is an instance of the internal class `TriggerArguments` which contains the following members:

Member	Description
Name	The name of the global variable that is about to be changed.
Indexers	If the assignment is some form of indexed assignment, <code>Indexers</code> is an array with the same shape as the sub-array that was assigned and contains the ravel-order, ⌈IO -sensitive, indices of the changed elements. Otherwise, <code>Indexers</code> is undefined.

EXAMPLE

```

▽ foo args
[1]   :Implements Trigger *
[2]   args.Name'has changed'
[3]   :If 2=args.⌈NC'Indexers'
[4]       'pIndexers'(pargs.Indexers)
[5]       'Indexers'(;args.Indexers)
[6]   :EndIf
▽

```

```
vec←15
vec has changed
```

```
a b←10 'Pete'
a has changed
b has changed
```

```
vec[2 4]←99
vec has changed
ρIndexers 2
Indexers 2 4
```

```
array←2 3 4ρ12
array has changed
```

```
(2 1 3↑array)←42
array has changed
ρIndexers 2 1 3
Indexers 1 2 3 13 14 15
```

Note

- like other Triggers, only the most recently fixed global trigger function will apply and be called on assignment to a global variable.
- global triggers do not apply to local names nor to semi-globals (names which are localised further up the stack).
- an assignment to a global variable will fire both its specific trigger (if defined) and the global trigger. However, the order of execution is undefined.
- do not use an argument name for your trigger function that may conflict with a global variable name in the namespace.

Further Example

A potential use for a global trigger is to detect the unintended creation of global variables due to localisation omissions. Note however that the timing of the activation of the Trigger is unpredictable. In this example, the trigger for the assignment to `b` activates after function `hoo` has exited. When Threads are involved, timing becomes even less predictable.

```

    ▼ CatchGlobals arg
[1]   a Displays a warning when a global is assigned
[2]   :Implements Trigger *
[3]   '*** assignment to global variable: ',
      arg.Name, ' from ', 1+[]SI
      ▼
      ▼ foo
[1]   goo
      ▼
      ▼ goo
[1]   hoo
      ▼
      ▼ hoo
[1]   a←10
[2]   b←a
      ▼
      foo
*** assignment to global variable: a from  hoo  goo  foo
*** assignment to global variable: b from  goo  foo

```

5.2.4 Trigger Fields

A field may act as a [Section 5.2.1](#) so that a function may be invoked whenever the value of the Field is changed.

As an example, it is often useful for the Display Form of an Instance to reflect the value of a certain Field. Naturally, when the Field changes, it is desirable to change the Display Form. This can be achieved by making the Field a Trigger as illustrated by the following example.

Notice that the Trigger function is invoked both by assignments made within the Class (as in the assignment in `ctor`) and those made from outside the Instance.

```

:Class MyClass
  :Field Public Name
  :Field Public Country←'England'
  ▽ ctor nm
    :Access Public
    :Implements Constructor
    Name←nm
  ▽
  ▽ format
    :Implements Trigger Name, Country
    □DF 'My name is ', Name, ' and I live in ', Country
  ▽
:EndClass A MyClass

  me←□NEW MyClass 'Pete'
  me
My name is Pete and I live in England

  me.Country←'Greece'
  me
My name is Pete and I live in Greece

  me.Name←'Kostas'
  me
My name is Kostas and I live in Greece

```

6 APL Files

6.1 Native Files

Introduction

Dyalog includes a wide selection of functions that allow reading/writing to files containing text or binary data, as well as the manipulation of the host filesystem, for example, deleting files or creating directories.

The characteristics of host filesystems vary across different platforms and even across devices on the same platform. Different host operating systems provide access to their filesystems in largely incompatible ways. Nevertheless, the native file functions within Dyalog work in broadly the same way across all platforms, which makes it relatively straightforward to create applications which are portable across all Dyalog environments.

Text Files

Characters within a text file may be encoded in one of a number of different ways, and different host environments tend to have different preferences. Although UTF-8 is increasingly becoming the *de facto* encoding standard, other encoding formats exist (particularly in legacy environments) and there are still multiple conventions for representing line endings.

Dyalog includes two powerful functions – *Dyalog APL Language: Nget* and *Dyalog APL Language: Nput* – which read and write text files to or from character arrays in the workspace. The encoding and line-ending types can be explicitly specified but by default `⎕NGET` will try to deduce the encoding automatically and `⎕NPUT` will use defaults appropriate for the host environment.

The search and replace functions *Dyalog APL Language: S* and *Dyalog APL Language: R* can also be used to read and write text files, filtering and modifying the content as they do so.

The simplest text files contain just plain text – variable length lines of text with no formatting such as italics etc. However, formatting or data encoding can be included within a text file using formats such as markup (e.g. HTML and XML), character separated

values (CSV) or JavaScript Object Notation (JSON). Dyalog includes the functions *Dyalog APL Language: Xml*, *Dyalog APL Language: Csv* and *Dyalog APL Language: Json* to decode or encode such file content.

Binary Data Files

Binary data files contain data in application-specific format, and are rarely read or written by anything other than the application which creates them and understands their format. Dyalog provides a number of functions which allow a Dyalog application to manage its own binary data files or binary data files from any source, by allowing them to be read or written as sequences of bytes or words. Regions of binary files can also be locked to coordinate shared access to the files.

Files are tied (opened) using *Dyalog APL Language: Ntie* or created using *Dyalog APL Language: Ncreate*. Both of these functions return a numeric tie number by which the file is subsequently identified when read (*Dyalog APL Language: Nread*), written (*Dyalog APL Language: Nappend*, *Dyalog APL Language: Nreplace*), renamed (*Dyalog APL Language: Nrename*) locked or unlocked (*Dyalog APL Language: Nlock*) or (re)sized (*Dyalog APL Language: Nsize*, *Dyalog APL Language: Nresize*). When a file tie is no longer needed it can be untied (closed) using *Dyalog APL Language: Nuntie* or closed and the file deleted using *Dyalog APL Language: Nerase*. *Dyalog APL Language: Nnums* and *Dyalog APL Language: Nnames* report the numbers and names respectively of all currently tied files.

Filesystem Manipulation

In addition to reading and writing files, an application may typically want to manipulate the filesystem in various ways. A diverse set of functions exist to do this: *Dyalog APL Language: Mkdir* creates directories, *Dyalog APL Language: Ndelete* deletes files and/or directories and their contents, *Dyalog APL Language: Nmove* and *Dyalog APL Language: Ncopy* move and copy files and/or directories and their contents. *Dyalog APL Language: Ninfo* queries and sets properties such as size or modification date on files and directories, and can also query the contents of directories. *Dyalog APL Language: Nexists* will report whether or not a name exists within the file system.

Additionally, 7391 can be used to obtain the name of the directory used by the host filesystem for temporary files (generally emptied at startup).

Progress Callbacks

The native file functions `⎕NCOPY`, `⎕NMOVE`, and `⎕NINFO` support the **ProgressCallback** variant option to enable progress callbacks.

Overview

If this option is enabled, the system function invokes an APL callback function at several stages as a file operation proceeds, meaning that the results of the system function can be accessed as they become available rather than waiting for them to all be available. A system object is used to communicate between the system function and the callback.

Each file operation has four distinct stages:

1. The start of the operation. The callback is invoked before any directories are scanned or files are processed. This gives the application the opportunity to set parameters that control the frequency of callbacks during the operation itself.
2. The optional scan phase during which the system function enumerates the files that will be involved in the copy or move operation. The file count obtained is used to set the `Limit` field. The application may use this subsequently to indicate the degree of progress.
3. The main processing of the files.
4. The end of the operation.

The callback function is invoked once at the start of the operation, during the (optional) scan and processing stages, and finally once at the end of the operation. During the scan and processing stages, the `Skip` and `Delay` options provide alternative ways to control the frequency with which the function is invoked.

If both options are 0, the callback will be invoked after every file is processed. However, if there are a large number of small files involved, and you simply want to update a progress bar, this may prove to be unnecessarily frequent, and will increase the total time required to complete the operation.

If you want to update a progress bar regularly (for example every second), the `Delay` option (1000 = 1 second) is the better choice. In other circumstances, you might choose to use `Skip`.

If you use both options, the callback will be invoked when *both* apply, so if you set `Skip` to 10 and `Delay` to 5000, the callback will be invoked after at least 10 files have been processed and at least 5 seconds have elapsed since the previous invocation of the callback.

The value of the **ProgressCallback** variant option may be:

<code>fn</code>	The name of the callback function.
<code>fn data</code>	The name of the callback function, and an array or namespace which is to be passed to the callback in its left argument.

The right argument given to the callback function is a 3-element vector:

[1]	Function	Character vector which identifies the function that caused the callback to be executed.
[2]	Event	Character vector describing the event that lead to the callback being executed. See below.
[3]	Info	Reference to a namespace containing information about the event. See below.

Event

Event is a character vector which indicates the stage of the copy or move operation.

'Start'	Reported by the first invocation of the callback which occurs before any files are scanned or processed. This may be used to set the parameters that control the operation. See Options .
'Scan'	Indicates that the system function is in the initial phase of scanning the files in order to calculate <code>Limit</code> . See ScanFirst .
'Progress'	Indicates that the system function is at the main stage of the operation and is processing the files.
'Done'	Indicates that all files have been processed.

Note that there will always be at least 2 invocations of the callback, to indicate the start and end of the operation.

Info

Info is a reference to a namespace that contains information about the event. This namespace persists for the duration of the execution of the system function and contains the following fields:

Progress	A number between 0 and Limit. When the event code is 'Start', Progress is 0. Every time a file or directory is processed, Progress is increased by 1. Finally when the event code is 'Done', Progress will be equal to Limit.
Limit	The maximum value of Progress. This value might change during the file operation if it does not do a full discovery first (the ScanFirst option is 0), or if the file structure changes between the scan and the file operation.
Last	A vector of file names which have been processed since the last invocation of the callback function. The user can specify the maximum length of this vector by setting the LastFileCount option. The names in this list are the source names rather than the destination names. The Last vector is always empty when the event is 'Start' and it is cleared when going from the 'Scan' phase to the 'Progress' phase, to avoid any confusion.
Data	A field that is reserved for the user to store data which persists between invocations of the callback. It could for example be used to keep a sequence number, to count the number of times the callback had been run.
Options	This is a namespace which contains the information that controls the future execution of the callback and it is described below.

Options

This Namespace contains options that control future invocations of the callback. The options persist between these invocations, so there is no need to set them again unless they should be changed. The fields and their default values are:

Field	Default	Description
<code>ScanFirst</code>	1	Specifies if the file operation should do a "scan pass" before processing the files. This stage just enumerates the files to determine how many there are. This will ensure <code>Limit</code> has a realistic value when the actual processing of the files happens. The <code>ScanFirst</code> field is only inspected right after the first invocation of the callback function, with the event code <code>'Start'</code> .
<code>Delay</code>	0	Specifies the number of milliseconds to wait until the callback will be called again. If all file operations finish before this time, the callback function is called anyway, with the event code <code>'Done'</code> . If a slow file operation is happening (such as copying a big file), the actual delay before the callback is invoked might be longer than the value of <code>Delay</code> .
<code>Skip</code>	0	Specifies a number of files to skip between invocations of the callback function. If you are only interested in getting a callback for each tenth file, you should set this option to 9, for example.
<code>LastFileCount</code>	1	An integer, specifying the maximum number of the latest filenames to be stored in the <code>Last</code> field. The default is to only store the last file processed, but if <code>Delay</code> or <code>Skip</code> are non-zero, multiple files could have been processed between calls to the callback function. A value of 5 for example, will make sure that the five last files processed before calling the callback, will have their names in the <code>Last</code> field. The <code>Last</code> field might have fewer elements than <code>LastFileCount</code> , if the number of files processed since the last call is less than <code>LastFileCount</code> . The value <code>-1</code> indicates that the <code>Last</code> field should contain all the last files since the last call (no limit).

The result of the callback function is a Boolean scalar indicating whether the system function should continue or stop:

1: Execution should continue.

0: Execution should stop. In this case, an `INTERRUPT` (event 1003) is signalled.

6.2 Component Files

6.2.1 Introduction

Most languages store programs and data separately. APL is unusual in that it allows you to store programs and data together in a workspace.

This can be inefficient if your dataset gets very large; when your workspace is loaded, you are loading ALL of your data, whether you need it or not.

It also makes it difficult for other users to access your data, particularly if you want them to be able to update it.

In these circumstances, you must extract your data from your workspace, and write it to a file on disk, thus separating your data from your program. There are many different kinds of file format. This section is concerned with the APL Component File system which preserves the idea that your data consists of APL objects; hence you can only access this type of file from within APL

The Component File system has a set of system functions through which you access the file. Although this means that you have to learn a whole new set of functions in order to use files, you will find that they provide you with a very powerful mechanism to control access to your data.

6.2.2 Component Files

Overview

A **component file** is a data file maintained by Dyalog APL. It contains a series of APL arrays known as **components** which are accessed by reference to their relative position or **component number** within the file. Component files are just like other data files and there are no special restrictions imposed on names or sizes.

A set of system functions is supplied to perform a range of file operations. These provide facilities to create or delete files, and to read and write components. Facilities are also provided for multi-user access, including the capability to determine who may do what, and file locking for concurrent updates.

Tying and Untying Files

To access an existing component file it must be **tied**, that is, opened for use. The tie may be **exclusive** (single-user access) or **shared** (multi-user access). A file is **untied**, that is, closed, using `□FUNTIE` or on terminating Dyalog APL. File ties survive `)LOAD`, `□LOAD` and `)CLEAR` operations.

Tie Numbers

A file is tied by associating a **file name** with a **tie number**. Tie numbers are integers in the range 1 - 2147483647 and, you can supply one explicitly, or have the interpreter allocate the next available one by specifying 0. The system functions which tie files return the tie number as a "shy" result.

Creating and Removing Files

A component file is created using `□FCREATE` which automatically ties the file for exclusive use. A newly created file is empty, that is, contains 0 components. A file is removed with `□FERASE`, although it must be exclusively tied to do so.

Adding and Removing Components

Components are added to a file using `□FAPPEND` and removed using `□FDROP`. Component numbers are allocated consecutively starting at 1. Thus a new component added by `□FAPPEND` is given a component number which is one greater than that of the last component in the file. Components may be removed from the beginning or end of the file, but not from the middle. Component numbers are therefore contiguous.

Reading and Writing Components

Components are read using `□FREAD` and overwritten using `□FREPLACE`. There are no restrictions on the size or type of array which may replace an existing component. Components are accessed by component number.

Component Information

In addition to the data held in a component, the user ID that wrote it and the time at which it was written is also recorded.

Multi-User Access

`□FSTIE` ties a file for **shared** (that is, multi-user) access. This kind of access would be appropriate for a multi-user UNIX system, a network of single user PCs, or multiple APL tasks under Microsoft Windows.

`□FHOLD` provides the means for the user to temporarily prevent other co-operating users from accessing one or more files. This is necessary to allow a single logical update involving more than one component, and perhaps more than one file, to be completed without interference from another user. `□FHOLD` is applicable to External Variables as well as Component Files

File Access Control

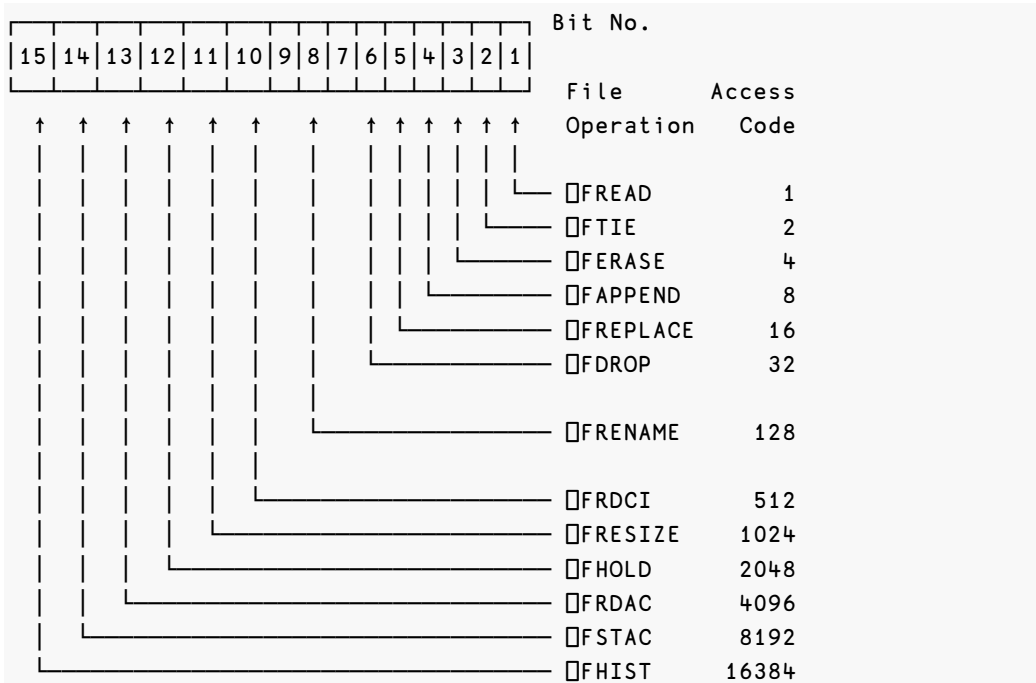
There are two levels of file access control. As a regular file, the operating system read/write controls for owner and other users apply. In addition, Dyalog manages its own access controls using the **access matrix**. This is an integer matrix with 3 columns and any number of rows. Column 1 contains user numbers, column 2 an encoding of permitted file operations, and column 3 passnumbers. Each row specifies which file operations may be performed by which user(s) with which passnumber. A value of 0 in column 1 specifies all users. A value of `~1` in column 2 specifies all file operations. A value of 0 in column 3 specifies no passnumber. If any row of the access matrix contains `(0 ~1 0)` it specifies that all users may perform all file operations with no passnumber.

User Number

Under Windows, this is a number which is defined by the **aplnid** parameter. If you intend to use Dyalog's **access matrix** to control file access in a multi-user environment, it is desirable to allocate to each user, a distinct **user number**. However, if you intend to rely on underlying operating system controls, allocating a user number of 0 (the default installation value) to everyone is more appropriate. Under non-Windows platforms the User Number is set to be the effective user-id of the APL process and cannot be altered. In both cases, a user number of 0 causes APL to circumvent the access matrix mechanism described below.

Permission Code

This is an integer representation of a Boolean mask. Each bit in the mask indicates whether or not a particular file operation is permitted as follows:



For example, if bits 1, 4 and 6 are set and all other relevant bits are zero only ☐FREAD, ☐FAPPEND and ☐FDROP are permitted. A convenient way to set up the mask is to sum the access codes associated with each operation.

For example, the value 41 (1+8+32) authorises ☐FREAD, ☐FAPPEND and ☐FDROP. A value of ~ 1 (all bits set) permits all operations. Thus by subtracting the access codes of operations to be forbidden, it is possible to permit all but certain types of access. For example, a value of ~ 133 ($\sim 1 - 4 + 128$) permits all operations except ☐FERASE and ☐FRENAME. Note that the value of unused bits is ignored. Any non-zero permission code allows ☐FSTIE and ☐FSIZE. ☐FCREATE, ☐FUNTIE, ☐FLIB, ☐FNAMES and ☐FNOMS are not subject to access control. Passnumbers may also be used to establish different levels of access for the same user.

When the user attempts to tie a file using ☐FTIE or ☐FSTIE a row of the access matrix is selected to control this and subsequent operations.

If the user is the owner, and the owner's user ID does not appear in the access matrix, the value (☐AI[1] ~ 1 0) is conceptually appended to the access matrix. This ensures that the owner has full access rights unless they are explicitly restricted.

The chosen row is the first row in which the value in column 1 of the access matrix matches the user ID and the value in column 3 matches the supplied passnumber which is taken to be zero if omitted.

If there is no match of user ID and passnumber in the access matrix (including implicitly added rows) then no access is granted and the tie fails with a `FILE ACCESS ERROR`.

Once the applicable row of the access matrix is selected, it is used to verify all subsequent file operations. The passnumber used to tie the file **MUST** be used for every subsequent operation. Secondly, the appropriate bit in the permission code corresponding to the file operation in question must be set. If either of these conditions is broken, the operation will fail with `FILE ACCESS ERROR`.

If the access matrix is changed while a user has the file tied, the change takes immediate effect. When the user next attempts to access the file, the applicable row in the access matrix will be reselected subject to the supplied passnumber being the same as that used to tie the file. If access with that password is rescinded the operation will fail with `FILE ACCESS ERROR`.

When a file is created using `□FCREATE`, the access matrix is empty. At this stage, the owner has full access with passnumber 0, but no access with a non-zero passnumber. Other users have no access permissions. Thus only the owner may initialise the access matrix.

User 0

If a user has an **aplnid** of 0, the access matrix and supplied passnumbers are ignored. This user is granted full and unrestricted access rights to all component files, subject only to underlying operating system restrictions.

General File Operations

`□FLIB` gives a list of **component files** in a given directory. `□FNAMES` and `□FNUMS` give a list of the names and tie numbers of tied files. These general operations which apply to more than one file are not subject to access controls.

Component File System Functions

See *Language Reference* for full details of the syntax of these system functions.

General	
□FAVAIL	Report file system availability
File Operations	
□FCREATE	Create a file
□FTIE	Tie an existing file (exclusive)
□FSTIE	Tie an existing file (shared)
□FUNTIE	Untie file(s)
□FCOPY	Copy a file
□FERASE	Erase a file
□FRENAME	Rename a file
File information	
□FHIST	Report file events
□FNUMS	Report tie numbers of tied files
□FNAMES	Report names of tied files
□FLIB	Report names of component files
□FPROPS	Report file properties
□FSIZE	Report size of file
Writing to the file	
□FAPPEND	Append a component to the file
□FREPLACE	Replace an existing component
Reading from a file	
□FREAD	Read one or more components
□FRDCI	Read component information
Manipulating a file	
□FDROP	Drop a block of components
□FRESIZE	Change file size (forces a compaction)
□FCHK	Check and repair a file
Access manipulation	

☐ FSTAC	Set file access matrix
☐ FRDAC	Read file access matrix
Control multi-user access	
☐ FHOLD	Hold file(s) - see later section for details

Using the Component File System

Let us suppose that you have written an APL system that builds a personnel database, containing the name, age and place of birth of each employee. Let us assume that you have created a variable DATA, which is a nested vector with each element containing a person's name, age and place of birth:

```
DISPLAY 2↑DATA
```

Then the following APL expressions can be used to access the database:

EXAMPLE 1

Show record 2

```
DISPLAY 2>DATA
```

EXAMPLE 2

How many people in the database?

```
ρDATA
```

123

EXAMPLE 3

Update Pauline's age

```
(2 2>DATA)←16
```

EXAMPLE 4

Add a new record to the database

```
DATA , ← c 'Maurice' 18 'London'
```

Now let's build a component file to hold our personnel database.

Create a new file, giving the file name, and the number you wish to use to identify it (the file tie number):

```
'COMPFILE'  FCREATE 1
```

If the file already exists, or you have already used this tie number, then APL will respond with the appropriate error message.

Now write the data to the file. We could write a function that loops to do this, but it is neater to take advantage of the fact that our data is a nested vector, and use each ("").

DATA ☐ FAPPEND** 1

Now we'll try our previous examples using this file.

EXAMPLE 1

Show record 2

DISPLAY ☐ FREAD 1 2

Pauline	21
Isleworth	

EXAMPLE 2

How many people in our database?

```

    □FSIZE 1      A First component, next
1 125 10324 4294967295 A component, file size,
                        A maximum file size

```

```

    ~1+2>□FSIZE 1  A Number of data items

```

The fourth element of □FSIZE indicates the file size limit. Dyalog APL does not impose a file size limit, although your operating system may do so, but the concept is retained in order to make this version of Component Files compatible with others.

EXAMPLE 3

Update Pauline's age

```

    REC ← □FREAD 1 2      A Read second component
    REC[2] ← 18           A Change age
    REC □FREPLACE 1 2      A And replace component

```

EXAMPLE 4

Add a new record

```

    ('Janet' 25 'Basingstoke') □FAPPEND 1

```

EXAMPLE 5

Rename our file

```

    'PERSONNEL' □FRENAME 1

```

EXAMPLE 6

Tie an existing file; give file name and have the interpreter allocate the next available tie number.

```

    'SALARIES' □FTIE 0
2

```

EXAMPLE 7

Give everyone access to the PERSONNEL file

```

    (1 3ρ0 ~1 0)□FSTAC 1

```

EXAMPLE 8

Set different permissions on SALARIES.

```
AM ← 1 3p1 1 0      ⍝ Owner ID 1 has full access
AM ⍵← 102 1 0        ⍝ User ID 102 has READ only
AM ⍵← 210 2073 0     ⍝ User ID 210 has
                     ⍝ READ+APPEND+REPLACE+HOLD
```

```
AM ⍵FSTAC 2          ⍝ Store access matrix
```

EXAMPLE 9

Report on file names and associated numbers

```
⍵FNAMES,⍵FNUMS
PERSONNEL 1
SALARIES  2
```

EXAMPLE 10

Untie all files

```
⍵FUNTIE ⍵FNUMS
```

6.2.3 Programming Techniques

Controlling Multi-User Access

Obviously, Dyalog APL contains mechanisms that prevent data getting mixed up if two users update a file at the same time. However, it is the programmer's responsibility to control the logic of multi-user updates.

For example, suppose two people are updating our database at the same time. The first checks to see if there is an entry for 'Geoff', sees that there isn't so adds a new record. Meanwhile, the second user is checking for the same thing, and so also adds a record for 'Geoff'. Each user would be running code similar to that shown below:

```

▼ UPDATE;DATA;NAMES
[1]  A Using the component file
[2]  'PERSONNEL' □FSTIE 1
[3]  NAMES←□FREAD " 1,"τ-1+2□FSIZE 1
[4]  →END×τ(←'Geoff')∈NAMES
[5]  ('Geoff' 41 'Houns low')□FAPPEND 1
[6]  END:□FUNTIE 1
▼

```

The system function □FHOLD provides the means for the user to temporarily prevent other co-operating users from accessing one or more files. This is necessary to allow a single logical update, perhaps involving more than one record or more than one file, to be completed without interference from another user.

The code above is replaced by that below:

```

▼ UPDATE;DATA;NAMES
[1]  A Using the component file
[2]  'PERSONNEL' □FSTIE 1
[3]  □FHOLD 1
[4]  NAMES←□FREAD " 1,"τ-1+2□FSIZE 1
[5]  →END×τ(←'Geoff')∈NAMES
[6]  ('Geoff' 41 'Houns low')□FAPPEND 1
[7]  END:□FUNTIE 1 ♦ □FHOLD τ0
▼

```

Successive □FHOLDS on a file executed by different users are queued by Dyalog APL; once the first □FHOLD is released, the next on the queue holds the file. □FHOLDS are released by return to immediate execution, by □FHOLD 0, or by erasing the external variable.

It is easy to misunderstand the effect of □FHOLD. It is NOT a file locking mechanism that prevents other users from accessing the file. It only works if the tasks that wish to access the file co-operate by queuing for access by issuing □FHOLDS. It would be very inefficient to issue a □FHOLD on a file then allow the user to interactively edit the data with the hold in operation. What happens if he goes to lunch? Any other user who wants to access the file and cooperates by issuing a □FHOLD would have to wait in the queue for 3 hours until the first user returns, finishes his update and his □FHOLD is released. It is usually more efficient (as well as more friendly) to issue □FHOLDS around a small piece of critical code.

Suppose we had a control file associated with our personnel data base. This control file could be an external variable, or a component file. In both cases, the concept is the

same; only the commands needed to access the file are different. In this example, we will use a component file:

```
'CONTROL' □FCREATE 1      A Create control file
(1 3p0 -1 0) □FSTAC 1     A Allow everyone access
⊕ □FAPPEND 1              A Set component 1 to empty
□FUNTIE 1                 A And untie it
```

Now we'll allow our man that likes long lunch breaks to edit the file, but will control the hold in a more efficient way:

```
▼ EDIT;CMP;CV
[1]  A Share-tie the control file
[2]  'CONTROL' □FSTIE 1
[3]  A Share-tie the data file
[4]  'PERSONNEL' □FSTIE 2
[5]  A Find out which component the user wants to edit
[6]  ASK: CMP←ASKΔWHICHΔRECORD
[7]  A Hold the control file
[8]  □FHOLD 1
[9]  A Read the control vector
[10] CV←□FREAD 1 1
[11] A Make control vector as big as the data file
[12] CV←(-1+2>□FSIZE 2)↑CV
[13] A Look at flag for this component
[14] →(FREE, INUSE)[1+CMP>CV]
[15] A In use - tell user and release hold
[16] INUSE: 'Record in use' ♦ □FHOLD ⊕ ♦ →ASK
[17] A Ok to use - flag in-use and release hold
[18] FREE: CV[CMP]←1 ♦ CV □FREPLACE 1 1 ♦ □FHOLD ⊕
[19] A Let user edit the record
[20] EDITΔRECORD RECORD
[21] A When he's finished, clear the control vector
[22] □FHOLD 1
[23] CV←□FREAD 1 1 ♦ CV[CMP]←0 ♦ CV □FREPLACE 1 1
[26] □FHOLD ⊕
[27] A And repeat
[28] →ASK
▼
```

Component 1 of our CONTROL file acts as a control vector. Its length is set equal to the number of components in the PERSONNEL file, and an element is set to 1 if a user wishes to access the corresponding data component. Only the control file is ever

subject to a `□F HOLD`, and then only for a split-second, with no user inter-action being performed whilst the hold is active.

When the first user runs the function, the relevant entry in the control vector will be set to 1. If a second user accesses the database at the same time, he will have to wait briefly whilst the control vector is updated. If he wants the same component as the first user, he will be told that it is in use, and will be given the opportunity to edit something else.

This simple mechanism allows us to lock the components of our file, rather than the entire file. You can set up more informative control vectors than the one above; for example, you could easily put the user name into the control vector and this would enable you to tell the next user who is editing the component he is interested in.

6.2.4 File Design

Our personnel database could be termed a *record oriented* system. All the information relating to one person is easily obtained, and information relating to a new person is easily added, but if we wish to find the oldest person, we have to read ALL the records in the file.

It is sometimes more useful to have separate components, perhaps stored on separate files, that hold indexes of the data fields that you may wish to search on. For example, suppose we know that we always want to access our personnel database by name. Then it would make sense to hold an index component of names:

```
A Extract name field from each data record
'PERSONNEL' □FSTIE 1
NAMES←□FREAD"1,""1+2□FSIZE 2
```

```
A Create index file, and append NAMES
'INDEX' □FCREATE 2
NAMES □FAPPEND 2
```

Then if we want to find Pauline's data record:

```
NAMES←□FREAD 2,1      A Read index of names
CMP←NAMES1<'Pauline'   A Search for Pauline
DATA←□FREAD 1,CMP      A Read relevant record
```

There are many different ways to structure data files; you must design a structure that is the most efficient for your application.

6.2.5 Internal Structure

If you are going to make a lot of use of APL files in your systems, it is useful for you to have a rough idea of how Dyalog APL organises and manages the disk area used by such files.

The internal structure of external variables and component files is the same, and the examples given below apply to both.

Consider a component file with 3 components:

```
'TEMP'  FCREATE 1
'One'  'Two'  'Three'  FAPPEND 1
```

Dyalog APL will write these components onto contiguous areas of disk:

1	2	3
One	Two	Three

Replace the second component with something the same size:

```
'Six'  []FREPLACE 1 2
```

This will fit into the area currently used by component 2.

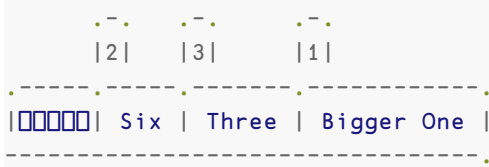
1	2	3
One	Six	Three

If your system uses fixed length records, then the size of your components never change, and the internal structure of the file remains static.

However, suppose we start replacing larger data objects:

```
'Bigger One'  □FREPLACE 1 1
```

This will not fit into the area currently assigned to component 1, so it is appended to the end of the file. Dyalog APL maintains internal tables which contain the location of each component; hence, even though the components may not be physically stored in order, they can always be accessed in order.

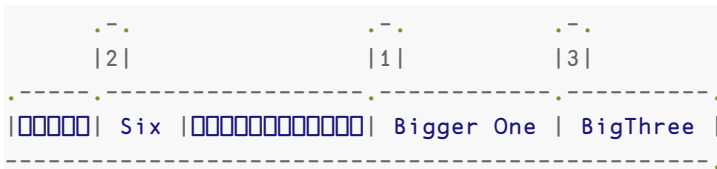


The area that was occupied by component 1 now becomes free.

Now we'll replace component 3 with something bigger:

```
'BigThree' ⎕FREPLACE 1 3
```

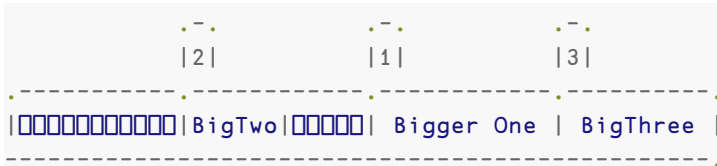
Component 3 is appended to the end of the file, and the area that was used before becomes free:



Dyalog APL keeps tables of the size and location of the free areas, as well as the actual location of your data. Now we'll replace component 2 with something bigger:

```
'BigTwo' ⎕FREPLACE 1 2
```

Free areas are used whenever possible, and contiguous holes are amalgamated.



You can see that if you are continually updating your file with larger data objects, then the file structure can become fragmented. At any one time, the disk area occupied by your file will be greater than the area necessary to hold your data. However, free areas are constantly being reused, so that the amount of unused space in the file will seldom exceed 30%.

Whenever you issue a monadic `⎕FRESIZE` command on a component file, Dyalog APL **COMPACTS** the file; that is, it restructures it by reordering the components and by amalgamating the free areas at the end of the file. It then truncates the file and releases the disk space back to the operating system (note that some versions of UNIX

do not allow the space to be released). For a large file with many components, this process may take a significant time.

Error Conditions

FILE SYSTEM NOT AVAILABLE

A `FILE SYSTEM NOT AVAILABLE` (Error code 28) error will be generated if the operating system returns an unexpected error when attempting to get a lock on a component file. In Windows environments this may indicate that *opportunistic locks* (aka *oplocks*) are in use; they should be disabled if Dyalog components files are being used.

FILE SYSTEM TIES USED UP

A `FILE SYSTEM TIES USED UP` (Error code 30) error will be generated when an attempt is made to open more component files than is possible.

FILE TIED

A `FILE TIED` error is reported if you attempt to tie a file which another user has exclusively tied.

Limitations

File Tie Quota

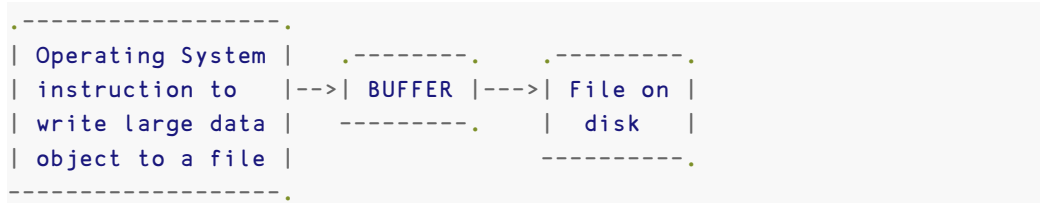
The File Tie Quota is the maximum number of files that a user may tie concurrently. Dyalog APL itself allows a maximum of 1024 under UNIX and 512 under Windows, although in either case your installation may impose a lower limit. When an attempt is made to exceed this limit, the report `FILE TIE QUOTA` (Error code 31) is given. This error will also be generated if an attempt is made to exceed the maximum number of open files that is imposed by the operating system.

File Name Quota

Dyalog APL records the names of each user's tied files in a buffer of 40960 bytes. When this buffer is full, the report `FILE NAME QUOTA USED UP` (Error code 32) will be given. This is only likely to occur if long pathnames are used to identify files.

6.2.6 The Effect of Buffering

Disk drives are fairly slow devices, so most operating systems take advantage of a facility called buffering. This is shown in simple terms below:



When you issue a write to a disk area, the data is not necessarily sent straight to the disk. Sometimes it is written to an internal buffer (or cache), which is usually held in (fast) main memory. When the buffer is full, the contents are passed to the disk. This means that at any one time, you could have data in the buffer, as well as on the disk. If your machine goes down whilst in this state, you could have a partially updated file on the disk. In these circumstances, the operating system generally recovers your file automatically.

If this facility is exploited, it offers very fast file updating. For systems that are I/O bound, this is a very important consideration. However, the disadvantage is that whilst it may appear that a write operation has completed successfully, part of the data may still be residing in the buffer, waiting to be flushed out to the disk. It is usually possible to force the buffer to empty; see your operating system manuals for details (UNIX automatically invokes the `sync()` command every few seconds to flush its internal buffers).

Dyalog APL exploits this facility, employing buffers internal to APL as well as making use of the system buffers. Of course, these techniques cannot be used when the file is shared with other users; obviously, the updates must be written immediately to the disk. However, if the file is exclusively tied, then several layers of buffers are employed to ensure that file access is as fast as possible.

You can ensure that the contents of all internal buffers are flushed to disk by issuing `⌈FUNTIE 0` at any time.

6.2.7 Integrity and Security

The structure of component files, the asynchronous nature of the buffering performed by APL, by the Operating System, and by the external device sub-system, introduces the potential danger that a component file might become damaged. To prevent this happening, the component file system includes optional journaling and check-sum

features. These are optional because the additional security these features provide comes at the cost of reduced performance. You can choose the level of security that is appropriate for your application.

When journaling is enabled (see `□FPROPS`), files are updated using a journal which effectively prevents system or network failures from causing file damage.

Additional security is provided by the check sum facility which enables component files to be repaired using the system function `□FCHK`. See File Check and Repair.

Level 1 journaling protects a component file from damage caused by an abnormal termination of the APL process. This could occur if the process is deliberately or accidentally terminated by the user or by the Operating System, or by an error in Dyalog APL.

Level 2 journaling provides protection not just against the possibility that the APL process terminates abnormally, but that the Operating System itself fails. However, a damaged component file must be explicitly repaired using the system function `□FCHK` which will repair any damaged components by rolling them back to their previous states.

Level 3 provides the same level of protection as Level 2, but following the abnormal termination of either APL or the Operating System, the rollback of an incomplete update will be automatic and no explicit repair will be needed.

Higher levels of Journaling inevitably reduce the performance of component file updates.

For further information, see `□FPROPS` and `□FCHK`.

Operating System Commands

APL files are treated as normal data files by the operating system, and may be manipulated by any of the standard operating system commands.

Do not use operating system commands to copy, erase or move component files that are tied and in use by an APL session.

Information

Component files that have both journalling and checksum properties set to `0` have been deprecated; from Dyalog v21.0, component files with this combination of properties will be read-only. Dyalog Ltd recommends using `⎕FPROPS` to convert any such files to have different properties. For information on how to identify component files that have both journalling and checksum properties set to `0` in your existing codebase, see the *Release Notes*.

7 Error Trapping

7.1 Standard Error Action

The standard system action in the event of an error or interrupt whilst executing an expression is to suspend execution and display an error report. If necessary, the state indicator is cut back to a statement such that there is no halted locked function visible in the state indicator.

The error report consists of up to three lines

1. The error message, preceded by the symbol $\perp$ if the error occurred while evaluating the Execute function.
2. The statement in which the error occurred (or expression being evaluated by the Execute function), preceded by the name of the function and line number where execution is suspended unless the state indicator has been cut back to immediate execution mode. If the state indicator has been cut back because of a locked function in execution, the displayed statement is that from which the locked function was invoked.
3. The symbol $\wedge$ under the last referenced symbol or name when the error occurred. All code to the right of the $\wedge$ symbol in the expression will have been evaluated.

EXAMPLES

```

      X PLUS U
VALUE ERROR
      X PLUS U
          ^

      FOO
INDEX ERROR
FOO[2] X←X+A[I]
          ^

      CALC
 $\perp$ DOMAIN ERROR
CALC[5] ÷0
          ^

```

7.2 Error Trapping Concepts

The purpose of this section is to show some of the ways in which the ideas of error trapping can be used to great effect to change the flow of control in a system.

First, we must have an idea of what is meant by error trapping. We are all used to entering some duff APL code, and seeing a (sometimes) rather obscure, esoteric error message echoed back:

```
10÷0
DOMAIN ERROR
10÷0
^
```

This message is ideal for the APL programmer, but not so for the end user. We need a way to bypass the default action of APL, so that we can take an action of our own, thereby offering the end user a more meaningful message.

Every error message reported by Dyalog APL has a corresponding error number (for a list of error codes and message, see `⌈TRAP`, Language Reference). Many of these error numbers plus messages are common across all versions of APL. We can see that the code for `DOMAIN ERROR` is 11, whilst `LENGTH ERROR` has code 5.

Dyalog APL provides two distinct but related mechanisms for the trapping and control of errors. The first is based on the control structure `:Trap ... :EndTrap`, and the second, on the system variable `⌈TRAP`. The control structure is easier to administer and so is recommended for normal use, while the system variable provides slightly finer control and may be necessary for specialist applications.

Last Error number and Diagnostic Message

Dyalog APL keeps a note of the last error that occurred, and provides this information through system functions: `⌈EN`, `⌈EM` and `⌈DM`.

```
10÷0
DOMAIN ERROR
10÷0
^
```

Error Number for last occurring error:

```
⌈EN
11
```

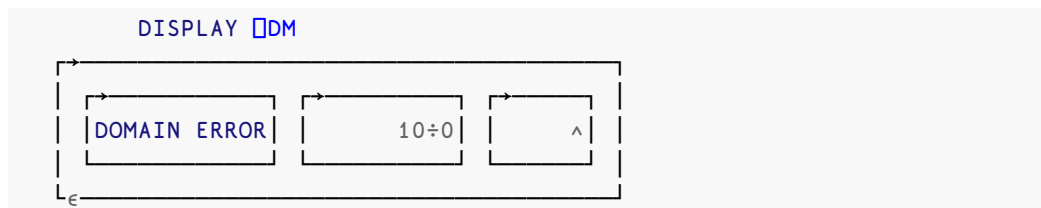
Error Message associated with code 11:

```
⌈EM 11
DOMAIN ERROR
```

⌈DM (Diagnostic Message) is a 3 element nested vector containing error message, expression and caret:

```
⌈DM
DOMAIN ERROR      10÷0      ^
```

Use function DISPLAY to show structure:



Mix (†) of this vector produces a matrix that displays the same as the error message produced by APL:

```
↑⌈DM
DOMAIN ERROR
10÷0
^
```

Error Trapping Control Structure

You can embed a number of lines of code in a :Trap control structure within a defined function.

```
[1] ...
[2] :Trap 0
[3] ...
[4] ...
[5] :EndTrap
[6] ...
```

Now, whenever *any* error occurs in one of the enclosed lines, or in a function called from one of the lines, processing stops immediately and control is transferred to the line following the :EndTrap. The 0 argument to :Trap, in this case represents any error. To trap only specific errors, you could use a vector of error numbers:

```
[2]      :Trap 11 2 3
```

Notice that in this case, no extra lines are executed after an error. Control is passed to line [6] either when an error has occurred, or if all the lines have been executed without error. If you want to execute some code *only* after an error, you could re-code the example like this:

```
[1]      ...
[2]      :Trap 0
[3]          ...
[4]          ...
[5]      :Else
[6]          ...
[7]          ...
[8]      :EndTrap
[9]      ...
```

Now, if an error occurs in lines [3-4], (or in a function called from those lines), control will be passed immediately to the line following the `:Else` statement. On the other hand, if all the lines between `:Trap` and `:Else` complete successfully, control will pass out of the control structure to (in this case) line [9].

The final refinement is that specific error cases can be accommodated using `:Case[List]` constructs in the same manner as the `:Select` control structure.

```
[1]      :Trap 17+121          A Component file errors.
[2]          tie+name 0FTIE 0    A Try to tie file
[3]          'OK'
[4]      :Case 22
[5]          'Can't find ',name
[6]      :CaseList 25+113
[7]          'Resource Problem'
[8]      :Else
[9]          'Unexpected Problem'
[10]     :EndTrap
```

Note that `:Trap` can be used in conjunction with `□SIGNAL` described below.

Traps can be nested. In the following example, code in the inner trap structure attempts to tie a component file, and if unsuccessful, tries to create one. In either case, the tie number is then passed to function `ProcessFile`. If an error other than 22 (`FILE NAME ERROR`) occurs in the inner trap structure, or an error occurs in function `ProcessFile` (or any of its called function), control passes to line immediately to line [9].

```

[1]      :Trap 0
[2]          :Trap 22
[3]              tie←name ⍵FTIE 0
[4]          :Else
[5]              tie←name ⍵FCREATE 0
[6]          :EndTrap
[7]          ProcessFile tie
[8]      :Else
[9]          'Unexpected Error'
[10]     :EndTrap

```

Trap System Variable: ⍵TRAP

The second way of trapping errors is to use the system variable: ⍵TRAP.

⍵TRAP, can be assigned a nested vector of **trap specifications**. Each trap specification is itself a nested vector, of length 3, with each element defined as:

list of error numbers	The error numbers we are interested in.
action code	Either 'E' (Execute) or 'C' (Cut Back). There are others, but they are seldom used.
action to be taken	APL expression, usually a branch statement or a call to an APL function.

So a single trap specification may be set up as:

```
⍵TRAP←5 'E' 'ACTION1'
```

and a multiple trap specification as:

```
⍵TRAP←(5 'E' 'ACTION1')((1 2 3) 'C' 'ACTION2')
```

The action code E tells APL that you want your action to be taken in the function in which the error occurred, whereas the code C indicates that you want your action to be taken in the function where the ⍵TRAP was *localised*. If necessary, APL must first travel back up the state indicator (cut-back) until it reaches that function.

7.3 Example Traps

Dividing by Zero

Let's try setting a `⌈TRAP` on `DOMAIN ERROR`:

```
MSG←'''Please give a non-zero right arg'''
⌈TRAP←11 'E' MSG
```

When we enter:

```
10÷0
```

APL executes the expression, and notes that it causes an error number 11. Before issuing the standard error, it scans its `⌈TRAP` table, to see if you were interested enough in that error to set a trap; you were, so APL executes the action specified by you:

```
10÷0
Please give non-zero right arg
```

Let's reset our `⌈TRAP`:

```
⌈TRAP←0p⌈TRAP      a No traps now set
```

and write a defined function to take the place of the primitive function `÷`:

```
▽ R←A DIV B
[1] R←A÷B
[2] ▽
```

Then run it:

```
10 DIV 0
DOMAIN ERROR
DIV[1] R←A÷B
      ^
```

Let's edit our function, and include a localised `⌈TRAP`:

```

    ▽ R←A DIV B; □TRAP
[1]  A Set the trap
[2]  □TRAP←11 'E' '→ERR1'
[3]  A Do the work; if it results in error 11,
[4]  A execute the trap
[5]  R←A÷B
[6]  A All OK if we got to here, so exit
[7]  →0
[8]  A Will get here only if error 11 occurred
[9]  ERR1: 'Please give a non-zero right arg'
    ▽

```

Running the function with good and bad arguments has the desired effect:

```

    10 DIV 2
5

```

```

    10 DIV 0
Please give a non-zero right arg

```

□TRAP is a variable like any other, and since it is localised in DIV, it is only effective in DIV and any other functions that may be called by DIV. So

```

    10÷0
DOMAIN ERROR
    10÷0
    ^

```

still gives an error, since there is no trap set in the global environment.

Other Errors

What happens to our function if we run it with other duff arguments:

```

    1 2 3 DIV 4 5
LENGTH ERROR
DIV [4] R←A÷B
    ^

```

Here is an error that we have taken no account of.

Change DIV to take this new error into account:

```

    ▽ R←A DIV B;⌈TRAP
[1]  ⌈ Set the trap
[2]  ⌈TRAP←(11 'E' '→ERR1')(5 'E' '→ERR2')
[3]  ⌈ Do the work; if it results in error 11,
[4]  ⌈ execute the trap
[5]  R←A ÷ B
[6]  ⌈ All OK if we got to here, so exit
[7]  →0
[8]  ⌈ Will get here only if error 11 occurred
[9]  ERR1:'Please give a non-zero right arg'⌈→0
[10] ⌈ Will get here only if error 5 occurred
[11] ERR2:'Arguments must be same length'
    ▽

```

```
)RESET
```

```

1 2 3 DIV 4 5
Arguments must be the same length

```

But here's yet another problem that we didn't think of:

```

(2 3⌈6) DIV (2 3 4⌈24)
RANK ERROR
DIV [4] R←A÷B
    ^

```

Global Traps

Often when we are writing a system, we can't think of everything that may go wrong ahead of time; so we need a way of catching "everything else that I may not have thought of". The error number used for "everything else" is zero:

```
)RESET
```

Set a global trap:

```
⌈TRAP ← 0 'E' ' ' 'Invalid arguments' ' '
```

And run the function:

```

(2 3⌈6) DIV (2 3 4⌈24)
Invalid arguments

```

In this case, when APL executed line 4 of our function `DIV`, it encountered an error number 4 (`RANK ERROR`). It searched the local trap table, found nothing relating to error

4, so searched further up the stack to see if the error was mentioned anywhere else. It found an entry with an associated Execute code, so executed the appropriate action AT THE POINT THAT THE ERROR OCCURRED. Let's see what's in the stack:

```
)SI
DIV[4]*
```

```
↑DM
RANK ERROR
DIV[4] R←A÷B
^
```

So although our action has been taken, execution has stopped where it normally would after a RANK ERROR.

Dangers

We must be careful when we set global traps; let's call the non-existent function BUG whenever we get an unexpected error:

```
)RESET
[TRAP ← 0 'E' 'BUG'
(2 3p16) DIV (2 3 4p124)
```

Nothing happens, since APL traps a RANK ERROR on line 4 of DIV, so executes the trap statement, which causes a VALUE ERROR, which activates the trap action, which causes a VALUE ERROR, which etc. etc. If we had also chosen to trap on 1000 (ALL INTERRUPTS), then we'd be in trouble!

Let's define a function BUG:

```
▽ BUG
[1] A Called whenever there is an unexpected error
[2] '*** UNEXPECTED ERROR OCCURRED IN: ',>1↓SI
[3] '*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
[4] '*** WORKSPACE SAVED AS BUG.',>1↓SI
[5] A Tidy up ... reset LX, untie files ... etc
[6] [SAVE 'BUG.',>1↓SI
[7] '*** LOGGING YOU OFF THE SYSTEM'
[8] [OFF
▽
```

Now, whenever we run our system and an unexpected error occurs, our BUG function will be called.

```
10 DIV 0
Please give non-zero right arg
```

```
(2 3p16) DIV (2 3 4p12)
```

```
*** UNEXPECTED ERROR OCCURRED IN: DIV
*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
*** WORKSPACE SAVED AS BUG.DIV
*** LOGGING YOU OFF THE SYSTEM'
```

The system administrator can then load BUG.DIV, look at the SI stack, discover the problem, and fix it.

Looking out for Specific Problems

In many cases, you can of course achieve the same effect of a trap by using APL code to detect the problem before it happens. Consider the function TIEΔFILE, which checks to see if a file already exists before it tries to access it:

```
▽ R←TIEΔFILE FILE;FILES
[1] A Tie file FILE with next available tie number
[2] A
[3] A All files in my directory
[4] FILES←⊞FLIB 'mydir'
[5] A Remove trailing blanks
[6] FILES←dbr"↓FILES
[7] A Required file in list?
[8] →ERR×1~(⊂FILE)∈FILES
[9] A Tie file with next number
[10] FILE ⊞FTIE R←1+[/0,⊞FNUMS
[11] A ... and exit
[12] →0
[13] A Error message
[14] ERR:R←'File does not exist'
▽
```

This function executes the same code whether the file name is right or wrong, and it could take a while to get all the file names in your directory. It would be neater, and more efficient to take action ONLY when the file name is wrong:

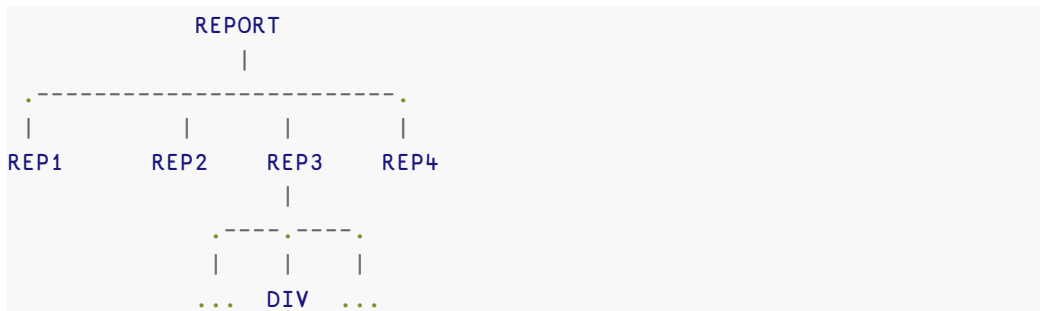
```

    ▽ R←TIEΔFILE FILE;□TRAP
[1]  A Tie file FILE with next available tie number
[2]  A
[3]  A Set trap
[4]  □TRAP←22 'E' '→ERR'
[5]  A Tie file with next number
[6]  FILE □FTIE R←1+[/0,□FNUMS
[7]  A ... and exit if OK
[8]  →0
[9]  A Error message
[10] ERR:R←'File does not exist'

```

Cut-Back versus Execute

Let us consider the effect of using Cut-Back instead of Execute. Consider the system illustrated below, in which the function REPORT gives the user the option of 4 reports to be generated:



```

    ▽ REPORT;OPTIONS;OPTION;□TRAP
[1]  A Driver functions for report sub-system. If an
[2]  A unexpected error occurs, take action in the
[3]  A function where the error occurred
[4]  A
[5]  A Set global trap
[6]  □TRAP←0 'E' 'BUG'
[7]  A Available options
[8]  OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
[9]  A Ask user to choose
[10] LOOP:→END×10=ρOPTION←MENU OPTIONS
[11] A Execute relevant function
[12]  ⚡OPTION
[13] A Repeat until EXIT
[14]  →LOOP
[15] A Now end
[16] END:

```

Suppose the user chooses REP3, and an unexpected error occurs in DIV. The good news is that the System Administrator gets a snapshot copy of the workspace that he can play about with:

```

    )LOAD BUG.DIV  A Load workspace
saved .....

```

```

    )SI              A Where did error occur?
DIV[4]*
REP3[6]
⚡
REPORT[7]

    ↑□DM            A What happened?
RANK ERROR
DIV[4] R←A÷B
    ^

    ▽              A Edit function on top of stack
[0]R←A DIV B
.....

```

The bad news is, our user is locked out of the whole system, even though it may only be REP3 that has a problem. We can get around this by making use of the CUT-BACK action code.

```

▽ REPORT;OPTIONS;OPTION;⌈TRAP
[1] ⌈ Driver functions for report sub-system. If an
[2] ⌈ unexpected error occurs, cut the stack back
[3] ⌈ to this function, then take action
[4] ⌈
[5] ⌈ Set global trap
[6] ⌈ ⌈TRAP←0 'C' '→ERR'
[7] ⌈ Available options
[8] ⌈ OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
[9] ⌈ Ask user to choose
[10] LOOP:→END×10=ρOPTION←MENU OPTIONS
[11] ⌈ Execute relevant function
[12] ⌈ ⌈OPTION
[13] ⌈ Repeat until EXIT
[14] ⌈ →LOOP
[15] ⌈ Tell user ...
[16] ⌈ ERR:MESSAGE'Unexpected error in',OPTION
[17] ⌈ ... what's happening
[18] ⌈ MESSAGE'Removing from list'
[19] ⌈ Remove option from list
[20] ⌈ OPTIONS←OPTIONS~cOPTION
[21] ⌈ And repeat
[22] ⌈ →LOOP
[23] ⌈ End
[24] ⌈ END:

```

Suppose the user runs this version of REPORT and chooses REP3. When the unexpected error occurs in DIV, APL will check its trap specifications, and see that the relevant trap was set in REPORT with a cut-back code. APL therefore **cuts back the stack to the function in which the trap was localised, THEN takes the specified action**. Looking at the SI stack above, we can see that APL must jump out of DIV, then REP3, then ⌈, to return to line 7 of REPORT; THEN it takes the specified action.

7.4 Signalling Events

It would be useful to be able to employ the idea of cutting back the stack and taking an alternative route through the code, when a condition other than an APL error occurs. To achieve this, we must be able to trap on errors other than APL errors, and we must be able to define these errors to APL. We do the former by using error codes in the range 500 to 999, and the latter by using ⌈SIGNAL.

Consider our system; ideally, when an unexpected error occurs, we want to save a snapshot copy of our workspace (execute BUG in place), then immediately jump back to

REPORT and reduce our options. We can achieve this by changing our functions a little, and using `□SIGNAL`:

```

▽ REPORT;OPTIONS;OPTION;□TRAP
[1] A Driver functions for report sub-system. If an
[2] A unexpected error occurs, make a snapshot copy
[3] A of the workspace, then cutback the stack to
[4] A this function, reduce the option list & resume
[5] A Set global trap
[6] □TRAP←(500 'C' '→ERR')(0 'E' 'BUG')
[7] A Available options
[8] OPTIONS←'REP1' 'REP2' 'REP3' 'REP4'
[9] A Ask user to choose
[10] LOOP:→END×10=ρOPTION←MENU OPTIONS
[11] A Execute relevant function
[12] ±OPTION
[13] A Repeat until EXIT
[14] →LOOP
[15] A Tell user ...
[16] ERR:MESSAGE'Unexpected error in',OPTION
[17] A ... what's happening
[18] MESSAGE'Removing from list'
[19] A Remove option from list
[20] OPTIONS←OPTIONS~cOPTION
[21] A And repeat
[22] →LOOP
[23] A End
[24] END:

```

```

▽ BUG
[1] A Called whenever there is an unexpected error
[2] '*** UNEXPECTED ERROR OCCURRED IN: ',>1↓□SI
[3] '*** PLEASE CALL YOUR SYSTEM ADMINISTRATOR'
[4] '*** WORKSPACE SAVED AS BUG.',>1↓□SI
[5] A Tidy up ... reset □LX, untie files ... etc
[6] □SAVE 'BUG.',>1↓□SI
[7] '*** RETURNING TO DRIVER FOR RESELECTION'
[8] □SIGNAL 500

```

Now when the unexpected error occurs, the first trap specification catches it, and the BUG function is executed in place. Instead of logging the user off as before, an error 500 is signalled to APL. APL checks its trap specifications, sees that 500 has been set in REPORT as a cut-back, so cuts back to REPORT before branching to ERR.

Flow Control

Error handling, which employs a combination of all the system functions and variables described, allows us to dynamically alter the flow of control through our system, as well as allow us to handle errors gracefully. It is a very powerful facility, which is simple to use, but is often neglected.

7.5 Handling Unexpected Application Errors in Windows

When running an APL application, it is possible that an unexpected error will occur.

It is advisable to set a trap at the top level of the application which traps all possible errors; in this way the programmer can cater for any errors that are not already explicitly trapped by, for example, writing information to a file, or saving the workspace. On UNIX in particular it may also be useful to call `EXIT` with a positive integer to the right of the `EXIT` - this is used as the exit code to APL.

It is also possible to generate an error which it is not possible to trap in APL code; examples include attempting to access the session in a runtime APL, or generating an error which causes APL to crash (for example, by the incorrect use of a shared library function).

By default in such cases, APL will pop up a message box, and cannot continue until the user selects the OK button.

It is possible to override this behaviour by setting the configuration parameter `DYALOG_NOPOPUPS` to 1. This will cause system popups to be suppressed; it does not suppress application popups generated by APL code.

With `DYALOG_NOPOPUPS=1` APL will terminate silently, except that an `aplcore` file will be generated. The location of the `aplcore` file can be controlled by the configuration parameter `APLCoreName`. It may be more useful to ask the operating system to handle the unexpected termination of the APL process, for example, by bringing up a debugger, or Dr Watson. This can be achieved by setting the configuration parameter `PassExceptionsToOpSys` to 1. In most cases it is useful to set `DYALOG_NOPOPUPS=1` too.

It is also possible to log such events to the Windows Event Log. Setting the configuration parameter `DYALOG_EVENTLOGGINGLEVEL` to a value greater than 0 will cause this to happen. If the configuration parameter `DYALOG_EVENTLOGNAME` is not set, then an event log called `Dyalog` will be created which can be viewed from the Windows Event Viewer. The first time that such an event occurs the following entries will be added to the Windows registry:

The key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Dyalog APL` with values

Value Name	Value
Sources	Dyalog APL
MaxSize	150000000

The key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Dyalog APL\Dyalog APL` with values

Value Name	Value
EventMessageFile	DYALOG\dyalog.exe
CategoryMessageFile	DYALOG\dyalog.exe
Category Count	5
TypesSupported	7

where DYALOG is the directory where Dyalog APL is installed.

If `DYALOG_EVENTLOGNAME` is set, it should contain the name of the log to which events will be logged. For example

```
DYALOG_EVENTLOGNAME="MyApp Event Log"
```

When set, no registry entries are added by Dyalog, but if the above registry entries have been manually created, the events will be logged to an event log which has the name "MyApp Event Log". If the registry entries described above have not been created, the events will instead be logged into the Application Log, and the Event Viewer will display text similar to the following when events are viewed:

The description for Event ID (1) in Source (MyApp Event Log) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. You may be able to use the /AUXSOURCE= flag to retrieve this description; see Help and Support for details. The following information is part of the event: Syserror: 995 code: 2 Aplcore "aplcore1" has been created.

8 Error Messages

8.1 Error Messages

8.1.1 Introduction

The error messages reported by APL are described in this section. Standard APL messages that provide information or report error conditions are summarised in [Section 8.1.2](#) and described later in alphabetical order.

APL also reports messages originating from the Operating System (WINDOWS or UNIX) which are summarised in [Section 8.1.3](#) and [Section 8.1.4](#). Only those Operating System error messages that might occur through normal usage of APL operations are described here. Other messages could occur as a direct or indirect consequence of using the Operating System interface functions `⌈CMD` and `⌈SH` or system commands `⌋CMD` and `⌋SH`, or when a non-standard device is specified for the system functions `⌈ARBIN` or `⌈AROUT`. Refer to the WINDOWS or UNIX reference manual for further information about these messages.

Most errors may be trapped using the system variable `⌈TRAP`, thereby retaining control and inhibiting the standard system action and error report. The table, *Dyalog APL Language: Trap* identifies the error code for trappable errors. The error code is also identified in the heading block for each error message when applicable.

See *Dyalog Programming Reference Guide* for a full description of the Error Handling facilities in Dyalog APL.

8.1.2 APL Errors

Table 8-1: *APL Error Messages*

Error Code	Report
	bad ws
	cannot create name
	clear ws
	copy incomplete
	defn error
	incorrect command
	insufficient resources
	is name
	Name already exists
	name is not a ws
	name saved date/time
	Namespace does not exist
	not copied name
	not found name
	not saved this ws is name
	sys error number
	too many names
	warning duplicate label
	warning duplicate name
	warning label name present in line 0
	warning pendent operation
	warning unmatched brackets
	warning unmatched parentheses
	was name
	ws not found
	ws too large
1	WS FULL

Error Code	Report
2	SYNTAX ERROR
3	INDEX ERROR
4	RANK ERROR
5	LENGTH ERROR
6	VALUE ERROR
7	FORMAT ERROR
10	LIMIT ERROR
11	DOMAIN ERROR
12	HOLD ERROR
16	NONCE ERROR
18	FILE TIE ERROR
19	FILE ACCESS ERROR
20	FILE INDEX ERROR
21	FILE FULL
22	FILE NAME ERROR
23	FILE DAMAGED
24	FILE TIED
25	FILE TIED REMOTELY
26	FILE SYSTEM ERROR
28	FILE SYSTEM NOT AVAILABLE
30	FILE SYSTEM TIES USED UP
31	FILE TIE QUOTA USED UP
32	FILE NAME QUOTA USED UP
34	FILE SYSTEM NO SPACE
35	FILE ACCESS ERROR - CONVERTING FILE
38	FILE COMPONENT DAMAGED
52	FIELD CONTENTS RANK ERROR

Error Code	Report
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
99	INTERNAL ERROR
1003	INTERRUPT
1005	EOF INTERRUPT
1006	TIMEOUT
1007	RESIZE
1008	DEADLOCK

8.1.3 Operating System Error Messages

The table below refers to UNIX Operating Systems under which the error code reported by Dyalog APL is (100 + the UNIX file error number). The text for the error message, which is obtained by calling `perror()`, will vary from one type of system to another.

See also [Section 8.1.4](#) for the equivalent error messages under Windows.

Typical Operating System Error Messages

Error Code	Report
101	FILE ERROR 1 Not owner
102	FILE ERROR 2 No such file or directory
103	FILE ERROR 3 No such process
104	FILE ERROR 4 Interrupted system call
105	FILE ERROR 5 I/O error
106	FILE ERROR 6 No such device or address
107	FILE ERROR 7 Arg list too long
108	FILE ERROR 8 Exec format error
109	FILE ERROR 9 Bad file number
110	FILE ERROR 10 No children
111	FILE ERROR 11 No more processes
112	FILE ERROR 12 Not enough code
113	FILE ERROR 13 Permission denied
114	FILE ERROR 14 Bad address
115	FILE ERROR 15 Block device required
116	FILE ERROR 16 Mount device busy
117	FILE ERROR 17 File exists
118	FILE ERROR 18 Cross-device link
119	FILE ERROR 19 No such device
120	FILE ERROR 20 Not a directory
121	FILE ERROR 21 Is a directory
122	FILE ERROR 22 Invalid argument
123	FILE ERROR 23 File table overflow
124	FILE ERROR 24 Too many open files
125	FILE ERROR 25 Not a typewriter
126	FILE ERROR 26 Text file busy
127	FILE ERROR 27 File too large

Error Code	Report
128	FILE ERROR 28 No space left on device
129	FILE ERROR 29 Illegal seek
130	FILE ERROR 30 Read-only file system
131	FILE ERROR 31 Too many links
132	FILE ERROR 32 Broken pipe
133	FILE ERROR 33 Math argument
134	FILE ERROR 34 Result too large

8.1.4 Windows Operating System Error Messages

Table 8-2: Windows Operating System Messages

Error Code	Report
101	FILE ERROR 1 No such file or directory
102	FILE ERROR 2 No such file or directory
103	FILE ERROR 3 Exec format error
105	FILE ERROR 5 Not enough memory
106	FILE ERROR 6 Permission denied
107	FILE ERROR 7 Argument list too big
108	FILE ERROR 8 Exec format error
109	FILE ERROR 9 Bad file number
111	FILE ERROR 11 Too many open files
112	FILE ERROR 12 Not enough memory
113	FILE ERROR 13 Permission denied
114	FILE ERROR 14 Result too large
115	FILE ERROR 15 Resource deadlock would occur
117	FILE ERROR 17 File exists
118	FILE ERROR 18 Cross-device link
122	FILE ERROR 22 Invalid argument
123	FILE ERROR 23 File table overflow
124	FILE ERROR 24 Too many open files
133	FILE ERROR 33 Argument too large
134	FILE ERROR 34 Result too large
145	FILE ERROR 45 Resource deadlock would occur

8.1.5 APL Error Messages

There follows an alphabetical list of error messages reported from within Dyalog APL.

8.1.6 APL Error Messages (A-Z)

bad ws

This report is given when an attempt is made to `)COPY` or `)PCOPY` from a file that is not a valid workspace file. Invalid files include workspaces that were created by a version of Dyalog APL later than the version currently being used.

cannot create name

This report is given when an attempt is made to `)SAVE` a workspace with a name that is either the name of an existing, non-workspace file, or the name of a workspace that the user does not have permission to overwrite or create.

clear ws

This message is displayed when the system command `)CLEAR` is issued.

EXAMPLE

```
)CLEAR
clear ws
```

copy incomplete

This report is given when an attempted `)COPY` or `)PCOPY` fails to complete. Reasons include:

- Failure to identify the incoming file as a workspace.
- Not enough active workspace to accommodate the copy.

DEADLOCK

1008

If two threads succeed in acquiring a hold of two different tokens, and then each asks to hold the other token, they will both stop and wait for the other to release its token. The interpreter detects such cases and issues an error (1008) DEADLOCK.

defn error

This report is given when either:

- The system editor is invoked in order to edit a function that does not exist, or the named function is pendent or locked, or the given name is an object other than a function.
- The system editor is invoked to define a new function whose name is already active.
- The header line of a function is replaced or edited in definition mode with a line whose syntax is incompatible with that of a header line. The original header line is re-displayed by the system editor with the cursor placed at the end of the line. Back-spacing to the beginning of the line followed by line-feed restores the original header line.

EXAMPLES

```

      X←1
      ▽X
defn error

      ▽FOO[0[]]
[0]   R←FOO
[0]   R←FOO:X
defn error
[0]   R←FOO:X

      []LOCK'FOO'
      ▽FOO[]
defn error

```

DOMAIN ERROR**11**

This report is given when either:

- An argument of a function is not of the correct type or its numeric value is outside the range of permitted values or its character value does not constitute valid name(s) in the context.
- An array operand of an operator is not an array, or it is not of the correct type, or its numeric value is outside the range of permitted values. A function operand of an operator is not one of a prescribed set of functions.
- A value assigned to a system variable is not of the correct type, or its numeric value is outside the range of permitted values

- The result produced by a function includes numeric elements which cannot be fully represented.

EXAMPLES

```

1÷0
DOMAIN ERROR
1÷0
^

(×∘'CAT')2 4 6
DOMAIN ERROR
(×∘'CAT')2 4 6
^

⌈IO←5
DOMAIN ERROR
⌈IO←5
^

```

EOF INTERRUPT

1005

This report is given on encountering the end-of-file when reading input from a file. This condition could occur when an input to APL is from a file.

EXCEPTION

90

This report is given when a Microsoft .NET object throws an exception. For details see *Dyalog APL Language: Exception*.

FIELD CONTENTS RANK ERROR

52

This report is given if a field content of rank greater than 2 is assigned to ⌈SM.

FIELD CONTENTS TOO MANY COLUMNS

53

This report is given if the content of a numeric or date field assigned to ⌈SM has more than one column.

FIELD POSITION ERROR**54**

This report is given if the location of the field assigned to `□SM` is outside the screen.

FIELD CONTENTS TYPE MISMATCH**56**

This report is given if the field contents assigned to `□SM` does not conform with the given field type, for example, character content with numeric type.

FIELD TYPE BEHAVIOUR UNRECOGNISED**57**

This report is given if the field type or behaviour code assigned to `□SM` is invalid.

FIELD ATTRIBUTES RANK ERROR**58**

This report is given if the current video attribute assigned to `□SM` is non-scalar but its rank does not match that of the field contents.

FIELD ATTRIBUTES LENGTH ERROR**59**

This report is given if the current video attribute assigned to `□SM` is non-scalar but its dimensions do not match those of the field contents.

FULL SCREEN ERROR**60**

This report is given if the required full screen capabilities are not available to `□SM`. This report is only generated in UNIX environments.

KEY CODE UNRECOGNISED**61**

This report is given if a key code supplied to `□SR` or `□PFKEY` is not recognised as a valid code. It will also be generated if you attempt to generate a KeyPress event with an invalid Input Code.

KEY CODE RANK ERROR**62**

This report is given if a key code supplied to `□SR` or `□PFKEY` is not a scalar or a vector.

KEY CODE TYPE ERROR**63**

This report is given if a key code supplied to `□SR` or `□PFKEY` is numeric or nested; that is, is not a valid key code.

FORMAT FILE ACCESS ERROR**70**

This report is given if the date format file to be used by `□SM` does not exist or cannot be accessed.

FORMAT FILE ERROR**71**

This report is given if the date format file to be used by `□SM` is ill-formed.

FILE ACCESS ERROR**19**

This report is given when the user attempts to execute a file system function for which the user is not authorised, or has supplied the wrong passnumber. It also occurs if the file specified as the argument to `□FERASE` or `□FRENAME` is not exclusively tied.

EXAMPLES

```

    'SALES' □FSTIE 1

    □FRDAC 1
0 4121 0
0 4137 99

    X □FREPLACE 1
FILE ACCESS ERROR
    X □FREPLACE 1
    ^

    'SALES' □FERASE 1
FILE ACCESS ERROR
    'SALES' □FERASE 1
    ^

```

FILE ACCESS ERROR CONVERTING

When a new version of Dyalog APL is used, it may be that improvements to the component file system demand that the internal structure of component files must alter. This alteration is performed by the interpreter on the first occasion that the file is accessed. If the operating system file permissions deny the ability to perform such a restructure, this report is given.

FILE COMPONENT DAMAGED**38**

This report is given if an attempt is made to access a component that is not a valid APL object. This will rarely occur, but may happen as a result of a previous computer system failure. Components files may be checked using `⌈FCHK`. See *Dyalog APL Language: Fchk*.

FILE DAMAGED**23**

This report is given if a component file becomes damaged. This rarely occurs but may result from a computer system failure. Components files may be checked using `⌈FCHK`. See *Dyalog APL Language: Fchk*.

FILE FULL**21**

This report is given if the file operation would cause the file to exceed its file size limit or would cause the component number to exceed the maximum permitted which is just below 2^{*32} .

FILE INDEX ERROR**20**

This report is given when an attempt is made to reference a non-existent component.

EXAMPLE

```

      □FSIZE 1
1 21 16578 4294967295

      □FREAD 1 34
FILE INDEX ERROR
      □FREAD 1 34
      ^
      □FDROP 1 50
FILE INDEX ERROR
      □FDROP 1 50
      ^

```

FILE NAME ERROR**22**

This report is given if:

- the user attempts to □FCREATE using the name of an existing file.
- the user attempts to □FTIE or □FSTIE a non-existent file, or a file that is not a component file.
- the user attempts to □FERASE a component file or □NERASE a native file with a name other than the EXACT name that was used when the file was tied.

FILE NAME QUOTA USED UP**32**

This report is given when the user attempts to execute a file system command that would result in the User's File Name Quota (see Dyalog Programming Reference Guide: Component Files) being exceeded.

This can occur with □FCREATE, □FTIE, □FSTIE or □FRENAME .

FILE SYSTEM ERROR**26**

This report is given if an input/output (I/O) error occurs when reading from or writing to the host file system. Contact your System Administrator.

If this occurs when the file is being written it may become damaged; it is therefore advisable to check the integrity of the file using □FCHK once the source of the I/O errors has been corrected. See *Dyalog APL Language: Fchk*.

FILE SYSTEM NO SPACE**34**

This report is given if the user attempts a file operation that cannot be completed because there is insufficient disk space.

FILE SYSTEM NOT AVAILABLE**28**

This error is generated if the operation system generates an unexpected error when attempting to get a lock on a component file. See *Dyalog Programming Reference Guide: Component Files* for details.

This error has been seen in Windows environments which have *opportunistic locks* (aka *oplocks*) enabled, either on the server that is running Dyalog APL, or on a server which has access to the same shared drives, or the disk array which contains the shared drives. In this scenario this error is not seen consistently, but rather is interspersed with other file-related errors. Oplocks should be disabled in environments where shared component files are used.

This error has also been seen when attempting to use component files on NFS mounted filesystems when the NFS lock daemon is not working properly: in this state component file operations may just hang (but can be interrupted) rather than this error being generated.

FILE SYSTEM TIES USED UP**30**

This error is generated when the maximum number of file ties for this APL instance has been reached. See *Dyalog Programming Reference Guide: Component Files* for details.

FILE TIE ERROR**18**

This report is given when the argument to a file system function contains a file tie number used as if it were tied when it is not or as if it were available when it is already tied. It also occurs if the argument to `□FHOLD` contains the names of non-existent external variables. It does not indicate that there is a problem with the underlying operating system's locking mechanism.

EXAMPLES

```

      □FNAMES, □FNUMS
SALES  1
COSTS  2
PROFIT 3

      X □FAPPEND 4
FILE TIE ERROR
      X □FAPPEND 4
      ^
      'NEWSALES' □FCREATE 2
FILE TIE ERROR
      'NEWSALES' □FCREATE 2
      ^

      'EXTVFILE' □XT 'BIGMAT'
      □FHOLD 'BIGMAT'
FILE TIE ERROR
      □FHOLD 'BIGMAT'
      ^
      □FHOLD< 'BIGMAT'

```

FILE TIED**24**

This report is given if the user attempts to tie a file that is exclusively tied by another task, or attempts to exclusively tie a file that is already share-tied by another task.

FILE TIED REMOTELY**25**

This report is given if the user attempts to tie a file that is exclusively tied by another task, or attempts to exclusively tie a file that is already share-tied by another task; and that task is running on other than the user's processor.

FILE TIE QUOTA USED UP**31**

This error is generated if an attempt is made to □FTIE, □FSTIE or □FCREATE a file when the user already has the maximum number of files tied. (See *Dyalog Programming Reference Guide:Component Files*)

FORMAT ERROR**7**

This report is given when the format specification in the left argument of system function □FMT is ill-formed.

EXAMPLE

```
'A1,1X,I5'□FMT CODE NUMBER
FORMAT ERROR
'A1,1X,I5'□FMT CODE NUMBER
^
```

(The correct specification should be 'A1,X1,I5' .)

HOLD ERROR**12**

This report is given when an attempt is made to save a workspace using the system function □SAVE if any external arrays or component files are currently held (as a result of a prior use of the system function □FHOLD).

EXAMPLE

```
▽HOLD△SAVE
[1] □FHOLD 1
[2] □SAVE 'TEST'
▽
'FILE' □FSTIE 1
HOLD△SAVE
HOLD ERROR
HOLD△SAVE[2] □SAVE 'TEST'
^
```

incorrect command

This report is given when an unrecognised system command is entered.

EXAMPLE

```
)CLERA
incorrect command
```

INDEX ERROR**3**

This report is given when either:

- The value of an index, whilst being within comparison tolerance of an integer, is outside the range of values defined by the index vector along an axis of the array being indexed. The permitted range is dependent on the value of $\square IO$.
- The value specified for an axis, whilst being within comparison tolerance of an integer for a derived function requiring an integer axis value or a non-integer for a derived function requiring a non-integer, is outside the range of values compatible with the rank(s) of the array argument(s) of the derived function. Axis is dependent on the value of $\square IO$.

EXAMPLES

```

      A
1 2 3
4 5 6

      A[1;4]
INDEX ERROR
      A[1;4]
      ^

      ↑ [2]'ABC' 'DEF'
INDEX ERROR
      ↑ [2]'ABC' 'DEF'
      ^

```

INTERNAL ERROR**99**

INTERNAL ERROR indicates that the system has reached an unexpected state from which Dyalog APL has recovered.

Should you encounter an **INTERNAL ERROR**, Dyalog strongly recommends that you save your work(space), and asks that you report the issue to Dyalog.

INTERRUPT**1003**

This report is given when execution is suspended by entering a hard interrupt. A hard interrupt causes execution to suspend as soon as possible without leaving the environment in a damaged state.

EXAMPLE

```
1 1 2 0(2 100p1200)0. |?1000p200
```

(Hard interrupt)

INTERRUPT

```
1 1 2 0(2 100p1200)0. |?1000p200
      ^
```

is name

This report is given in response to the system command `)WSID` when used without a parameter. `name` is the name of the active workspace including directory references given when loaded or named. If the workspace has not been named, the system reports `is CLEAR WS.`

EXAMPLE

```
)WSID
is WS/UTILITY
```

LENGTH ERROR

5

This report is given when the shape of the arguments of a function do not conform, but the ranks do conform.

EXAMPLE

```
2 3+4 5 6
LENGTH ERROR
2 3+4 5 6
  ^
```

LIMIT ERROR

10

This report is given when a system limit is exceeded. System limits are installation dependent.

EXAMPLE

```
(16p1)p1
LIMIT ERROR
(16p1)p1
^
```

NONCE ERROR**16**

This report is given when a system function or piece of syntax is not currently implemented but is reserved for future use.

NO PIPES**72**

This message applies to the UNIX environment ONLY.

This message is given when the limit on the number of pipes communicating between tasks is exceeded. An installation-set quota is assigned for each task. An associated task may require more than one pipe. The message occurs on attempting to exceed the account's quota when either:

- An APL session is started
- A non-APL task is started by the system function □SH
- An external variable is used.

It is necessary to release pipes by terminating sufficient tasks before proceeding with the required activity. In practice, the error is most likely to occur when using the system function □SH.

EXAMPLES

```
'via' □SH 'via'
NO PIPES
'via' □SH 'via'
^

'EXT/ARRAY' □XT 'EXVAR'
NO PIPES
'EXT/ARRAY' □XT 'EXVAR'
^
```

name is not a ws

This report is given when the name specified as the parameter of the system commands `)LOAD`, `)COPY` or `)PCOPY` is a reference to an existing file or directory that is not identified as a workspace.

This will also occur if an attempt is made to `)LOAD` a workspace that was `)SAVE`'d using a later version of Dyalog APL.

EXAMPLE

```
)LOAD EXT\ARRAY
EXT\ARRAY is not a ws
```

Name already exists

This report is given when an `)NS` command is issued with a name which is already in use for a workspace object other than a namespace.

Namespace does not exist

This report is given when a `)CS` command is issued with a name which is not the name of a global namespace.

not copied name

This report is given for each object named or implied in the parameter list of the system command `)PCOPY` which was not copied because of an existing global referent to that name in the active workspace.

EXAMPLE

```
)PCOPY WS/UTILITY A FOO Z
WS/UTILITY saved Mon Nov 1 13:11:19 1993
not copied Z
```

not found name

This report is given when either:

- An object named in the parameter list of the system command `)ERASE` is not erased because it was not found or it is not eligible to be erased.

- An object named in the parameter list (or implied list) of names to be copied from a saved workspace for the system commands)COPY or)PCOPY is not copied because it was not found in the saved workspace.

EXAMPLES

```
)ERASE []IO
not found []IO

)COPY WS/UTILITY UND
WS/UTILITY saved Mon Nov 1 13:11:19 1993
not found UND
```

not saved this ws is name

This report is given in the following situations:

- When the system command)SAVE is used without a name, and the workspace is not named. In this case the system reports `not saved this ws is CLEAR WS`.
- When the system command)SAVE is used with a name, and that name is not the current name of the workspace, but is the name of an existing file.

In neither case is the workspace renamed.

EXAMPLES

```
)CLEAR
)SAVE
not saved this ws is CLEAR WS

)WSID JOHND
)SAVE
)WSID ANDYS
)SAVE JOHND
not saved this ws is ANDYS
```

PROCESSOR TABLE FULL

76

This report can only occur in a UNIX environment.

This report is given when the limit on the number of processes (tasks) that the computer system can support would be exceeded. The limit is installation dependent. The report is given when an attempt is made to initiate a further process, occurring when an APL session is started.

It is necessary to wait until active processes are completed before the required task may proceed. If the condition should occur frequently, the solution is to increase the limit on the number of processes for the computer system.

EXAMPLE

```
'prefect' □SH 'prefect'
PROCESSOR TABLE FULL
'prefect' □SH 'prefect'
^
```

RANK ERROR**4**

This report is given when the rank of an argument or operand does not conform to the requirements of the function or operator, or the ranks of the arguments of a function do not conform.

EXAMPLE

```
2 3 + 2 2p10 11 12 13
RANK ERROR
2 3 + 2 2p10 11 12 13
^
```

RESIZE**1007**

This report is given when the user resizes the □SM window. It is only applicable to Dyalog APL/X and Dyalog APL/W.

name saved date time

This report is given when a workspace is saved, loaded or copied.

date/time is the date and time at which the workspace was most recently saved.

EXAMPLES

```
)LOAD WS/UTILITY
WS/UTILITY saved Fri Sep 11 10:34:35 1998

)COPY SPACES GEOFF JOHND VINCE
./SPACES saved Wed Sep 30 16:12:56 1998
```

SYNTAX ERROR**2**

This report is given when a line of characters does not constitute a meaningful statement. This condition occurs when either:

- An illegal symbol is found in an expression.
- Brackets, parentheses or quotes in an expression are not matched.
- Parentheses in an expression are not matched.
- Quotes in an expression are not matched.
- A value is assigned to a function, label, constant or system constant.
- A strictly dyadic function (or derived function) is used monadically.
- A monadic function (or derived function) is used dyadically.
- A monadic or dyadic function (or derived function) is used without any arguments.
- The operand of an operator is not an array when an array is required.
- The operand of an operator is not a function (or derived function) when a function is required.
- The operand of an operator is a function (or derived function) with incorrect valency.
- A dyadic operator is used with only a single operand.
- An operator is used without any operands.

EXAMPLES

```

A>10)/A
SYNTAX ERROR
A>10)/A
^

```

```

τ2 4 8
SYNTAX ERROR
τ2 4 8
^

```

```

A.+1 2 3
SYNTAX ERROR
A.+1 2 3
^

```

sys error number

This report is given when an internal error occurs in Dyalog APL.

Under UNIX it may be necessary to enter a hard interrupt to obtain the UNIX command prompt, or even to **kill** your processes from another screen. Under WINDOWS it may be necessary to reboot your PC.

If this error occurs, please submit a fault report to your Dyalog APL distributor.

TIMEOUT

1006

This report is given when the time limit specified by the system variable `⌈RTL` is exceeded while awaiting input through character input (`⌈`) or `⌈SR`.

It is also reported by `⌈FHOLD` if it times out.

It is usual for this error to be trapped.

EXAMPLE

```
⌈RTL←5 ⋄ ⌈←'RESPOND WITHIN 5 SECONDS: ' ⋄ R←⌈
RESPOND WITHIN 5 SECONDS:
TIMEOUT
⌈RTL←5 ⋄ ⌈←'RESPOND WITHIN 5 SECONDS: ' ⋄ R←⌈
      ^
```

TRANSLATION ERROR

92

This report is given when the system cannot convert a character from Unicode to an Atomic Vector index or vice versa. Conversion is controlled by the value of `⌈AVU`. Note that this error can occur when you **reference a variable** whose value has been obtained by reading data from a `TCPsocket` or by calling an external function. This is because in these cases the conversion to/from `⌈AV` is deferred until the value is used.

TRAP ERROR

84

This report is given when a workspace full condition occurs whilst searching for a definition set for the system variable `⌈TRAP` after a trappable error has occurred. It does not occur when an expression in a `⌈TRAP` definition is being executed.

too many names

This report is given by the function editor when the number of distinct names (other than distinguished names beginning with the symbol `⌈`) referenced in a defined function exceeds the system limit of 4096.

VALUE ERROR**6**

This report is given when either:

- There is no active definition for a name encountered in an expression.
- A function does not return a result in a context where a result is required.

EXAMPLES

```

      X
VALUE ERROR
      X
      ^

      ▼ HELLO
[1]    'HI THERE'
[2]    ▼

      2+HELLO
HI THERE
VALUE ERROR
      2+HELLO
      ^

```

warning duplicate label

This warning message is reported on closing definition mode when one or more labels are duplicated in the body of the defined function. This does not prevent the definition of the function in the active workspace. The value of a duplicated label is the lowest of the line-numbers in which the labels occur.

warning duplicate name

This warning message is reported on closing definition mode when one or more names are duplicated in the header line of the function. This may be perfectly valid. Definition of the function in the active workspace is not prevented. The order in which values are associated with names in the header line is described in "*Defined Functions & Operators*".

warning pendent operation

This report is given on opening and closing definition mode when attempting to edit a pendant function or operator.

EXAMPLE

```

[0]   ▽FOO
[1]   GOO
[2]   ▽

[0]   ▽GOO
[1]   ◦
[2]   ▽

      FOO
SYNTAX ERROR
GOO[1] ◦
      ^

      ▽FOO
warning pendent operation
[0]   ▽FOO
[1]   GOO
[2]   ▽
warning pendent operation

```

warning label name present

This warning message is reported on closing definition mode when one or more label names also occur in the header line of the function. This does not prevent definition of the function in the active workspace. The order in which values are associated with names is described in "*Defined Functions & Operators*".

warning unmatched brackets

This report is given after adding or editing a function line in definition mode when it is found that there is not an opening bracket to match a closing bracket, or vice versa, in an expression. This is a warning message only. The function line will be accepted even though syntactically incorrect.

EXAMPLE

```

[3]   A[;B[;2]←0
warning unmatched brackets
[4]

```

warning unmatched parentheses

This report is given after adding or editing a function line in definition mode when it is found that there is not an opening parenthesis to match a closing parenthesis, or vice versa, in an expression. This is a warning message only. The function line will be accepted even though syntactically incorrect.

EXAMPLE

```
[4] X←(E>2)^E<10)≠A
warning unmatched parentheses
[5]
```

was name

This report is given when the system command)WSID is used with a parameter specifying the name of a workspace. The message identifies the former name of the workspace. If the workspace was not named, the given report is was CLEAR WS.

EXAMPLE

```
)WSID TEMP
was UTILITY
```

WS FULL**1**

This report is given when there is insufficient workspace in which to perform an operation. Workspace available is identified by the system constant □WA.

The maximum workspace size allowed is defined by the environment variable MAXWS. See **maxws parameter** for details.

EXAMPLE

```
□WAρ1.2
WS FULL
□WAρ1.2
^
```

ws not found

This report is given when a workspace named by the system commands `)LOAD`, `)COPY` or `)PCOPY` does not exist as a file, or when the user does not have read access authorisation for the file.

EXAMPLES

```
)LOAD NOWS  
ws not found  
  
)COPY NOWS A FOO X  
ws not found
```

ws too large

This report is given when:

- the user attempts to `)LOAD` a workspace that needs a greater work area than the maximum that the user is currently permitted.
- the user attempts to `)COPY` or `)PCOPY` from a workspace that would require a greater work area than the user is currently permitted if the workspace were to be loaded.

The maximum work area permitted is set using the environment variable `MAXWS`.

8.1.7 Operating System Error Messages

There follows a numerically sorted list of error messages emanating from a typical operating system and reported through Dyalog APL.

8.1.8 Operating System Error Messages (A-Z)

FILE ERROR 1 Not owner

101

This report is given when an attempt is made to modify a file in a way which is forbidden except to the owner or super-user, or in some instances only to a super-user.

FILE ERROR 2 No such file

This report is given when a file (which should exist) does not exist, or when a directory in a path name does not exist.

FILE ERROR 5 I O error

105

This report is given when a physical I/O error occurred whilst reading from or writing to a device, indicating a hardware fault on the device being accessed.

FILE ERROR 6 No such device

This report is given when a device does not exist or the device is addressed beyond its limits. Examples are a tape which has not been mounted or a tape which is being accessed beyond the end of the tape.

FILE ERROR 13 Permission denied

113

This report is given when an attempt is made to access a file in a way forbidden to the account.

FILE ERROR 20 Not a directory

120

This report is given when the request assumes that a directory name is required but the name specifies a file or is not a legal name.

FILE ERROR 21 Is a directory**121**

This report is given when an attempt is made to write into a directory.

FILE ERROR 23 File table overflow**123**

This report is given when the system limit on the number of open files is full and a request is made to open another file. It is necessary to wait until the number of open files is reduced. If this error occurs frequently, the system limit should be increased.

FILE ERROR 24 Too many open

This report is given when the task limit on the number of open files is exceeded. It may occur when an APL session is started or when a shell command is issued to start an external process through the system command `□SH`. It is necessary to reduce the number of open files. It may be necessary to increase the limit on the number of open files to overcome the problem.

FILE ERROR 26 Text file busy**126**

This report is given when an attempt is made to write a file which is a load module currently in use. This situation could occur on assigning a value to an external variable whose associated external file name conflicts with an existing load module's name.

FILE ERROR 27 File too large**127**

This report is given when a write to a file would cause the system limit on file size to be exceeded.

FILE ERROR 28 No space left

This report is given when a write to a file would exceed the capacity of the device containing the file.

FILE ERROR 30 Read only file

This report is given when an attempt is made to write to a device which can only be read from. This would occur with a write-protected tape.

8.1.9 System Errors

Introduction

Dyalog APL will generate a system error and (normally) terminate in one of two circumstances:

- As a result of the failure of a workspace integrity check
- As a result of a System Exception

On Windows, if the **DYALOG_NOPOPUPS** parameter is 0 (the default), it will display the System Error dialog box (see **System Error Dialog Box**). This is suppressed if **DYALOG_NOPOPUPS** is 1.

aplcore file

When a system error occurs, APL normally saves an *aplcore* file which may be sent to Dyalog for diagnosis. The name and location of the *aplcore* file may be specified by the *Installation/Configuration: Configuration Parameters* parameter. If this parameter is not specified, the *aplcore* file is named `aplcore` and is saved in the current working directory.

Normally a new *aplcore* will replace a file of the same name. However, if *Installation/Configuration: Configuration Parameters* contains an asterisk (*), the system will create a new file, replacing the asterisk with a number incremented from the largest numbered file present.

The number of *aplcore* files retained by the system is specified by the *Installation/Configuration: Configuration Parameters* parameter. If *Installation/Configuration: Configuration Parameters* is 0, the system will not save an *aplcore*. However, under Windows, if **DYALOG_NOPOPUPS** is 0, and the user checks the *Create an aplcore file* checkbox when the *System Error* dialog box is displayed, an *aplcore* will be saved regardless of the value of *Installation/Configuration: Configuration Parameters*. See **System Error Dialog Box**.

Be aware that if your application contains any secure data, this data may be present in an *aplcore* file, and it may be appropriate to set both *Installation/Configuration: Configuration Parameters* and **DYALOG_NOPOPUPS** to 0 to prevent such data being saved on disk.

Information that may prove useful in debugging the problem, including (where possible) the SI stack at the point where the *aplcore* was generated, is by default written to the end of *aplcore* files; the section begins with the string

```
'===== Interesting Information'
```

Under UNIX, this interesting information section can be extracted from the *aplcore* as follows:

```
sed -n '/===== Interesting Information/, $p' aplcore
```

To prevent this information from being written to the *aplcore* file, the *Installation/Configuration: Apl Textinaplcore* parameter should be set to 0.

Workspace Integrity

When you)SAVE your workspace, Dyalog APL first performs a workspace integrity check. If it detects any discrepancy or violation in the internal structure of your workspace, APL does not overwrite your existing workspace on disk. Instead, it displays the System Error dialog box and saves the workspace, together with diagnostic information, in an *aplcore* file before terminating.

A System Error code is displayed in the dialog box and should be reported to Dyalog for diagnosis. This information also appears in the Interesting Information section of the *aplcore* file.

Note that the internal error that caused the discrepancy could have occurred at any time prior to the execution of)SAVE and it may not be possible for Dyalog to identify the cause from this *aplcore* file.

If APL is started in debug mode with the **-Dc**, **-Dw** or **-DW** flags, the Workspace Integrity check is performed more frequently, and it is more likely that the resulting *aplcore* file will contain information that will allow the problem to be identified and corrected. It is also possible to enable or alter the debugging level from within APL using the SetDFlags method; Dyalog support will direct the use of this feature when necessary.

System Exceptions

Non-specific System Errors are the result of Operating System exceptions that can occur due to a fault in Dyalog APL itself, an error in a Windows or other DLL, or even as a result of a hardware fault. The following system exceptions are separately identified.

Code	Description	Suggested Action
900	A Paging Fault has occurred	As the most likely cause is a temporary network fault, recommended course of action is to restart your program.
990 & 991	An exception has occurred in the Development or Run-Time DLL.	
995	An exception has occurred in a DLL function called via <code>□NA</code>	Carefully check your <code>□NA</code> statement and the arguments that you have passed to the DLL function
996	An exception has occurred in a DLL function called via a threaded <code>□NA</code> call	As above
997	An exception has occurred while processing an incoming OLE call	
999	An exception has been caused by Dyalog APL or by the Operating System	

Recovering Data from *aplcore* files

Objects may often (but not always) be recovered from *aplcore* using `)COPY` or `□CY`. Note that if the *aplcore* contains a workspace with more than one instance of the same name on the stack, `□CY` copies the most local object whereas `)COPY` copies the global one.

Be aware that in many cases an attempt to `)COPY` from or `)LOAD` an *aplcore* is likely to result in a further syserror; this may result in the original *aplcore* being overwritten, thus losing the contents of that file. It is therefore worth while taking a copy of the *aplcore* before attempting to `)COPY` from it. Attempting to copy specific items is more likely to be successful than copying the entire workspace from the *aplcore*.

Note that in previous versions under Windows because (by default) the *aplcore* file has no extension, it was necessary to explicitly add a dot, or APL would attempt to find the non-existent file `aplcore.dws`. This is no longer true in version 14.1 onwards.

Reporting Errors to Dyalog

If APL crashes and saves an *aplcore* file, please email the following information to support@dyalog.com:

1. a brief description of the circumstances surrounding the error
2. details of your version of Dyalog APL: the full version number, whether it is Unicode or Classic Edition, and the BuildID. This information appears in the *Help->About* box; the *Copy* button copies this information into the clipboard, from where it can be pasted into an email etc.
3. a compressed form of the *aplcore* file itself

If the problem is reproducible, that is, can be easily repeated, please also send the appropriate description, workspace, and other files required to do so.

System Error Dialog Box

The System Error Dialog illustrated below was produced by deliberately inducing a system exception in the DLL function `memcpy()`. The functions used were:

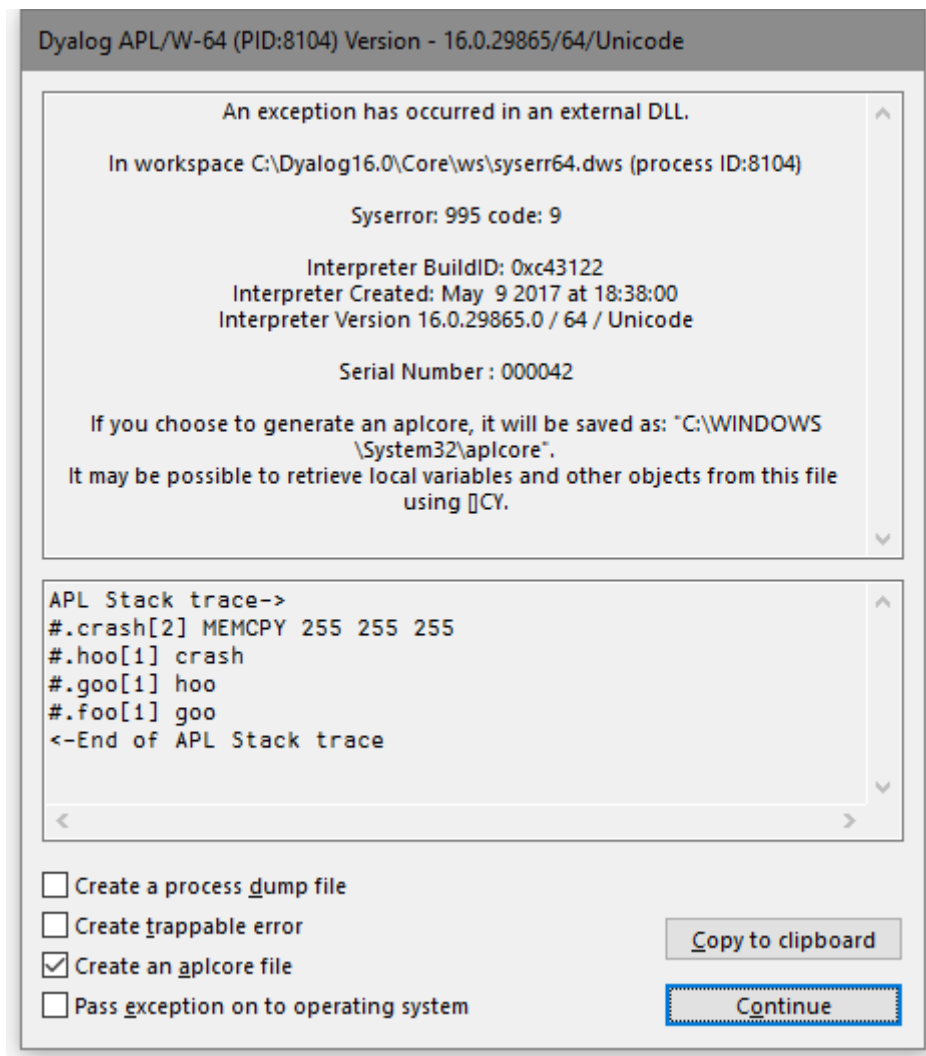
```

    ▼ foo
[1]    goo
    ▼
    ▼ goo
[1]    hoo
    ▼
    ▼ hoo
[1]    crash
    ▼

    ▼ crash
[1]    □NA'dyalog64|MEMCPY u u u'
[2]    MEMCPY 255 255 255
    ▼

```

Note: Under a 32-bit interpreter the `□NA` call should refer to `dyalog32`.



Options

Item	Description
Create a process dump file	Dumps a complete core image, see below.
Create Trappable Error	If you check this box (only enabled on System Error codes 995 and 996), APL will not terminate but will instead generate an error 91 (<code>EXTERNAL DLL EXCEPTION</code>) when you press <i>Dismiss</i> .
Create an aplcore file	If this box is checked, an <i>aplcore</i> file will be created.
Pass exception on to operating system	If this box is checked, the exception will be passed on to your current debugging tool (for example, <i>Visual Studio</i>). See PassExceptionsToOpSys .
Copy to clipboard	Copies the contents of the APL stack trace window to the Clipboard.

Create a process dump file

Under Windows the *Create a process dump file* option creates a user-mode process dump file , also known as a minidump file, called `dya log .dmp` in the current directory. This file allows post-mortem debugging of a crash in the interpreter or a loaded DLL. It contains much more debug information than a normal *aplcore* (and is much larger than an *aplcore*) and can be sent to Dyalog Limited (zip it first please). Alternatively the file can be loaded into *Visual Studio .NET* to do your own debugging.

Debugging your own DLLs

If you are using Visual Studio, the following procedure should be used to debug your own DLLs when an appropriate Dyalog APL System Error occurs.

Ensure that the *Pass Exception* box is checked, then click on *Dismiss* to close the *System Error* dialog box.

The system exception dialog box appears. Click on *Debug* to start the process in the Visual Studio debugger.

After debugging, the System Exception dialog box appears again. Click on *Don't send to terminate Windows' exception handling*.

ErrorOnExternalException Parameter

This parameter allows you to prevent APL from taking the actions described above when an exception caused by an external DLL occurs. The following example illustrates what happens when the functions above are run, but with the *Installation/Configuration: Configuration Parameters* parameter set to 1.

```

1  ↵←2  ↵NQ'.' 'GetEnvironment' 'ErrorOnExternalException'
    foo
EXTERNAL DLL EXCEPTION
crash[2] MEMCPY 255 255 255
    ^
    ↵EN
91
    )SI
crash[2]*
hoo[1]
goo[1]
foo[1]

```

Note: Dyalog recommends that enabling *Installation/Configuration: Configuration Parameters* should only be done while developing or debugging an application; ignoring such errors may result in corruption in the workspace which could result to unexpected errors later in the application.

What should I do if Dyalog hangs?

If Dyalog for Windows hangs, you should generate a process dump file and send it to Dyalog Support, along with your Build ID.

To do this:

1. Start *Task Manager* (as a user who has administrative privileges)
2. Go to the *Processes* tab
3. Right click on the `dyalog.exe` process and choose *Create Dump File*. Windows will create a process dump file in `C:\Users\<your name>\AppData\Local\Temp\dyalog.DMP`
4. Compress this file and send it to Dyalog. If it is less than 10 Mb in size, send it to Dyalog Support as an email attachment. If it is more than 10 Mb, upload it via the *MyDyalog/My Account* page or contact Dyalog support to request an account on our FTP server.

9 APLMon

9.1 Introduction

APLMon is a built-in Dyalog feature that, when enabled, monitors the execution of APL primitives and functions and writes the results to a CSV file. When not enabled (the default), it has no impact on execution.

When running an APL application with APLMon enabled, Dyalog logs the use of every primitive operation; how many times it was called, how much time was spent running it and details about the arguments it was processing. This makes it possible to identify the primitive operations on which the application spends the most time. Dyalog Ltd can use this data to help determine which optimisations to prioritise and where to concentrate performance work. For additional information on this topic, please contact support@dyalog.com.

APLMon does not return any application-side information. APL programmers should use `⌈PROFILE` or the `cmpx` function from the `dfns` workspace to determine which parts of the application can be optimized programmatically. However, APLMon can be useful for APL programmers who want to verify that their application uses primitives as expected or that it does not perform unnecessary computations such as disclosing simple scalars.

9.2 Using APLMON

APLMon is enabled and disabled using the APLMon method of Root. It takes a single argument, the name of the file to which the results are to be written. If the file name is empty, APLMon is disabled.

Accumulated data is only actually written to the file when the method is called or when Dyalog terminates. The name of the file to which the results are written is returned as a shy result. This mechanism allows you to log application results in separate files.

EXAMPLE

Run APLMon and output the data to `myapplication.csv`:

```

a←2 □NQ '.' 'APLMON' 'C:\aplmon\myapplication.csv'
a

pa

0

```

The empty result indicates that APLMON was disabled prior to this call.

EXAMPLE

Output the data and disable APLMON:

```

+2 □NQ '.' 'APLMON' ''
C:\aplmon\myapplication.csv

```

The file name in the result indicates that data has been output to that file and APLMON has been disabled.

9.2.1 Time measurement

The overhead of using APLMON reduces the overall speed of execution significantly. However, this does not affect the relative accuracy of the measurements.

For optimal accuracy, time measurement should be done with as few other processes running as possible, and in a process that has a higher priority than others.

To change process priority on Microsoft Windows: Start Dyalog, then launch the *Task Manager*. In the *Processes* tab, right-click on *Dyalog* and select *Go to details* from the pop-up menu that appears. Right-click on *dyalog.exe* in the *Details* tab; move the cursor to *Set priority* in the pop-up menu that appears, then select *Above normal* from the additional pop-up menu. A higher priority is not needed (everything else is *Normal* by default) and if set higher, the Dyalog process could cause the operating system to hang and make user control difficult.

9.3 APLMON output

The output file is a CSV matrix with values separated by commas. Fields can be empty. The file may be read by □CSV and applications such as Microsoft Excel.

The eleven columns of the APLMON output file are :

token	Name of the primitive's token (symbol, if you prefer)
lfn	Name of the left operand's token, if the primitive was an operator
rfn	Name of the right operand's token, if the primitive was an operator
ltype	Data type of the left argument of the function
rtype	Data type of the right argument of the function
lrank	Rank of the left argument of the function. Arrays of rank greater or equal to 3 will be logged as « 3+ »
rrank	Rank of the right argument of the function. Arrays of rank greater or equal to 3 will be logged as « 3+ »
lbound	Range of the left argument sizes. A range is made of two numbers separated by a « < » sign. The real size is between these two numbers. So arguments with similar sizes are grouped together.
rbound	Range of the right argument sizes. A range is made of two numbers separated by a « < » sign. The real size is between these two numbers. So arguments with similar sizes are grouped together.
hitcount	Number of times the primitive was called
time	Time spent by the primitive (in seconds)

These fields are empty if not applicable (which will appear in the CSV file as two consecutive commas).

EXAMPLE

In the following example, APLMON is used to analyse the execution of the expression `10 foo 20 30`. Note that expressions executed in the Session when APLMON is enabled also contribute to the analysis.

```

▽foo▽
▽ r←a foo b
[1]   r←(a+b)(a-b)(a×b)(a÷b)
▽
  ⍣2 ⍵NQ '.' 'APLMON' 'c:\dyalog17.0\aplmon'

    10 foo 20 30
30 40 -10 -20 200 300 0.5 0.3333333333
  ⍣2 ⍵NQ '.' 'APLMON' ''
  rslt←⍵CSV 'c:\dyalog17.0\aplmon'
  prslt
15 11

```

rslt											
token	lfn	rfn	ltype	rtype	lrank	rrank	lboun				
d rbound	hitcount	time									
plus			int8	int8	0	1					
1<1	2<4	1	0.000001								
minus			int8	int8	0	1					
1<1	2<4	1	0.000001								
times			int8	int8	0	1					
1<1	2<4	1	0.000002								
divide			int8	int8	0	1					
1<1	2<4	1	0.000003								
left arrow				nested							
1	2<4	1	0.000001								
right tack				wchar8							
1	0<0	1	0.000000								
stranding			int16	float64	1	1					
2<4	2<4	1	0.000001								
stranding			int8	nested	1	1					
2<4	2<4	2	0.000001								
stranding			wchar8	wchar8	1	1					
5<9	0<0	1	0.000001								
name											
lookup											
10	0.000002										
tradfn											
init											
1	0.000002										
aplmon measurement											
time											
0.000013											
aplmon unmeasured											
time											
19.772147											
aplmon											
total											
19.772174											

APLMON measures time only for the atomic calls (that is, not complex expressions or user-defined functions). Primitives that evaluate APL expressions (such as `⍎`, `⍋FIX` and `⍋NEW`) are not included. For example :

```
foo←{1+ω} ⍋ foo''(Np1)
```

The each will not appear in the log, but only the primitives called by `foo`, meaning that `+` will appear as being called N times

```
+.*/(Nρ1
```

The `/` will not be measured, but `+.*` will appeared as called $(N-1)$ times.

This ensures that time measurements do not overlap, and can therefore be compared.

However, some non-atomic expressions are logged, and the non-primitive parts are logged as non primitive.

9.4 Special syntax

APL has some special syntax that is treated in a special way by APLMON.

For example, cases of Simple Indexing (where the indexer is a set of semicolon-separated simple arrays) are reported as follows.

The *token* is `simple indexing`, there are no *lfn* or *rfn*, the left argument is the indexed array, and the right argument is as representative of the indices as possible. Dyalog has chosen to make it appear as an array, the rank of which is the number of specified subscripts, and the shape of which is the number of items in these subscripts.

APL expression	token	lfn	rfn	ltype	rtype	lrank	rrank	lbound	rbound
'abcd'[3 3ρ3]	simple indexing			unicode8	int8	1	1	2<4	5<9
(2 2ρ4) [1;]	simple indexing			int8	int8	2	1	2<4	1<1

Reach and Choose Indexing (where the indexer is a nested array)

The *token* is `reach indexing`, the left argument is the indexed array, and the right argument is the indexer array.

Axis specification

The *token* is `axis`, *lfn* is the function whose axis is specified, *rfn* is non-primitive, as it is the axis specification constant.

Assignment

The *token* is `left arrow` and it is considered as a monadic function.

Modified Assignment

The *token* is `left arrow`, *lfn* is the modifier function, *rfn* is empty, *larg* is the modified value and *rarg* the modifier value.

Indexed Assignment

The *token* is `indexed assignment`, *lfn* and *rfn* are empty, *larg* is the indexed array, *rarg* is the new value.

Modified Indexed Assignment

The *token* is `indexed assignment`, *lfn* is the modifier function, *rfn* is empty, *larg* is the indexed array, *rarg* is the new value.

Selective Assignment

The *token* is `selective assignment`, *lfn* and *rfn* are empty, *larg* is the indexed array, *rarg* is the new value.

Modified Selective Assignment

The *token* is `selective assignment`, *lfn* is the modifier function, *rfn* is empty, *larg* is the indexed array, *rarg* is the new value.

Stranding

The *token* is `stranding`, *larg* and *rarg* are the arrays being stranded together.

Name Lookup

The *token* is `name lookup`. This represents time spent in the parser looking up names in namespaces.

Defined Function Initialisation

The *token* is `tradfn init`. This represents time spent on entry to a traditional defined function.

Selective Assignment Pre-scan

The *token* is `selassi pre-scan`. This represents extra time spent in the parser to analyse the syntax of a selective assignment expression.

Control Structures

The *token* is `control structures`. This represents time spent processing control structures in a traditional defined function.

Namespace dot syntax

The *token* is `nsref`, *lfn* and *rfn* are right and left operands respectively, *larg* and *rarg* are as expected.

For example, in the expression: `SE.EX 'var'`

token is `nsref`, *lfn* is `EX`, *rfn* is `non-primitive`, *larg* is empty, *rarg* is the character vector.

9.5 Notes

Indexing

Simple indexing is the case when the indexer is a list of semicolon-separated simple arrays, as opposed to choose and reach indexing, where indexer is a nested array:

Simple Indexing Example

```
A[:,2;(2 3p1)]
```

Choose Indexing Example

```
A[(2 1)(2 2)]
```

Reach Indexing Example

```
A[∘((2 1) 1 5)]
```

For choose and reach indexing, the indexer will be represented by itself, in the monitor log, because it is a true array. A simple indexer, though, is not a true array but a collection of arrays, so it is treated specially : it is represented in the monitor log by a character array, the rank of which is the number of subscript, and the shape of which is the number of items in each subscript.

Indexer Examples

[2 3ρ1] will be represented as a rank-1 character array of shape 6

[1;4 5;7 8 9] will be represented as a rank-3 character array of shape 1 2 3

[;] will be represented as a rank-0 character array

[;5;0 2ρ1] will be represented as a rank-2 character array of shape 1 0

They cannot be misinterpreted because character arrays are illegal as an indexer.

In the monitor, for indexing, $A[I]$, the left argument will be A and the right argument will be the representation of I as described above.

For indexed assignment, $A[I]←B$, the right argument will be B , and the left argument will be the representation of I as described above.

Operators

In an expression in which an operator takes primitive operands, APLMON will report the time for the operator itself. If one of the operands is non-primitive, APLMON will report the time used by the operands.

EXAMPLES

```

    ▽ APLMON expr
[1] 2 [NQ]'.' 'APLMON' 'c:\dyalog17.0\aplmon'
[2] ±expr
[3] [CSV 2 [NQ]'.' 'APLMON' ''
    ▽

    APLMON '+/1 2 3'
6
    token                lfn  rfn  ltype  rtype  lrank  rrank  lboun
d rbound hitcount time
    slash                plus                int8
1                2<4      1                0.000003
    stranding                wchar8  wchar8  1      1
5<9      0<0      1                0.000001
    name
lookup
    1                0.000001
    aplmon measurement
time
    0.000005
    aplmon unmeasured
time
    0.003220
    aplmon
total
                0.003229
    APLMON '{α+ω}/1 2 3'
6
    token                lfn  rfn  ltype  rtype  lrank  rrank
lboun rbound hitcount time
    plus                int8  int8  0      0
1<1      1<1      2                0.000001
    stranding                wchar8  wchar8  1      1
5<9      0<0      1                0.000001
    name
lookup
    1                0.000001
    aplmon measurement
time
    0.000007
    aplmon unmeasured
time
    0.003490
    aplmon

```

```

total
    0.003500
    APLMON '+.x/1 2 3'
6
  token          lfn  rfn  ltype  rtype  lrank  rrank  lbo
und rbound hitcount time
  dot            plus times int8   int8   0     0
1<1    1<1      1      0.000007
  dot            plus times int8   int32  0     0
1<1    1<1      1      0.000001
  stranding                        wchar8 wchar8 1     1
5<9    0<0      1      0.000001
  name
lookup
    1            0.000001
  aplmon measurement
time
    0.000005
  aplmon unmeasured
time
    0.003058
  aplmon
total
    0.003073

```