

Release Notes

DYALOG APL
The tool of thought for expert programming

Version 13.0

*Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2011 by Dyalog Limited.*

All rights reserved.

Version 13.0

First Edition April 2011

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

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

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.

UNIX is a registered trademark of The Open Group.

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

All other trademarks and copyrights are acknowledged.

Contents

CHAPTER 1 Introduction	1
Summary.....	1
System Requirements	2
Microsoft Windows	2
Microsoft .Net Interface.....	2
Unix and Linux	2
Installation Changes	2
Version Numbering.....	2
Files and Directories	2
Interoperability and Compatibility.....	4
Introduction.....	4
Code	4
“Ordinary” Arrays.....	4
32 vs. 64-bit Component Files	5
External Variables.....	5
32 vs. 64-bit Interpreters	5
Unicode vs. Classic Editions.....	5
□AVU changes	6
DECFs and Complex numbers.....	6
Very large array components	7
File Journaling	7
TCPSockets.....	7
Auxiliary Processors	7
Session Files	7
Improvements to Take, Drop and Index	8
New Functions	9
Left and Same	9
Identity and Right	9
Floating-Point Representation.....	9
Profiling	9
Space Indicator.....	9
Regular Expression Operators.....	9
Variant Operator	10
Update DataTable (2010 \mathbb{I}).....	10
Read DataTable (2020 \mathbb{I}).....	10
Fork New User: Unix only (4000 \mathbb{I})	10
Change User: Unix only (4001 \mathbb{I}).....	10
Reap Forked Tasks: Unix only (4002 \mathbb{I})	10
Signal Counts: Unix only (4007 \mathbb{I})	10
Changes to Formatting.....	11
New Idioms	13
Retained translate table for Grade	14
Correction to Dyadic Grade.....	14

Correction to Index Generator.....	14
Dyalog Unicode IME	15
Configuring the Dyalog Unicode IME	17
OLE interface changes	21
New Parameters	22
Internal Error (99)	23
Deprecation of support for 32-bit component files	24
CHAPTER 2 128 Bit Decimal Floating-Point Support	25
Introduction.....	25
Floating-Point Representation.....	25
Conversion between Decimal and Binary	27
Decimal Comparison Tolerance	27
Passing floating-point values.....	28
Decimal Floats and Microsoft.NET.....	28
CHAPTER 3 Complex Numbers.....	31
Overview	31
Notation.....	31
Arithmetic.....	31
Circular functions	32
Different Result for Power	33
Comparison	34
CHAPTER 4 Search and Replace System Operators	35
Overview.....	35
Syntax	36
Input Document.....	37
Output	38
[R].....	38
[S].....	39
Search pattern.....	39
Transformation pattern.....	39
Transformation codes.....	41
Transformation Function.....	42
Options	43
IC.....	43
Mode.....	44
DotAll.....	44
EOL	45
NEOL	45
ML.....	46
Greedy	46
OM	47
InEnc	47

OutEnc	48
Enc	48
ResultText	48
Line, document and mixed modes	49
Technical Considerations	49
Further Examples	50

CHAPTER 5 Reference to Language Enhancements.....55

Add:	58
And, Lowest Common Multiple:	59
Binomial:	60
Ceiling:	60
Circular:	61
Conjugate:	62
Decode:	63
Direction (Signum):	65
Divide:	65
Drop:	66
Equal:	67
Exponential:	69
Factorial:	69
Floor:	70
Format (Dyadic):	71
Identity:	73
Index:	73
Left:	76
Logarithm:	77
Magnitude:	77
Matrix Divide:	78
Matrix Inverse:	80
Multiply:	81
Natural Logarithm:	81
Negative:	81
Pi Times:	82
Power:	83
Reciprocal:	84
Residue:	84
Right:	85
Same:	85
Subtract:	86
Take:	87
Variant:	88
I-Beam:	91
Update DataTable:	92
Read DataTable:	94
Fork New Task: (UNIX only)	97
Change User: (UNIX only)	98

Reap Forked Tasks: (UNIX only)	R←4002IY	98
Signal Counts: (UNIX only)	R←4007IY	101
Decimal Comparison Tolerance:	□DCT	101
Data Representation (Monadic):.....	R←□DR Y.....	102
File Create:	{R}←X □FCREATE Y	103
Floating-Point Representation:	□FR	105
Name Association:	{R}←{X}□NA Y.....	107
Variant:	{R}←{X}(f □OPT B)Y.....	133
Profile Application:	R←□PROFILE Y.....	133
Space Indicator:	R←□RSI	139
Appendices: PCRE Specifications.....		141
Appendix A – Search Pattern syntax.....		141
Appendix B – Search Pattern syntax summary		178
Appendix C – License		184
Index.....		187

CHAPTER 1

Introduction

Summary

Dyalog APL Version 13.0 provides the following new features and enhancements:

- Support for 128-bit Decimal Floating-Point Number. See Chapter 2.
- Support for Complex Numbers. See Chapter 3.
- New regular expression operators - `⌈R`, `⌈S`. See Chapter 4.
- Short left arguments for Take (`↑`), Drop (`↓`) and Index (`⌈`).
- New primitive functions Left, Same, Identity and Right (tacks).
- New system function `⌈PROFILE` for profiling and performance tuning your applications.
- New system function `⌈RSI`.
- Implementation changes to Format `⌈` and `⌈FMT`
- Matrix append idiom and retained translate table for dyadic `⌈` and `⌈`.
- Correction to dyadic `⌈`.
- Change to the result of `⌈`.
- New I-beam functions `2010⌈` and `2020⌈` to improve the performance when using the ADO.Net DataTable object.
- New I-beam functions `4000⌈`, `4001⌈`, `4002⌈`, and `4007⌈` to perform certain system level operations under UNIX.
- New Input Method Editor (IME) for Unicode Edition.
- OLE Interface changes for default indexers.
- New parameters to control the defaults for component file checksum and journaling, and the accidental input of complex constants.
- Advance notice of the withdrawal of support for 32-bit Component Files.

System Requirements

Microsoft Windows

Dyalog APL Version 13.0 supports all current versions of Windows from Windows 2000 up to and including Windows 7 and Windows Server 2008.

Dyalog APL Version 13.0 is not supported for versions of Windows prior to Windows 2000, such as Windows 95, Windows 98, Windows ME and Windows NT4.

Microsoft .Net Interface

Dyalog APL Version 13.0 .Net Interface requires Version 2.x or greater of the Microsoft .Net Framework. It does *not* operate with .Net Version 1.0.

Unix and Linux

For an up-to-date list of supported Unix and Linux platforms, please contact support@dyalog.com.

Installation Changes

Version Numbering

From Version 13.0 onwards the Version numbers in Dyalog APL have been changed.

The new numbering is in the form 13.0.7626. This allows Windows installers to correctly identify the version numbers. The third number uniquely identifies a particular state of the source code, whereas the BuildID uniquely identifies a specific interpreter executable. The complete version number will always be incremented whenever a new build is made.

Files and Directories

The location of some of the files installed by Dyalog has changed. The following notes refer to a default installation of the 32-bit Unicode edition on a 64 bit version of Windows; the details for Classic and for 64-bit editions will differ slightly. See User Guide for further details.

For a default installation of the 32-bit Unicode edition of Dyalog APL on a 64 bit version of Windows \$DYALOG defaults to:

```
C:\Program Files (x86)\Dyalog\Dyalog APL 13.0 Unicode.
```


Dyalog APL uses the `LoadLibrary()` function to load DLLs. The first directory searched is the directory containing the image file used to create the calling process.

- The files that were previously installed in the GAC (Global Assembly Cache, `\Windows\assembly`) are no longer installed in this way but are located only within the `$DYALOG` directory tree.
- Files that were previously installed in `Program Files\Dyalog\Dyalog 13.0\bin` are now only installed in `$DYALOG`.
- All files that were previously installed in `$DYALOG\bin` are now only installed in `$DYALOG`.

This new policy simplifies the installation of Dyalog APL, and removes a number of issues associated with updating files in the Global Assembly Cache (GAC).

Note that the `$DYALOG\Samples\asp.net` directory and its subdirectories is now self-contained except for the Dyalog script compiler and the Dyalog interpreter DLL. This means that this directory can be moved elsewhere, and as long as the `DyalogCompilerFullPath` key in `web.config` points to the Dyalog script compiler (`dialogc.exe` or `dialogc_unicode.exe`), and the `dyalog130rt.dll` or `dyalog130rt_unicode.dll` is in the same directory as the Dyalog script compiler, the samples will work.

If later patches to the .Net DLLs are installed on the computer, the `web.config` file will need to be updated to refer to the correct version numbers.

Interoperability and Compatibility

Introduction

Workspaces and component files are stored on disk in a binary format (illegible to text editors). This format differs between machine architectures and among versions of Dyalog. For example a file component written by a PC may well have an internal format that is different from one written by a UNIX machine. Similarly, a workspace saved from Dyalog Version 13.0 will differ internally from one saved by a previous version of Dyalog APL.

It is convenient for versions of Dyalog APL running on different platforms to be able to *interoperate* by sharing workspaces and component files. From Version 11.0, component files and workspaces can generally be shared between Dyalog interpreters running on different platforms. However, this is not always possible. For example, component files created by Version 10.1 can often not be shared across platforms, even when used by later versions (the system function `⌘FCOPY` can be used to make a logically identical copy of an old file, which is fully inter-operable).

The following sections describe other limitations in inter-operability:

Code

Code which is saved in workspaces, or embedded within `⌘OR`s stored in component files, can generally only be read by the version which saved them and later versions of the interpreter. In the case of workspaces, a load (or copy) from an older version would fail with the message:

```
      this WS requires a later version of the interpreter.
```

In the case of `⌘OR`, unpredictable behaviour may result if an older version reads a `⌘OR` saved by a later version of the system. In addition each time that a `⌘OR` object is read into a later interpreter time will be spent in converting the internal representation into the latest form. Dyalog recommends that `⌘OR` should not be used as a mechanism for sharing code or objects between different versions of APL

“Ordinary” Arrays

With the exception of the Unicode restrictions described in the following paragraphs, Dyalog APL provides inter-operability for arrays which only contain (nested) character and numeric data. Such arrays can be stored in component files - or transmitted using `TCPsocket` objects and Conga connections, and shared between all versions and across all platforms.

As mentioned in the introduction, full cross-platform interoperability of component files is only available for large component files (see the following section), and for small component files created by Version 11.0 or later.

32 vs. 64-bit Component Files

Large (64-bit-addressing) component files are inaccessible to versions of the interpreter that pre-dated their introduction (versions earlier than 10.1).

The second item in the right argument of `⎕FCREATE` determines the addressing type of the file.

```
'small'⎕fcreate 1 32      A create small file.
'large'⎕fcreate 1 64     A create large file.
```

If the second item is missing, the file type defaults to 64-bit-addressing. In versions prior to 12.0, the default was 32-bit-addressing. It is possible to override these defaults on the command line.

Note that *small* (32-bit-addressing) cannot contain Unicode data. Unicode editions of Dyalog APL can only write character data which would be readable by a Classic edition (consisting of elements of `⎕AV`).

External Variables

External variables are implemented as small (32-bit-addressing) component files, and subject to the same restrictions as these files. External variables are unlikely to be developed further; Dyalog recommends that applications which use them should switch to using mapped files or traditional component files. Please contact Dyalog if you need further advice on this topic.

32 vs. 64-bit Interpreters

From Dyalog APL Version 11.0 onwards, there are two separate versions of programs for 32-bit and 64-bit machine architectures (the 32-bit versions will also run on 64-bit machines running 64-bit operating systems). There is complete inter-operability between 32- and 64-bit interpreters, except that 32-bit interpreters are unable to work with arrays greater than 2GB in size.

Unicode vs. Classic Editions

From Version 12.0 onwards, a Unicode edition is available, which is able to work with the entire Unicode character set. Classic editions (a term which includes versions prior to 12.0) are limited to the 256 characters defined in the atomic vector, `⎕AV`.

Large (64-bit-addressing) component files have a Unicode property; when this is enabled (it is the default), all characters will be written as Unicode data to the file. The Unicode property is always off for small (32-bit addressing) files, which may not contain Unicode data. The Unicode property can be toggled on and off using `⎕FPROPS`.

When a Unicode edition writes to a component file which may not contain Unicode data, character data is mapped to `⎕AV`, and can therefore be read without problems by Classic editions.

A `TRANSLATION ERROR` will occur if a Unicode edition writes to a non-Unicode component (that is either a 32-bit file, or a 64-bit file when the Unicode property is currently off) if the data being written contains characters which are not in `⎕AV` (see `⎕AVU` for more details).

Likewise, a Classic edition (Version 12.0 or later) will issue a `TRANSLATION ERROR` if it attempts to read a component containing Unicode data from a component file. Version 11.0 cannot read components containing Unicode data and issues a `NONCE ERROR`.

A `TRANSLATION ERROR` will also issued when a Classic edition `)LOADs` or `)COPYs` a workspace containing Unicode data which cannot be mapped to `⎕AV`.

`TCPSocket` objects have an `APL` property which corresponds to the Unicode property of a file, if this is set to `Classic` (the default) the data in the socket will be restricted to `⎕AV`, if `Unicode` it will contain Unicode character data. As a result, `TRANSLATION ERRORs` can occur on transmission or reception in the same way as

⎕AVU changes

The implementation of the function `Right` in Version 13.0 led to the discovery that `⎕AVU` incorrectly defined `⎕AV[59+⎕IO]` as `⍤` (`⎕UCS 164`) rather than `⍤` (`Right Tack, ⎕UCS 8866`). This error has been corrected in the default `⎕AVU` and in workspace `AVU`. If you are operating in a mixed Unicode/Classic environment, this error will have caused earlier Classic editions to map `⎕AV[59+⎕IO]` to the wrong Unicode character (`⍤`). This may cause `TRANSLATION ERRORs` when a Version 13.0 Classic system attempts to read the data, as it will not be able to represent `⍤` in the Atomic Vector.

DECFs and Complex numbers

Version 13.0 introduces two new data types; `DECFs` and `Complex numbers`. Attempts to read components of these types in earlier interpreters will result in a `DOMAIN ERROR`.

Very large array components

The maximum size of a component written by Version 12.1 and prior is 2GB. This is the size of the component as held on disk; the maximum size of an array in APL will be slightly smaller. In Version 13.0 the maximum size of a component written by a 64 bit interpreter is 4GB. An attempt to read such a component in 32-bit interpreters will result in a `WS FULL`. An attempt to read such a component in 64-bit Versions 12.0 and 12.1 patched after 1st April 2011 will result in a `NONCE ERROR`; earlier patches generate a `FILE COMPONENT DAMAGED` error.

File Journaling

Version 12.0 introduces File Journaling (level 1), and 12.1 adds levels 2 and 3. Versions earlier than 12.0 cannot tie files which have any form of journaling enabled. Version 12.0 cannot tie files with journaling levels other than 1. Files can be shared with earlier versions by using `⎕FPROPS` to switch journaling off.

TCPSockets

TCPSockets used to communicate between differing versions of Dyalog APL are subject to similar limitations to those described above for component files. In particular TCPSockets with `'Style' 'APL'` will only be able to pass arrays that are supported by both versions.

Auxiliary Processors

A Dyalog APL process is restricted to starting an AP of exactly the same architecture. In other words, the AP must share the same word-width and byte-ordering as its interpreter process.

Session Files

Session (.dse) files may only be used on the platform on which they were created and saved.

Improvements to Take, Drop and Index

You may now elide trailing items of the left argument to the primitive functions Take (\uparrow), Drop (\downarrow) and Index ($\boxed{}$). Instead of causing a RANK ERROR or LENGTH ERROR, the missing items default to appropriate values.

`A←3 4 5 6⍳360` A rank-4 array.

Take: missing trailing items of the left argument default to the length of the corresponding axis.

```
2↑A ↔ 2 4 5 6↑A
1 1↑A ↔ 1 1 5 6↑A
0↑A ↔ 3 4 5 6↑A ↔ A
```

Drop: missing trailing items of the left argument default to 0:

```
2↓A ↔ 2 0 0 0↓A
1 1↓A ↔ 1 1 0 0↓A
0↓A ↔ 0 0 0 0↓A ↔ A
```

Index: missing trailing items of the left argument default to the index vector of the corresponding axis:

```
2⍳A ↔ 2(⍳4)(⍳5)(⍳6)⍳A
1 1⍳A ↔ 1 1 (⍳5)(⍳6)⍳A
0⍳A ↔ (⍳3)(⍳4)(⍳5)(⍳6)⍳A ↔ A
```

New Functions

Left and Same

Monadic \leftarrow (left tack) is called Same and returns its argument. Dyadic \leftarrow is called Left and returns its left argument.

Identity and Right

Monadic \vdash (right tack) is called Identity and returns its argument. Dyadic \vdash is called Right and returns its right argument.

Note that monadic $+$ is now called Conjugate and it is recommended that its use as "Identity" be replaced by monadic \vdash .

Floating-Point Representation

The new system variable $\square FR$ specifies whether floating-point arithmetic is performed using 64-bit binary floating-point or 128-bit decimal floating-point. These options are selected by setting $\square FR$ to 645 or 1287 respectively. The default value of $\square FR$ is configurable. There is also a new system variable $\square DCT$ which controls comparison tolerance for decimal128 comparisons. See Chapter 2 for details.

Profiling

The new System Function $\square PROFILE$ facilitates the profiling of either CPU consumption or elapsed time for an application. It does so by collecting and retaining time measurements for APL functions/operators and function/operator lines. $\square PROFILE$ is used to both control the state of profiling and retrieve the collected profiling data.

Space Indicator

The new System Function $\square RSI$ is identical to $\square NSI$ except that $\square RSI$ returns refs to the spaces whereas $\square NSI$ returns their names, ie. $\square NSI \leftrightarrow \text{⌘} \square RSI$.

Regular Expression Operators

The new regular expression search and replace feature is implemented by 2 new system operators: $\square R$ and $\square S$. See Chapter 4 for details.

Variant Operator

`⍤` or `⍤OPT` (Variant) is a new system operator designed to facilitate the processing of name/value pairs used to set options and parameters, such as those required by `⍤R` and `⍤S`.¹

Update DataTable (2010⍤)

This function performs a *block update* of an instance of the ADO.NET object `System.Data.DataTable`.

Read DataTable (2020⍤)

This function performs a *block read* from an instance of the ADO.NET object `System.Data.DataTable`.

Fork New User: Unix only (4000⍤)

This function *forks* the current APL task. This means that it initiates a new separate copy of the APL program, with exactly the same APL execution stack.

Change User: Unix only (4001⍤)

The function changes the *userid* (*uid*) and *groupid* (*gid*) of the process to values that correspond to the specified user name.

Reap Forked Tasks: Unix only (4002⍤)

Under UNIX, when a child process terminates, it signals to its parent that it has terminated and waits for the parent to acknowledge that signal. 4002⍤ is the mechanism to allow the APL programmer to issue such acknowledgements.

Signal Counts: Unix only (4007⍤)

Obtains a count of the number of signals that have been generated since the last call to this function, or since the start of the process.

¹ The new symbol `⍤` is not present in the Classic Edition so the system function `⍤OPT` is provided as an alternative.

Changes to Formatting.

Monadic \uparrow , Dyadic \uparrow and \square FMT now use 128-bit decimal representation internally when processing floating-point numbers. This has been done to improve the accuracy with which floating-point numbers are converted to text for display and printing.

As a result of this change, the APLFormatBias parameter is no longer supported and if defined will be ignored.

Monadic \uparrow formats complex numbers using J notation (always with a capital J).

Dyadic \uparrow and \square FMT generate DOMAIN ERROR if their right argument contains a complex number.

When \square FR is set to 1287, \uparrow and 'E50. \uparrow ' \square FMT 1, where the right hand side is an integer, will now print 34 "reliable" digits instead of 16 before filling in with underscores.

```

       $\uparrow$  Version 12.1
1.0000000000000000_____

```

```

       $\square$ FR←1287  $\uparrow$  Version 13.0
       $\uparrow$ 
1.00000000000000000000000000000000_____

```

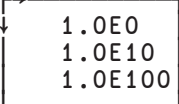
Alignment (E Format)

The following rules of alignment apply to \square FMT (E format), and dyadic \uparrow with a negative left argument.

When \square FR is 1287, dyadic \uparrow (with negative precision) and dyadic \square FMT (E format) will try to move everything left two spaces, to leave room for an extra two digits in the exponent.

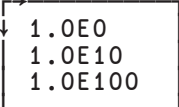
When `⎕FR` is 645, the result is aligned so that there is room for three characters after the E:

```
⎕FR←645
]display 'E10.2'⎕FMT 1 1E10 1E100
```



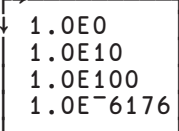
When `⎕FR` is 1287, the result is aligned to leave room for five characters after the E:

```
⎕FR←1287
]display 'E10.2'⎕FMT 1 1E10 1E100
```



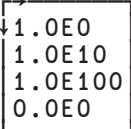
This is done so that there is room for the largest negative exponent that decimal floating point supports:

```
]display 'E10.2'⎕FMT 1 1E10 1E100 1E-6176
```



However, if the field width isn't large enough to allow for the extra trailing spaces, they are truncated (rather than printing a line of asterisks):

```
]display 'E7.2'⎕FMT 1 1E10 1E100 1E-1000
```



New Idioms

The following new idioms are implemented in Version 13.0.

Expression	Description
$A \bar{\leftarrow} A$	Catenate along first axis
$\bar{\leftarrow} / PV$	Join along first axis
$\bar{\leftarrow} A$	First sub-array along first axis
$\bar{\leftarrow} / A$	First sub-array along last axis
$\bar{\leftarrow} A$	Last sub-array along first axis
$\bar{\leftarrow} / A$	Last sub-array along last axis
$* \circ N$	Euler's idiom

Notes

Existing idiom "Catenate To": $V, \leftarrow A$ has been generalised to allow arrays of higher rank on the left, so that its new template expression should be $A, \leftarrow A$. This means, for example, that concatenating a new column to a matrix will be optimised.

New idiom $A \bar{\leftarrow} A$ is identical to $A, \leftarrow A$ except concatenation occurs along the first, rather than the last, axis.

Join along first axis: $\bar{\leftarrow} / PV$ is analogous to existing idiom Join $(, / PV)$ except that conformable items of the argument vector are concatenated along their first, rather than last, axis.

Sub-array selection idioms $\bar{\leftarrow} A$, $\bar{\leftarrow} / A$, $\bar{\leftarrow} A$, and $\bar{\leftarrow} / A$ return the first (respectively. last) rank $(0 \leq i < \rho A)$ sub-array along the first (respectively last) axis of A . For example, if V is a vector, then:

$\bar{\leftarrow} / V$	First item of vector
$\bar{\leftarrow} V$	Last item of vector

Similarly, if M is a matrix, then:

$\bar{\leftarrow} M$	First row of matrix
$\bar{\leftarrow} / M$	First column of matrix
$\bar{\leftarrow} M$	Last row of matrix
$\bar{\leftarrow} / M$	Last column of matrix

The idiom generalises uniformly to higher-rank arrays.

Euler's idiom $\ast\circ N$ 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, even though the machine cannot represent multiples of pi, including $0J1$, accurately.

Retained translate table for Grade

The expressions: $\text{alphabet}\circ\Delta$ and $\text{alphabet}\circ\Psi$ now retain the translate table in a similar way as the set functions ($\iota \in$ etc)

```
f ← a ∘ Δ
f x  ⍝ first run builds the translate table
f y  ⍝ subsequent runs are significantly faster than a Δ y
```

Note that the expression $(a \circ \Delta)''x \ y$ does the same as the above example, without saving the derived function.

Correction to Dyadic Grade

The implementation of dyadic Δ has been fixed to follow the Extended APL Standard.

$X\Delta Y$ is now equivalent to $X\Delta((1\uparrow\rho Y), \times/1\uparrow\rho Y)\rho Y$. Previously, when the rank of Y was greater than 2, $X\Delta Y$ would incorrectly grade the sub-vectors of Y along the last axis, instead of the sub-arrays of Y along the first axis

Correction to Index Generator

The result of the expression $\iota\theta$ has been changed from $\square IO$ to $(\epsilon\theta)$. The previous behaviour was incorrect.

Version 12.1

```

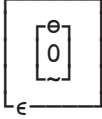
1       $\iota\theta$ 
       $\square IO \leftarrow 0$ 
       $\iota\theta$ 
0       $\square IO \equiv \iota\theta$ 
1
```

Version 13.0

```

]display  $\iota\theta$ 

```



```

1       $(\epsilon\theta) \equiv \iota\theta$ 

```

Dyalog Unicode IME

The mechanism for entering APL characters using the keyboard has been completely revised.

- A brand-new Input Method Editor, the Dyalog Unicode IME, replaces all previous mechanisms in the Unicode Edition. The Dyalog Unicode IME can be used with previous Unicode Editions of Dyalog APL provided that they are patched to a Version created on or after 1st April 2011.
- The Ctrl and AltGr keyboards that were created using the Microsoft Keyboard Layout Creator (MSKLC) and introduced with Version 12.0 are no longer supplied nor formally supported for use with Version 13.0.
- The Comfort On-Screen keyboard is no longer included with nor supported by Version 13.0.

The Dyalog Unicode IME uses the same *Microsoft Input Method Editor* technology as the previous version of the IME, but differs in the following respects:

- It maps keystrokes to Unicode code points (as opposed to positions in the Dyalog Atomic Vector)
- It specifies only the keystrokes for entering APL (and potentially other special) symbols. It does not define or redefine characters entered using the standard keyboard conventions.

This means that the Dyalog Unicode IME acts effectively as an overlay to whatever keyboard is employed, and may be used to enter APL symbols in any application that supports IME technology, i.e. most Windows applications.

Input Translate Tables

The Dyalog Unicode IME is driven by one of a set of Input Translate tables installed in `Program Files\Dyalog\UnicodeIME` directory. For example, the US-English Input table is named:

```
Program Files\Dyalog\UnicodeIME\en_US.din
```

The tables contain entries each of which maps a keystroke to the Unicode code point of an APL symbol. For example, the following entries (from `en_US.din`) map:

- \diamond (Unicode 8900) to Ctrl+`
- $\ddot{\cdot}$ (Unicode 168) to Ctrl+1
- and so forth.



+UNI Desc	Shift	Key	+ Name
8900=Ctrl+` :	2	192	+ Diamond
168=Ctrl+1 :	2	49	+ Diaeresis (Each)
175=Ctrl+2 :	2	50	+ Overbar (High Minus)
60=Ctrl+3 :	2	51	+ Less Than

The Dyalog Unicode IME supplied with Version 13.0 includes support for Danish, Finnish, French, German, Italian, Swedish and British and American English keyboards, based on the Version 12.1 Dyalog Ctrl layouts.

The Dyalog Unicode IME also has support for the Danish, British and American English physical keyboards, which are available from Dyalog Ltd.

The default keyboard mapping for unsupported languages is American English.

To differentiate the new from the old, the Dyalog Unicode IME uses a different icon to that used by the old IME as shown below:

	The Dyalog Unicode IME
	The old APL IME:

See the new *IME User Guide* for further details.

Special Commands

The special commands used by Dyalog APL for Windows Unicode Edition are mapped to keystrokes under *Options/Configure/Keyboard Shortcuts* (see User Guide Chapter 2).

However, the Dyalog Unicode IME translate tables also include default mappings for the special command keystrokes used by Dyalog APL. This mechanism is intended primarily for use by terminal emulators used for running UNIX-based versions of Dyalog APL,

These commands are defined in

```
Program Files\Dyalog\UnicodeIME\special_keys.din
```

For example the command **ER** (execute) is mapped to the Enter key, **ED** (edit) to Shift+Enter, and so forth

```
+ER=Enter:          13  + Enter
ED=Shift+Enter:    1 13  + Edit
TC=Ctrl+Enter:     2 13  + Trace
FD=Ctrl+Shift+Enter: 3 13  + Forward.
```

These command keystroke mappings are ignored by applications unless the application is explicitly named by the `WantsSpecialKeys` parameter.

Configuring the Dyalog Unicode IME

The following description uses screenshots taken from a Windows 7 PC with three Input Languages configured for the current user: English (United Kingdom) – the default Input Language, Danish (Denmark) and English (United States).

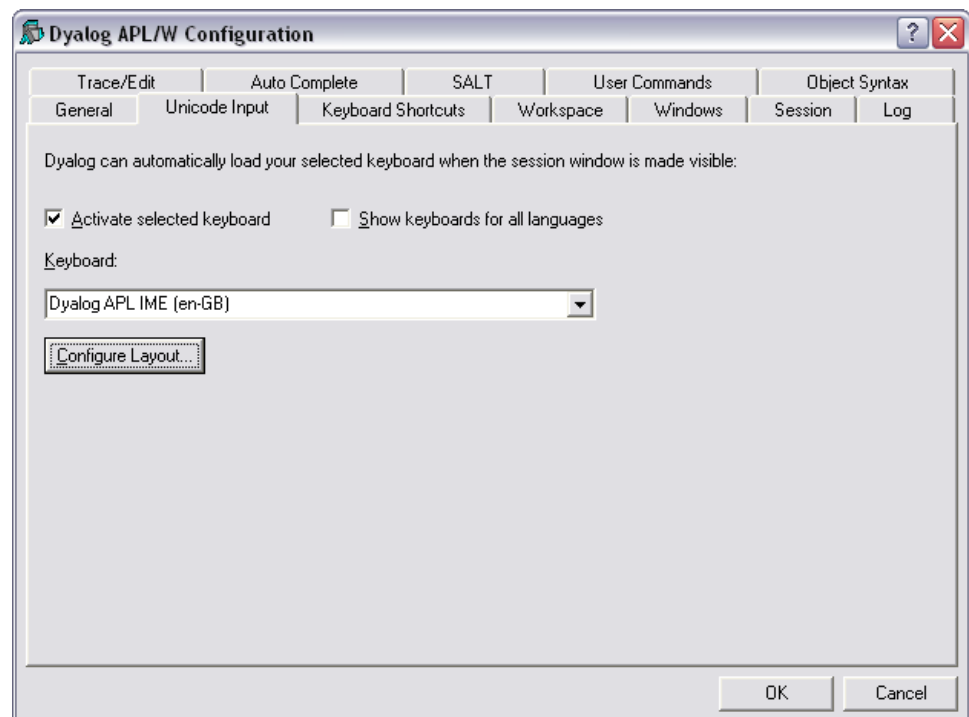
The Dyalog Unicode IME is added as an additional service to all keyboards defined to the user and the administrator at the time that the IME was installed.

For each IME the underlying keyboard layout file will be the same as that defined for the base keyboard. The layout file is a DLL created by Microsoft.

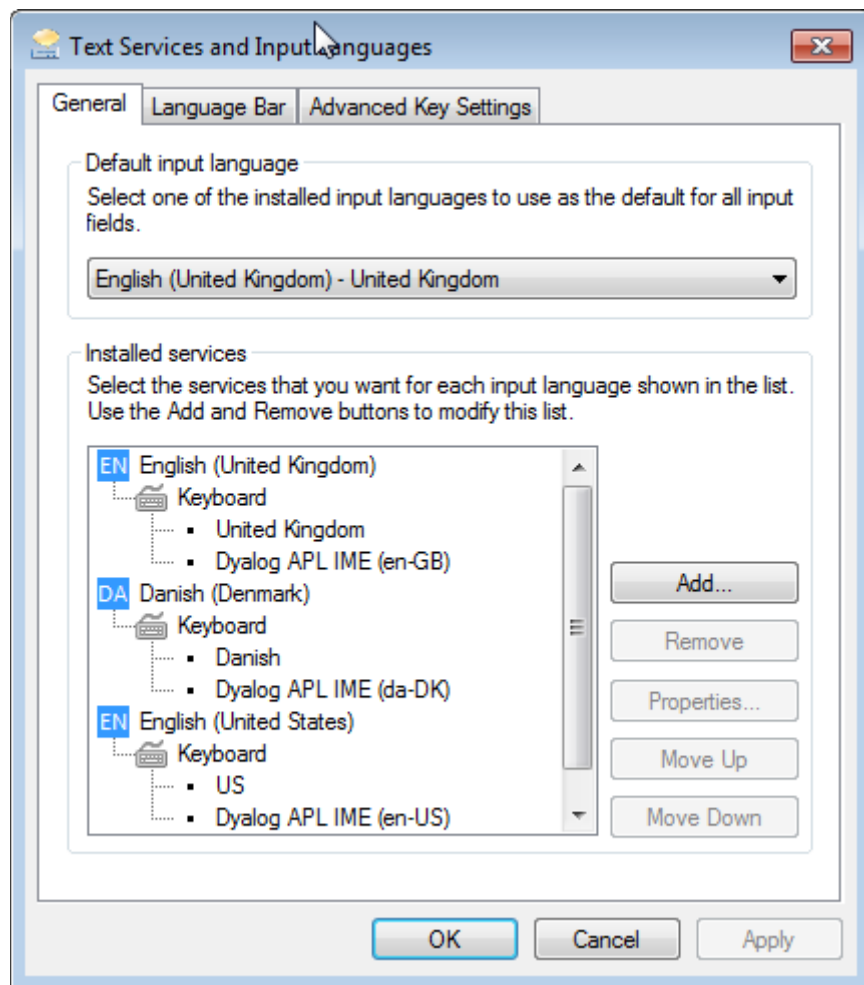
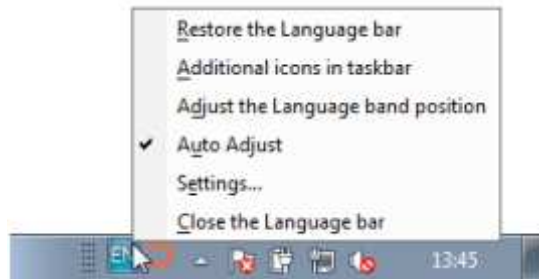
The language specified in the description of the IME is the name of the IME translate table that has been associated with the IME for the specific keyboard. In the case of languages not supported by the IME the keyboard will default to en-US. With the IME as supplied with Version 13.0 altering this text requires editing the appropriate Registry value.

The IME may be configured from within APL or from Windows.

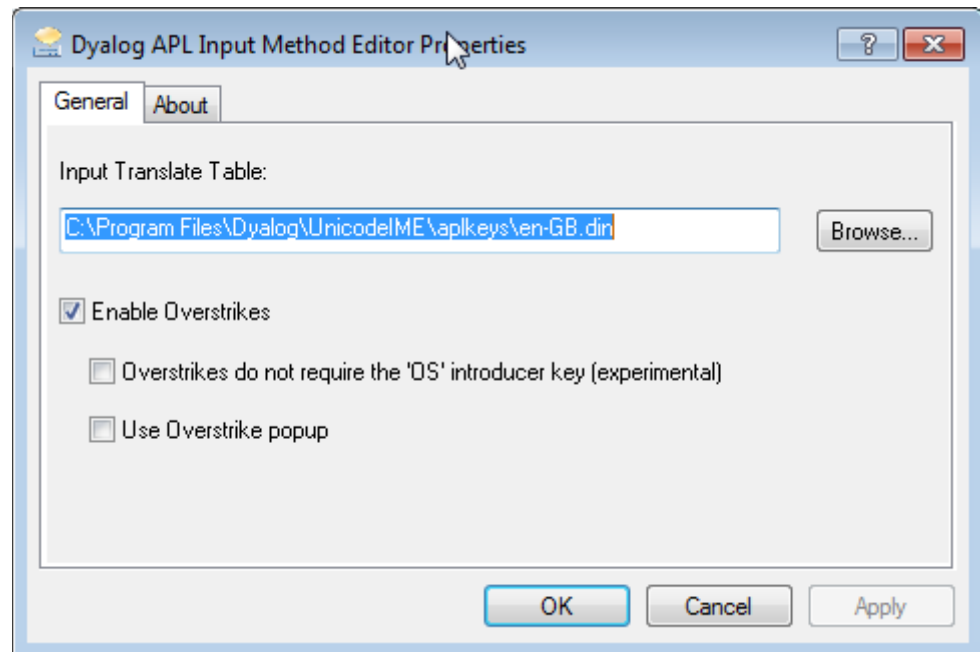
1. **From within Dyalog APL**, to change the properties of the IME go to *Options/Configure/Unicode Input* tab and select *Configure Layout*:



2. **From Windows**, right click on either the Input Language icon or the Keyboard layout icon in the TaskBar and select Settings:



To alter the configuration of any of the installed IMEs, select that IME and click on Properties:



Input translate table:

The translate table defines the mapping between APL characters and the keystrokes that generate those APL characters. It is possible to alter the mapping or to create support for new keyboards by altering the translate table, or by selecting a different translate table. See the *IME User Guide* for more details.

Overstrikes:

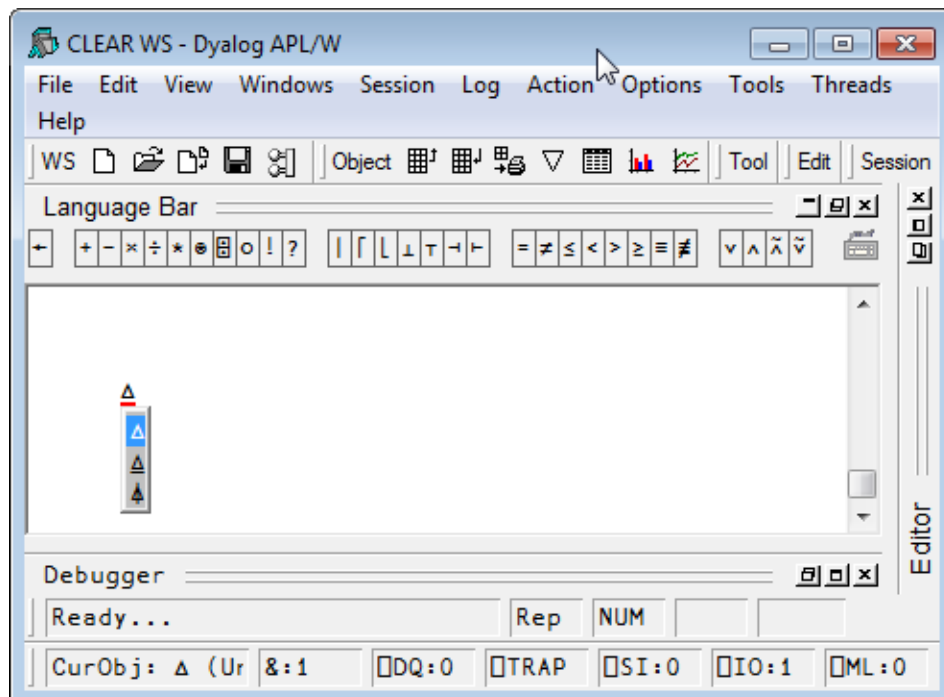
In the original implementations of APL, many of the special symbols could only be generated by overstriking one character on top of another as is reflected in the appearance of the glyphs. For example, the symbol for Grade Up (Δ) is actually the symbol for delta (Δ) superimposed on the symbol for vertical bar (|)

In Dyalog APL such symbols can be generated either by a single keystroke, or (in Replace mode) by overtyping one symbol with another. For example Δ may be generated using Shift+Ctrl+4, or by switching to *Replace* mode and typing the three keystrokes Ctrl+h, Left-Cursor, Ctrl+m.

Using the Dyalog Unicode IME the character can also be entered by pressing Ctrl+Bksp, Ctrl+m, Ctrl+h. Note that Ctrl+Bksp is the default *Overstrike Introducer Key* (key code OS).

Use Overstrike popup:

With this option selected, when the character following the Overstrike Introducer Key is pressed, a popup box displays all the overstrikes which contain the last character typed: in the example below Ctrl+Bksp has been followed by Ctrl+h:



Note the fine (red) line under the Δ in the Session window. This indicates that an overstrike creation operation is in progress.

The input of the symbol Δ can be completed by pressing Ctrl+m, or by moving the selection up and down the pop-up list using Cursor-Up or Cursor-Down

Overstrikes do not require the OS introducer key (experimental):

With this option selected, the IME identifies characters which are part of a valid overstrike, and when such a character is entered into the session, begins an overstrike creation operation. This mode is experimental in the IME supplied with Version 13.0.

OLE interface changes

Default indexers which are implemented as methods rather than as properties, must now be accessed using indexing and not using function calls. For example, to obtain the individual members of the Shapes collection of the Excel Work Sheet object, it was previously necessary to call its Item method as a function.

Version 12.1

```
Excel.Application.ActiveSheet.Shapes.Item 1
```

Version 13.0

In Version 13.0, this is no longer supported, and you must use indexing instead.

```
Excel.Application.ActiveSheet.Shapes[1]
```

New Parameters

APL_FCREATE_PROPS_C

This parameter specifies the default checksum level for newly-created component files.

APL_FCREATE_PROPS_J

This parameter specifies the default journaling level for newly-created component files.

APL_FAST_FCHK

This parameter specifies whether Dyalog APL should optimise `⎕FCHK` by allowing it to reliably determine whether a component file had been properly untied and therefore does not need to be checked (this is overridable using the `⎕FCHK` option 'force').

Optimising `⎕FCHK` in this way has a performance impact on `⎕FUNTIE` and it is recommended this optimisation is switched off if your application frequently ties and unties files.

Note: this only affects component files with journaling enabled.

The values of the parameter are:

- 0 Do not optimise `⎕FCHK` (optimise `⎕FUNTIE` instead)
- 1 Optimise `⎕FCHK`

The default values of the parameter reflect the existing behaviour in Version 12.1: 0 on Windows and 1 on Linux / AIX. On Windows, setting the value 1 has no effect.

APL_EXTERN_DECF

By default, arrays of type DECF (128-bit decimal) will be passed unchanged to Auxiliary Processors, and to DLLs using A or Z argument types. However, if `APL_EXTERN_DECF` is set to 0, DECF arrays will be converted to DOUBLE before they are passed to AP's and DLL's. This will allow user-written Auxiliary Processors and DLLs to continue to work at least temporarily while users determine how to change their code to cater for the new data type. This parameter will not be supported beyond Version 13.0.

APL_COMPLEX_AS_V12

Support for Complex Numbers (Chapter 3) means that some functions produce different results from previous Versions of Dyalog APL. If `APL_COMPLEX_AS_V12` is set to 1 the behaviour of code developed using Version 12.1 or earlier will be unchanged; in particular:

- Power (`*`) and logarithm (`⊗`) do not produce Complex Numbers as results from non-complex arguments.
- `⊖VFI` will not honour "J" or "j" as part of a number.
- `⊖4○Y` will be evaluated as $(-1+Y*2)*0.5$, which is positive for negative real arguments.

If `APL_COMPLEX_AS_V12` is set to any other value or is not set at all then code developed using version 12.1 or earlier may now generate Complex Numbers.

Note that this feature is provided to simplify the transition of older code to Version 13.0. It does not prevent the generation and use of Complex Numbers using features new to 13.0 (such as explicitly specifying a Complex Number literal), and it will be removed in a future release of Dyalog APL.

Internal Error (99)

`INTERNAL ERROR` indicates a severe system error from which Dyalog APL has recovered; in the past such errors would always have resulted in a `Syserror`.

Should you encounter `INTERNAL ERROR`, Dyalog strongly recommends that you save your work(space) , and report the issue. No existing `Syserror` reports have yet been changed into `INTERNAL ERROR`.

Deprecation of support for 32-bit component files

Since Version 10.1, Dyalog APL has supported **large span** (64-bit) component files, and since Version 12.0 `⎕FCREATE` has created these by default. Existing **small span** (32-bit) component files are still supported and 32-bit component files may still be created if suitable options are specified, but they have restrictions which 64-bit files do not, including:

- The maximum file size is 4GB.
- The files are not fully architecture-independent meaning that there are limitations sharing them between, for example, Windows or Linux and AIX machines.
- Components may not contain Unicode data.

Dyalog intends to withdraw support for 32-bit component files in future releases.

If you have any existing 32-bit component files, or applications which create and/or use them, Dyalog recommends that you prepare for this in the following ways:

- Ensure that Dyalog is not started with the command-line option `-F32`. This option sets the default component file type which is created to 32-bit.
- Ensure that no `⎕FCREATE` within your applications explicitly specifies that 32-bit files are to be created.
- Make plans to convert any existing 32-bit component files to 64-bit using `⎕FCOPY`. `⎕FCOPY` will create a 64-bit copy even if the file being copied is 32-bit.

Note: in order to allow the use of legacy files retrieved from backups etc., Dyalog will continue to provide a means to convert 32-bit files to supported formats for a minimum of 10 years after direct support is withdrawn.

CHAPTER 2

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, Version 13.0 introduces support for the 128-bit decimal data type described by IEEE-754-2008 as an alternative representation for floating-point values.

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 chips starting with the “P6”, and recent “z” series mainframes). Even with hardware support, a slowdown of a factor of 4 can be expected. For this reason, Dyalog Version 13 allows users to decide whether they need the higher-precision decimal representation, or prefer to stay with the faster and smaller binary representation.

A new 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  A Type of a floating-point constant
⎕FR = ⎕DR 3÷4   A 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 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 3’s. Using the tolerance which applies to binary floats (type 645), the numbers are equal. However, the “reverse” experiment yields 0, as tolerance is much narrower in the 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 new 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).

Passing floating-point values

⌈NA now supports a new 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 AP’s and DLL’s 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.

Version 13.0 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.

The .Net type System.Int64 will now always be cast to a 128-bit decimal number when entering Dyalog APL, regardless of the setting of []FR. So long as no 64-bit arithmetic is performed on such a value, it will remain a 128-bit number and can be passed back to .Net without loss.

CHAPTER 3

Complex Numbers

Overview

Dyalog APL Version 13.0 introduces support for complex numbers. In simple terms, a complex number² 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 $i^2 = -1$.

Notation

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.

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

.3j.5
0.3J0.5

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

Arithmetic

The arithmetic primitive functions have been extended accordingly to handle complex numbers.

```
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+bd i2
-0.9J1.9      A          = (ac-bd)+(bc+ad)i
```

² http://en.wikipedia.org/wiki/Complex_number

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

```
5      |3j4
```

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

```
3j4   +3j4
3j-4
```

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

```
25      3j4*3j-4
```

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

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

The famous Euler's Identity may be expressed as follows:

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

Circular functions

The circular functions $X \circ Y$ have been extended for complex values in Y . In addition, the following new functions have been added where 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.o3.5J-1.2 2J3 3J4
3.5 2 3
-1.2 3 4
```

Different Result for Power

In Version 13.0, the implementation of X^Y (Power) gives a different answer for negative real X than in all previous Versions of Dyalog APL. This change is however in accordance with the ISO/IEC 13751 Standard for Extended APL.

In Version 13.0, the result is the principal value; whereas in previous Versions the result is a negative or positive real number or DOMAIN ERROR. The following examples illustrate this point:

```

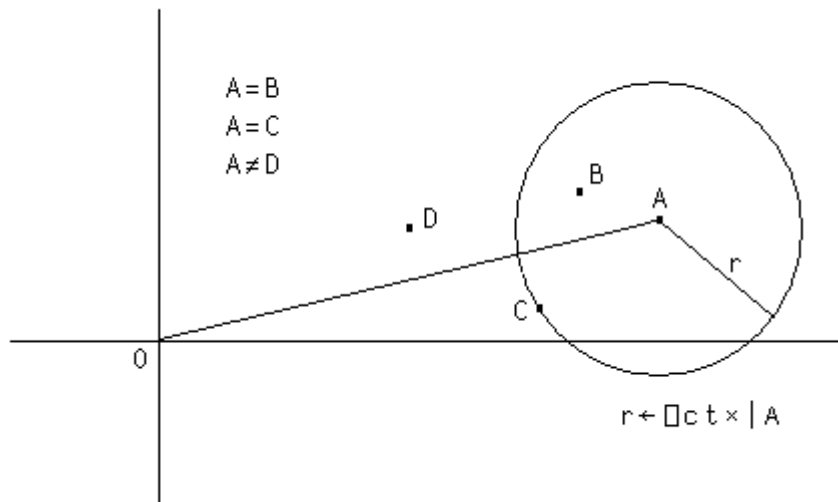
      -8 * 1 2 ÷ 3          A Version 12.1
-2 4
      -8 * 1 2 ÷ 3          A Version 13.0
1J1.732050808 -2J3.464101615

      * (1 2 ÷ 3) * ⍉ -8    A Version 13.0
1J1.732050808 -2J3.464101615

```

Comparison

In comparing two complex numbers X and Y , $X=Y$ is 1 if the magnitude of $X-Y$ does not exceed ϵ_{CT} times the larger of the magnitudes of X and Y ; geometrically, $X=Y$ if the number smaller in magnitude lies on or within a circle centred on the one with larger magnitude, having radius ϵ_{CT} 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 ϵ_{FR} . Thus, comparisons between complex numbers and decimal floating-point numbers are subject to ϵ_{CT} , not ϵ_{DCT} . 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 plane.

CHAPTER 4

Search and Replace System Operators

Overview

`⊠R` (Replace) and `⊠S` (Search) are system operators which take search pattern(s) as their left arguments and transformation rule(s) as their right arguments; the derived function operates on text data to perform either a **search**, or a search and **replace** operation.

The search patterns may include *Regular Expressions* so that complex searches may be performed. `⊠R` and `⊠S` utilise the open-source regular-expression search engine PCRE, which is built into Dyalog APL and distributed according to the license in Appendix C.

The transformation rules are applied to the text which matches the search patterns; they may be given as a simple character vector, numeric codes, or – for greatest flexibility – a function.

The operators use the Variant operator to set options.

Examples of replace operations

```
('.at' ⊠R '\u0') 'The cat sat on the mat'
The CAT SAT on the MAT
```

In the search pattern the dot matches any character, so the pattern as a whole matches sequences of three characters ending 'at'. The transformation is given as a character string, and causes the entire matching text to be folded to upper case.

```
('w+' ⊠R {ϕω.Match}) 'The cat sat on the mat'
ehT tac tas no eht tam
```

The search pattern matches each word. The transformation is given as a function, which receives a namespace containing various variables describing the match, and it returns the match in reverse, which in turn replaces the matched text.

Examples of search operations

```
STR←'The cat sat on the mat'  
('.at' □S '\u0') STR  
CAT SAT MAT
```

The example is identical to the first, above, except that after the transformation is applied to the matches the results are returned in a vector, not substituted into the source text.

```
('.at' □S {ω.((1↑Offsets),1↑Lengths)}) STR  
4 3 8 3 19 3
```

When searching, the result vector need not contain only text and in this example the function returns the numeric position and length of the match given to it; the resultant vector contains these values for each of the three matches.

```
('.at' □S 0 1) STR  
4 3 8 3 19 3
```

Here the transformation is given as a vector of numeric codes which are a short-hand for the position and length of each match; the overall result is therefore identical to the previous example.

These examples all operate on a simple character vector containing text, but the text may be given in several forms - character vectors, vectors of character vectors, and external data streams. These various forms constitute a 'document'. When the result also takes the form of a document it may be directed to a stream.

Syntax

$$\{R\} \leftarrow \{X\} (A \square R B) Y$$

The two system operators, $\square R$ for replace and $\square S$ for search, are syntactically identical. With $\square R$, the input document is examined; text which matches the search pattern is amended and the remainder is left unchanged. With $\square S$, each match in the input document results in an item in the result whose type is dependent on the transformation specified.

A specifies one or more search patterns, being given as a single character, a character vector, a vector of character vectors or a vector of both characters and character vectors. See 'search pattern' following.

B is the transformation to be performed on matches within the input document; it may be either one or more transformation patterns (specified as a character, a character vector, a vector of character vectors, or a vector of both characters and character vectors), one or more transformation codes (specified as a numeric scalar or a numeric vector) or a function; see 'transformation pattern', 'transformation codes' and 'transformation function' following.

Y specifies the input document; see ‘input document’ below.

X optionally specifies an output stream; see ‘output’ below.

R is the result value; see ‘output’ below.

Input Document

The input document may be an array or a data stream.

When it is an array it may be given in one of two forms:

1. A character scalar or vector
2. A vector of character vectors

In Version 13.0 the only supported data stream is a native file, specified as tie number, which is read from the current position to the end. If the file is read from the start, and there is a valid Byte Order Mark (BOM) at the start of it, the data encoding is determined by this BOM. Otherwise, data in the file is assumed to be encoded as specified by the **InEnc** option.

Hint: once a native file has been read to the end by `{}R` or `{}S` it is possible to reset the file position to the start so that it may be read again using:

```
{ } {}NREAD t ienum 82 0 0
```

The input document is comprised of lines of text. Line breaks may be included in the data:

Implicitly,

- Between each item in the outer vector (type 2, above)

Explicitly, as

- carriage return
- line feed
- carriage return and line feed together, in that order
- vertical tab (U+000B)
- newline (U+0085)
- form Feed (U+000C)
- line Separator (U+2028)
- paragraph Separator (U+2029)

The implicit line ending character may be set using the **EOL** option. Explicit line ending characters may also be replaced by this character - so that all line endings are normalised - using the **NEOL** option.

The input document may be processed in **line** mode, **document** mode or **mixed** mode. In document mode and mixed mode, the entire input document, line ending characters included, is passed to the search engine; in line mode the document is split on line endings and passed to the search engine in sections without the line ending characters. The choice of mode affects both memory usage and behaviour, as documented in the section 'Line, document and mixed modes'.

Output

The format of the output is dependent on whether `⎕S` or `⎕R` are in use, whether an output stream is specified and, for `⎕R`, the form of the input and whether the **ResultText** option is specified.

An output data stream may optionally be specified. In Version 13.0 the only supported data stream is a native file, specified as file number, and all output will be appended to it. Data in the stream is encoded as specified by the **OutEnc** option. If this encoding specifies a Byte Order Mark and the file is initially empty then the Byte Order Mark will be written at the start. Appending to existing data using a different encoding is permitted but unlikely to produce desirable results. If an input stream is also used, care must be taken to ensure the input and output streams are not the same.

`⎕R`

With no output stream specified and unless overridden by the **ResultText** option, the derived function result will be a document which closely matches the format of the input document, as follows:

A **character scalar or vector** input will result in a **character vector** output. Any and all line endings in the output will be represented by line ending characters within the character vector.

A **vector of character vectors** as input will result in a **vector of character vectors** as document output. Any and all line endings in the output document will be implied at the end of each character vector.

A **stream** as input will result in a **vector of character vectors** document output. Any and all line endings in the output document will be implied at the end of each character vector.

Note that the shape of the output document may be significantly different to that of the input document.

If the **ResultText** option is specified, the output type may be forced to be a **character vector** or **vector of character vectors** as described above, regardless of the input document.

With an output stream specified there is no result - instead the text is appended to the stream. If the appended text does not end with a line ending character then the line ending character specified by the **EOL** option is also appended.

□S

With no output stream specified, the result will be a vector containing one item for each match in the input document, of types determined by the transformation performed on each match.

With an output stream specified there is no result - instead each match is appended to the stream. If any match does not end with a line ending character then the line ending character specified by the **EOL** option is also appended. Only text may be written to the stream, which means:

- When a transformation function is used, the function may only generate a character vector result.
- Transformation codes may not be used.

Search pattern

The syntax of the search pattern is reproduced from the PCRE documentation verbatim in appendices A and B.

There may be multiple search patterns. If more than one search pattern is specified and more than one pattern matches the same part of the input document then priority is given to the pattern specified first.

Transformation pattern

For each match in the input document, the transformation pattern causes the creation of text which, for □R, replaces the matching text and, for □S, generates one item in the result.

There may be either one transformation pattern, or the same number of transformation patterns as search patterns. If there are multiple search patterns and multiple transformation patterns then the transformation pattern used corresponds to the search pattern which matched the input text.

Transformation patterns may not be mixed with transformation codes or functions.

The following characters have special meaning:

%	acts as a placeholder for the entire line (line mode) or document (document mode or mixed mode) which contained the match
&	acts as a placeholder for the entire portion of text which matched
\n	represents a line feed character
\r	represents a carriage return
\0	equivalent to &
\n	acts as a placeholder for the text which matched the first to ninth subpattern; <i>n</i> may be any single digit value from 1 to 9
\(n)	acts as a placeholder for the text which matched the numbered subpattern; <i>n</i> may have an integer value from 0 to 63.
\<name>	acts as a placeholder for the text which matched the named subpattern
\\	represents the backslash character
\%	represents the percent character
\&	represents the ampersand character

The above may be qualified to fold matching text to upper- or lower-case by using the **u** and **l** modifiers respectively. Character sequences beginning with the backslash place the modifier after the backslash; character sequences with no leading backslash add both a backslash and the modifier to the start of the sequence, for example:

\u&	acts as a placeholder for the entire portion of text which matched, folded to upper case
\l0	equivalent to \l&

Character sequences beginning with the backslash other than those shown are invalid. All characters other than those shown are literal values and are included in the text without modification.

Transformation codes

The transformation codes are a numeric scalar or vector. For each match in the input document, a numeric scalar or vector of the same shape as the transformation codes is created, with the codes replaced with values as follows:

0. The offset from the start of the line (line mode) or document (document mode or mixed mode) of the start of the match, origin zero.
1. The length of the match.
2. In line mode, the block number in the source document of the start of the match. The value is origin zero. In document mode or mixed mode this value is always zero.
3. The pattern number which matched the input document, origin zero.

Transformation codes may only be used with `{}S`.

Transformation Function

The transformation function is called for each match within the input document. The function is monadic and is passed a namespace, containing the following variables:

Block	The entire line (line mode) or document (document mode or mixed mode) in which the match was found.
BlockNum	With line mode, the block (line) number in the source document of the start of the match. The value is origin zero. With document mode or mixed mode the entire document is contained within one block and this value is always zero.
Pattern	The search pattern which matched.
PatternNum	The index-zero pattern number which matched.
Match	The text within Block which matched Pattern.
Offsets	A vector of one or more index-zero offsets relative to the start of Block. The first value is the offset of the entire match; any and all additional values are the offsets of the portions of the text which matched the subpatterns, in the order of the subpatterns within Pattern.
Lengths	A vector of one or more lengths, corresponding to each value in Offset.
Names	A vector of one or more character vectors corresponding to each of the values in Offsets, specifying the names given to the subpatterns within Pattern. The first entry (corresponding to the match) and all subpatterns with no name are included as length zero character vectors.
ReplaceMode	A Boolean indicating whether the function was called by <code>⎕R</code> (value 1) or <code>⎕S</code> (value 0).
TextOnly	A Boolean indicating whether the return value from the function must be a character vector (value 1) or any value (value 0).

The return value from the function is used as follows:

With `⎕R` the function must return a character vector. The contents of this vector are used to replace the matching text.

With `⎕S` the function may return no value. If it does return a value:

- When output is being directed to a stream it must be a character vector.
- Otherwise, it may be any value. The overall result of the derived function is the catenation of the enclosure of each returned value into a single vector.

The passed namespace exists over the lifetime of `SR` or `SS`; the function may therefore preserve state by creating variables in the namespace.

The function may itself call `SR` or `SS`.

The locations of the match within `Block` and subpatterns within `Match` are given as offsets rather than positions, i.e. the values are the number of characters preceding the data, and are not affected by the Index Origin.

There may be only one transformation function, regardless of the number of search patterns.

Options

Options are specified using the Variant operator. The Principal option is `IC`.

Default values are indicated with a tick (✓).

IC

When set, case is ignored in searches.

1	Matches are not case sensitive.
0 ✓	Matches are case sensitive.

Example:

```
( '[AEIOU]' SR 'X' [ 'IC' 1 ) 'ABCDE abcde'
XBCDX XbcdX
( '[AEIOU]' SR 'X' [ 1 ) 'ABCDE abcde'
XBCDX XbcdX
```

Mode

Specifies whether the input document is interpreted in **line** mode, **document** mode or **mixed** mode.

'L' ✓	When line mode is set, the input document is split into separate lines (discarding the line ending characters themselves), and each line is processed separately. This means that the ML option applies per line, and the '^' and '\$' anchors match the start and end respectively of each line. Because the document is split, searches can never match across multiple lines, nor can searches for line ending characters ever succeed. Setting line mode can result in significantly reduced memory requirements compared with the other modes.
'D'	When document mode is set, the entire input document is processed as a single block. The ML option applies to this entire block, and the '^' and '\$' anchors match the start and end respectively of the block - not the lines within it. Searches can match across lines, and can match line ending characters.
'M'	When mixed mode is set, the '^' and '\$' anchors match the start and end respectively of each line, as if line mode is set, but in all other respects behaviour is as if document mode is set - the entire input document is processed in a single block.

Examples:

```
( '$' ⍋R '[Endline]' ⍋ 'Mode' 'L' ) 'ABC' 'DEF'
ABC[Endline] DEF[Endline]
```

```
( '$' ⍋R '[Endline]' ⍋ 'Mode' 'D' ) 'ABC' 'DEF'
ABC DEF[Endline]
```

```
( '$' ⍋R '[Endline]' ⍋ 'Mode' 'M' ) 'ABC' 'DEF'
ABC[Endline] DEF[Endline]
```

DotAll

Specifies whether the dot ('.') character in search patterns matches line ending characters.

0 ✓	The '.' character in search patterns matches most characters, but not line endings.
1	The '.' character in search patterns matches all characters.

This option is invalid in line mode, because line endings are stripped from the input document.

Example:

```
( '.' □R 'X' □('Mode' 'D') 'ABC' 'DEF'
XXX  XXX
( '.' □R 'X' □('Mode' 'D')('DotAll' 1)) 'ABC' 'DEF'
XXXXXXXX
```

EOL

Sets the line ending character which is implicitly present between character vectors, when the input document is a vector of character vectors.

CR	Carriage Return (U+000D)
LF	Line Feed (U+000A)
CRLF ✓	Carriage Return followed by New Line
VT	Vertical Tab (U+000B)
NEL	New Line (U+0085)
FF	Form Feed (U+000C)
LS	Line Separator (U+2028)
PS	Paragraph Separator (U+2029)

In the Classic Edition, setting a value which is not in □AVU may result in a TRANSLATION ERROR.

Example:

```
(' \n' □R 'X' □('Mode' 'D')('EOL' 'LF')) 'ABC' 'DEF'
ABCXDEF
```

Here, the implied line ending between ‘ABC’ and ‘DEF’ is ‘\n’, not the default ‘\r\n’.

NEOL

Specifies whether explicit line ending sequences in the input document are normalised by replacing them with the character specified using the **EOL** option.

0 ✓	Line endings are not normalised.
1	Line endings are normalised.

Example:

```
a←'ABC',(1↑2↓⊞AV),'DEF',(1↑3↓⊞AV),'GHI'
('⋄\n'⋄S 0 ⋄ 'Mode' 'D' ⋄ 'NEOL' 1 ⋄ 'EOL' 'LF') a
3 7
```

'\n' has matched both explicit line ending characters in the input, even though they are different.

ML

Sets a limit to the number of processed pattern matches per line (line mode) or document (document mode and mixed mode).

Positive value n	Sets the limit to the first n matches.
0 ✓	Sets no limit.
Negative value \bar{n}	Sets the limit to exactly the n th match.

Examples:

```
('. ' ⋄R 'x' ⋄ 'ML' 2) 'ABC' 'DEF'
xxC  xxF
('.' ⋄R 'x' ⋄ 'ML'  $\bar{2}$ ) 'ABC' 'DEF'
AxC  DxF
('.' ⋄R 'x' ⋄ 'ML'  $\bar{4}$  ⋄ 'Mode' 'D') 'ABC' 'DEF'
ABC  xEF
```

Greedy

Controls whether patterns are “greedy” (and match the maximum input possible) or are not (and match the minimum). Within the pattern itself it is possible to specify greediness for individual elements of the pattern; this option sets the default.

1 ✓	Greedy by default.
0	Not greedy by default.

Examples:

```
('[A-Z].*[0-9]' ⋄R 'X' ⋄ 'Greedy' 1) 'ABC123 DEF456'
X
('[A-Z].*[0-9]' ⋄R 'X' ⋄ 'Greedy' 0) 'ABC123 DEF456'
X23 X56
```

OM

Specifies whether matches may overlap.

1	Searching continues for all patterns and then from the character following the <i>start</i> of the match, thus permitting overlapping matches.
0 ✓	Searching continues from the character following the <i>end</i> of the match.

This option may only be used with `□S`. With `□R` searching always continues from the character following the end of the match (the characters following the start of the match will have been changed).

Examples:

```

([0-9]+ □S '\0' ☒ 'OM' 0) 'A 1234 5678 B'
1234 5678
([0-9]+ □S '\0' ☒ 'OM' 1) 'A 1234 5678 B'
1234 234 34 4 5678 678 78 8
    
```

InEnc

This option specifies the encoding of the input stream when it cannot be determined automatically.

When the stream is read from its start, and the start of the stream contains a recognised Byte Order Mark (BOM), the encoding is taken as that specified by the BOM and this option is ignored. Otherwise, the encoding is assumed to be as specified by this option.

UTF8 ✓	The stream is processed as UTF-8 data. Note that ASCII is a subset of UTF-8, so this default is also suitable for ASCII data.
UTF16LE	The stream is processed as UTF16 little-endian data.
UTF16BE	The stream is processed as UTF16 big-endian data.
ASCII	The stream is processed as ASCII data. If the stream contains any characters outside of the ASCII range then an error is produced.
ANSI	The stream is processed as ANSI (Windows-1252) data.

For compatibility with the **OutEnc** option, the above UTF formats may be qualified with `-BOM` (e.g. `UTF-BOM`). For input streams, the qualified and unqualified options are equivalent.

OutEnc

When the output is written to a stream, the data may be encoded on one of the following forms:

Implied ✓	If input came from a stream then the encoding format is the same as the input stream, otherwise UTF-8
UTF8	The data is written in UTF-8 format.
UTF16LE	The data is written in UTF-16 little-endian format.
UTF16BE	The data is written in UTF-16 big-endian format.
ASCII	The data is written in ASCII format.
ANSI	The data is written in ANSI (Windows-1252) format.

The above UTF formats may be qualified with -BOM (e.g. UTF8-BOM) to specify that a Byte Order Mark should be written at the start of the stream. For files, this is ignored if the file already contains any data.

Enc

This option sets both **InEnc** and **OutEnc** simultaneously, with the same given value. Any option value accepted by those options except Implied may be given.

ResultText

For `⎕R`, this option determines the format of the result.

Implied ✓	The output will either be a character vector or a vector of character vectors , dependent on the input document type
Simple	The output will be a character vector . Any and all line endings in the output will be represented by line ending characters within the character vector.
Nested	The output will be a vector of character vectors . Any and all line endings in the output document will be implied at the end of each character vector.

This option may only be used with `⎕R`.

Examples:

```

⎕UCS ⋄ ('A' ⎕R 'x') 'AB' 'CD'
120 66 67 68
⎕UCS ('A' ⎕R 'x' ⎕ 'ResultText' 'Simple') 'AB' 'CD'
120 66 13 10 67 68

```

Line, document and mixed modes

The Mode setting determines how the input document is packaged as a block and passed to the search engine. In line mode each line is processed separately; in document mode and mixed mode the entire document is presented to the search engine. This affects both the semantics of the search expression, and memory usage.

Semantic differences

- The **ML** option applies per block of data.
- In line mode, search patterns cannot be constructed to span multiple lines. Specifically, patterns that include line ending characters (such as `'r'`) will never match because the line endings are never presented to the search engine.
- By default the search pattern metacharacters `'^'` and `'$'` match the start and end of the block of data. In line mode this is always the start and end of each line. In document mode this is the start and end of the document. In mixed mode the behaviour of `'^'` and `'$'` are amended by setting the PCRE option `'MULTILINE'` so that they match the start and end of each line within the document.

Memory usage differences

- Blocks of data passed to the search engine are processed and stored in the workspace. Processing the input document in line mode limits the total memory requirements; in particular this means that large streams can be processed without holding all the data in the workspace at the same time.

Technical Considerations

`□R` and `□S` utilise the open-source regular-expression search engine PCRE, which is built into the Dyalog software and distributed according to the license in Appendix C.

Before data is passed to PCRE it is converted to UTF-8 format. This converted data is buffered in the workspace; processing large documents may have significant memory requirements. In line mode, the data is broken into individual lines and each is processed separately, potentially reducing memory demands.

It is possible to save a workspace with an active `□R` or `□S` on the stack and execution can continue when the workspace is reloaded with the same interpreter version. Later versions of the interpreter may not remain compatible and may signal a `DOMAIN ERROR` with explanatory message in the status window if it is unable to continue execution.

PCRE has a buffer length limit of 2^{31} bytes (2GB). UTF-8 encodes each character using between 1 and 6 bytes (typically 1 or 3). In the very worst case, where every character is encoded in 6 bytes, the maximum block length which can be searched would be 357,913,940 characters.

Further Examples

Several of the examples use the following vector as the input document:

```
text
To be or not to be- that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles
```

Replace all upper and lower-case vowels by 'X':

```
('[aeiou]' ⍋R 'X' ⍋ ('IC' 1)) text
TX bX Xr nXt tX bX- thXt Xs thX qXXstXXn:
WhXthXr 'tXs nXblXr Xn thX mXnd tX sXffXr
ThX slXngs Xnd XrrXws Xf XXtrXgXXs fXrtXnX,
Xr tX tXkX Xrms XgXXnst X sXX Xf trXXblXs
```

Replace only the second vowel on each line by '\VOWEL\':

```
('[aeiou]' ⍋R '\VOWEL\' ⍋ ('IC' 1)('ML' -2)) text
To b\VOWEL\ or not to be- that is the question:
Wheth\VOWEL\r 'tis nobler in the mind to suffer
The sl\VOWEL\ngs and arrows of outrageous fortune,
Or t\VOWEL\ take arms against a sea of troubles
```

Case fold each word:

```
('(?<first>\w)(?<remainder>\w*)' ⍋R
'\u<first>\l<remainder>') text
To Be Or Not To Be- That Is The Question:
Whether 'Tis Nobler In The Mind To Suffer
The Slings And Arrows Of Outrageous Fortune,
Or To Take Arms Against A Sea Of Troubles
```

Extract only the lines with characters 'or' (in upper or lower case) on them:

```
↑('or' ⍋S '%' ⍋ ('IC' 1)('ML' 1)) text
To be or not to be- that is the question:
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles
```

Identify which lines contain the word 'or' (in upper or lower case) on them:

```
('\bor\b' ⍋S 2⍋ ('IC' 1)('ML' 1)) text
0 3
```

Note the difference between the characters 'or' (which appear in 'fortune') and the word 'or'.

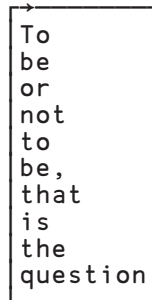
Place every non-space sequence of characters in brackets:

```
('[^\s]+' ⍋R '(&)' ) 'To be or not to be, that is the
question'
(To) (be) (or) (not) (to) (be,) (that) (is) (the) (question)
```


Replace all sequences of one or more spaces by newline. Note that the effect of this is dependent on the input format:

Character vector input results in a single character vector output with embedded newlines:

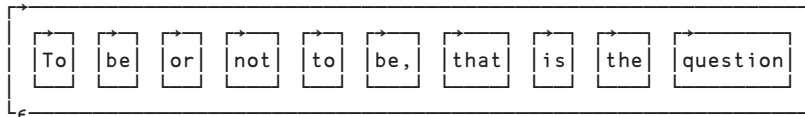
```
]display ('\s+' [R '\r']) 'To be or not to be, that  
is the question'
```



```
To  
be  
or  
not  
to  
be,  
that  
is  
the  
question
```

A vector of two character vectors as input results in a vector of 10 character vectors output:

```
]display ('\s+' [R '\r']) 'To be or not to be,' 'that  
is the question'
```



```
To be or not to be, that is the question
```

Change numerals to their expanded names, using a function:

```
∇r←f a
[1] r←' ',>(⊡a.Match)↓'zero' 'one' 'two' 'three' 'four'
      'five' 'six' 'seven' 'eight' 'nine'
∇
  verbose←('[0-9]' ⍋R f)
  verbose ⌘27×56×87
one three one five four four
```

Swap 'red' and 'blue':

```
('red' 'blue' ⍋R 'blue' 'red') 'red hat blue coat'
blue hat red coat
```

Convert a comma separated values (CSV) file so that

- a) dates in the first field are converted from European format to ISO, and
- b) currency values are converted from Deutsche Marks (DEM) to Euros (DEM 1.95583 to €1).

The currency conversion requires the use of a function. Note the nested use of ⍋R.

Input file:

```
01/03/1980,Widgets,DEM 10.20
02/04/1980,Bolts,DEM 61.75
17/06/1980,Nuts; special rate DEM 17.00,DEM 17.00
18/07/1980,Hammer,DEM 1.25
```

Output file:

```
1980-03-01,Widgets,€ 5.21
1980-04-02,Bolts,€ 31.57
1980-06-17,Nuts; special rate DEM 17.00,€ 8.69
1980-07-18,Hammer,€ 0.63
```

```

▽ ret←f a;d;m;y;v
[1]   □IO←0
[2]   :Select a.PatternNum
[3]   :Case 0
[4]     d m y←{a.Match[a.Offsets[ω+1]+ia.Lengths[ω+1]]}''i3
[5]     ret←y,'-',m,'-',d,',',
[6]   :Else
[7]     v←±a.Block[a.Offsets[1]+ia.Lengths[1]]
[8]     v÷←1.95583
[9]     ret←',€ ',('(\d+\.\d\d)*'□R'\1')⌘v
[10]  :EndSelect

```

▽

```

in ← 'x.csv' □NTIE 0
out ← 'new.csv' □NCREATE 0
dateptn←'(\d{2})/(\d{2})/(\d{4}),'
valptn←',DEM ([0-9.]+'
out (dateptn valptn □R f) in
□nuntie in out

```

Create a simple profanity filter. For the list of objectionable words:

```
profanity←'bleeding' 'heck'
```

first construct a pattern which will match the words:

```
ptn←(('^' '$' '\r\n') □R '\\b(' ')\\b' '|')
□OPT 'Mode' 'D') profanity
```

```
ptn
```

```
\b(bleeding|heck)\b
```

then a function that uses this pattern:

```
sanitise←ptn □R '****' □opt 1
sanitise '"Heck", I said'
"****", I said
```


CHAPTER 5

Reference to Language Enhancements

This Chapter provides new sections for each of the primitive and system functions that have been introduced or enhanced in Version 13.0.

New Primitive Operators, Functions, System Functions & Variables

Identity	$R \leftarrow Y$
Right	$R \leftarrow X \vdash Y$
Same	$R \leftarrow \neg Y$
Left	$R \leftarrow X \dashv Y$
Variant operator	$\{R\} \leftarrow \{X\} (f \square B) Y$ $\{R\} \leftarrow \{X\} (f \square \text{OPT } B) Y$
Decimal Comparison Tolerance	$\square \text{DCT}$
Floating-point Representation	$\square \text{FR}$
Space Indicator	$\square \text{RSI}$

Enhanced Primitive Functions, System Functions & Variables

Drop	$R \leftarrow X \downarrow Y$
Take	$R \leftarrow X \uparrow Y$
Index	$R \leftarrow \{X\} \square Y$
Data Representation	$R \leftarrow \square \text{DR } Y$

Complex Numbers

The following table lists the Primitive Functions & System Functions which have been updated to support Complex Numbers

Add	$R \leftarrow X + Y$
And, Lowest Common Multiple	$R \leftarrow X \wedge Y$
Binomial	$R \leftarrow X ! Y$
Ceiling	$R \leftarrow \lceil Y$
Circular	$R \leftarrow X \circ Y$
Conjugate ³	$R \leftarrow + Y$
Decode	$R \leftarrow X \perp Y$
Divide	$R \leftarrow X \div Y$
Direction ⁴	$R \leftarrow \times Y$
Reciprocal	$R \leftarrow \div Y$
Equal	$R \leftarrow X = Y$
Exponential	$R \leftarrow * Y$
Factorial	$R \leftarrow ! Y$
Floor	$R \leftarrow \lfloor Y$
Logarithm	$R \leftarrow X \oplus Y$
Magnitude	$R \leftarrow Y$
Matrix Divide	$R \leftarrow X \boxed{\div} Y$
Matrix Inverse	$R \leftarrow \boxed{\div} Y$
Multiply	$R \leftarrow X \times Y$
Natural Logarithm	$R \leftarrow \ominus Y$
Negative	$R \leftarrow - Y$
Pi Times	$R \leftarrow \circ Y$
Power	$R \leftarrow X * Y$
Reciprocal	$R \leftarrow \div Y$
Subtract	$R \leftarrow X - Y$

³ Previously known as Identity.

⁴ Previously known as Signum.

Other functions such as Match, Membership, Unique, Union, Intersection and functions that operate on the structure of arrays have also been modified to support complex numbers. However the changes have minimal or no impact on the documentation and are therefore not included in these Release Notes.

The following table lists those functions that do not accept complex numbers in arguments: Note that a number is complex if its has a non-zero imaginary part.

Functions outside the domain of complex numbers

Deal	$R \leftarrow X ? Y$
Grade Down	$R \leftarrow \Psi Y$
Grade Up	$R \leftarrow \Delta Y$
Greater than	$R \leftarrow X > Y$
Greater or Equal	$R \leftarrow X \geq Y$
Less than	$R \leftarrow X < Y$
Less or Equal	$R \leftarrow X \leq Y$
Maximum	$R \leftarrow X \lceil Y$
Minimum	$R \leftarrow X \lfloor Y$
Roll	$R \leftarrow ? Y$

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

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

Examples

```
      1 2 + 3 4
4 6
```

```
      1 2 + 3,=4 5
4 6 7
```

```
      1J1 2J2 + 3J3
4J4 5J5
```

```
      -5+4J4 5J5
-1J4 0J5
```


And, Lowest Common Multiple: **$R \leftarrow X \wedge Y$** **Case 1: X and Y are Boolean**

R is Boolean is determined as follows:

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

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

```

      0 1 0 1 ^ 0 0 1 1
0 0 0 1

```

Case 2: Either or both X and Y are numeric (non-Boolean)R is the lowest common multiple of X and Y. Note that in this case, \square CT is an implicit argument.**Example**

```

      15 1 2 7 ^ 35 1 4 0
105 1 4 0

      2 3 4^0j1 1j2 2j3
2 3J6 8J12

      2j2 2j4^5j5 4j4
10J10 -4J12

```

Binomial: **$R \leftarrow X!Y$**

X and Y may be any numbers except that if Y is a negative integer then X must be a whole number (integer). R is numeric. An element of R is integer if corresponding elements of X and Y are integers. Binomial is defined in terms of the function Factorial for positive integer arguments:

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

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

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

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

Example

```

      1 1.2 1.4 1.6 1.8 2!5
5 6.105689248 7.219424686 8.281104786 9.227916704 10
      2!3j2
1J5

```

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

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

Y must be numeric.

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

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

Examples

```

      ⌈-2.3 0.1 100 3.3
-2 1 100 4
      ⌈1.2j2.5 1.2j-2.5
1J3 1J-2

```

For further explanation, see Floor.

\lceil CT is an implied argument of Ceiling.

Circular:**R←X○Y**

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

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

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

Examples

```

      0 -1 o 1
0 1.570796327

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

      2oPI÷3
0.5

      9 11o3.5J-1.2
3.5 -1.2

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

```

Conjugate:**R←+Y**

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

If Y is real or non-numeric, R is the same array unchanged.

Examples

```

      +3j4
3J-4

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

      3j4++3j4
6

      3j4x+3j4
25

      +A←ι5
1 2 3 4 5

      +□EX'A'
1

```

Decode: **$R \leftarrow X \downarrow Y$**

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

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

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

$$\rho R \leftrightarrow (-1 \downarrow \rho X), 1 \downarrow \rho Y$$

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

This function is also known as Base Value.

Examples

```
60 60↓3 13
193
```

```
0 60↓3 13
193
```

```
60↓3 13
193
```

```
2↓1 0 1 0
10
```

Polynomial Evaluation

If X is a scalar and Y a vector of length n , decode evaluates the polynomial

(Index origin 1)

```
2↓1 2 3 4
26
```

```
3↓1 2 3 4
58
```

```
1j1↓1 2 3 4
5J9
```

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

Examples

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

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

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

Scalar extension may be applied:

```
      2⊥M
0 1 2 3 4 5 6 7
```

Extension along a unit axis may be applied:

```
      +A←2 1ρ2 10
2
10
      A⊥M
0 1 2 3 4 5 6 7
0 1 10 11 100 101 110 111
```

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

Y may be any numeric array.

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

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

Examples

```

      x^-15.3 0 101
-1 0 1

```

```

      x3j4 4j5
0.6j0.8 0.6246950476j0.7808688094

```

```

      {w÷|w}3j4 4j5
0.6j0.8 0.6246950476j0.7808688094

```

```

      |x3j4 4j5
1 1

```

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

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

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

If $\square DIV=1$ and $Y=0$, the result of $X \div Y$ is 0 for all values of X .

Examples

```

      2 0 5÷4 0 2
0.5 1 2.5

```

```

      3j1 2.5 4j5÷2 1j1 .2
1.5j0.5 1.25j^-1.25 20j25

```

```

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

```

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

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

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

1. if $X[I]$ is positive, all but the first $X[I]$ elements of the vectors result.
2. if $X[I]$ is negative, all but the last $X[I]$ elements of the vectors result.

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

Examples

```

BOARD      4↓ 'OVERBOARD '
OVER      -5↓ 'OVERBOARD '
0         ρ10↓ 'OVERBOARD '

ONE
FAT
FLY      M
0      -2↓M
O
F
F

ON      -2  -1↓M
1↓M
FAT
FLY     M3←2 3 4ρ□A

QRST    1 1↓M3
UVWX   -1  -1↓M3
ABCD
EFGH

```


Equal: $R \leftarrow X = Y$

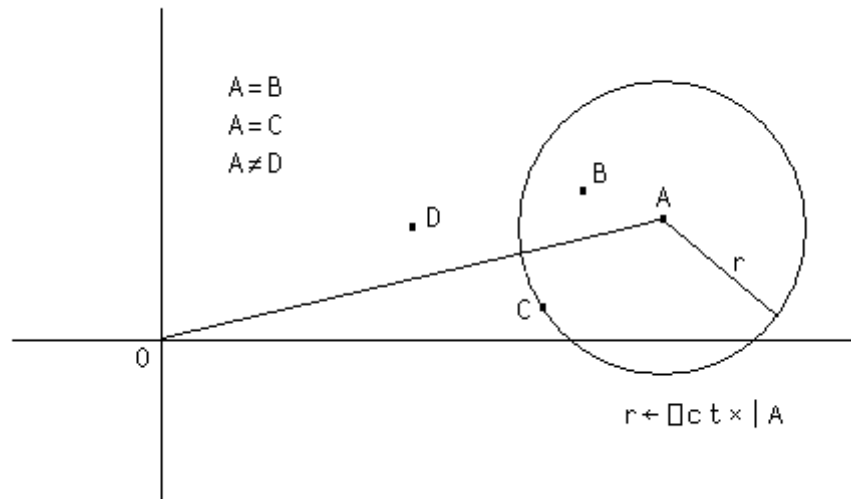
Y may be any array. X may be any array. R is Boolean. ϵ CT is an implicit argument of Equal.

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

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

For real numbers X and Y, X is considered equal to Y if $(|X - Y|)$ is not greater than ϵ CT $\times (|X| + |Y|)$.

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



Examples

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

```
a←2+0j1×⊞CT
a
2J1E-14
1 0 1
a=2j.000000000000001 2j.000000000000001
```

```
'CAT'='FAT'
0 1 1
```

```
'CAT'=1 2 3
0 0 0
```

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

```
⊞CT←1E-10
1=1.000000000000001
1
```

```
1=1.0000001
0
```

Exponential:**R ← * Y**

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

Example

```
*1 0
2.718281828 1
```

```
*0j1 1j2
0.5403023059j0.8414709848 -1.131204384j2.471726672
```

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

Factorial:**R ← ! Y**

Y must be numeric excluding negative integers. R is numeric. R is the product of the first Y integers for positive integer values of Y. For non-integral values of Y, !Y is equivalent to the gamma function of Y+1.

Examples

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

```
!^-1.5 0 1.5 3.3
^-3.544907702 1 1.329340388 8.85534336
```

```
!0j1 1j2
0.4980156681j^-0.1549498283 0.1122942423j0.3236128855
```

Floor: **$R \leftarrow \lfloor Y$**

Y must be numeric.

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

Examples

```
⌊-2.3 0.1 100 3.3
-3 0 100 3
```

```
⌊0.5 + 0.4 0.5 0.6
0 1 1
```

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

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

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

```
▽ fl←CpxFloor cpxs;a;b
[1]  A Complex floor of scalar complex number (a+ib)
[2]  a b←9 110cpxs
[3]  :If 1>(a-⌊a)+b-⌊b
[4]    fl←(⌊a)+0J1×⌊b
[5]  :Else
[6]    :If (a-⌊a)<b-⌊b
[7]      fl←(⌊a)+0J1×1+⌊b
[8]    :Else
[9]      fl←(1+⌊a)+0J1×⌊b
[10]  :EndIf
[11] :EndIf
▽
```

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

ϵ CT is an implicit argument of Floor.

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

Y must be a simple real (non-complex) numeric array. X must be a simple integer scalar or vector. R is a character array displaying the array Y according to the specification X . R has rank $1 \uparrow \rho Y$ and ${}^{-1} \downarrow \rho R$ is ${}^{-1} \downarrow \rho Y$.

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

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

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

Examples

```

      5 0 2 3 6
      1 2 3
      4 5 6

```

```

      4 0 1.1 2 2.5 7
      1 2 4 3

```

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

Example

```

      4 1 1.1 2 2.5 7
      1.1 2.0 4.0 2.5

```

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

Example

```

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

```

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

Example

```

      3⌞2 3ρ10 15.2346 -17.1 2 3 4
10.000 15.235 -17.100
 2.000  3.000  4.000

```

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

Example

```

      3 0 6 2 ⌞ 3 2ρ10.1 15 1001 22.357 101 1110.1
10 15.00
*** 22.36
101*****

```

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

Example

```

      26⌞2*100
1267650600228229_____ . _____

      ρ26⌞2*100
59

      0 20⌞÷3
0.3333333333333333_____

      0 -20⌞÷3
3.3333333333333333_____E-1

```

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

```

      ρ5 2 ⌞ 2 3 4ρ124
2 3 20

```

If any element of **Y** is complex, dyadic **⌞** reports a **DOMAIN ERROR**.

Identity: $R \leftarrow \vdash Y$

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

Example

```

       $\vdash$  'abc' 1 2 3
abc    1 2 3

```

Index: $R \leftarrow \{X\} \square Y$ **Dyadic case**

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

$$(I \ J \ \dots \ \square \ Y) \equiv Y[I;J;\dots]$$

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

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

$$(\rho X \square Y) \equiv \uparrow, / \rho^* X$$

Index is sometimes referred to as *squad indexing*.

Note that index may be used with selective specification.

$\square IO$ is an implicit argument of index.

Examples

```

      []IO←1

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

      []←MAT←10⊥∘∘3 4
11 12 13 14
21 22 23 24
31 32 33 34

      2 1[]MAT
21
      2[]MAT
21 22 23 24

      3(2 1)[]MAT
32 31
      (2 3)1[]MAT
21 31
      (2 3)(,1)[]MAT
21
31
      ρ(2 1ρ1)(3 4ρ2)[]MAT
2 1 3 4
      ρθ θ[]MAT
0 0
      (3(2 1)[]MAT)←0 ⋄ MAT      a Selective assignment.
11 12 13 14
21 22 23 24
0 0 33 34

```

Monadic case

If *Y* is an array, *Y* is returned.

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

```
imc.Item≡[]imc
```


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

Note that the *values* of the index set are obtained or assigned by calls to the corresponding PropertyGet and PropertySet functions. Furthermore, if there is a sequence of primitive functions to the left of the Index function, that operate on the index set itself (functions such as dyadic ρ , \uparrow , \downarrow , \Rightarrow) as opposed to functions that operate on the *values* of the index set (functions such as $+$, \lceil , \lfloor , ρ''), calls to the PropertyGet and PropertySet functions are deferred until the required index set has been completely determined. The full set of functions that cause deferral of calls to the PropertyGet and PropertySet functions is the same as the set of functions that applies to selective specification.

If for example, `CompFile` is an Instance of a Class with a Default Numbered Property, the expression:

$$1\uparrow\phi\lceil\text{CompFile}$$

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

Note that similarly, the expression

$$10000\rho\lceil\text{CompFile}$$

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

Left: **$R \leftarrow X \rightarrow Y$**

X and Y may be any arrays.

The result R is the left argument X.

Example

```

      42 → 'abc' 1 2 3
42

```

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

Examples

```

      → / 1 2 3
1

      mat ← ↑ 'scent' 'canoe' 'arson' 'rouse' 'fleet'
      → / mat  A first row
scent
      → / mat  A first column
scarf

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

```

Similarly, with expansion:

```

      → \ mat
sssss
ccccc
aaaaa
rrrrr
fffff
      → \ mat
scent
scent
scent
scent
scent

```

Logarithm: $R \leftarrow X \otimes Y$

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

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

$$X \otimes Y \leftrightarrow (\ominus Y) \div \ominus X$$

Examples

$$10 \otimes 100 \quad 2$$

$$2 \quad 0.3010299957$$

$$2 \quad 10 \otimes 0J1 \quad 1J2$$

$$0J2.2661800709 \quad 0.34948500217J0.48082857878$$

$$1 \quad 1 \otimes 1$$

$$2 \quad 2 \otimes 1$$

$$0$$

Magnitude: $R \leftarrow | Y$

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

Note that the magnitude of a complex number is defined to be _____

Examples

$$|2 \quad -3.4 \quad 0 \quad -2.7$$

$$2 \quad 3.4 \quad 0 \quad 2.7$$

$$|3j4$$

$$5$$

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

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

R is the result of matrix division of X by Y. That is, the matrix product $Y + . \times R$ is X.

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

The shape of R is $(1 \downarrow \rho Y), 1 \downarrow \rho X$.

Examples

```
⎕PP←5
```

```
B
```

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

```
35 89 79 ⎕ B
```

```
2.14444 8.2111 5.0889
```

```
A
```

```
35 36
89 88
79 75
```

```
A ⎕ B
```

```
2.14444 2.1889
8.2111 7.1222
5.0889 5.5778
```

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

```

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

      P
12.03 8.78 6.01 3.75 -0.31 -2.79

      P#Q
14.941 -2.9609
    
```

Example: linear regression on complex numbers

```

x←j^-50+?2 13 4ρ100
y←(x+.x3 4 5 6) + j^0.0001x^-50+?2 13ρ100
ρx
13 4
ρy
13
y # x
2.99999J0.0000134459 4.00001J^-0.000044302
4.99995J0.0000031282 5.99999J^-0.00000939231
A i.e. y#x recovered the coefficients 3 4 5 6
    
```

Matrix Inverse: **$R \leftarrow \boxed{\ominus} Y$**

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

R is the inverse of Y if Y is a square matrix, or the left inverse of Y if Y is not a square matrix. That is, $R+. \times Y$ is an identity matrix.

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

Examples

```
M
4 -1
2 1
```

```
+A← $\boxed{\ominus}$ M
0.1666666667 0.1666666667
-0.3333333333 0.6666666667
```

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

```
A+. \times M
1 0
0 1
```

```
 $\boxed{\ominus}$ RL←7*5
j←{α←0 ⋄ α+0J1×ω}
x←j/50+?2 5 5ρ100
x
-37J-41 25J015 -5J-09 3J020 -29J041
-46J026 17J-24 17J-46 43J023 -12J-18
1J013 33J025 -47J049 -45J-14 2J-26
17J048 -50J022 -12J025 -44J015 -9J-43
18J013 8J038 43J-23 34J-07 2J026
ρx
5 5
id←{o.=~ιω} ρ identity matrix of order ω
[/,| (id 1↑ρx) - x+. \times x
3.66384E-16
```

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

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

This function is also known as Times.

Example

```

      3 2 1 0 × 2 4 9 6
6 8 9 0

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

```

Natural Logarithm: **$R \leftarrow \odot Y$**

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

Example

```

      *1 2
0 0.6931471806

      *2 2p0j1 1j2 2j3 3j4
0.0000000000J1.5707963268 0.80471895622J1.1071487178
1.2824746787J0.98279372325 1.60943791240J0.927295218

```

Negative: **$R \leftarrow -Y$**

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

Example

```

      -4 2 0 -3 -5
-4 -2 0 3 5

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

```

Pi Times:**R←○Y**

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

Example

```
○0.5 1 2
1.570796327 3.141592654 6.283185307
```

```
○0J1
0J3.1415926536
```

```
*○0J1 π Euler
-1
```


Power: **$R \leftarrow X * Y$**

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

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

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

If X is negative, and Y can be approximated as a rational number of the form $P \div Q$ where P and Q are relatively prime integers, then:

- if Q is even, $X * Y$ gives a DOMAIN ERROR
- if Q is odd and P is even, then $X * Y \leftrightarrow (|X) * Y$
- if Q and P are both odd, then $X * Y \leftrightarrow -(|X) * Y$

If X is negative, and Y cannot be approximated as a rational number, then:

$$X * Y \leftrightarrow -(|X) * Y.$$

Examples

$$4 \text{ } 0.25 \quad 2 * 2^{-2}$$

$$3 \text{ } 8 \quad 9 \text{ } 64 * 0.5$$

$$729 \text{ }^{-27} * 2 \text{ }^3, (1 \text{ }^2 \div 3), 1.2 \quad 52.19591521$$

$$\begin{aligned} & * 2 \text{ } 2 \rho 0 j 1 \text{ } 1 j 2 \text{ } 2 j 3 \text{ }^{-4} j^{-5} \\ & 0.5403023059 J 0.8414709848 \text{ }^{-1.131204384000 J 2.471726672} \\ &^{-7.3151100950 J 1.042743656} \quad 0.005195454155 J 0.01756331074 \end{aligned}$$

$$\text{ }^{-1} \quad * \circ 0 J 1 \text{ } \rho \text{ Euler}$$

Reciprocal:**R←÷Y**

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

$\square \text{DIV}$ is an implicit argument of Reciprocal.

Examples

```

      ÷4 2 5
0.25 0.5 0.2

```

```

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

```

```

      □DIV ← 1

```

```

      ÷0 0.5
0 2

```

Residue:**R←X|Y**

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

For positive arguments, R is the remainder when Y is divided by X. If $X=0$, R is Y. For other argument values, R is $Y - N \times X$ where N is some integer such that R lies between 0 and X, but is not equal to X.

$\square \text{CT}$ is an implicit argument of Residue.

Examples

```

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

```

```

      0.5 | 3.12 -1 -0.6
0.12 0 0.4

```

```

      -1 0 1 | -5.25 0 2.41
-0.25 0 0.41

```

```

      1j2 | 2j3 3j4 5j6
1J1 -1J1 0J1

```

Note that the ASCII pipe (|) may also be interpreted as Residue (|).

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

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

Example

```

      42  $\vdash$  'abc' 1 2 3
abc   1 2 3

```

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

Examples

```

       $\vdash$ /1 2 3
3

```

```

      mat $\leftarrow$  $\vdash$ 'scent' 'canoe' 'arson' 'rouse' 'fleet'
       $\vdash$ /mat  # last row
fleet
       $\vdash$ /mat  # last column
Tenet

```

```

       $\vdash$ /[2]2 3 4p124 # last row from each plane
      9 10 11 12
      21 22 23 24

```

Same: **$R \leftarrow \vdash Y$**

Y may be any array.

The result R is the argument Y .

Examples

```

       $\vdash$ 'abc' 1 2 3
abc   1 2 3

```

Subtract:**R←X-Y**

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

This function is also known as Minus.

Example

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

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

Take: **R←X↑Y**

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

If Y is a scalar, it is treated as a one-element array of shape (ρ, X)ρ1. The length of X must be the same as or less than the rank of Y. If the length of X is less than the rank of Y, the missing elements of X default to the length of the corresponding axis of Y.

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

If more elements are taken than exist on axis I, the extra positions in R are filled with the fill element of Y (⊖∈Y).

Examples

```

      5↑'ABCDEF'
ABCDE

      5↑1 2 3
1 2 3 0 0

      -5↑1 2 3
0 0 1 2 3

      5↑(ι3) (ι4) (ι5)
1 2 3 1 2 3 4 1 2 3 4 5 0 0 0 0 0 0

      M
1 2 3 4
5 6 7 8

      2 3↑M
1 2 3
5 6 7

      -1 -2↑M
7 8

      M3←2 3 4ρA
      1↑M3
ABCD
EFGH
IJKL
      -1↑M3
MNOP
QRST
UVWX
    
```

Variant: **$\{R\} \leftarrow \{X\} (f \text{ ⍷ } B) Y$**

The Variant operator ⍷ specifies the value of an *option* to be used by its left operand function f . An *option* is a named property of a function whose value in some way affects the operation of that function.

For example, the Search and Replace operators include options named **IC** and **Mode** which respectively determine whether or not *case* is ignored and in what manner the input document is processed.

One of the set of options may be designated as the *Principal option* whose value may be set using a short-cut form of syntax as described below. For example, the Principal option for the Search and Replace operators is **IC**.

⍷ and ⍷OPT are synonymous though only the latter is available in the Classic Edition.

In Version 13.0 the Variant operator is used solely to specify options for the ⍷S and ⍷R operators but it is anticipated that its use will become more widespread in later versions.

For the operand function with right argument Y and optional left argument X , the right operand B specifies the values of one or more options that are applicable to that function. B may be a scalar, a 2-element vector, or a vector of 2-element vectors which specifies values for one or more options as follows:

- If B is a 2-element vector and the first element is a character vector, it specifies an option name in the first element and the option value (which may be any suitable array) in the second element.
- If B is a vector of 2-element vectors, each item of B is interpreted as above.
- If B is a scalar (a rank-0 array of any depth), it specifies the value of the Principal option

Option names and their values must be appropriate for the left operand function, otherwise an **OPTION ERROR** (error code 13) will be reported.

The following illustrations and examples apply to functions derived from the Search and Replace operators.

Examples of operand B

The following expression sets the `IC` option to 1, the `Mode` option to 'D' and the `EOL` option to 'LF'.

```
⊠('Mode' 'D')('IC' 1)('EOL' 'LF')
```

The following expression sets just the `EOL` property to 'CR'.

```
⊠'EOL' 'CR'
```

The following expression sets just the `Principal` option (which for the Search and Replace operators is `IC`) to 1.

```
⊠ 1
```

The order in which options are specified is typically irrelevant but if the same option is specified more than once, the rightmost one dominates. The following expression sets the option `IC` to 1:

```
⊠('IC' 0) ('IC' 1)
```

The `Variant` operator generates a derived function $f \boxplus B$ and may be assigned to a name. The derived function is effectively function `f` bound with the option values specified by `B`.

The derived function may itself be used as a left operand to `Variant` to produce a second derived function whose options are further modified by the second application of the operator. The following sets the same options as the first example above:

```
⊠'Mode' 'D'⊠'IC' 1⊠'EOL' 'LF'
```

When the same option is specified more than once in this way, the outermost (rightmost) one dominates. The following expression also sets the option `IC` to 1:

```
⊠'IC' 0⊠'IC' 1
```

Further Examples

The following derived function returns the location of the word 'variant' within its right argument using default values for all the options.

```
f1 ← 'variant' ⍋S 0
f1 'The variant Variant operator'
4
```

It may be modified to perform a case-insensitive search:

```
(f1 ⍋ 1) 'The variant Variant operator'
4 12
```

This modified function may be named:

```
f2 ← f1 ⍋ 1
f2 'The variant Variant operator'
4 12
```

The modified function may itself be modified, in this case to revert to a case sensitive search:

```
f3 ← f2 ⍋ 0
f3 'The variant Variant operator'
4
```

This is equivalent to:

```
(f1 ⍋ 1 ⍋ 0) 'The variant Variant operator'
4
```


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

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

WARNING: Although documentation is provided for I-Beam functions, any service provided using I-Beam should be considered as “experimental” and subject to change – without notice - from one release to the next. Any use of I-Beams in applications should therefore be carefully isolated in cover-functions that can be adjusted if necessary.

A is an integer that specifies the type of operation to be performed as shown in the table below. Y is an array that supplies further information about what is to be done.

X is currently unused.

R is the result of the derived function.

A	Derived Function
200	Syntax Colouring
1111	Number of Threads
1112	Parallel Execution Threshold
1113	Thread Synchronisation Mechanism
2000	Memory Manager Statistics
<u>2010</u>	<u>Update DataTable</u>
<u>2020</u>	<u>Read DataTable</u>
2100	Export to Memory
<u>4000</u>	<u>Fork New Task</u>
<u>4001</u>	<u>Change User</u>
<u>4002</u>	<u>Reap Forked Tasks</u>
<u>4007</u>	<u>Signal Counts</u>

Functions shown underlined are new in Version 13.0.

Update DataTable:**{X}2010IY**

This function performs a *block update* of an instance of the ADO.NET object `System.Data.DataTable`. This object may only be updated using an explicit row-wise loop, which is slow at the APL level. `2010I` implements an *internal* row-wise loop which is much faster on large arrays. Furthermore, the function handles NULL values and the conversion of internal APL data to the appropriate .Net datatype in a more efficient manner than can be otherwise achieved. These 3 factors together mean that the function provides a significant improvement in performance compared to calling the row-wise programming interface directly at the APL level.

Y is a 2, 3 or 4-item array containing `dtRef`, `Data`, `NullValues` and `Rows` as described in the table below.

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

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

Example

First for comparison is shown the type of code that is required to update a `DataTable` by looping,

```

USING←'System' 'System.Data,system.data.dll'
dt←NEW DataTable
ac←{dt.Columns.Add α ω}
'S1' 'S2' 'I1' 'D1' ac String String Int32 DateTime
S1 S2 I1 D1

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

```

```

ar←{(row←dt.NewRow).ItemArray←ω ◊ dt.Rows.Add row}
t←3>⊖ai ◊ ar⌵data ◊ (3>⊖ai)-t
449

```

This result shows that this code can only insert roughly 100 rows per second (3>⊖AI returns elapsed time in milliseconds), because of the need to *loop* on each row and perform a noticeable amount of work each time around the loop.

2010I does all the looping in compiled code:

```

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

```

So in this case, using 2010I achieves something like 10,000 rows per second.

Using ParseFlags

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

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

```

NextYear←⌈"DateTime.Now+{⊖NEW TimeSpan (4↑ω)}"⌵⌵⌵365
data←(⌈"⌵⌵⌵"),(np'odd' 'even'),(10|⌵⌵),NextYear
~2 4↑data
364 even 4 18-01-2011 14:03:29
365 odd 5 19-01-2011 14:03:29
SetDT←2010I
0 0 0 1 SetDT dt data

```

Handling Nulls

If applicable, NullValues is a vector with as many elements as the DataTable has columns, indicating the value that should be converted to System.DBNull as data is written. For example, using the same DataTable as above:

```

t
<null> odd 1 21-01-2010 14:50:19
two even 2 22-01-2010 14:50:19
three odd 99 23-01-2010 14:50:19
dt.Rows.Clear # Clear the contents of dt
SetDT dt t ('<null>' 'even' 99 '')

```

Above, we have declares that the string '`<null>`' should be considered to be a null value in the first column, '`even`' in the second column, and the integer `99` in the third.

Updating Selected Rows

Sometimes, you may have read a very large number of rows from a DataTable, but only want to update a single row, or a very small number of rows. Row indices can be provided as the fourth element of the argument to `2010⍲`. If you are not using `NullValues`, you can just use an empty vector as a placeholder. Continuing from the example above, we could replace the first row in our DataTable using:

```
SetDT←2010⍲
SetDT dt (1 4ρ'one' 'odd' 1 DateTime.Now) ⊖ 0
```

Note

- the values must be provided as a matrix, even if you only want to update a single row,
- row indices are zero origin (the first row has number 0).

Warning

If you are experimenting with writing to a DataTable, note that you should call `dt.Rows.Clear` each time to clear the current contents of the table. Otherwise you will end up with a very large number of rows after a while.

Read DataTable:

`R←{X}2020⍲Y`

This function performs a *block read* from an instance of the ADO.NET object `System.Data.DataTable`. This object may only be read using an explicit row-wise loop, which is slow at the APL level. `2020⍲` implements an *internal* row-wise loop which is much faster on large arrays. Furthermore, the function handles NULL values and the conversion of .Net datatypes to the appropriate internal APL form in a more efficient manner than can be otherwise achieved. These 3 factors together mean that the function provides a significant improvement in performance compared to calling the row-wise programming interface directly at the APL level.

`Y` is a scalar or a 2-item array containing `dtRef`, and `NullValues` as described in the table below.

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

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

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

Example

First for comparison is shown the type of code that is required to read a `DataTable` by looping:

```
t←3>[]AI ◊ data1←↑([]dt.Rows).ItemArray ◊ (3>[]AI)-t
191
```

The above expression turns the `dt.Rows` collection into an array using `[]`, and *mixes* the `ItemArray` properties to produce the result. Although here there is no explicit loop, involved, there is an implicit loop required to reference each item of the collection in succession. This operation performs at about 200 rows/sec.

`2010I` does the looping entirely in compiled code and is significantly faster:

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

ParseFlags Example

In the example shown above, `2020I` created 365 instances of `System.DateTime` objects in the workspace. If we are willing to receive the timestamps in the form of strings, we can read the data almost an order of magnitude faster:

```
t←3>[]AI ◊ data3←0 0 0 1 GetDT dt ◊ (3>[]AI)-t
3
```

The left argument to `2020⍲` allows you to flag columns which should be returned as the `ToStRiNg()` value of each object in the flagged columns. Although the resulting array looks identical to the original, it is not: The fourth column contains character vectors:

```
      ~2 4↑data3
364  even  4  18-01-2011 14:03:29
365  odd   5  19-01-2011 14:03:29
```

Depending on your application, you may need to process the text in the fourth column in some way – but the overall performance will probably still be very much better than it would be if `DateTime` objects were used.

Handling Nulls

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

```
      GetDT←2020⍲
      □←z←GetDT dt
      odd  1  21-01-2010 14:50:19
two      2  22-01-2010 14:50:19
three   odd  23-01-2010 14:50:19

      (1 1⍴z).GetType
System.DBNull System.DBNull System.DBNull
```

However, by supplying a `NullValues` argument to `2020⍲`, we can request that nulls in each column are mapped to a corresponding value of our choice; in this case, `'<null>'`, `'even'`, and `99` respectively.

```
      GetDT dt ('<null>' 'even' 99 '')
<null>  odd  1  21-01-2010 14:50:19
two     even  2  22-01-2010 14:50:19
three   odd  99 23-01-2010 14:50:19
```

Fork New Task: (UNIX only)**R←4000IY**

Y must be is a simple empty vector but is ignored.

This function *forks* the current APL task. This means that it initiates a new separate copy of the APL program, with exactly the same APL execution stack.

Following the execution of this function, there will be two identical APL processes running on the machine, each with the same execution stack and set of APL objects and values. However, none of the external interfaces and resources in the parent process will exist in the newly forked child process.

- The function will return a result in both processes.
- In the parent process, R is the process id of the child (forked) process.
- In the child process, R is a scalar zero.

The following external interfaces and resources that may be present in the parent process are not replicated in the child process:

- Component file ties
- Native file ties
- Mapped file associations
- Auxiliary Processors
- .NET objects
- Edit windows
- Clipboard entries
- GUI objects (all children of ' . ')
- I/O to the current terminal

Note that External Functions established using `⌈NA` are replicated in the child process.

The function will fail with a `DOMAIN ERROR` if there is more than one APL thread running.

The function will fail with a `FILE ERROR 11 Resource temporarily unavailable` if an attempt is made to exceed the maximum number of processes allowed per user.

Change User: (UNIX only)**R←4001Y**

Y is a character vector that specifies a valid UNIX user name. The function changes the *userid* (*uid*) and *groupid* (*gid*) of the process to values that correspond to the specified user name.

Note that it is only possible to change the user name if the current user name is `root` (`uid=0`).

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

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

If the operation fails, the function generates a `FILE ERROR 1 Not Owner` error.

If the argument to `4001I` is other than a non-empty simple character vector, the function generates a `DOMAIN ERROR`.

If the argument is not the name of a valid user the function generates a `FILE ERROR 3 No such process`.

If the argument is the same name as the current effective user, then the function returns that name, but has no effect.

If the argument is a valid name other than the name of the effective user id of the current process, and that effective user id is not `root` the function generates a `FILE ERROR 1 Not owner`.

Reap Forked Tasks: (UNIX only)**R←4002Y**

Under UNIX, when a child process terminates, it signals to its parent that it has terminated and waits for the parent to acknowledge that signal. `4002I` is the mechanism to allow the APL programmer to issue such acknowledgements.

Y must be a simple empty vector but is ignored.

The result R is a matrix containing the list of the newly-terminated processes which have been terminated as a result of receiving the acknowledgement, along with information about each of those processes as described below.

R[; 1] is the process ID (PID) of the terminated child

R[; 2] is `-1` if the child process terminated normally, otherwise it is the signal number which caused the child process to terminate.

R[; 3] is `-1` if the child process terminated as the result of a signal, otherwise it is the exit code of the child process

The remaining 15 columns are the contents of the `rusage` structure returned by the underlying `wait3()` system call. Note that the two `timeval` structs are each returned as a floating point number.

The current `rusage` structure contains:

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss; /* maximum resident set size
*/
    long ru_ixrss; /* integral shared memory
size */
    long ru_idrss; /* integral unshared data
size */
    long ru_isrss; /* integral unshared stack
size */
    long ru_minflt; /* page reclaims */
    long ru_majflt; /* page faults */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* messages sent */
    long ru_msgrcv; /* messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches
*/
    long ru_nivcsw; /* involuntary context
switches */
};
```

4002I may return the PID of an abnormally terminated Auxiliary Processor; APL code should check that the list of processes that have been reaped is a superset of the list of processes that have been started.

Example

```

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

Signal Counts: (UNIX only)**R←4007IY**

Y must be a simple empty vector but is ignored.

The result R is an integer vector of signal counts. The length of the vector is system dependent. On AIX 32-bit it is 63 on AIX 64-bit it is 256 but code should not rely on the length.

Each element is a count of the number of signals that have been generated since the last call to this function, or since the start of the process. R[1] is the number of occurrences of signal 1 (SIGHUP), R[2] the number of occurrences of signal 2, and so forth.

Each time the function is called it zeros the counts; it is therefore inadvisable to call it in more than one APL thread.

Currently, only SIGHUP, SIGINT, SIGQUIT, SIGTERM and SIGWINCH are counted and all other corresponding elements of R are 0.

Decimal Comparison Tolerance:**⍵DCT**

The value of ⍵DCT determines the precision with which two numbers are judged to be equal when the value of ⍵FR is 1287. If ⍵FR is 645, the system uses ⍵CT.

⍵DCT may be assigned any value in the range from 0 to $2.3283064365386962890625E^{-10}$. A value of 0 ensures exact comparison. The value in a clear workspace is $1E^{-28}$.

For further information, see ⍵CT

Examples

```

⍵DCT←1E-10
1.00000000001 1.0000001 = 1
1 0

```

Data Representation (Monadic): **$R \leftarrow \square DR \ Y$**

Monadic $\square DR$ returns the type of its argument Y . The result R is an integer scalar containing one of the following values. Note that the internal representation and data types for character data differ between the Unicode and Classic Editions.

Value	Data Type
11	1 bit Boolean
80	8 bits character
83	8 bits signed integer
160	16 bits character
163	16 bits signed integer
320	32 bits character
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating
1287	128 bits Decimal

Unicode Edition

Value	Data Type
11	1 bit Boolean
82	8 bits character
83	8 bits signed integer
163	16 bits signed integer
323	32 bits signed integer
326	32 bits Pointer
645	64 bits Floating
1287	128 bits Decimal

Classic Edition

Note that types **80**, **160** and **320** and **83** and **163** and **1287** are exclusive to Dyalog APL.

File Create:	{R}←X □FCREATE Y
---------------------	-------------------------

Y must be a simple integer scalar or a 1 or 2 element vector containing the *file tie number* followed by an optional *address size*. .

The *file tie number* must not be the tie number associated with another tied file.

The *address size* is an integer and may be either 32 or 64. A value of 32 causes the internal component addresses to be represented by 32-bit values which allow a maximum file size of 4GB. A value of 64 (the default) causes the internal component addresses to be represented by 64-bit values which allows file sizes up to operating system limits. Note that 32-bit component files will. See below.

Note:

- a 32-bit component file *may not* contain Unicode character data.
- a 64-bit component file may not be accessed by versions of Dyalog APL prior to Version 10.1.0

X must be either

- a) a simple character scalar or vector which specifies the name of the file to be created. See *User Guide* for file naming conventions under UNIX and Windows.
- b) a vector of length 1 or 2 whose items are:
 - i. a simple character scalar or vector as above.
 - ii. an integer scalar specifying the file size limit in bytes.

The newly created file is tied for exclusive use.

The shy result of □FCREATE is the tie number of the new file.

Automatic Tie Number Allocation

A tie number of 0 as argument to a create or tie operation, allocates, and returns as an explicit result, the first (closest to zero) available tie number. This allows you to simplify code. For example:

from:

```
tie←1+[/0,□FNUMS      A With next available number,
file □FCREATE tie   A ... create file.
```

to:

```
tie←file □FCREATE 0 A Create with first available..
```

Examples

```
'..\BUDGET\SALES'   □FCREATE 2      A Windows
'../budget/SALES.85' □FCREATE 2      A UNIX

'COSTS' 200000 □FCREATE 4           A max size 200000

'LARGE' □FCREATE 5 64              A 64-bit file
'SMALL' □FCREATE 6 32              A 32-bit file
```

Important Note

Dyalog intends to withdraw support for 32-bit component files in future releases.

If you have any existing 32-bit component files, or applications which create and/or use them, Dyalog recommends that you prepare for this in the following ways:

- Ensure that Dyalog is not started with the command-line option `-F32`. This option sets the default component file type which is created to 32-bit.
- Ensure that no `□FCREATE` within your applications explicitly specifies that 32-bit files are to be created.
- Make plans to convert any existing 32-bit component files to 64-bit using `□FCOPY`. `□FCOPY` will create a 64-bit copy even if the file being copied is 32-bit.

Note: in order to allow the use of legacy files retrieved from backups etc., Dyalog will continue to provide a means to convert 32-bit files to supported formats for a minimum of 10 years after direct support is withdrawn.

Floating-Point Representation:

`format`

The value of `format` determines the way that floating-point operations are performed.

If `format` is 645, all floating-point calculations are performed using IEEE 754 64-bit floating-point operations and the results of these operations are represented internally using *binary64*⁵ floating-point format.

If `format` is 1287, all floating-point calculations are performed using IEEE 754-2008 128-bit decimal floating-point operations and the results of these operations are represented internally using *decimal128*⁶ format.

Note that when you change `format`, its new value only affects subsequent floating-point operations and results. Existing floating-point values stored in the workspace remain unchanged.

The default value of `format` (its value in a `clear workspace`) is configurable.

`format` 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 `format` *can* vary, the system is *not designed* to allow “seamless” modification during the running of an application and the dynamic alteration of is not recommended. Strange effects may occur. For example, the type of a constant contained in a line of code (in a function or class), will depend on the value of `format` *when the function is fixed*. 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:

```
format←1287
x←1÷3
```

```
format←645
x=1÷3
```

1

The decimal number has 17 more 3’s. Using the tolerance which applies to binary floats (type 645), the numbers are equal. However, the “reverse” experiment yields 0, as tolerance is much narrower in the decimal universe:

```
format←645
x←1÷3
```

```
format←1287
x=1÷3
```

0

⁵ http://en.wikipedia.org/wiki/Double_precision_floating-point_format

⁶ http://en.wikipedia.org/wiki/Decimal128_floating-point_format

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
```

When experimenting with `⎕FR` it is important to note that numeric constants entered into the Session are evaluated (and assigned a data type) before the line is actually executed. This means that constants are evaluated according to the value of `⎕FR` that pertained before the line was entered. For example:

```
⎕FR←645
⎕FR
645
⎕FR←1287 ⋄ ⎕DR 0.1
645
⎕DR 0.1
1287
```

WARNING: The use of `COMPLEX` numbers when `⎕FR` is 1287 is not recommended, because:

- any 128-bit decimal array into which a complex number is inserted or appended will be forced in its entirety into complex representation, potentially losing precision
- all comparisons are done using `⎕DCT` when `⎕FR` is 1287, and this is equivalent to 0 for complex numbers.

Name Association:**{R}←{X}□NA Y**

□NA provides access from APL to compiled functions within a **Dynamic Link Library (DLL)**. A DLL is a collection of functions typically written in C (or C++) each of which may take arguments and return a result.

Instructional examples using □NA can be found in supplied workspace: `QUADNA.DWS`.

The DLL may be part of the standard operating system software, purchased from a third party supplier, or one that you have written yourself.

The right argument `Y` is a character vector that identifies the name and syntax of the function to be associated. The left argument `X` is a character vector that contains the name to be associated with the external function. If the □NA is successful, a function (name class 3) is established in the active workspace with name `X`. If `X` is omitted, the name of the external function itself is used for the association.

The shy result `R` is a character vector containing the name of the external function that was fixed.

For example, `math.dll` might be a library of mathematical functions containing a function `divide`. To associate the APL name `div` with this external function:

```
'div' □NA 'F8 math|divide I4 I4'
```

where `F8` and `I4`, specify the types of the result and arguments expected by `divide`. The association has the effect of establishing a new function: `div` in the workspace, which when called, passes its arguments to `divide` and returns the result.

```
)fns
div      div 10 4
2.5
```

Type Declaration

In a compiled language such as C, the types of arguments and results of functions must be declared explicitly. Typically, these types will be published with the documentation that accompanies the DLL. For example, function `divide` might be declared:

```
double divide(int32_t, int32_t);
```

which means that it expects two long (4-byte) integer arguments and returns a double (8-byte) floating point result. Notice the correspondence between the C declaration and the right argument of `⎕NA`:

```
C:           double    divide      (int32_t,  int32_t);
APL: 'div'  ⎕NA 'F8  math|divide      I4      I4  '
```

It is imperative that care be taken when coding type declarations. A DLL *cannot* check types of data passed from APL. A wrong type declaration will lead to erroneous results or may even cause the workspace to become corrupted and crash.

The full syntax for the right argument of `⎕NA` is:

[result] library|function [arg1] [arg2] ...

Note that functions associated with DLLs are never dyadic. All arguments are passed as items of a (possibly nested) vector on the right of the function.

Locating the DLL

The DLL may be specified using a full pathname, file extension, and function type.

Pathname: APL uses the `LoadLibrary()` system function under Windows and `dlopen()` under UNIX and LINUX to load the DLL. If a full or relative pathname is omitted, these functions search standard operating system directories in a particular order. For further details, see the operating system documentation about these functions.

Alternatively, a full or relative pathname may be supplied in the usual way:

```
⎕NA '... c:\mydir\mydll|foo ...'
```

Errors: If the specified DLL (or a dependent DLL) fails to load it will generate:

```
FILE ERROR 1 No such file or directory
```

If the DLL loads successfully, but the specified library function is not accessible, it will generate:

```
VALUE ERROR
```

File Extension: Under Windows, if the file extension is omitted, **.dll** is assumed. Note that some DLLs are in fact **.exe** files, and in this case the extension must be specified explicitly:

```
␣NA'... mydll.exe|foo ...'
```

Example

```
␣NA'... mydll.exe.P32|foo ...' ␣ 32 bit Pascal
```

Call by Ordinal Number

Under Windows, a DLL may associate an *ordinal number* with any of its functions. This number may then be used to call the function as an alternative to calling it by name. Using `␣NA` to call by ordinal number uses the same syntax but with the function name replaced with its ordinal number. For example:

```
␣NA'... mydll|57 ...'
```

Multi-Threading

Appending the `'&'` character to the function name causes the external function to be run in its own system thread. For example:

```
␣NA'... mydll|foo& ...'
```

This means that other APL threads can run concurrently with the one that is calling the `␣NA` function.

Data Type Coding Scheme

The type coding scheme introduced above is of the form:

[direction] [special] type [width] [array]

The options are summarised in the following table and their functions detailed below.

Description	Symbol	Meaning
Direction	<	Pointer to array <i>input</i> to DLL function.
	>	Pointer to array <i>output</i> from DLL function
	=	Pointer to input/output array.
Special	0	Null-terminated string.
	#	Byte-counted string
Type	I	int
	U	unsigned int
	C	char
	T	Classic Edition char: translated to/from ANSI Unicode Edition char
	F	float
	D	decimal
	J	complex
	P	uintptr-t (equivalent to U4 on 32-bit Versions and U8 on 64-bit Versions)
	A	APL array
	Z	APL array with header (as passed to a TCP/IP socket)
	PP	Pocket pointer This provides support for direct access to data in the workspace.
Width	1	1-byte
	2	2-byte
	4	4-byte
	8	8-byte
	16	16-byte (128-bit)
Array	[n]	Array of length <i>n</i> elements
	[]	Array, length determined at call-time
Structure	{ . . . }	Structure.

In the Classic Edition, C specifies untranslated character, whereas T specifies that the character data will be translated to/from `AV`.

In the Unicode Edition, C and T are identical (no translation of character data is performed) except that for C the default width is 1 and for T the default width is "wide" (2 bytes under Windows, 4 bytes under UNIX).

The use of T with default width is recommended to ensure portability between Editions.

Direction

C functions accept data arguments either by *value* or by *address*. This distinction is indicated by the presence of a '*' or '[' in the argument declaration:

```
int num1;           // value of num1 passed.
int *num2;          // Address of num2 passed.
int num3[];         // Address of num3 passed.
```

An argument (or result) of an external function of type pointer, must be matched in the `NA` call by a declaration starting with one of the characters: <, >, or =.

In C, when an address is passed, the corresponding value can be used as either an *input* or an *output* variable. An output variable means that the C function overwrites values at the supplied address. Because APL is a call-by-value language, and doesn't have pointer types, we accommodate this mechanism by distinguishing output variables, and having them returned explicitly as part of the result of the call.

This means that where the C function indicates a *pointer type*, we must code this as starting with one of the characters: <, > or =.

- < indicates that the address of the argument will be used by C as an input variable and values at the address will *not* be over-written.
- > indicates that C will use the address as an output variable. In this case, APL must allocate an output array over which C can write values. After the call, this array will be included in the nested result of the call to the external function.
- = indicates that C will use the address for both input and output. In this case, APL duplicates the argument array into an output buffer whose address is passed to the external function. As in the case of an output only array, the newly modified copy will be included in the nested result of the call to the external function.

Examples

- <I2 Pointer to 2-byte integer - *input* to external function
- >C Pointer to character *output* from external function.
- =T Pointer to character *input* to and *output* from function.
- =A Pointer to APL array *modified* by function.

Special

In C it is common to represent character strings as *null-terminated* or *byte counted* arrays. These special data types are indicated by inserting the symbol **0** (null-terminated) or **#** (byte counted) between the direction indicator (<, >, =) and the type (T or C) specification. For example, a pointer to a null-terminated input character string is coded as <0T[], and an output one coded as >0T[].

Note that while appending the array specifier '[]' is formally correct, because the presence of the special qualifier (0 or #) *implies* an array, the '[]' may be omitted: <0T, >0T, =#C, etc.

Note also that the 0 and # specifiers may be used with data of all types (excluding A, Z and PP) and widths. For example, in the Classic Edition, <0U2 may be useful for dealing with Unicode.

Type

The data type of the argument is represented by one of the symbols **i**, **u**, **c**, **t**, **f**, **a**, which may be specified in lower or upper case:

	Type	Description
I	Integer	The value is interpreted as a 2s complement signed integer.
U	Unsigned integer	The value is interpreted as an unsigned integer.
C	Character	<p>The value is interpreted as a character.</p> <p>In the Unicode Edition, the value maps directly onto a Unicode code point.</p> <p>In the Classic Edition, the value is interpreted as an index into $\square AV$. This means that $\square AV$ positions map onto corresponding ANSI positions.</p> <p>For example, with $\square IO=0$: $\square AV[35] = 's'$, maps to $ANSI[35] = '$</p>

	Type	Description
T	Translated character	<p>The value is interpreted as a character.</p> <p>In the Unicode Edition, the value maps directly onto a Unicode code point.</p> <p>In the Classic Edition, the value is <i>translated</i> using standard Dyalog <code>⎕AV</code> to ANSI translation. This means that <code>⎕AV</code> characters map onto corresponding ANSI characters.</p> <p>For example, with <code>⎕IO=0</code>:</p> <p><code>⎕AV[35] = 's'</code>, maps to <code>ANSI[115] = 's'</code>.</p>
F	Float	The value is interpreted as an IEEE 754-2008 binary64 floating point number.
D	Decimal	The value is interpreted as an IEEE 754-2008 decimal128 floating point number (DPD format).
J	Complex	
P	uintptr-t	This is equivalent to U4 on 32-bit versions and U8 on 64-bit Versions.
A	APL array	A pointer to the whole array (including header information) is passed. This type is used to communicate with DLL functions which have been written specifically to work with Dyalog APL. See the <i>User Guide</i> section on Writing Auxiliary Processors. Note that type A is always passed as a pointer, so is of the form <code><A</code> , <code>=A</code> or <code>>A</code> .
Z	APL array with header	This is the same format as is used to transmit APL arrays over TCP/IP Sockets.
PP	Pocket Pointer	Provides direct access to data in the workspace.

Width

The type specifier may be followed by the width of the value in bytes. For example:

I4 4-byte signed integer.
U2 2-byte unsigned integer.
F8 8-byte floating point number.
F4 4-byte floating point number.
D16 16-byte decimal floating-point number

Type	Possible values for Width	Default value for Width
I	1, 2, 4, 8	4
U	1, 2, 4, 8	4
C	1,2,4	1
T	1,2,4	wide character(see below)
F	4, 8	8
D	16	16
J	16	16
P	Not applicable	
A	Not applicable	
Z	Not applicable	
PP	Not applicable	

In the Unicode Edition, the default width is the width of a *wide character* according to the convention of the host operating system. This translates to T2 under Windows and T4 under UNIX or Linux.

Note that 32-bit versions can support 64-bit integer *arguments*, but not 64-bit integer *results*.

Examples

I2 16-bit integer
<I4 Pointer to input 4-byte integer
U Default width unsigned integer.
=F4 Pointer to input/output 4-byte floating point number.

Arrays

Arrays are specified by following the basic data type with `[n]` or `[]`, where `n` indicates the number of elements in the array. In the C declaration, the number of elements in an array may be specified explicitly at compile time, or determined dynamically at runtime. In the latter case, the size of the array is often passed along with the array, in a separate argument. In this case, `n`, the number of elements is omitted from the specification. Note that C deals only in scalars and rank 1 (vector) arrays.

```
int vec[10];           // explicit vector length.
unsigned size, list[]; // undetermined length.
```

could be coded as:

```
I[10] vector of 10 ints.
U U[] unsigned integer followed by an array of unsigned integers.
```

Confusion sometimes arises over a difference in the declaration syntax between C and `⊠NA`. In C, an argument declaration may be given to receive a pointer to either a single scalar item, or to the first element of an array. This is because in C, the address of an array is deemed to be the address of its first element.

```
void foo (char *string);

char ch = 'a', ptr = "abc";

foo(&ch);           // call with address of scalar.
foo(ptr);           // call with address of array.
```

However, from APL's point of view, these two cases are distinct and if the function is to be called with the address of (pointer to) a *scalar*, it must be declared: `'<T'`. Otherwise, to be called with the address of an *array*, it must be declared: `'<T[]'`. Note that it is perfectly acceptable in such circumstances to define more than one name association to the same DLL function specifying different argument types:

```
'FooScalar'⊠NA'mydll|foo <T'   ⋄ FooScalar'a'
'FooVector'⊠NA'mydll|foo <T[]' ⋄ FooVector'abc'
```

Structures

Arbitrary data structures, which are akin to nested arrays, are specified using the symbols `{}`. For example, the code `{F8 I2}` indicates a structure comprised of an 8-byte *float* followed by a 2-byte *int*. Furthermore, the code `<{F8 I2}[3]` means an input pointer to an array of 3 such structures.

For example, this structure might be defined in C thus:

```
typedef struct
{
    double f;
    short i;
} mystruct;
```

A function defined to receive a count followed by an *input* pointer to an array of such structures:

```
void foo(unsigned count, mystruct *str);
```

An appropriate `⌈NA` declaration would be:

```
⌈NA 'mydll.foo U <{F8 I2}[]'
```

A call on the function with two arguments - a count followed by a vector of structures:

```
foo 4, c(1.4 3)(5.9 1)(6.5 2)(0 0)
```

Notice that for the above call, APL converts the two Boolean `(0 0)` elements to an 8-byte float and a 2-byte int, respectively.

Specifying Pointers Explicitly

⊞NA syntax enables APL to pass arguments to DLL functions by *value* or *address* as appropriate. For example if a function requires an integer followed by a *pointer* to an integer:

```
void fun(int valu, int *addr);
```

You might declare and call it:

```
⊞NA 'mydll|fun I <I' ⋄ fun 42 42
```

The interpreter passes the *value* of the first argument and the *address* of the second one.

Two common cases occur where it is necessary to pass a pointer explicitly. The first is if the DLL function requires a *null pointer*, and the second is where you want to pass on a pointer which itself is a result from a DLL function.

In both cases, the pointer argument should be coded as P. This causes APL to pass the pointer unchanged, *by value*, to the DLL function.

In the previous example, to pass a null pointer, (or one returned from another DLL function), you must code a separate ⊞NA definition.

```
'fun_null'⊞NA 'mydll|fun I P' ⋄ fun_null 42 0
```

Now APL passes the *value* of the second argument (in this case 0 - the null pointer), rather than its address.

Note that by using P, which is 4-byte for 32-bit processes and 8-byte for 64-bit processes, you will ensure that the code will run unchanged under both 32-bit and 63-bit Versions of Dyalog APL.

Using a Function

A DLL function may or may not return a result, and may take zero or more arguments. This syntax is reflected in the coding of the right argument of `⌈NA`. Notice that the corresponding associated APL function is niladic or monadic (never dyadic), and that it *always* returns a vector result - a null one if there is no output from the function. See Result Vector section below. Examples of the various combinations are:

DLL function Non-result-returning:

```
⌈NA 'mydll|fn1'           A Niladic
⌈NA 'mydll|fn2 <0T'      A Monadic - 1-element arg
⌈NA 'mydll|fn3 =0T <0T' A Monadic - 2-element arg
```

DLL function Result-returning:

```
⌈NA 'I4 mydll|fn4'       A Niladic
⌈NA 'I4 mydll|fn5 F8'    A Monadic - 1-element arg
⌈NA 'I4 mydll|fn6 >I4[] <0T' A Monadic - 2-element arg
```

When the external function is called, the number of elements in the argument must match the number defined in the `⌈NA` definition. Using the example functions defined above:

```
fn1           A Niladic Function.
fn2, <'Single String' A 1-element arg
fn3 'This' 'That'   A 2-element arg
```

Note in the second example, that you must enclose the argument string to produce a single item (nested) array in order to match the declaration. Dyalog converts the type of a numeric argument if necessary, so for example in `fn5` defined above, a Boolean value would be converted to double floating point (F8) prior to being passed to the DLL function.

Pointer Arguments

When passing pointer arguments there are three cases to consider.

< **Input pointer:** In this case you must supply the data array itself as argument to the function. A pointer to its first element is then passed to the DLL function.

```
fn2 c'hello'
```

> **Output pointer:** Here, you must supply the **number of elements** that the output will need in order for APL to allocate memory to accommodate the resulting array.

```
fn6 10 'world'  # 1st arg needs space for 10 ints.
```

Note that if you were to reserve fewer elements than the DLL function actually used, the DLL function would write beyond the end of the reserved array and may cause the interpreter to crash with a System Error (syserr 999 on Windows or SIGSEGV on Unix).

= **Input/Output:** As with the input-only case, a pointer to the first element of the argument is passed to the DLL function. The DLL function then overwrites some or all of the elements of the array, and the new value is passed back as part of the result of the call. As with the output pointer case, if the input array were too short, so that the DLL wrote beyond the end of the array, the interpreter would almost certainly crash.

```
fn3 '.....' 'hello'
```

Result Vector

In APL, a function cannot overwrite its arguments. This means that any output from a DLL function must be returned as part of the explicit result, and this includes output via 'output' or 'input/output' pointer arguments.

The general form of the result from calling a DLL function is a nested vector. The first item of the result is the defined explicit result of the external function, and subsequent items are implicit results from output, or input/output pointer arguments.

The length of the result vector is therefore: 1 (if the function was declared to return an explicit result) + the number of output or input/output arguments.

⊞NA Declaration	Result	Output Arguments	Result Length
mydll fn1	0		0
mydll fn2 <OT	0	0	0
mydll fn3 =OT <OT	0	1 0	1
I4 mydll fn4	1		1
I4 mydll fn5 F8	1	0	1
I4 mydll fn6 >I4[] <OT	1	1 0	2

As a convenience, if the result would otherwise be a 1-item vector, it is disclosed. Using the third example above:

```
ρfn3 '.....' 'abc'
5
```

fn3 has no explicit result; its first argument is input/output pointer; and its second argument is input pointer. Therefore as the length of the result would be 1, it has been disclosed.

ANSI /Unicode Versions of Library Calls

Under Windows, most library functions that take character arguments, or return character results have two forms: one Unicode (Wide) and one ANSI. For example, a function such as `MessageBox()`, has two forms `MessageBoxA()` and `MessageBoxW()`. The `A` stands for ANSI (1-byte) characters, and the `W` for wide (2-byte Unicode) characters.

It is essential that you associate the form of the library function that is appropriate for the Dialog Edition you are using, i.e. `MessageBoxA()` for the Classic Edition, but `MessageBoxW()` for the Unicode Edition.

To simplify writing portable code for both Editions, you may specify the character `*` instead of `A` or `W` at the end of a function name. This will be replaced by `A` in the Classic Edition and `W` in the Unicode Edition.

The default name of the associated function (if no left argument is given to `□NA`), will be without the trailing letter (`MessageBox`).

Type Definitions (typedefs)

The C language encourages the assignment of defined names to primitive and complex data types using its `#define` and `typedef` mechanisms. Using such abstractions enables the C programmer to write code that will be portable across many operating systems and hardware platforms.

Windows software uses many such names and Microsoft documentation will normally refer to the type of function arguments using defined names such as `HANDLE` or `LPSTR` rather than their equivalent C primitive types: `int` or `char*`.

It is beyond the scope of this manual to list *all* the Microsoft definitions and their C primitive equivalents, and indeed, DLLs from sources other than Microsoft may well employ their own distinct naming conventions.

In general, you should consult the documentation that accompanies the DLL in order to convert typedefs to primitive C types and thence to `□NA` declarations. The documentation may well refer you to the 'include' files which are part of the Software Development Kit, and in which the types are defined.

The following table of some commonly encountered Windows typedefs and their `□NA` equivalents might prove useful.

Windows typedef	ANA equivalent
HWND	P
HANDLE	P
GLOBALHANDLE	P
LOCALHANDLE	P
DWORD	U4
WORD	U2
BYTE	U1
LPSTR	=0T[] (note 1)
LPCSTR	<0T[] (note 2)
WPARAM	U
LPARAM	U4
LRESULT	I4
BOOL	I
UINT	U
ULONG	U4
ATOM	U2
HDC	P
HBITMAP	P
HBRUSH	P
HFONT	P
HICON	P
HMENU	P
HPALETTE	P
HMETAFILE	P
HMODULE	P
HINSTANCE	P
COLORREF	{U1[4]}
POINT	{I I}
POINTS	{I2 I2}
RECT	{I I I I}
CHAR	T or C

Notes

1. LPSTR is a pointer to a null-terminated string. The definition does not indicate whether this is input or output, so the safest coding would be =0T[] (providing the vector you supply for input is long enough to accommodate the result). You may be able to improve simplicity or performance if the documentation indicates that the pointer is ‘input only’ (<0T[]) or ‘output only’ (>0T[]). See **Direction** above.
2. LPCSTR is a pointer to a *constant* null-terminated string and therefore coding <0T[] is safe.
3. Note that the use of type T with default width ensures portability of code between Classic and Unicode Editions. In the Classic Edition, T (with no width specifier) implies 1-byte characters which are translated between \square AV and ASCII, while In the Unicode Edition, T (with no width specifier) implies 2-byte (Unicode) characters.

Dyalog32.dll or Dyalog64.dll

Included with Dyalog APL are utility DLLs called dyalog32.dll and dyalog64.dll. These DLLs contain two functions: MEMCPY and STRNCPY.

MEMCPY

MEMCPY is an extremely versatile function used for moving arbitrary data between memory buffers.

Its C definition is:

```
void *MEMCPY(           // copy memory
    void *to,           // target address
    void *fm,           // source address
    size_t size         // number of bytes to copy
);
```

MEMCPY copies *size* bytes starting from source address *fm*, to destination address *to*. The source and destination areas should not overlap; if they do the behaviour is undefined and the result is the first argument.

MEMCPY’s versatility stems from being able to associate to it using many different type declarations.

Example

Suppose a global buffer (at address: `addr`) contains (`numb`) double floating point numbers. To copy these to an APL array, we could define the association:

```
'doubles' ⍵NA 'dyalog32|MEMCPY >F8[] I4 U4'  
doubles numb addr (numb×8)
```

Notice that:

As the first argument to `doubles` is an output argument, we must supply the number of elements to reserve for the output data.

`MEMCPY` is defined to take the number of *bytes* to copy, so we must multiply the number of elements by the element size in bytes.

Example

Suppose that a database application requires that we construct a record in global memory prior to writing it to file. The record structure might look like this:

```
typedef struct {  
    int empno;           // employee number.  
    float salary;       // salary.  
    char name[20];      // name.  
} person;
```

Then, having previously allocated memory (`addr`) to receive the record, we can define:

```
'prec' ⍵NA 'dyalog32|MEMCPY I4 <{P F4 T[20]} U4'  
prec addr(99 12345.60 'Charlie Brown'  
) (4+4+20)
```

STRNCPY

`STRNCPY` is used to copy null-terminated strings between memory buffers.

Its C definition is:

```
void *STRNCPY(  
    char *to,           // copy null-terminated string  
    char *fm,          // target address  
    char *fm,          // source address  
    size_t size        // MAX number of chars to copy  
);
```

`STRNCPY` copies a maximum of `size` characters from the null-terminated source string at address `fm`, to the destination address `to`. If the source and destination strings overlap, the result is the first argument.

If the source string is shorter than `size`, null characters are appended to the destination string.

If the source string (including its terminating null) is longer than `size`, only `size` characters are copied and the resulting destination string is not null-terminated

Example

Suppose that a database application returns a pointer (`addr`) to a structure that contains two pointers to (max 20-char) null-terminated strings.

```
typedef struct {      // null-terminated strings:
    char *first;     // first name (max 19 chars + 1 null).
    char *last;      // last name. (max 19 chars + 1 null).
} name;
```

To copy the names *from* the structure:

```
'get' NA'dialog32|STRNCPY >0T[] P U4'
get 20 addr 20
Charlie
get 20 (addr+4) 20
Brown
```

Note that on a 64-bit Version, `FR` will need to be `1287` for the addition to be reliable.

To copy data *from* the workspace *into* an already allocated (`new`) structure:

```
'put' NA'dialog32|STRNCPY I4 <0T[] U4'
put new 'Bo' 20
put (new+4) 'Peep' 20
```

Notice in this example that you must ensure that names no longer than 19 characters are passed to `put`. More than 19 characters would not leave `STRNCPY` enough space to include the trailing null, which would probably cause the application to fail.

Examples

The following examples all use functions from the Microsoft Windows user32.dll.

This DLL should be located in a standard Windows directory, so you should not normally need to give the full path name of the library. However if trying these examples results in the error message 'FILE ERROR 1 No such file or directory', you must locate the DLL and supply the full path name (and possibly extension).

Example 1

The Windows function "GetCaretBlinkTime" retrieves the caret blink rate. It takes no arguments and returns an unsigned *int* and is declared as follows:

```
UINT GetCaretBlinkTime(void);
```

The following statements would provide access to this routine through an APL function of the same name.

```
      □NA 'U user32|GetCaretBlinkTime'  
      GetCaretBlinkTime  
530
```

The following statement would achieve the same thing, but using an APL function called **BLINK**.

```
      'BLINK' □NA 'U user32|GetCaretBlinkTime'  
      BLINK  
530
```

Example 2

The Windows function "SetCaretBlinkTime" sets the caret blink rate. It takes a single unsigned *int* argument, does not return a result and is declared as follows:

```
void SetCaretBlinkTime(UINT);
```

The following statements would provide access to this routine through an APL function of the same name:

```
      □NA 'user32|SetCaretBlinkTime U'  
      SetCaretBlinkTime 1000
```

Example 3

The Windows function "MessageBox" displays a standard dialog box on the screen and awaits a response from the user. It takes 4 arguments. The first is the window handle for the window that owns the message box. This is declared as an unsigned *int*. The second and third arguments are both pointers to null-terminated strings containing the message to be displayed in the Message Box and the caption to be used in the window title bar. The 4th argument is an unsigned *int* that specifies the Message Box type. The result is an *int* which indicates which of the buttons in the message box the user has pressed. The function is declared as follows:

```
int MessageBox(HWND, LPCSTR, LPCSTR, UINT);
```

The following statements provide access to this routine through an APL function of the same name. Note that the 2nd and 3rd arguments are both coded as input pointers to type T null-terminated character arrays which ensures portability between Editions.

```
⊞NA 'I user32|MessageBox* P <0T <0T U'
```

The following statement displays a Message Box with a stop sign icon together with 2 push buttons labelled OK and Cancel (this is specified by the value 19).

```
MessageBox 0 'Message' 'Title' 19
```

The function works equally well in the Unicode Edition because the <0T specification is portable.

```
MessageBox 0 'Το Μήνυμα' 'Ο Τίτλος' 19
```

Note that a simpler, portable (and safer) method for displaying a Message Box is to use Dyalog APL's primitive `MsgBox` object.

Example 4

The Windows function "FindWindow" obtains the window handle of a window which has a given character string in its title bar. The function takes two arguments. The first is a pointer to a null-terminated character string that specifies the window's class name. However, if you are not interested in the class name, this argument should be a NULL pointer. The second is a pointer to a character string that specifies the title that identifies the window in question. This is an example of a case described above where two instances of the function must be defined to cater for the two different types of argument. However, in practice this function is most often used without specifying the class name. The function is declared as follows:

```
HWND FindWindow(LPCSTR, LPCSTR);
```

The following statement associates the APL function **FW** with the second variant of the FindWindow call, where the class name is specified as a NULL pointer. To indicate that APL is to pass the *value* of the NULL pointer, rather than its address, we need to code this argument as **I4**.

```
'FW' ⍋NA 'P user32|FindWindow* I4 <0T'
```

To obtain the handle of the window entitled "CLEAR WS - Dyalog APL/W":

```
⍋←HNDL←FW 0 'CLEAR WS - Dyalog APL/W'
59245156
```

Example 5

The Windows function "GetWindowText" retrieves the caption displayed in a window's title bar. It takes 3 arguments. The first is an unsigned *int* containing the window handle. The second is a pointer to a buffer to receive the caption as a null-terminated character string. This is an example of an output array. The third argument is an *int* which specifies the maximum number of characters to be copied into the output buffer. The function returns an *int* containing the actual number of characters copied into the buffer and is declared as follows:

```
int GetWindowText(HWND, LPSTR, int);
```

The following associates the "GetWindowText" DLL function with an APL function of the same name. Note that the second argument is coded as ">0T" indicating that it is a pointer to a character output array.

```
⍋NA 'I user32|GetWindowText* P >0T I'
```

Now change the Session caption using **)WSID** :

```
)WSID MYWS
was CLEAR WS
```

Then retrieve the new caption (max length 255) using window handle **HNDL** from the previous example:

```
]display GetWindowText HNDL 255 255
```

```
→-----
| 19 |MYWS - Dyalog APL/W|
←-----
```

There are three points to note. Firstly, the number 255 is supplied as the second argument. This instructs APL to allocate a buffer large enough for a 255-element character vector into which the DLL routine will write. Secondly, the result of the APL function is a nested vector of 2 elements. The first element is the result of the DLL function. The second element is the output character array.

Finally, notice that although we reserved space for 255 elements, the result reflects the length of the actual text (19).


An alternative way of coding and using this function is to treat the second argument as an input/output array.

e.g.

```

⊞NA 'I User32|GetWindowText* P =0T I'
]display GetWindowText HNDL (255ρ' ') 255

```



In this case, the second argument is coded as =0T, so when the function is called an array of the appropriate size must be supplied. This method uses more space in the workspace, although for small arrays (as in this case) the real impact of doing so is negligible.

Example 6

The function "GetCharWidth" returns the width of each character in a given range. Its first argument is a device context (handle). Its second and third arguments specify font positions (start and end). The third argument is the resulting integer vector that contains the character widths (this is an example of an output array). The function returns a Boolean value to indicate success or failure. The function is defined as follows. Note that this function is provided in the library: gdi32.dll.

```

BOOL GetCharWidth(HDC, UINT, UINT, int FAR*);

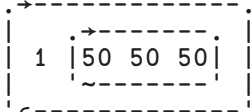
```

The following statements provide access to this routine through an APL function of the same name:

```

      ⍎NA 'U4 gdi32|GetCharWidth* P U U >I[]'
      'P'⍎WC'Printer'
      ]display GetCharWidth ('P' ⍎WG 'Handle') 65 67 3

```



Note: 'P'⍎WG'Handle' returns a handle This is represented as a number. The number will be in the range (0 - 2*32] on a 32-bit Version and (0 - 2*64] on a 64-bit Version. These can be passed to a P type parameter. Older Versions used a 32-bit signed integer.

Example 7

The following example from the supplied workspace: QUADNA.DWS illustrates several techniques which are important in advanced ⍎NA programming. Function DLLVersion returns the major and minor version number for a given DLL.

In advanced DLL programming, it is often necessary to administer memory outside APL's workspace. In general, the procedure for such use is:

1. Allocate global memory.
2. Lock the memory.
3. Copy any DLL input information from workspace into memory.
4. Call the DLL function.
5. Copy any DLL output information from memory to workspace.
6. Unlock the memory.
7. Free the memory.

Notice that steps 1 and 7 and steps 2 and 6 complement each other. That is, if you allocate global system memory, you must free it after you have finished using it. If you continue to use global memory without freeing it, your system will gradually run out of resources. Similarly, if you lock memory (which you must do before using it), then you should unlock it before freeing it. Although on some versions of Windows, freeing the memory will include unlocking it, in the interests of good style, maintaining the symmetry is probably a good thing.


```

▽ version←DllVersion file;Alloc;Free;Lock;Unlock;Size
;Info;Value;Copy;size;hndl;addr;buff;ok
[1]
[2] 'Alloc'□NA'P kernel32|GlobalAlloc U4 U4'
[3] 'Free'□NA'P kernel32|GlobalFree P'
[4] 'Lock'□NA'P kernel32|GlobalLock P'
[5] 'Unlock'□NA'U4 kernel32|GlobalUnlock P'
[6]
[7] 'Size'□NA'U4 version|GetFileVersionInfoSize* <OT >U4'
[8] 'Info'□NA'U4 version|GetFileVersionInfo*<OT U4 U4 P'
[9] 'Value'□NA'U4 version|VerQueryValue* P <OT >P >U4'
[10]
[11] 'Copy'□NA'dialog64|MEMCPY >U4[] P P'
[12]
[13] :If xsize↔Size file 0 A Size of info
[14] :AndIf xhndl←Alloc 0 size A Alloc memory
[15] :If xaddr←Lock hndl A Lock memory
[16] :If xInfo file 0 size addr A Version info
[17] ok buff size←Value addr'\ ' 0 0 A Version value
[18] :If ok
[19] buff←Copy(size÷4)buff size A Copy info
[20] version←(2/2*16)τ=2↓buff A Split version
[21] :EndIf
[22] :EndIf
[23] ok←Unlock hndl A Unlock memory
[24] :EndIf
[25] ok←Free hndl A Free memory
[26] :EndIf
▽

```

Lines [2-11] associate APL function names with the DLL functions that will be used.

Lines [2-5] associate functions to administer global memory.

Lines [7-9] associate functions to extract version information from a DLL.

Line[11] associates `Copy` with `MEMCPY` function from `diallog64.dll`.

Lines [13-26] call the DLL functions.

Line [13] requests the size of buffer required to receive version information for the DLL. A size of 0 will be returned if the DLL does not contain version information.

Notice that care is taken to balance memory allocation and release:

On line [14], the `:If` clause is taken only if the global memory allocation is successful, in which case (and only then) a corresponding `Free` is called on line [25].

`Unlock` on line[23] is called if and only if the call to `Lock` on line [15] succeeds.

A result is returned from the function *only* if all the calls are successful Otherwise, the calling environment will sustain a `VALUE ERROR`.

More Examples

```

□NA'I4 advapi32 |RegCloseKey P'
□NA'I4 advapi32 |RegCreateKeyEx* P <OT U4 <OT U4 U4 P >P >U4'
□NA'I4 advapi32 |RegEnumValue* P U4 >OT =U4 =U4 >U4 >OT =U4'
□NA'I4 advapi32 |RegOpenKey* P <OT >P'
□NA'I4 advapi32 |RegOpenKeyEx* P <OT U4 U4 >P'
□NA'I4 advapi32 |RegQueryValueEx* P <OT =U4 >U4 >OT =U4'
□NA'I4 advapi32 |RegSetValueEx* P <OT =U4 U4 <OT U4'
□NA'P dyalog32 |STRNCYPY P P P'
□NA'P dyalog32 |STRNCYPYA P P P'
□NA'P dyalog32 |STRNCYPYW P P P'
□NA'P dyalog32 |MEMCPY P P P'
□NA'I4 gdi32 |AddFontResource* <OT'
□NA'I4 gdi32 |BitBlt P I4 I4 I4 I4 P I4 I4 U4'
□NA'U4 gdi32 |GetPixel P I4 I4'
□NA'P gdi32 |GetStockObject I4'
□NA'I4 gdi32 |RemoveFontResource* <OT'
□NA'U4 gdi32 |SetPixel P I4 I4 U4'
□NA' glu32 |gluPerspective F8 F8 F8 F8'
□NA'I4 kernel32 |CopyFile* <OT <OT I4'
□NA'P kernel32 |GetEnvironmentStrings'
□NA'U4 kernel32 |GetLastError'
□NA'U4 kernel32 |GetTempPath* U4 >OT'
□NA'P kernel32 |GetProcessHeap'
□NA'I4 kernel32 |GlobalMemoryStatusEx = {U4 U4 U8 U8 U8 U8 U8 U8}'
□NA'P kernel32 |HeapAlloc P U4 P'
□NA'I4 kernel32 |HeapFree P U4 P'
□NA' opengl32 |glClearColor F4 F4 F4 F4'
□NA' opengl32 |glClearDepth F8'
□NA' opengl32 |glEnable U4'
□NA' opengl32 |glMatrixMode U4'
□NA'I4 user32 |ClientToScreen P = {I4 I4}'
□NA'P user32 |FindWindow* <OT <OT'
□NA'I4 user32 |ShowWindow P I4'
□NA'I2 user32 |GetAsyncKeyState I4'
□NA'P user32 |GetDC P'
□NA'I4 User32 |GetDialogBaseUnits'
□NA'P user32 |GetFocus'
□NA'U4 user32 |GetSysColor I4'
□NA'I4 user32 |GetSystemMetrics I4'
□NA'I4 user2 |InvalidateRgn P P I4'
□NA'I4 user32 |MessageBox* P <OT <OT U4'
□NA'I4 user32 |ReleaseDC P P'
□NA'P user32 |SendMessage* P U4 P P'
□NA'P user32 |SetFocus P'
□NA'I4 user32 |WinHelp* P <OT U4 P'
□NA'I4 winnm |sndPlaySound <OT U4'

```

Variant:	$\{R\} \leftarrow \{X\} (f \square OPT B) Y$
-----------------	--

$\square OPT$ is synonymous with the Variant Operator symbol \square and is the only form available in the Classic Edition.

See *Variant Operator*.

Profile Application:	$R \leftarrow \square PROFILE Y$
-----------------------------	--

$\square PROFILE$ facilitates the profiling of either CPU consumption or elapsed time for a workspace. It does so by retaining time measurements collected for APL functions/operators and function/operator lines. $\square PROFILE$ is used to both control the state of profiling and retrieve the collected profiling data.

Y specifies the action to perform and any options for that action, if applicable. Y is case-insensitive.

Use	Description
$state \leftarrow \square PROFILE 'start' \{timer\}$	Turn profiling on using the specified timer or resume if profiling was stopped
$state \leftarrow \square PROFILE 'stop'$	Suspend the collection of profiling data
$state \leftarrow \square PROFILE 'clear'$	Turn profiling off, if active, and discard any collected profiling data
$state \leftarrow \square PROFILE 'calibrate'$	Calibrate the profiling timer
$state \leftarrow \square PROFILE 'state'$	Query profiling state
$data \leftarrow \square PROFILE 'data'$	Retrieve profiling data in flat form
$data \leftarrow \square PROFILE 'tree'$	Retrieve profiling data in tree form

`⎕PROFILE` has 2 states:

- active – the profiler is running and profiling data is being collected.
- inactive – the profiler is not running.

For most actions, the result of `⎕PROFILE` is its current state and contains:

[1]	character vector indicating the <code>⎕PROFILE</code> state having one of the values 'active' or 'inactive'
[2]	character vector indicating the timer being used having one of the values 'CPU' or 'elapsed'
[3]	call time bias in milliseconds. This is the amount of time, in milliseconds, that is consumed for the system to take a time measurement.
[4]	timer granularity in milliseconds, or 0 if the granularity cannot be accurately determined. This is the resolution of the timer being used.

state←`⎕PROFILE 'start' {timer}`

Turn profiling on; `timer` is an optional case-independent character vector containing 'CPU' or 'elapsed'. If omitted, it defaults to 'CPU'.

The first time a particular timer is chosen, `⎕PROFILE` will spend 1000 milliseconds (1 second) to approximate the call time bias and granularity for that timer.

```
⎕PROFILE 'start'  
active CPU 0.0001037499999 0.0001037499999
```

state←`⎕PROFILE 'stop'`

Suspends the collection of profiling data.

```
⎕PROFILE 'stop'  
inactive CPU 0.0001037499999 0.0001037499999
```

state←`⎕PROFILE 'clear'`

Clears any collected profiling data and, if profiling is active, places profiling in an inactive state.

```
⎕PROFILE 'clear'  
inactive 0 0
```

state←PROFILE 'calibrate'

Causes PROFILE to perform a 1000 millisecond calibration to approximate the call time bias and granularity for the current timer. Note, a timer must have been previously selected by using PROFILE 'start'.

PROFILE will retain the lesser of the current timer values compared to the new values computed by the calibration. The rationale for this is to use the smallest possible values of which we can be certain.

```
PROFILE 'calibrate'
active CPU 0.0001037499997 0.0001037499997
```

state←PROFILE 'state'

Returns the current profiling state.

```
)clear
clear ws
PROFILE 'state'
inactive 0 0

PROFILE 'start' 'CPU'
active CPU 0.0001037499997 0.0001037499997
PROFILE 'state'
active CPU 0.0001037499997 0.0001037499997
```

data←PROFILE 'data'

Retrieves the collected profiling data. Specifying 'data' returns:

[;1]	function name
[;2]	function line number or θ for a whole function entry
[;3]	number of times the line or function was executed
[;4]	accumulated time (ms) for this entry exclusive of items called by this entry
[;5]	accumulated time (ms) for this entry inclusive of items called by this entry
[;6]	number of times the timer function was called for the exclusive time
[;7]	number of times the timer function was called for the inclusive time

Example: (numbers have been truncated for formatting)

```

      □PROFILE 'data'
#.foo          1 1.04406 39347.64945 503 4080803
#.foo          1 1 0.12488 0.12488 1 1
#.foo          2 100 0.58851 39347.19390 200 4080500
#.foo          3 100 0.21340 0.21340 100 100
#.NS1.goo      100 99.44404 39346.6053 50300 4080300
#.NS1.goo      1 100 0.61679 0.61679 100 100
#.NS1.goo      2 10000 67.80292 39314.9642 20000 4050000
#.NS1.goo      3 10000 19.60274 19.6027 10000 10000

```

data←□PROFILE 'tree'

Retrieve the collected profiling data in tree format:

[; 1]	depth level
[; 2]	function name
[; 3]	function line number or θ for a whole function entry
[; 4]	number of times the line or function was executed
[; 5]	accumulated time (ms) for this entry exclusive of items called by this entry
[; 6]	accumulated time (ms) for this entry inclusive of items called by this entry
[; 7]	number of times the timer function was called for the exclusive time
[; 8]	number of times the timer function was called for the inclusive time

Example:

```

      □PROFILE 'tree'
0 #.foo          1 1.04406 39347.64945 503 4080803
1 #.foo          1 1 0.12488 0.12488 1 1
1 #.foo          2 100 0.58851 39347.19390 200 4080500
2 #.NS1.goo      100 99.44404 39346.60538 50300 4080300
3 #.NS1.goo      1 100 0.61679 0.61679 100 100
3 #.NS1.goo      2 10000 67.80292 39314.96426 20000 4050000
4 #.NS2.moo      10000 39247.16133 39247.16133 4030000 4030000
5 #.NS2.moo      1 10000 39.28315 39.28315 10000 10000
5 #.NS2.moo      2 1000000 36430.65236 36430.65236 1000000 1000000
5 #.NS2.moo      3 1000000 1645.36214 1645.36214 1000000 1000000
3 #.NS1.goo      3 10000 19.60274 19.60274 10000 10000
1 #.foo          3 100 0.21340 0.21340 100 100

```

Note that rows with an even depth level in column [; 1] represent function summary entries and odd depth level rows are function line entries. Recursive functions will generate separate rows for each level of recursion.

Notes

Profile Data Entry Types

The results of `PROFILE 'data'` and `PROFILE 'tree'` have two types of entries; function summary entries and function line entries. Function summary entries contain \emptyset in the line number column, whereas function line entries contain the line number. Dynamic functions line entries begin with 0 as they do not have a header line like traditional functions. The timer data and timer call counts in function summary entries represent the aggregate of the function line entries plus any time spent that cannot be directly attributed to a function line entry. This could include time spent during function initialisation, etc.

Example:

```
#.foo          1  1.04406 39347.649450  503 4080803
#.foo      1  1  0.12488  0.124887  1  1
#.foo      2 100 0.58851 39347.193900 200 4080500
#.foo      3 100 0.21340  0.213406 100  100
```

Timer Data Persistence

The profiling data collected is stored outside the workspace and will not impact workspace availability. The data is cleared upon workspace load, clear workspace, `PROFILE 'clear'`, or interpreter sign off.

The PROFILE User Command

`PROFILE` is a utility which implements a high-level interface to `PROFILE` and provides reporting and analysis tools that act upon the profiling data. For further information, see *Tuning Applications using the Profile User Command*.

Using PROFILE Directly

If you choose to use `PROFILE` directly, the following guidelines and information may be of use to you.

Note: Running your application with `PROFILE` turned on incurs a significant processing overhead and will slow your application down.

Decide which timer to use

`PROFILE` supports profiling of either CPU or elapsed time. CPU time is generally of more interest in profiling application performance.

Simple Profiling

To get a quick handle on the top CPU time consumers in an application, use the following procedure:

- Make sure the application runs long enough to collect enough data to overcome the timer granularity – a reasonable rule of thumb is to make sure the application runs for at least $(4000 \times 4 \div \text{PROFILE 'state'})$ milliseconds.
- Turn profiling on with `PROFILE 'start' 'CPU'`
- Run your application.
- Pause the profiler with `PROFILE 'stop'`
- Examine the profiling data from `PROFILE 'data'` or `PROFILE 'tree'` for entries that consume large amounts of resource.

This should identify any items that take more than 10% of the run time.

To find finer time consumers, or to focus on elapsed time rather than CPU time, take the following additional steps prior to running the profiler:

- Turn off as much hardware as possible. This would include peripherals, network connections, etc.
- Turn off as many other tasks and processes as possible. These include anti-virus software, firewalls, internet services, background tasks.
- Raise the priority on the Dyalog APL task to higher than normal, but in general avoid giving it the highest priority.
- Run the profiler as described above.

Doing this should help identify items that take more than 1% of the run time.

Advanced Profiling

The timing data collected by `PROFILE` is not adjusted for the timer's call time bias; in other words, the times reported by `PROFILE` include the time spent calling the timer function. One effect of this can be to make “cheap” lines that are called many times seem to consume more resource. If you desire more accurate profiling measurements, or if your application takes a short amount of time to run, you will probably want to adjust for the timer call time bias. To do so, subtract from the timing data the timer's call time bias multiplied by the number of times the timer was called.

Example:

```
CallTimeBias←3÷PROFILE 'state'  
RawTimes←PROFILE 'data'  
Adjusted←RawTimes[;4 5]-RawTimes[;6 7]×CallTimeBias
```


Space Indicator:**R←R SI**

R is a vector of refs to the spaces from which functions in the state indicator were called ($\rho R SI \leftrightarrow \rho NSI \leftrightarrow \rho SI$).

R SI and NSI are identical except that R SI returns refs to the spaces whereas NSI returns their names. ie. $NSI \leftrightarrow \text{⌈} R SI$.

Note that R SI returns refs to the spaces *from which* functions were called not those *in which* they are currently running.

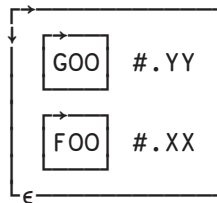
Example

```

)OBJECTS
XX      YY
      VR 'YY.FOO'
      ▽ R←FOO
[1]    R←SE.GOO
      ▽
      VR 'SE.GOO'
      ▽ R←GOO
[1]    R←SI, [1.5] NSI
      ▽

)CS XX
#.XX   ]display #.YY.FOO

```



Appendices: PCRE Specifications

PCRE (Perl Compatible Regular Expressions) is an open source library used by the `PR` and `PS` system operators. The regular expression syntax which the library supports is not unique to APL nor is it an integral part of the language. Its documentation is reproduced verbatim in these appendices. There are two named sections: *pcrpattern*, which describes the full syntax and semantics; and *pcresyntax*, a quick reference summary.

Appendix A – Search Pattern syntax

PCREPATTERN(3)

PCREPATTERN(3)

NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions that are supported by PCRE are described in detail below. There is a quick-reference syntax summary in the *pcresyntax* page. PCRE tries to match Perl syntax and semantics as closely as it can. PCRE also supports some alternative regular expression syntax (which does not conflict with the Perl syntax) in order to provide some compatibility with regular expressions in Python, .NET, and Oniguruma.

Perl's regular expressions are described in its own documentation, and regular expressions in general are covered in a number of books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

The original operation of PCRE was on strings of one-byte characters. However, there is now also support for UTF-8 character strings. To use this, PCRE must be built to include UTF-8 support, and you must call `pcre_compile()` or `pcre_compile2()` with the `PCRE_UTF8` option. There is also a special sequence that can be given at the start of a pattern:

```
(*UTF8)
```

Starting a pattern with this sequence is equivalent to setting the `PCRE_UTF8` option. This feature is not Perl-compatible. How setting UTF-8 mode affects pattern matching is mentioned in several places below. There is also a summary of UTF-8 features in the section on UTF-8 support in the main *pcre* page.

The remainder of this document discusses the patterns that are supported by PCRE when its main matching function, `pcre_exec()`, is used. From release 6.0, PCRE offers a second matching function, `pcre_dfa_exec()`, which matches using a different algorithm that is not Perl-compatible. Some of the features discussed below are not available when `pcre_dfa_exec()` is used. The advantages and disadvantages of the alternative function, and how it differs from the normal function, are discussed in the *pcrematching* page.

NEWLINE CONVENTIONS

PCRE supports five different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (line-feed) character, the two-character sequence CRLF, any of the three preceding, or any Unicode newline sequence. The `pcreapi` page has further discussion about newlines, and shows how to set the newline convention in the options arguments for the compiling and matching functions.

It is also possible to specify a newline convention by starting a pattern string with one of the following five sequences:

```
(*CR)      carriage return
(*LF)      linefeed
(*CRLF)    carriage return, followed by linefeed
(*ANYCRLF) any of the three above
(*ANY)     all Unicode newline sequences
```

These override the default and the options given to `pcre_compile()` or `pcre_compile2()`. For example, on a Unix system where LF is the default newline sequence, the pattern

```
(*CR)a.b
```

changes the convention to CR. That pattern matches "a\nb" because LF is no longer a newline. Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used.

The newline convention does not affect what the `\R` escape sequence matches. By default, this is any Unicode newline sequence, for Perl compatibility. However, this can be changed; see the description of `\R` in the section entitled "Newline sequences" below. A change of `\R` setting can be combined with a change of newline convention.

CHARACTERS AND METACHARACTERS

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (the `PCRE_CASELESS` option), letters are matched independently of case. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the

pattern by the use of metacharacters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

```

\  

^      general escape character with several uses  

^      assert start of string (or line, in multiline mode)  

$      assert end of string (or line, in multiline mode)  

.      match any character except newline (by default)  

[      start character class definition  

|      start of alternative branch  

(      start subpattern  

)      end subpattern  

?      extends the meaning of (  

       also 0 or 1 quantifier  

       also quantifier minimizer  

*      0 or more quantifier  

+      1 or more quantifier  

       also "possessive quantifier"  

{      start min/max quantifier

```

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

```

\  

^      general escape character  

^      negate the class, but only if the first character  

-      indicates character range  

[      POSIX character class (only if followed by POSIX  

       syntax)  

]      terminates the character class

```

The following sections describe the use of each of the metacharacters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a * character, you write \
in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\
.

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that \$ and @ are handled as literals in \Q...\E

sequences in PCRE, whereas in Perl, \$ and @ cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
<code>\Qabc\$xyz\E</code>	<code>abc\$xyz</code>	abc followed by the contents of \$xyz
<code>\Qabc\ \$xyz\E</code>	<code>abc\ \$xyz</code>	<code>abc\ \$xyz</code>
<code>\Qabc\E\ \$\Qxyz\E</code>	<code>abc\$xyz</code>	<code>abc\$xyz</code>

The `\Q...\E` sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences than the binary character it represents:

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	linefeed (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or back reference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh..

The precise effect of `\cx` is as follows: if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

After `\x`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between `\x{` and `}`, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2^{31} in UTF-8 mode. That is, the maximum value in hexadecimal is 7FFFFFFF. Note that this is bigger than the largest Unicode code point, which is 10FFFF.

If characters other than hexadecimal digits appear between `\x{` and `}`, or if there is no terminating `}`, this form of escape is not recognized. Instead, the initial `\x` will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for `\x`. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}`.

After `\0` up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x\07` specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

```

\040  is another way of writing a space
\40   is the same, provided there are fewer than 40
       previous capturing subpatterns
\7    is always a back reference
\11   might be a back reference, or another way of
       writing a tab
\011  is always a tab
\0113 is a tab followed by the character "3"
\113  might be a back reference, otherwise the
       character with octal code 113
\377  might be a back reference, otherwise
       the byte consisting entirely of 1 bits
\81   is either a back reference, or a binary zero
       followed by the two characters "8" and "1"

```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by an unsigned or a negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed later, following the discussion of parenthesized subpatterns.

Absolute and relative subroutine calls

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a "subroutine". Details are discussed later. Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

```
\d    any decimal digit
\D    any character that is not a decimal digit
\h    any horizontal whitespace character
\H    any character that is not a horizontal whitespace character
\s    any whitespace character
\S    any character that is not a whitespace character
\v    any vertical whitespace character
\V    any character that is not a vertical whitespace character
\w    any "word" character
\W    any "non-word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32). If "use locale;" is included in a Perl script, `\s` may match the VT character. In PCRE, it never does.

In UTF-8 mode, characters with values greater than 128 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`. This is true even when Unicode character property support is available. These sequences retain their original meanings from before UTF-8 support was available, mainly for efficiency reasons. Note that this also affects `\b`, because it is defined in terms of `\w` and `\W`.

The sequences `\h`, `\H`, `\v`, and `\V` are Perl 5.10 features. In contrast to the other sequences, these do match certain high-valued codepoints in UTF-8 mode. The horizontal space characters are:

```
U+0009    Horizontal tab
U+0020    Space
U+00A0    Non-break space
U+1680    Ogham space mark
U+180E    Mongolian vowel separator
U+2000    En quad
U+2001    Em quad
U+2002    En space
U+2003    Em space
U+2004    Three-per-em space
U+2005    Four-per-em space
U+2006    Six-per-em space
U+2007    Figure space
U+2008    Punctuation space
U+2009    Thin space
U+200A    Hair space
U+202F    Narrow no-break space
U+205F    Medium mathematical space
U+3000    Ideographic space
```


The vertical space characters are:

```

U+000A    Linefeed
U+000B    Vertical tab
U+000C    Formfeed
U+000D    Carriage return
U+0085    Next line
U+2028    Line separator
U+2029    Paragraph separator

```

A "word" character is an underscore or any character less than 256 that is a letter or digit. The definition of letters and digits is controlled by PCRE's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the pcreapi page). For example, in a French locale such as "fr FR" in Unix-like systems, or "french" in Windows, some character codes greater than 128 are used for accented letters, and these are matched by \w. The use of locales with Unicode is discouraged.

Newline sequences

Outside a character class, by default, the escape sequence \R matches any Unicode newline sequence. This is a Perl 5.10 feature. In non-UTF-8 mode \R is equivalent to the following:

```
(?>\r\n|\n|\x0b|\f|\r|\x85)
```

This is an example of an "atomic group", details of which are given below. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

In UTF-8 mode, two additional characters whose codepoints are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

It is possible to restrict \R to match only CR, LF, or CRLF (instead of the complete set of Unicode line endings) by setting the option PCRE_BSR_ANYCRLF either at compile time or when the pattern is matched. (BSR is an abbreviation for "backslash R".) This can be made the default when PCRE is built; if this is the case, the other behaviour can be requested via the PCRE_BSR_UNICODE option. It is also possible to specify these settings by starting a pattern string with one of the following sequences:

```

(*BSR_ANYCRLF)  CR, LF, or CRLF only
(*BSR_UNICODE)  any Unicode newline sequence

```

These override the default and the options given to pcre_compile() or pcre_compile2(), but they can be overridden by options given to pcre_exec() or pcre_dfa_exec(). Note that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used. They can be combined with a change of newline convention, for example, a pattern can start with:

```
(*ANY)(*BSR_ANYCRLF)
```

Inside a character class, `\R` matches the letter "R".

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences that match characters with specific properties are available. When not in UTF-8 mode, these sequences are of course limited to testing characters whose codepoints are less than 256, but they do work in this mode. The extra escape sequences are:

```
\p{xx}  a character with the xx property
\P{xx}  a character without the xx property
\X      an extended Unicode sequence
```

The property names represented by `xx` above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Other properties such as "InMusical-Symbols" are not currently supported by PCRE. Note that `\P{Any}` does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

```
\p{Greek}
\P{Han}
```

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kharoshthi, Khmer, Lao, Latin, Limbu, Linear_B, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Oriya, Osmanya, Phags_Pa, Phoenician, Runic, Shavian, Sinhala, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Yi.

Each character has exactly one general category property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, `\p{^Lu}` is the same as `\P{Lu}`.

If only one letter is specified with `\p` or `\P`, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

```
\p{L}
\pL
```

The following general category property codes are supported:

```
C      Other
Cc     Control
Cf     Format
Cn     Unassigned
```

Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

The special property `L&` is also supported: it matches a character that has the `Lu`, `Ll`, or `Lt` property, in other words, a letter that is not classified as a modifier or "other".

The `Cs` (Surrogate) property applies only to characters in the range `U+D800` to `U+DFFF`. Such characters are not valid in UTF-8 strings (see RFC 3629) and so cannot be tested by PCRE, unless UTF-8 validity checking has been turned off (see the discussion of `PCRE_NO_UTF8_CHECK` in the `pcreapi` page). Perl does not support the `Cs` property.

The long synonyms for property names that Perl supports (such as `\p{Letter}`) are not supported by PCRE, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the `Cn` (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, `\p{Lu}` always matches only upper case letters.

The `\X` escape matches any number of Unicode characters that form an extended Unicode sequence. `\X` is equivalent to

```
(?>\PM\pM*)
```

That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character. None of them have codepoints less than 256, so in non-UTF-8 mode `\X` matches any one character.

Matching characters by Unicode property is not fast, because PCRE has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as `\d` and `\w` do not use Unicode properties in PCRE.

Resetting the match start

The escape sequence `\K`, which is a Perl 5.10 feature, causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:

```
foo\Kbar
```

matches "foobar", but reports that it has matched "bar". This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of `\K` does not interfere with the setting of captured substrings. For example, when the pattern

```
(foo)\Kbar
```

matches "foobar", the first substring is still set to "foo".

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

```
\b    matches at a word boundary
\B    matches when not at a word boundary
\A    matches at the start of the subject
\Z    matches at the end of the subject
      also matches before a newline at the end of the subject
\z    matches only at the end of the subject
\G    matches at the first matching position in the subject
```

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the

string if the first or last character matches `\w`, respectively. Neither PCRE nor Perl has a separate "start of word" or "end of word" metasequence. However, whatever follows `\b` normally determines which it is. For example, the fragment `\ba` matches "a" at the start of a word.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the `PCRE_NOTBOL` or `PCRE_NOTEOL` options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the `startoffset` argument of `pcre_exec()` is non-zero, indicating that matching is to start at a point other than the beginning of the subject, `\A` can never match. The difference between `\Z` and `\z` is that `\Z` matches before a newline at the end of the string as well as at the very end, whereas `\z` matches only at the end.

The `\G` assertion is true only when the current matching position is at the start point of the match, as specified by the `startoffset` argument of `pcre_exec()`. It differs from `\A` when the value of `startoffset` is non-zero. By calling `pcre_exec()` multiple times with appropriate arguments, you can mimic Perl's `/g` option, and it is in this kind of implementation where `\G` can be useful.

Note, however, that PCRE's interpretation of `\G`, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with `\G`, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the `startoffset` argument of `pcre_exec()` is non-zero, circumflex can never match if the `PCRE_MULTILINE` option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the

very end of the string, by setting the `PCRE_DOLLAR_ENDONLY` option at compile time. This does not affect the `\Z` assertion.

The meanings of the circumflex and dollar characters are changed if the `PCRE_MULTILINE` option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when `PCRE_MULTILINE` is set. When newline is specified as the two-character sequence `CRLF`, isolated CR and LF characters do not indicate newlines.

For example, the pattern `^abc$/` matches the subject string `"def\nabc"` (where `\n` represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode, and a match for circumflex is possible when the `startoffset` argument of `pcre_exec()` is non-zero. The `PCRE_DOLLAR_ENDONLY` option is ignored if `PCRE_MULTILINE` is set.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether or not `PCRE_MULTILINE` is set.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line. In UTF-8 mode, the matched character may be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence `CRLF` is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

The behaviour of dot with regard to newlines can be changed. If the `PCRE_DOTALL` option is set, a dot matches any one character, without exception. If the two-character sequence `CRLF` is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence `\C` matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the `\C` escape sequence is best avoided.

PCRE does not allow `\C` to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special by default. However, if the `PCRE_JAVASCRIPT_COMPAT` option is set, a lone closing square bracket causes a compile-time error. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may be more than one byte long. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `[^aeiou]` matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion; it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the `\x{` escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `[^aeiou]` does not match "A", whereas a caseful version would. In UTF-8 mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching in UTF8-mode for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF-8 support.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the `PCRE_DOTALL` and `PCRE_MULTILINE` options is used. A class such as `[^a]` always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037]. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example [\x{100}-\x{2ff}]

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\^_wxyzabc], matched caselessly, and in non-UTF-8 mode, if character tables for a French locale are in use, [\xc8-\xcb] matches accented E characters in both cases. In UTF-8 mode, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character types \d, \D, \p, \P, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

alnum	letters and digits
alpha	letters
ascii	character codes 0 - 127
blank	space or tab only
cntrl	control characters
digit	decimal digits (same as \d)
graph	printing characters, excluding space
lower	lower case letters
print	printing characters, including space
punct	printing characters, excluding letters and digits
space	white space (not quite the same as \s)


```
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to `\s`, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 128 do not match any of the POSIX character classes.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the `PCRE_CASELESS`, `PCRE_MULTILINE`, `PCRE_DOTALL`, and `PCRE_EXTENDED` options (which are Perl-compatible) can be changed from within the pattern by a sequence of Perl option letters enclosed between `"?"` and `""`. The option letters are

```
i for PCRE_CASELESS
m for PCRE_MULTILINE
s for PCRE_DOTALL
x for PCRE_EXTENDED
```

For example, `(?im)` sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as `(?im-sx)`, which sets `PCRE_CASELESS` and `PCRE_MULTILINE` while unsetting `PCRE_DOTALL` and `PCRE_EXTENDED`, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options `PCRE_DUPNAMES`, `PCRE_UNGREEDY`, and `PCRE_EXTRA` can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

When one of these option changes occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the `pcre_fullinfo()` function).

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches `abc` and `aBc` and no other strings (assuming `PCRE_CASELESS` is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches `"ab"`, `"aB"`, `"c"`, and `"C"`, even though when matching `"C"` the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

Note: There are other PCRE-specific options that can be set by the application when the `compile` or `match` functions are called. In some cases the pattern can contain special leading sequences such as `(*CRLF)` to override what the application has set or what has been defaulted. Details are given in the section entitled "Newline sequences" above. There is also the `(*UTF8)` leading sequence that can be used to set UTF-8 mode; this is equivalent to setting the `PCRE_UTF8` option.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words `"cat"`, `"cataract"`, or `"caterpillar"`. Without the parentheses, it would match `"cataract"`, `"erpillar"` or an empty string.

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the `ovector` argument of `pcre_exec()`. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string `"the red king"` is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

DUPLICATE SUBPATTERN NUMBERS

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (?! and is itself a non-capturing subpattern. For example, consider this pattern:

```
(?! (Sat)ur|(Sun) )day
```

Because the two alternatives are inside a (?! group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (?! group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing buffers that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

```
# before -----branch-reset----- after
/ ( a ) (?! x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1          2          2 3          2    3    4
```

A back reference to a numbered subpattern uses the most recent value that is set for that number by any subpattern. The following pattern matches "abcabc" or "defdef":

```
/(?!(abc)|(def))\1/
```

In contrast, a recursive or "subroutine" call to a numbered subpattern always refers to the first one in the pattern with the given number. The following pattern matches "abcabc" or "defabc":

```
/(?|(abc)|(def))(?1)/
```

If a condition test for a subpattern's having matched refers to a non-unique number, the test is true if any of the subpatterns of that number have matched.

An alternative approach to using this "branch reset" feature is to use duplicate named subpatterns, as described in the next section.

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax. Perl allows identically numbered subpatterns to have different names, but PCRE does not.

In PCRE, a subpattern can be named in one of three ways: (?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as back references, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. There is also a convenience function for extracting a captured substring by name.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the PCRE_DUPNAMES option at compile time. (Duplicate names are also always permitted for subpatterns with the same number, set up as described in the previous section.) Duplicate names can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a "branch reset" subpattern, as described in the previous section.)

The convenience function for extracting the data by name returns the

substring for the first (and in this example, the only) subpattern of that name that matched. This saves searching to find which numbered subpattern it was.

If you make a back reference to a non-unique named subpattern from elsewhere in the pattern, the one that corresponds to the first occurrence of the name is used. In the absence of duplicate numbers (see the previous section) this is the one with the lowest number. If you use a named reference in a condition test (see the section about conditions below), either to check whether a subpattern has matched, or to check for recursion, all subpatterns with the same name are tested. If the condition is true for any one of them, the overall condition is true. This is the same behaviour as testing by number. For further details of the interfaces for handling named subpatterns, see the `pcreapi` documentation.

Warning: You cannot use different names to distinguish between two subpatterns with the same number because PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are given to subpatterns with the same number. However, you can give the same name to subpatterns with the same number, even when `PCRE_DUPNAMES` is not set.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the `\C` escape sequence
- the `\X` escape sequence (in UTF-8 mode with Unicode properties)
- the `\R` escape sequence
- an escape such as `\d` that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)
- a recursive or "subroutine" call to a subpattern

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a

position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, \x{100}{2} matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, when Unicode property support is available, \X{3} matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present. This may be useful for subpatterns that are referenced as subroutines from elsewhere in the pattern. Items other than subpatterns that have a {0} quantifier are omitted from the compiled pattern.

For convenience, the three most common quantifiers have single-character abbreviations:

```
*   is equivalent to {0,}
+   is equivalent to {1,}
?   is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\.*\*/
```

to the string

```
/* first comment */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\.*?\*/
```

does the right thing with the C comments. The meaning of the various

quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` or `.{0,}` and the PCRE_DOTALL option (equivalent to Perl's `/s`) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by `\A`.

In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL in order to obtain this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When `.*` is inside capturing parentheses that are the subject of a back reference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy")

repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

Note that a possessive quantifier can be used with an entire group, for example:

```
(abc|xyz){2,3}+
```

Possessive quantifiers are always greedy; the setting of the `PCRE_UNGREEDY` option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers

should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun's Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A's when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|\<d+\>)*[!?)]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?) rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?)\D+|\<d+\>)*[!?)]
```

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal.

See the subsection entitled "Non-printing characters" above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the `\g` escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. These examples are all identical:

```
(ring), \1
(ring), \g1
(ring), \g{1}
```

An unsigned number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence `\g{-1}` is a reference to the most recently started capturing subpattern before `\g`, that is, is it equivalent to `\2`. Similarly, `\g{-2}` would be equivalent to `\1`. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \1libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax `\k{name}` and the Perl syntax `\k<name>` or `\k'name'` are supported, as is the Python syntax `(?P=name)`. Perl 5.10's unified back reference syntax, in which `\g` can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

```
(?<p1>(i)rah)\s+\k<p1>
(?'p1'(i)rah)\s+\k{p1}
(?P<p1>(i)rah)\s+(?P=p1)
(?<p1>(i)rah)\s+\g{p1}
```

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail by default. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". However, if the PCRE_JAVASCRIPT_COMPAT option is set at compile time, a back reference to an unset value matches an empty string.

Because there may be many capturing parentheses in a pattern, all digits following a backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise, the `\g{` syntax or an empty comment (see "Comments" below) can be used.

Recursive back references

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Back references of this type cause the group that they reference to be treated as an atomic group. Once the whole group has been matched, a subsequent matching failure cannot cause backtracking into the middle of the group.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `\A`, `\G`, `\Z`, `\z`, `^` and `$` are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail. The Perl 5.10 backtracking control verb (*FAIL) or (*F) is essentially a synonym for (!).

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (5.8 and 5.10), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable to PCRE if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

In some cases, the Perl 5.10 escape sequence `\K` (see above) can be used instead of a lookbehind assertion to get round the fixed-length restriction.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the `\C` escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The `\X` and `\R` escapes, which can match different numbers of bytes, are also not permitted.

"Subroutine" calls (see below) such as `(?2)` or `(?&X)` are permitted in lookbehinds, as long as the subpattern matches a fixed-length string. Recursion, however, is not supported.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching of fixed-length strings at the end of subject strings. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abc-foo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a specific capturing subpattern has already been matched. The two possible forms of conditional subpattern are:

```
(?(condition)yes-pattern)  
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if a capturing subpattern of that number has previously matched. If there is more than one capturing subpattern with the same number (see the earlier section about duplicate subpattern numbers), the condition is true if any of them have been set. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by `?(-1)`, the next most recent by `?(-2)`, and so on. In looping constructs it can also make sense to refer to subsequent groups with constructs such as `?(+2)`.

Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \ ( ) ?    [ ^ ( ) ] +    ( ? ( 1 ) \ ) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The

third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

```
...other stuff... ( \ ( )? [^()] + (?(-1) \ ) ) ...
```

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax `?(<name>...)` or `?('name'...)` to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax `?(name)...` is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

```
?(<OPEN> \ ( )? [^()] + (?(<OPEN>) \ ) )
```

If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them has matched.

Checking for pattern recursion

If the condition is the string `(R)`, and there is no subpattern with the name `R`, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter `R`, for example:

```
?(R3...) or (?(&name)...) 
```

the condition is true if the most recent recursion is into a subpattern whose number or name is given. This condition does not check the entire recursion stack. If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them is the most recent recursion.

At "top level", all these recursion test conditions are false. The syntax for recursive patterns is described below.

Defining subpatterns for use by reference only

If the condition is the string `(DEFINE)`, and there is no subpattern with the name `DEFINE`, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of

DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) )
\b (?&byte) (\.(?&byte)){3} \b
```

The first part of the pattern is a DEFINE group inside which another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition. The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:


```
$re = qr\( (? : (?>[^()]+) | (?p{$re}) ) * \) x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was subsequently introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

This PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
\( ( [^()]+ | (?R) ) * \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis. Note the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \ ( ( [^()]+ | (?1) ) * \ ) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references (a Perl 5.10 feature). Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always "subroutine" calls, as described in the next section.

An alternative approach is to use named parentheses instead. The Perl syntax for this is (?&name); PCRE's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

```
(?<pn> \ ( ( [^()]+ | (?&pn) ) * \ ) )
```

If there is more than one subpattern with the same name, the earliest one is used.

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of a possessive quantifier for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa())
```

it yields "no match" quickly. However, if a possessive quantifier is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values of capturing parentheses are those from the outermost level. If you want to obtain intermediate values, a callout function can be used (see below and the `pcrecallout` documentation). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the inner capturing parentheses (numbered 2) is "ef", which is the last value taken on at the top level. If a capturing subpattern is not matched at the top level, its final value is unset, even if it is (temporarily) set at a deeper level.

If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using `pcre_malloc`, freeing it via `pcre_free` afterwards. If no memory can be obtained, the match fails with the `PCRE_ERROR_NOMEMORY` error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (?R) \d+ | [^<]+) | (?R) * >
```

In this pattern, `(?R)` is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The `(?R)` item is the actual recursive call.

Recursion difference from Perl

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. This can be illustrated by the following pattern, which purports to match a palindromic string that contains an odd number of characters (for example, "a", "aba", "abcba", "abcdcba"):

```
^(.|.) (?1)\2$
```

The idea is that it either matches a single character, or two identical characters surrounding a sub-palindrome. In Perl, this pattern works; in PCRE it does not if the pattern is longer than three characters. Consider the subject string "abcba":

At the top level, the first character is matched, but as it is not at the end of the string, the first alternative fails; the second alternative is taken and the recursion kicks in. The recursive call to subpattern 1 successfully matches the next character ("b"). (Note that the beginning and end of line tests are not part of the recursion).

Back at the top level, the next character ("c") is compared with what subpattern 2 matched, which was "a". This fails. Because the recursion is treated as an atomic group, there are now no backtracking points, and so the entire match fails. (Perl is able, at this point, to re-enter the recursion and try the second alternative.) However, if the pattern is written with the alternatives in the other order, things are different:

```
^(.)(?1)\2|.)$
```

This time, the recursing alternative is tried first, and continues to recurse until it runs out of characters, at which point the recursion fails. But this time we do have another alternative to try at the higher level. That is the big difference: in the previous case the remaining alternative is at a deeper recursion level, which PCRE cannot use.

To change the pattern so that matches all palindromic strings, not just those with an odd number of characters, it is tempting to change the pattern to this:

```
^(.)(?1)\2|.?)$
```

Again, this works in Perl, but not in PCRE, and for the same reason. When a deeper recursion has matched a single character, it cannot be entered again in order to match an empty string. The solution is to separate the two cases, and write out the odd and even cases as alternatives at the higher level:

```
^(?:((.)(?1)\2)|((.)(?3)\4|.))
```

If you want to match typical palindromic phrases, the pattern has to ignore all non-word characters, which can be done like this:

```
^\W*(?:((.\W*(?1)\W*\2)|((.\W*(?3)\W*\4|\W*.\W*))\W*$
```

If run with the PCRE_CASELESS option, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works well in both PCRE and Perl. Note the use of the possessive quantifier `+` to avoid backtracking into sequences of non-word characters. Without this, PCRE takes a great deal longer (ten times or more) to match typical phrases, and Perl takes so long that you think it has gone into a loop.

WARNING: The palindrome-matching patterns above work only if the subject string does not start with a palindrome that is shorter than the entire string. For example, although "abcba" is correctly matched, if the subject is "ababa", PCRE finds the palindrome "aba" at the start, then fails at top level because the end of the string does not follow. Once again, it cannot jump back into the recursion to try other alternatives, so the entire match fails.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

```
(...(absolute)...)(?2)...  
(...(relative)...)(?-1)...  
(...(?)...)(relative)...
```

An earlier example pointed out that the pattern

```
(sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?1)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a subroutine call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. Any capturing parentheses that are set during the subroutine call revert to their previous values afterwards.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

```
(abc)(?i:(?-1))
```

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

ONIGURUMA SUBROUTINE SYNTAX

For compatibility with Oniguruma, the non-Perl syntax `\g` followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a subroutine, possibly recursively. Here are two of the examples used above, rewritten using this syntax:

```
(?<pn> \{ ( (?>[^\s()]+) | \g<pn> )* \} )  
(sens|respons)e and \g'1'ibility
```

PCRE supports an extension to Oniguruma: if a number is preceded by a plus or a minus sign it is taken as a relative reference. For example:

```
(abc)(?i:\g<-1>)
```

Note that `\g{...}` (Perl syntax) and `\g<...>` (Oniguruma syntax) are not synonymous. The former is a back reference; the latter is a subroutine call.

CALLOUTS

Perl has a feature whereby using the sequence `{...}` causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable `pcre_callout`. By default, this variable contains `NULL`, which disables all calling out.

Within a regular expression, `(?C)` indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter `C`. The default value is zero. For example, this pattern has two callout points:

```
(?C1)abc(?C2)def
```

If the `PCRE_AUTO_CALLOUT` flag is passed to `pcre_compile()`, callouts are automatically installed before each item in the pattern. They are all numbered 255.

During matching, when PCRE reaches a callout point (and `pcre_callout` is set), the external function is called. It is provided with the number of the callout, the position in the pattern, and, optionally, one item of data originally supplied by the caller of `pcre_exec()`. The callout function may cause matching to proceed, to backtrack, or to fail altogether. A complete description of the interface to the callout function is given in the `pcrecallout` documentation.

BACKTRACKING CONTROL

Perl 5.10 introduced a number of "Special Backtracking Control Verbs", which are described in the Perl documentation as "experimental and subject to change or removal in a future version of Perl". It goes on to say: "Their usage in production code should be noted to avoid problems during upgrades." The same remarks apply to the PCRE features described in this section.

Since these verbs are specifically related to backtracking, most of them can be used only when the pattern is to be matched using `pcre_exec()`, which uses a backtracking algorithm. With the exception of `(*FAIL)`, which behaves like a failing negative assertion, they cause an error if encountered by `pcre_dfa_exec()`.

If any of these verbs are used in an assertion or subroutine subpattern (including recursive subpatterns), their effect is confined to that subpattern; it does not extend to the surrounding pattern. Note that such subpatterns are processed as anchored at the point where they are tested.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. In Perl, they are generally of the form `(*VERB:ARG)` but PCRE does not support the use of arguments, so

its general form is just (*VERB). Any number of these verbs may occur in a pattern. There are two kinds:

Verbs that act immediately

The following verbs act as soon as they are encountered:

(*ACCEPT)

This verb causes the match to end successfully, skipping the remainder of the pattern. When inside a recursion, only the innermost pattern is ended immediately. If (*ACCEPT) is inside capturing parentheses, the data so far is captured. (This feature was added to PCRE at release 8.00.) For example:

A(?:A|B(*ACCEPT)|C)D

This matches "AB", "AAD", or "ACD"; when it matches "AB", "B" is captured by the outer parentheses.

(*FAIL) or (*F)

This verb causes the match to fail, forcing backtracking to occur. It is equivalent to (!) but easier to read. The Perl documentation notes that it is probably useful only when combined with ({} or (?!)). Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

a+(?C)(*FAIL)

A match with the string "aaaa" always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

Verbs that act after backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is no subsequent match, a failure is forced. The verbs differ in exactly what kind of failure occurs.

(*COMMIT)

This verb causes the whole match to fail outright if the rest of the pattern does not match. Even if the pattern is unanchored, no further attempts to find a match by advancing the starting point take place. Once (*COMMIT) has been passed, pcre_exec() is committed to finding a match at the current starting point, or not at all. For example:

a+(*COMMIT)b

This matches "xxaab" but not "aacaab". It can be thought of as a kind of dynamic anchor, or "I've started, so I must finish."

(*PRUNE)

This verb causes the match to fail at the current position if the rest of the pattern does not match. If the pattern is unanchored, the normal "bumpalong" advance to the next starting character then happens. Backtracking can occur as usual to the left of (*PRUNE), or when matching

to the right of (*PRUNE), but if there is no match to the right, backtracking cannot cross (*PRUNE). In simple cases, the use of (*PRUNE) is just an alternative to an atomic group or possessive quantifier, but there are some uses of (*PRUNE) that cannot be expressed in any other way.

(*SKIP)

This verb is like (*PRUNE), except that if the pattern is unanchored, the "bumpalong" advance is not to the next character, but to the position in the subject where (*SKIP) was encountered. (*SKIP) signifies that whatever text was matched leading up to it cannot be part of a successful match. Consider:

a+(*SKIP)b

If the subject is "aaaac...", after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at "c". Note that a possessive quantifier does not have the same effect as this example; although it would suppress backtracking during the first match attempt, the second attempt would start at the second character instead of skipping on to "c".

(*THEN)

This verb causes a skip to the next alternation if the rest of the pattern does not match. That is, it cancels pending backtracking, but only within the current alternation. Its name comes from the observation that it can be used for a pattern-based if-then-else block:

(COND1 (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ) ...

If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds); on failure the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If (*THEN) is used outside of any alternation, it acts exactly like (*PRUNE).

AUTHOR

Philip Hazel
 University Computing Service
 Cambridge CB2 3QH, England.

REVISION

Last updated: 11 January 2010
 Copyright (c) 1997-2010 University of Cambridge.

Appendix B – Search Pattern syntax summary

PCRESYNTAX (3)

PCRESYNTAX (3)

NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION SYNTAX SUMMARY

The full syntax and semantics of the regular expressions that are supported by PCRE are described in the `pcrepattern` documentation. This document contains just a quick-reference summary of the syntax.

QUOTING

<code>\x</code>	where x is non-alphanumeric is a literal x
<code>\Q...\E</code>	treat enclosed characters as literal

CHARACTERS

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	newline (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or backreference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh..

CHARACTER TYPES

<code>.</code>	any character except newline; in <code>dotall</code> mode, any character whatsoever
<code>\C</code>	one byte, even in UTF-8 mode (best avoided)
<code>\d</code>	a decimal digit
<code>\D</code>	a character that is not a decimal digit
<code>\h</code>	a horizontal whitespace character
<code>\H</code>	a character that is not a horizontal whitespace character
<code>\p{xx}</code>	a character with the xx property
<code>\P{xx}</code>	a character without the xx property
<code>\R</code>	a newline sequence
<code>\s</code>	a whitespace character
<code>\S</code>	a character that is not a whitespace character
<code>\v</code>	a vertical whitespace character
<code>\V</code>	a character that is not a vertical whitespace character
<code>\w</code>	a "word" character
<code>\W</code>	a "non-word" character
<code>\X</code>	an extended Unicode sequence

In PCRE, `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` recognize only ASCII characters.

GENERAL CATEGORY PROPERTY CODES FOR \p and \P

C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
L&	Ll, Lu, or Lt
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

SCRIPT NAMES FOR \p AND \P

Arabic, Armenian, Balinese, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lycian, Lydian, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Ol_Chiki, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Saurash-

tra, Shavian, Sinhala, Sudanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, Yi.

CHARACTER CLASSES

[...]	positive character class
[^...]	negative character class
[x-y]	range (can be used for hex characters)
[:xxx:]	positive POSIX named set
[:^xxx:]	negative POSIX named set
alnum	alphanumeric
alpha	alphabetic
ascii	0-127
blank	space or tab
cntrl	control character
digit	decimal digit
graph	printing, excluding space
lower	lower case letter
print	printing, including space
punct	printing, excluding alphanumeric
space	whitespace
upper	upper case letter
word	same as \w
xdigit	hexadecimal digit

In PCRE, POSIX character set names recognize only ASCII characters. You can use \Q...\E inside a character class.

QUANTIFIERS

?	0 or 1, greedy
?+	0 or 1, possessive
??	0 or 1, lazy
*	0 or more, greedy
*+	0 or more, possessive
*?	0 or more, lazy
+	1 or more, greedy
++	1 or more, possessive
??	1 or more, lazy
{n}	exactly n
{n,m}	at least n, no more than m, greedy
{n,m}+	at least n, no more than m, possessive
{n,m}?	at least n, no more than m, lazy
{n,}	n or more, greedy
{n,}+	n or more, possessive
{n,}?	n or more, lazy

ANCHORS AND SIMPLE ASSERTIONS

\b	word boundary (only ASCII letters recognized)
\B	not a word boundary
^	start of subject also after internal newline in multiline mode
\A	start of subject
\$	end of subject

	also before newline at end of subject
	also before internal newline in multiline mode
\Z	end of subject
	also before newline at end of subject
\z	end of subject
\G	first matching position in subject

MATCH POINT RESET

\K	reset start of match
----	----------------------

ALTERNATION

expr|expr|expr...

CAPTURING

(...)	capturing group
(?<name>...)	named capturing group (Perl)
(?'name'...)	named capturing group (Perl)
(?P<name>...)	named capturing group (Python)
(?:...)	non-capturing group
(? ...)	non-capturing group; reset group numbers for capturing groups in each alternative

ATOMIC GROUPS

(?>...)	atomic, non-capturing group
---------	-----------------------------

COMMENT

(?#....)	comment (not nestable)
----------	------------------------

OPTION SETTING

(?i)	caseless
(?J)	allow duplicate names
(?m)	multiline
(?s)	single line (dotall)
(?U)	default ungreedy (lazy)
(?x)	extended (ignore white space)
(?-...)	unset option(s)

The following is recognized only at the start of a pattern or after one of the newline-setting options with similar syntax:

(*UTF8)	set UTF-8 mode
---------	----------------

LOOKAHEAD AND LOOKBEHIND ASSERTIONS

(?=...)	positive look ahead
(?!...)	negative look ahead
(?<=...)	positive look behind

(?!...) negative look behind

Each top-level branch of a look behind must be of a fixed length.

BACKREFERENCES

\n	reference by number (can be ambiguous)
\gn	reference by number
\g{n}	reference by number
\g{-n}	relative reference by number
\k<name>	reference by name (Perl)
\k'name'	reference by name (Perl)
\g{name}	reference by name (Perl)
\k{name}	reference by name (.NET)
(?P=name)	reference by name (Python)

SUBROUTINE REFERENCES (POSSIBLY RECURSIVE)

(?R)	recurse whole pattern
(?n)	call subpattern by absolute number
(?+n)	call subpattern by relative number
(?-n)	call subpattern by relative number
(?&name)	call subpattern by name (Perl)
(?P>name)	call subpattern by name (Python)
\g<name>	call subpattern by name (Oniguruma)
\g'name'	call subpattern by name (Oniguruma)
\g<n>	call subpattern by absolute number (Oniguruma)
\g'n'	call subpattern by absolute number (Oniguruma)
\g<+n>	call subpattern by relative number (PCRE extension)
\g'+n'	call subpattern by relative number (PCRE extension)
\g<-n>	call subpattern by relative number (PCRE extension)
\g'-n'	call subpattern by relative number (PCRE extension)

CONDITIONAL PATTERNS

(?(condition)yes-pattern)	
(?(condition)yes-pattern no-pattern)	
(?(n)...	absolute reference condition
(?(+n)...	relative reference condition
(?(-n)...	relative reference condition
(?(<name>)...	named reference condition (Perl)
(?('name')...	named reference condition (Perl)
(?(name)...	named reference condition (PCRE)
(?(R)...	overall recursion condition
(?(Rn)...	specific group recursion condition
(?(R&name)...	specific recursion condition
(?(DEFINE)...	define subpattern for reference
(?(assert)...	assertion condition

BACKTRACKING CONTROL

The following act immediately they are reached:

(*ACCEPT)	force successful match
(*FAIL)	force backtrack; synonym (*F)

The following act only when a subsequent match failure causes a backtrack to reach them. They all force a match failure, but they differ in what happens afterwards. Those that advance the start-of-match point do so only if the pattern is not anchored.

(*COMMIT)	overall failure, no advance of starting point
(*PRUNE)	advance to next starting character
(*SKIP)	advance start to current matching position
(*THEN)	local failure, backtrack to next alternation

NEWLINE CONVENTIONS

These are recognized only at the very start of the pattern or after a (*BSR_...) or (*UTF8) option.

(*CR)	carriage return only
(*LF)	linefeed only
(*CRLF)	carriage return followed by linefeed
(*ANYCRLF)	all three of the above
(*ANY)	any Unicode newline sequence

WHAT \R MATCHES

These are recognized only at the very start of the pattern or after a (*...) option that sets the newline convention or UTF-8 mode.

(*BSR_ANYCRLF)	CR, LF, or CRLF
(*BSR_UNICODE)	any Unicode newline sequence

CALLOUTS

(?C)	callout
(?Cn)	callout with data n

AUTHOR

Philip Hazel
 University Computing Service
 Cambridge CB2 3QH, England.

REVISION

Last updated: 11 April 2009
 Copyright (c) 1997-2009 University of Cambridge.

Appendix C – License

Part of the functionality of `⎕R` and `⎕S` is implemented using the PCRE library which is redistributed according to the following license:

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Release 8 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

The basic library functions are written in C and are freestanding. Also included in the distribution is a set of C++ wrapper functions.

THE BASIC LIBRARY FUNCTIONS

Written by: Philip Hazel
Email local part: ph10
Email domain: cam.ac.uk

University of Cambridge Computing Service,
Cambridge, England.

Copyright (c) 1997-2009 University of Cambridge
All rights reserved.

THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.

Copyright (c) 2007-2008, Google Inc.
All rights reserved.

THE "BSD" LICENCE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the University of Cambridge nor the name of Google

Inc. nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

A

absolute value *See* magnitude
 add arithmetic function 58
 and boolean function 59
 APL_COMPLEX_AS_V12 parameter 23
 APL_EXTERN_DECF parameter 22
 APL_FAST_FCHK parameter 22
 APL_FCREATE_PROPS_C parameter 22
 APL_FCREATE_PROPS_J parameter 22
 APLFormatBias parameter 11

B

best fit approximation 79
 beta function 60
 binary integer decimal 28
 binomial function 60
 Boolean functions
 and (conjunction) 59
 byte order mark 37

C

ceiling function 60
 change user 10, 98
 circular function 23
 circular functions 61
 classic edition 45, 88, 133
 Classic Edition 123
 Compatibility 4
 complex numbers *See* Chapter 3
 circular functions 32, 61
 floating-point representation 27, 106
 component files 22, 24
 compatibility 4
 conjugate 9
 conjunction *See* and
 creating component files 103

D

data representation
 monadic 102
 decimal comparison tolerance 9, 27, 101
 default property 74

densely packed decimal 28
 deprecated features
 32-bit component files 104
 direction function 65
 divide arithmetic function 65
 DOMAIN ERROR 49
 DotAll option 44
 drop function 8, 66
 dyadic primitive functions
 add 58
 and 59
 circular 23
 divide 65
 drop 8, 66
 format 71
 index function 8, 73
 left 9, 76
 logarithm 23, 77
 matrix divide 78
 power 23, 33, 83
 residue 84
 right 9, 85
 subtract 86
 take 8, 87
 dyadic primitive operators
 replace 35, 88
 search 35, 88
 variant 10, 35, 43, 88, 133
 Dyalog Unicode IME 1, 15
 dynamic link libraries 107

E

Enc option 48
 EOL option 45
 equal relational function 67
 Euler identity 14, 32, 69
 exponential function 69

F

factorial function 69
 file check 22
 file copy 24, 104
 file create 24, 103

file untie22
floating-point representation . 9, 25, 26, 27, 101, 105
 complex numbers27, 106
floor function..... 70
fork new task 10, 97
format function
 dyadic 11, 71
 monadic 11
format system function..... 11

G

gamma function.....69
grade down function..... 14
grade up function..... 14
Greedy option.....46

I

i-beam.....91
 change user 10, 98
 fork new task 10, 97
 read DataTable 10, 94
 reap forked tasks..... 10, 98
 signal counts..... 10, 101
 update DataTable..... 10, 92
IC option..... 43, 88
identity9, 73
identity function62
identity matrix80
IME 15
index function.....8, 73
index-generator function 14
InEnc option47
Input Method Editor 15
Interoperability4
iota..... *See* index generator

K

keyboard shortcuts..... 16

L

least squares solution.....79
left9, 76
logarithm function 23, 77
logical conjunction *See* and

logical operations.....*See* Boolean functions

M

magnitude function77
matrix product*See* inner product
matrix-divide function78
matrix-inverse function80
ML option.....46, 49
Mode option44, 49, 88
monadic primitive functions
 ceiling60
 conjugate 9
 direction65
 exponential69
 factorial69
 floor70
 format 11
 grade down 14
 grade up 14
 identity.....9, 62, 73
 index generator14
 magnitude77
 matrix inverse80
 natural logarithm81
 negative81
 pi times82
 reciprocal84
 same.....9, 85
 signum65
multiply arithmetic function 81

N

name association.....107
namespace indicator9, 139
Naperian logarithm function.....81
natural logarithm function81
negate..... *See* negative function
negative function81
NEOL option45

O

OM option47
OutEnc option.....48
overstrike introducer key20
overstrikes popup.....20

P

PCRE 35
 pi-times function 82
 power function 23, 33, 83
 primitive operators
 replace 35, 88
 search 35, 88
 variant 35, 88, 133
 Principal option 43, 88, 89
 profile 9
 profile application 9, 133
 profile user command 137
 properties
 propertyget Function 75
 propertyset function 75

R

read DataTable **10, 94**
 reap forked tasks 10, 98
 reciprocal function 84
 regular expressions 35
 replace operator 35, 88
 DotAll 44
 Enc 48
 EOL 45
 Greedy 46
 IC 43, 88
 InEnc 47
 ML 46, 49
 Mode 44, 49, 88
 NEOL 45
 OutEnc 48
 residue function 84
 right 9, 85

S

same 9, 85

search operator 35, 88
 DotAll 44
 Enc 48
 EOL 45
 Greedy 46
 IC 43, 88
 InEnc 47
 ML 46, 49
 Mode 44, 49, 88
 NEOL 45
 OM 47
 OutEnc 48
 signal counts 10, 101
 signum function 65
 space indicator 9, 139
 squad indexing 8, 73
 subtract arithmetic function 86

T

take function 8, 87
 TRANSLATION ERROR 45

U

Unicode Edition 123
 update DataTable **10, 92**

V

variant operator 10, 35, 43, 88, 133
 verify and fix input 23
 version number 2

W

WansSpecialKeys parameter 16
 wide character 114

Z

zilde 14



DYALOG

APL

Dyalog Ltd
Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom
www.dyalog.com