NAME

PCRE2 - Perl-compatible regular expressions (revised API)

PCRE2 REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions that are supported by PCRE2 are described in detail below. There is a quick-reference syntax summary in the **pcre2syntax** page. PCRE2 tries to match Perl syntax and semantics as closely as it can. PCRE2 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 PCRE2's regular expressions is intended as reference material.

This document discusses the regular expression patterns that are supported by PCRE2 when its main matching function, **pcre2_match()**, is used. PCRE2 also has an alternative matching function, **pcre2_dfa_match()**, which matches using a different algorithm that is not Perl-compatible. Some of the features discussed below are not available when DFA matching is used. The advantages and disadvantages of the alternative function, and how it differs from the normal function, are discussed in the **pcre2matching** page.

SPECIAL START-OF-PATTERN ITEMS

A number of options that can be passed to **pcre2_compile()** can also be set by special items at the start of a pattern. These are not Perl-compatible, but are provided to make these options accessible to pattern writers who are not able to change the program that processes the pattern. Any number of these items may appear, but they must all be together right at the start of the pattern string, and the letters must be in upper case.

UTF support

In the 8-bit and 16-bit PCRE2 libraries, characters may be coded either as single code units, or as multiple UTF-8 or UTF-16 code units. UTF-32 can be specified for the 32-bit library, in which case it constrains the character values to valid Unicode code points. To process UTF strings, PCRE2 must be built to include Unicode support (which is the default). When using UTF strings you must either call the compiling function with one or both of the PCRE2_UTF or PCRE2_MATCH_INVALID_UTF options, or the pattern must start with the special sequence (*UTF), which is equivalent to setting the relevant PCRE2_UTF. How setting a UTF mode affects pattern matching is mentioned in several places below. There is also a summary of features in the **pcre2unicode** page.

Some applications that allow their users to supply patterns may wish to restrict them to non-UTF data for security reasons. If the PCRE2_NEVER_UTF option is passed to **pcre2_compile()**, (*UTF) is not allowed, and its appearance in a pattern causes an error.

Unicode property support

Another special sequence that may appear at the start of a pattern is (*UCP). This has the same effect as setting the PCRE2_UCP option: it causes sequences such as \d and \w to use Unicode properties to determine character types, instead of recognizing only characters with codes less than 256 via a lookup table. If also causes upper/lower casing operations to use Unicode properties for characters with code points greater than 127, even when UTF is not set.

Some applications that allow their users to supply patterns may wish to restrict them for security reasons. If the PCRE2_NEVER_UCP option is passed to **pcre2_compile()**, (*UCP) is not allowed, and its appearance in a pattern causes an error.

Locking out empty string matching

Starting a pattern with (*NOTEMPTY) or (*NOTEMPTY_ATSTART) has the same effect as passing the

PCRE2_NOTEMPTY or PCRE2_NOTEMPTY_ATSTART option to whichever matching function is subsequently called to match the pattern. These options lock out the matching of empty strings, either entirely, or only at the start of the subject.

Disabling auto-possessification

If a pattern starts with (*NO_AUTO_POSSESS), it has the same effect as setting the PCRE2_NO_AUTO_POSSESS option. This stops PCRE2 from making quantifiers possessive when what follows cannot match the repeated item. For example, by default a+b is treated as a++b. For more details, see the **pcre2api** documentation.

Disabling start-up optimizations

If a pattern starts with (*NO_START_OPT), it has the same effect as setting the PCRE2_NO_START_OPTIMIZE option. This disables several optimizations for quickly reaching "no match" results. For more details, see the **pcre2api** documentation.

Disabling automatic anchoring

If a pattern starts with (*NO_DOTSTAR_ANCHOR), it has the same effect as setting the PCRE2_NO_DOTSTAR_ANCHOR option. This disables optimizations that apply to patterns whose top-level branches all start with .* (match any number of arbitrary characters). For more details, see the **pcre2api** documentation.

Disabling JIT compilation

If a pattern that starts with (*NO_JIT) is successfully compiled, an attempt by the application to apply the JIT optimization by calling **pcre2_jit_compile()** is ignored.

Setting match resource limits

The **pcre2_match()** function contains a counter that is incremented every time it goes round its main loop. The caller of **pcre2_match()** can set a limit on this counter, which therefore limits the amount of computing resource used for a match. The maximum depth of nested backtracking can also be limited; this indirectly restricts the amount of heap memory that is used, but there is also an explicit memory limit that can be set.

These facilities are provided to catch runaway matches that are provoked by patterns with huge matching trees. A common example is a pattern with nested unlimited repeats applied to a long string that does not match. When one of these limits is reached, **pcre2_match()** gives an error return. The limits can also be set by items at the start of the pattern of the form

```
(*LIMIT_HEAP=d)
(*LIMIT_MATCH=d)
(*LIMIT_DEPTH=d)
```

where d is any number of decimal digits. However, the value of the setting must be less than the value set (or defaulted) by the caller of **pcre2_match()** for it to have any effect. In other words, the pattern writer can lower the limits set by the programmer, but not raise them. If there is more than one setting of one of these limits, the lower value is used. The heap limit is specified in kibibytes (units of 1024 bytes).

Prior to release 10.30, LIMIT_DEPTH was called LIMIT_RECURSION. This name is still recognized for backwards compatibility.

The heap limit applies only when the **pcre2_match()** or **pcre2_dfa_match()** interpreters are used for matching. It does not apply to JIT. The match limit is used (but in a different way) when JIT is being used, or when **pcre2_dfa_match()** is called, to limit computing resource usage by those matching functions. The depth limit is ignored by JIT but is relevant for DFA matching, which uses function recursion for recursions within the pattern and for lookaround assertions and atomic groups. In this case, the depth limit controls the

depth of such recursion.

Newline conventions

PCRE2 supports six different conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (linefeed) character, the two-character sequence CRLF, any of the three preceding, any Unicode newline sequence, or the NUL character (binary zero). The **pcre2api** page has further discussion about newlines, and shows how to set the newline convention when calling **pcre2_compile()**.

It is also possible to specify a newline convention by starting a pattern string with one of the following sequences:

```
(*CR)       carriage return
(*LF)       linefeed
(*CRLF)     carriage return, followed by linefeed
(*ANYCRLF)  any of the three above
(*ANY)      all Unicode newline sequences
(*NUL)      the NUL character (binary zero)
```

These override the default and the options given to the compiling function. 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. If more than one of these settings is present, the last one is used.

The newline convention affects where the circumflex and dollar assertions are true. It also affects the interpretation of the dot metacharacter when PCRE2_DOTALL is not set, and the behaviour of \N when not followed by an opening brace. However, it 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 next section and 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.

Specifying what \R matches

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 PCRE2_BSR_ANYCRLF at compile time. This effect can also be achieved by starting a pattern with (*BSR_ANYCRLF). For completeness, (*BSR_UNICODE) is also recognized, corresponding to PCRE2_BSR_UNICODE.

EBCDIC CHARACTER CODES

PCRE2 can be compiled to run in an environment that uses EBCDIC as its character code instead of ASCII or Unicode (typically a mainframe system). In the sections below, character code values are ASCII or Unicode; in an EBCDIC environment these characters may have different code values, and there are no code points greater than 255.

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 PCRE2_CASELESS option or (?i) within the pattern), letters are matched independently of case. Note that

there are two ASCII characters, K and S, that, in addition to their lower case ASCII equivalents, are case-equivalent with Unicode U+212A (Kelvin sign) and U+017F (long S) respectively when either PCRE2_UTF or PCRE2_UCP is set.

The power of regular expressions comes from the ability to include wild cards, character classes, 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 group or control verb
)      end group or control verb
*      0 or more quantifier
+      1 or more quantifier; also "possessive quantifier"
?      0 or 1 quantifier; also quantifier minimizer
{      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 (if followed by POSIX syntax)
]      terminates the character class
```

If a pattern is compiled with the PCRE2_EXTENDED option, most white space in the pattern, other than in a character class, and characters between a # outside a character class and the next newline, inclusive, are ignored. An escaping backslash can be used to include a white space or a # character as part of the pattern. If the PCRE2_EXTENDED_MORE option is set, the same applies, but in addition unescaped space and horizontal tab characters are ignored inside a character class. Note: only these two characters are ignored, not the full set of pattern white space characters that are ignored outside a character class. Option settings can be changed within a pattern; see the section entitled "Internal Option Setting" below.

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 character that is not a digit or a letter, 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 must 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 \\.

Only ASCII digits and letters have any special meaning after a backslash. All other characters (in particular, those whose code points are greater than 127) are treated as literals.

If you want to treat all characters in a sequence as literals, 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 PCRE2, whereas in Perl, $ and @ cause variable interpolation. Also, Perl does "double-quotish backslash interpolation" on any backslashes between \Q and \E which, its documentation says, "may lead to confusing results". PCRE2 treats a backslash between \Q and \E just like any other character. Note the following examples:

Pattern	PCRE2 matches	Perl matches
\Qabc$xyz\E	abc$xyz	abc followed by the contents of $xyz
\Qabc\$xyz\E	abc\$xyz	abc\$xyz
\Qabc\E\$\Qxyz\E	abc$xyz	abc$xyz
\QA\B\E	A\B	A\B
\Q\\E	\	\\E

The \Q...\E sequence is recognized both inside and outside character classes. An isolated \E that is not preceded by \Q is ignored. If \Q is not followed by \E later in the pattern, the literal interpretation continues to the end of the pattern (that is, \E is assumed at the end). If the isolated \Q is inside a character class, this causes an error, because the character class is not terminated by a closing square bracket.

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 in a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences instead of the binary character it represents. In an ASCII or Unicode environment, these escapes are as follows:

\a	alarm, that is, the BEL character (hex 07)
\cx	"control-x", where x is any printable ASCII character
\e	escape (hex 1B)
\f	form feed (hex 0C)
\n	linefeed (hex 0A)
\r	carriage return (hex 0D) (but see below)
\t	tab (hex 09)
\0dd	character with octal code 0dd
\ddd	character with octal code ddd, or backreference
\o{ddd..}	character with octal code ddd..
\xhh	character with hex code hh
\x{hhh..}	character with hex code hhh..
\N{U+hhh..}	character with Unicode hex code point hhh..

By default, after \x that is not followed by {, 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 }. If a character other than a hexadecimal digit appears between \x{ and }, or if there is no terminating }, an error occurs.

Characters whose code points are less than 256 can be defined by either of the two syntaxes for \x or by an octal sequence. There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc} or \334. However, using the braced versions does make such sequences easier to read.

Support is available for some ECMAScript (aka JavaScript) escape sequences via two compile-time options. If PCRE2_ALT_BSUX is set, the sequence \x followed by { is not recognized. Only if \x is followed by two hexadecimal digits is it recognized as a character escape. Otherwise it is interpreted as a literal "x" character. In this mode, support for code points greater than 256 is provided by \u, which must be followed by four hexadecimal digits; otherwise it is interpreted as a literal "u" character.

PCRE2_EXTRA_ALT_BSUX has the same effect as PCRE2_ALT_BSUX and, in addition, \u{hhh..} is recognized as the character specified by hexadecimal code point. There may be any number of hexadecimal digits. This syntax is from ECMAScript 6.

The \N{U+hhh..} escape sequence is recognized only when PCRE2 is operating in UTF mode. Perl also uses \N{name} to specify characters by Unicode name; PCRE2 does not support this. Note that when \N is not followed by an opening brace (curly bracket) it has an entirely different meaning, matching any character that is not a newline.

There are some legacy applications where the escape sequence \r is expected to match a newline. If the PCRE2_EXTRA_ESCAPED_CR_IS_LF option is set, \r in a pattern is converted to \n so that it matches a LF (linefeed) instead of a CR (carriage return) character.

The precise effect of \cx on ASCII characters 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 \cA to \cZ become hex 01 to hex 1A (A is 41, Z is 5A), but \c{ becomes hex 3B ({ is 7B), and \c; becomes hex 7B (; is 3B). If the code unit following \c has a value less than 32 or greater than 126, a compile-time error occurs.

When PCRE2 is compiled in EBCDIC mode, \N{U+hhh..} is not supported. \a, \e, \f, \n, \r, and \t generate the appropriate EBCDIC code values. The \c escape is processed as specified for Perl in the **perlebcdic** document. The only characters that are allowed after \c are A-Z, a-z, or one of @, [, \,], ^, _, or ?. Any other character provokes a compile-time error. The sequence \c@ encodes character code 0; after \c the letters (in either case) encode characters 1-26 (hex 01 to hex 1A); [, \,], ^, and _ encode characters 27-31 (hex 1B to hex 1F), and \c? becomes either 255 (hex FF) or 95 (hex 5F).

Thus, apart from \c?, these escapes generate the same character code values as they do in an ASCII environment, though the meanings of the values mostly differ. For example, \cG always generates code value 7, which is BEL in ASCII but DEL in EBCDIC.

The sequence \c? generates DEL (127, hex 7F) in an ASCII environment, but because 127 is not a control character in EBCDIC, Perl makes it generate the APC character. Unfortunately, there are several variants of EBCDIC. In most of them the APC character has the value 255 (hex FF), but in the one Perl calls POSIX-BC its value is 95 (hex 5F). If certain other characters have POSIX-BC values, PCRE2 makes \c? generate 95; otherwise it generates 255.

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\015 specifies two binary zeros followed by a CR character (code value 13). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The escape \o must be followed by a sequence of octal digits, enclosed in braces. An error occurs if this is not the case. This escape is a recent addition to Perl; it provides way of specifying character code points as octal numbers greater than 0777, and it also allows octal numbers and backreferences to be unambiguously specified.

For greater clarity and unambiguity, it is best to avoid following \ by a digit greater than zero. Instead, use \o{} or \x{} to specify numerical character code points, and \g{} to specify backreferences. The following paragraphs describe the old, ambiguous syntax.

The handling of a backslash followed by a digit other than 0 is complicated, and Perl has changed over time, causing PCRE2 also to change.

Outside a character class, PCRE2 reads the digit and any following digits as a decimal number. If the number is less than 10, begins with the digit 8 or 9, or if there are at least that many previous capture groups in the expression, the entire sequence is taken as a *backreference*. A description of how this works is given later, following the discussion of parenthesized groups. Otherwise, up to three octal digits are read to form a character code.

Inside a character class, PCRE2 handles \8 and \9 as the literal characters "8" and "9", and otherwise reads up to three octal digits following the backslash, using them to generate a data character. Any subsequent digits stand for themselves. For example, outside a character class:

```
\040   is another way of writing an ASCII space
\40    is the same, provided there are fewer than 40
         previous capture groups
```

```
\7     is always a backreference
\11    might be a backreference, 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 backreference, otherwise the
         character with octal code 113
\377   might be a backreference, otherwise
         the value 255 (decimal)
\81    is always a backreference
```

Note that octal values of 100 or greater that are specified using this syntax must not be introduced by a leading zero, because no more than three octal digits are ever read.

Constraints on character values

Characters that are specified using octal or hexadecimal numbers are limited to certain values, as follows:

```
8-bit non-UTF mode    no greater than 0xff
16-bit non-UTF mode   no greater than 0xffff
32-bit non-UTF mode   no greater than 0xffffffff
All UTF modes         no greater than 0x10ffff and a valid code point
```

Invalid Unicode code points are all those in the range 0xd800 to 0xdfff (the so-called "surrogate" code points). The check for these can be disabled by the caller of **pcre2_compile()** by setting the option PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES. However, this is possible only in UTF-8 and UTF-32 modes, because these values are not representable in UTF-16.

Escape sequences in character classes

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, \b is interpreted as the backspace character (hex 08).

When not followed by an opening brace, \N is not allowed in a character class. \B, \R, and \X are not special inside a character class. Like other unrecognized alphabetic escape sequences, they cause an error. Outside a character class, these sequences have different meanings.

Unsupported escape sequences

In Perl, the sequences \F, \l, \L, \u, and \U are recognized by its string handler and used to modify the case of following characters. By default, PCRE2 does not support these escape sequences in patterns. However, if either of the PCRE2_ALT_BSUX or PCRE2_EXTRA_ALT_BSUX options is set, \U matches a "U" character, and \u can be used to define a character by code point, as described above.

Absolute and relative backreferences

The sequence \g followed by a signed or unsigned number, optionally enclosed in braces, is an absolute or relative backreference. A named backreference can be coded as \g{name}. Backreferences are discussed later, following the discussion of parenthesized groups.

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 capture group as a subroutine. Details are discussed later. Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are*not* synon y-mous. The former is a backreference; the latter is a subroutine call.

Generic character types

Another use of backslash is for specifying generic character types:

```
\d     any decimal digit
\D     any character that is not a decimal digit
\h     any horizontal white space character
\H     any character that is not a horizontal white space character
\N     any character that is not a newline
\s     any white space character
\S     any character that is not a white space character
\v     any vertical white space character
\V     any character that is not a vertical white space character
\w     any "word" character
\W     any "non-word" character
```

The \N escape sequence has the same meaning as the "." metacharacter when PCRE2_DOTALL is not set, but setting PCRE2_DOTALL does not change the meaning of \N. Note that when \N is followed by an opening brace it has a different meaning. See the section entitled "Non-printing characters" above for details. Perl also uses \N{name} to specify characters by Unicode name; PCRE2 does not support this.

Each pair of lower and upper case escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair. The 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, because there is no character to match.

The default \s characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32), which are defined as white space in the "C" locale. This list may vary if locale-specific matching is taking place. For example, in some locales the "non-breaking space" character (\xA0) is recognized as white space, and in others the VT character is not.

A "word" character is an underscore or any character that is a letter or digit. By default, the definition of letters and digits is controlled by PCRE2's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the **pcre2api** page). For example, in a French locale such as "fr_FR" in Unix-like systems, or "french" in Windows, some character codes greater than 127 are used for accented letters, and these are then matched by \w. The use of locales with Unicode is discouraged.

By default, characters whose code points are greater than 127 never match \d, \s, or \w, and always match \D, \S, and \W, although this may be different for characters in the range 128-255 when locale-specific matching is happening. These escape sequences retain their original meanings from before Unicode support was available, mainly for efficiency reasons. If the PCRE2_UCP option is set, the behaviour is changed so that Unicode properties are used to determine character types, as follows:

```
\d  any character that matches \p{Nd} (decimal digit)
\s  any character that matches \p{Z} or \h or \v
\w  any character that matches \p{L} or \p{N}, plus underscore
```

The upper case escapes match the inverse sets of characters. Note that \d matches only decimal digits, whereas \w matches any Unicode digit, as well as any Unicode letter, and underscore. Note also that PCRE2_UCP affects \b, and \B because they are defined in terms of \w and \W. Matching these sequences is noticeably slower when PCRE2_UCP is set.

The sequences \h, \H, \v, and \V, in contrast to the other sequences, which match only ASCII characters by default, always match a specific list of code points, whether or not PCRE2_UCP is set. The horizontal space characters are:

```
U+0009     Horizontal tab (HT)
```

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 (LF)
U+000B Vertical tab (VT)
U+000C Form feed (FF)
U+000D Carriage return (CR)
U+0085 Next line (NEL)
U+2028 Line separator
U+2029 Paragraph separator

In 8-bit, non-UTF-8 mode, only the characters with code points less than 256 are relevant.

Newline sequences

Outside a character class, by default, the escape sequence \R matches any Unicode newline sequence. In 8-bit 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 (form feed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). Because this is an atomic group, the two-character sequence is treated as a single unit that cannot be split.

In other modes, two additional characters whose code points are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode 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 PCRE2_BSR_ANYCRLF at compile time. (BSR is an abbreviation for "backslash R".) This can be made the default when PCRE2 is built; if this is the case, the other behaviour can be requested via the PCRE2_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 the compiling function. 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)

They can also be combined with the (*UTF) or (*UCP) special sequences. Inside a character class, \R is treated as an unrecognized escape sequence, and causes an error.

Unicode character properties

When PCRE2 is built with Unicode support (the default), three additional escape sequences that match characters with specific properties are available. They can be used in any mode, though in 8-bit and 16-bit non-UTF modes these sequences are of course limited to testing characters whose code points are less than U+0100 and U+10000, respectively. In 32-bit non-UTF mode, code points greater than 0x10ffff (the Unicode limit) may be encountered. These are all treated as being in the Unknown script and with an unassigned type. The extra escape sequences are:

\p{*xx*} a character with the*xx* property
\P{*xx*} a character without the*xx* property
\X a Unicode extended grapheme cluster

The property names represented by *xx* above are case-sensitive. There is support for Unicode script names, Unicode general category properties, "Any", which matches any character (including newline), and some special PCRE2 properties (described in the next section). Other Perl properties such as "InMusicalSymbols" are not supported by PCRE2. 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}

Unassigned characters (and in non-UTF 32-bit mode, characters with code points greater than 0x10FFFF) are assigned the "Unknown" script. Others that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Adlam, Ahom, Anatolian_Hieroglyphs, Arabic, Armenian, Avestan, Balinese, Bamum, Bassa_Vah, Batak, Bengali, Bhaiksuki, Bopomofo, Brahmi, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Caucasian_Albanian, Chakma, Cham, Cherokee, Chorasmian, Common, Coptic, Cuneiform, Cypriot, Cypro_Minoan, Cyrillic, Deseret, Devanagari, Dives_Akuru, Dogra, Duployan, Egyptian_Hieroglyphs, Elbasan, Elymaic, Ethiopic, Georgian, Glagolitic, Gothic, Grantha, Greek, Gujarati, Gunjala_Gondi, Gurmukhi, Han, Hangul, Hanifi_Rohingya, Hanunoo, Hatran, Hebrew, Hiragana, Imperial_Aramaic, Inherited, Inscriptional_Pahlavi, Inscriptional_Parthian, Javanese, Kaithi, Kannada, Katakana, Kayah_Li, Kharoshthi, Khitan_Small_Script, Khmer, Khojki, Khudawadi, Lao, Latin, Lepcha, Limbu, Linear_A, Linear_B, Lisu, Lycian, Lydian, Mahajani, Makasar, Malayalam, Mandaic, Manichaean, Marchen, Masaram_Gondi, Medefaidrin, Meetei_Mayek, Mende_Kikakui, Meroitic_Cursive, Meroitic_Hieroglyphs, Miao, Modi, Mongolian, Mro, Multani, Myanmar, Nabataean, Nandinagari, New_Tai_Lue, Newa, Nko, Nushu, Nyakeng_Puachue_Hmong, Ogham, Ol_Chiki, Old_Hungarian, Old_Italic, Old_North_Arabian, Old_Permic, Old_Persian, Old_Sogdian, Old_South_Arabian, Old_Turkic, Old_Uyghur, Oriya, Osage, Osmanya, Pahawh_Hmong, Palmyrene, Pau_Cin_Hau, Phags_Pa, Phoenician, Psalter_Pahlavi, Rejang, Runic, Samaritan, Saurashtra, Sharada, Shavian, Siddham, SignWriting, Sinhala, Sogdian, Sora_Sompeng, Soyombo, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tai_Tham, Tai_Viet, Takri, Tamil, Tangsa, Tangut, Telugu, Thaana, Thai, Tibetan, Tifinagh, Tirhuta, Toto, Ugaritic, Unknown, Vai, Vithkuqi, Wancho,

Warang_Citi, Yezidi, Yi, Zanabazar_Square.

Each character has exactly one Unicode 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 whose code points are in the range U+D800 to U+DFFF. These characters are no different to any other character when PCRE2 is not in UTF mode (using the 16-bit or 32-bit library). However, they are not valid in Unicode strings and so cannot be tested by PCRE2 in UTF mode, unless UTF validity checking has been turned off (see the discussion of PCRE2_NO_UTF_CHECK in the **pcre2api** page).

The long synonyms for property names that Perl supports (such as \p{Letter}) are not supported by PCRE2, 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. This is different from the behaviour of current versions of Perl.

Matching characters by Unicode property is not fast, because PCRE2 has to do a multistage table lookup in order to find a character's property. That is why the traditional escape sequences such as \d and \w do not use Unicode properties in PCRE2 by default, though you can make them do so by setting the PCRE2_UCP option or by starting the pattern with (*UCP).

Extended grapheme clusters

The \X escape matches any number of Unicode characters that form an "extended grapheme cluster", and treats the sequence as an atomic group (see below). Unicode supports various kinds of composite character by giving each character a grapheme breaking property, and having rules that use these properties to define the boundaries of extended grapheme clusters. The rules are defined in Unicode Standard Annex 29, "Unicode Text Segmentation". Unicode 11.0.0 abandoned the use of some previous properties that had been used for emojis. Instead it introduced various emoji-specific properties. PCRE2 uses only the Extended Pictographic property.

\X always matches at least one character. Then it decides whether to add additional characters according to the following rules for ending a cluster:

1. End at the end of the subject string.

2. Do not end between CR and LF; otherwise end after any control character.

3. Do not break Hangul (a Korean script) syllable sequences. Hangul characters are of five types: L, V, T, LV, and LVT. An L character may be followed by an L, V, LV, or LVT character; an LV or V character may be followed by a V or T character; an LVT or T character may be followed only by a T character.

4. Do not end before extending characters or spacing marks or the "zero-width joiner" character. Characters with the "mark" property always have the "extend" grapheme breaking property.

5. Do not end after prepend characters.

6. Do not break within emoji modifier sequences or emoji zwj sequences. That is, do not break between characters with the Extended_Pictographic property. Extend and ZWJ characters are allowed between the characters.

7. Do not break within emoji flag sequences. That is, do not break between regional indicator (RI) characters if there are an odd number of RI characters before the break point.

8. Otherwise, end the cluster.

PCRE2's additional properties

As well as the standard Unicode properties described above, PCRE2 supports four more that make it

possible to convert traditional escape sequences such as \w and \s to use Unicode properties. PCRE2 uses these non-standard, non-Perl properties internally when PCRE2_UCP is set. However, they may also be used explicitly. These properties are:

```
Xan   Any alphanumeric character
Xps   Any POSIX space character
Xsp   Any Perl space character
Xwd   Any Perl "word" character
```

Xan matches characters that have either the L (letter) or the N (number) property. Xps matches the characters tab, linefeed, vertical tab, form feed, or carriage return, and any other character that has the Z (separator) property. Xsp is the same as Xps; in PCRE1 it used to exclude vertical tab, for Perl compatibility, but Perl changed. Xwd matches the same characters as Xan, plus underscore.

There is another non-standard property, Xuc, which matches any character that can be represented by a Universal Character Name in C++ and other programming languages. These are the characters $, @, ‘ (grave accent), and all characters with Unicode code points greater than or equal to U+00A0, except for the surrogates U+D800 to U+DFFF. Note that most base (ASCII) characters are excluded. (Universal Character Names are of the form \uHHHH or \UHHHHHHHH where H is a hexadecimal digit. Note that the Xuc property does not match these sequences but the characters that they represent.)

Resetting the match start

In normal use, the escape sequence \K causes any previously matched characters not to be included in the final matched sequence that is returned. For example, the pattern:

```
foo\Kbar
```

matches "foobar", but reports that it has matched "bar". \K does not interact with anchoring in any way. The pattern:

```
^foo\Kbar
```

matches only when the subject begins with "foobar" (in single line mode), though it again reports the matched string as "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".

From version 5.32.0 Perl forbids the use of \K in lookaround assertions. From release 10.38 PCRE2 also forbids this by default. However, the PCRE2_EXTRA_ALLOW_LOOKAROUND_BSK option can be used when calling **pcre2_compile()** to re-enable the previous behaviour. When this option is set, \K is acted upon when it occurs inside positive assertions, but is ignored in negative assertions. Note that when a pattern such as (?=ab\K) matches, the reported start of the match can be greater than the end of the match. Using \K in a lookbehind assertion at the start of a pattern can also lead to odd effects. For example, consider this pattern:

```
(?<=\Kfoo)bar
```

If the subject is "foobar", a call to **pcre2_match()** with a starting offset of 3 succeeds and reports the matching string as "foobar", that is, the start of the reported match is earlier than where the match started.

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 groups 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
```

Inside a character class, \b has a different meaning; it matches the backspace character. If any other of these assertions appears in a character class, an "invalid escape sequence" error is generated.

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. When PCRE2 is built with Unicode support, the meanings of \w and \W can be changed by setting the PCRE2_UCP option. When this is done, it also affects \b and \B. Neither PCRE2 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 PCRE2_NOTBOL or PCRE2_NOTEOL options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the *startoffset* argument of **pcre2_match()** 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 matching process, as specified by the *startoffset* argument of **pcre2_match()**. It differs from \A when the value of *startoffset* is non-zero. By calling **pcre2_match()** 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 PCRE2's implementation of \G, being true at the starting character of the matching process, is subtly different from Perl's, which defines it as true at the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE2 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

The circumflex and dollar metacharacters are zero-width assertions. That is, they test for a particular condition being true without consuming any characters from the subject string. These two metacharacters are concerned with matching the starts and ends of lines. If the newline convention is set so that only the two-character sequence CRLF is recognized as a newline, isolated CR and LF characters are treated as ordinary data characters, and are not recognized as newlines.

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 **pcre2_match()** is non-zero, or if PCRE2_NOTBOL is set, circumflex can never match if the PCRE2_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.)

The 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), unless PCRE2_NOTEOL is set. Note, however, that it does not actually match the newline. 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 PCRE2_DOLLAR_ENDONLY option at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar metacharacters are changed if the PCRE2_MULTILINE option is set. When this is the case, a dollar character matches before any newlines in the string, as well as at the very end, and 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, for compatibility with Perl. However, this can be changed by setting the PCRE2_ALT_CIRCUMFLEX option.

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 **pcre2_match()** is non-zero. The PCRE2_DOLLAR_ENDONLY option is ignored if PCRE2_MULTILINE is set.

When the newline convention (see "Newline conventions" below) recognizes the two-character sequence CRLF as a newline, this is preferred, even if the single characters CR and LF are also recognized as newlines. For example, if the newline convention is "any", a multiline mode circumflex matches before "xyz" in the string "abc\r\nxyz" rather than after CR, even though CR on its own is a valid newline. (It also matches at the very start of the string, of course.)

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 PCRE2_MULTILINE is set.

FULL STOP (PERIOD, DOT) AND \N

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.

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 PCRE2_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.

The escape sequence \N when not followed by an opening brace behaves like a dot, except that it is not affected by the PCRE2_DOTALL option. In other words, it matches any character except one that signifies the end of a line.

When \N is followed by an opening brace it has a different meaning. See the section entitled "Non-printing characters" above for details. Perl also uses \N{name} to specify characters by Unicode name; PCRE2 does

not support this.

MATCHING A SINGLE CODE UNIT

Outside a character class, the escape sequence \C matches any one code unit, whether or not a UTF mode is set. In the 8-bit library, one code unit is one byte; in the 16-bit library it is a 16-bit unit; in the 32-bit library it is a 32-bit unit. Unlike a dot, \C always matches line-ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode, but it is unclear how it can usefully be used.

Because \C breaks up characters into individual code units, matching one unit with \C in UTF-8 or UTF-16 mode means that the rest of the string may start with a malformed UTF character. This has undefined results, because PCRE2 assumes that it is matching character by character in a valid UTF string (by default it checks the subject string's validity at the start of processing unless the PCRE2_NO_UTF_CHECK or PCRE2_MATCH_INVALID_UTF option is used).

An application can lock out the use of \C by setting the PCRE2_NEVER_BACKSLASH_C option when compiling a pattern. It is also possible to build PCRE2 with the use of \C permanently disabled.

PCRE2 does not allow \C to appear in lookbehind assertions (described below) in UTF-8 or UTF-16 modes, because this would make it impossible to calculate the length of the lookbehind. Neither the alternative matching function **pcre2_dfa_match()** nor the JIT optimizer support \C in these UTF modes. The former gives a match-time error; the latter fails to optimize and so the match is always run using the interpreter.

In the 32-bit library, however, \C is always supported (when not explicitly locked out) because it always matches a single code unit, whether or not UTF-32 is specified.

In general, the \C escape sequence is best avoided. However, one way of using it that avoids the problem of malformed UTF-8 or UTF-16 characters is to use a lookahead to check the length of the next character, as in this pattern, which could be used with a UTF-8 string (ignore white space and line breaks):

```
(?| (?=[\x00-\x7f])(\C) |
   (?=[\x80-\x{7ff}])(\C)(\C) |
   (?=[\x{800}-\x{ffff}])(\C)(\C)(\C) |
   (?=[\x{10000}-\x{1fffff}])(\C)(\C)(\C)(\C))
```

In this example, a group that starts with (?| resets the capturing parentheses numbers in each alternative (see "Duplicate Group Numbers" below). The assertions at the start of each branch check the next UTF-8 character for values whose encoding uses 1, 2, 3, or 4 bytes, respectively. The character's individual bytes are then captured by the appropriate number of \C groups.

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. 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. This means that, by default, an empty class cannot be defined. However, if the PCRE2_ALLOW_EMPTY_CLASS option is set, a closing square bracket at the start does end the (empty) class.

A character class matches a single character in the subject. 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.

Characters in a class may be specified by their code points using \o, \x, or \N{U+hh..} in the usual way. 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. Note that there are two ASCII characters, K and S, that, in addition to their lower case ASCII equivalents, are case-equivalent with Unicode U+212A (Kelvin sign) and U+017F (long S) respectively when either PCRE2_UTF or PCRE2_UCP is set.

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 PCRE2_DOTALL and PCRE2_MULTILINE options is used. A class such as [^a] always matches one of these characters.

The generic character type escape sequences \d, \D, \h, \H, \p, \P, \s, \S, \v, \V, \w, and \W may appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. In UTF modes, the PCRE2_UCP option affects the meanings of \d, \s, \w and their upper case partners, just as it does when they appear outside a character class, as described in the section entitled "Generic character types" above. The escape sequence \b has a different meaning inside a character class; it matches the backspace character. The sequences \B, \R, and \X are not special inside a character class. Like any other unrecognized escape sequences, they cause an error. The same is true for \N when not followed by an opening brace.

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, or immediately after a range. For example, [b-d-z] matches letters in the range b to d, a hyphen character, or z.

Perl treats a hyphen as a literal if it appears before or after a POSIX class (see below) or before or after a character type escape such as as \d or \H. However, unless the hyphen is the last character in the class, Perl outputs a warning in its warning mode, as this is most likely a user error. As PCRE2 has no facility for warning, an error is given in these cases.

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 normally include all code points between the start and end characters, inclusive. They can also be used for code points specified numerically, for example [\000-\037]. Ranges can include any characters that are valid for the current mode. In any UTF mode, the so-called "surrogate" characters (those whose code points lie between 0xd800 and 0xdfff inclusive) may not be specified explicitly by default (the PCRE2_EXTRA_ALLOW_SURROGATE_ESCAPES option disables this check). However, ranges such as [\x{d7ff}-\x{e000}], which include the surrogates, are always permitted.

There is a special case in EBCDIC environments for ranges whose end points are both specified as literal letters in the same case. For compatibility with Perl, EBCDIC code points within the range that are not letters are omitted. For example, [h-k] matches only four characters, even though the codes for h and k are 0x88 and 0x92, a range of 11 code points. However, if the range is specified numerically, for example, [\x88-\x92] or [h-\x92], all code points are included.

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 a non-UTF mode, if character tables for a French locale are in use, [\xc8-\xcb] matches accented E characters in both cases.

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, whereas [\w] includes underscore. A positive character class should be read as "something OR something OR ..." and a negative class as "NOT something AND NOT something AND NOT ...".

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, or for a special compatibility feature - see the next two sections), 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. PCRE2 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 and space
space    white space (the same as \s from PCRE2 8.34)
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The default "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). If locale-specific matching is taking place, the list of space characters may be different; there may be fewer or more of them. "Space" and \s match the same set of characters.

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. PCRE2 (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.

By default, characters with values greater than 127 do not match any of the POSIX character classes, although this may be different for characters in the range 128-255 when locale-specific matching is happening. However, if the PCRE2_UCP option is passed to **pcre2_compile()**, some of the classes are changed so that Unicode character properties are used. This is achieved by replacing certain POSIX classes with other sequences, as follows:

```
[:alnum:]  becomes  \p{Xan}
[:alpha:]  becomes  \p{L}
[:blank:]  becomes  \h
[:cntrl:]  becomes  \p{Cc}
[:digit:]  becomes  \p{Nd}
[:lower:]  becomes  \p{Ll}
[:space:]  becomes  \p{Xps}
[:upper:]  becomes  \p{Lu}
```

[:word:] becomes \p{Xwd}

Negated versions, such as [:^alpha:] use \P instead of \p. Three other POSIX classes are handled specially in UCP mode:

[:graph:] This matches characters that have glyphs that mark the page when printed. In Unicode property terms, it matches all characters with the L, M, N, P, S, or Cf properties, except for:

U+061C Arabic Letter Mark
U+180E Mongolian Vowel Separator
U+2066 - U+2069 Various "isolate"s

[:print:] This matches the same characters as [:graph:] plus space characters that are not controls, that is, characters with the Zs property.

[:punct:] This matches all characters that have the Unicode P (punctuation) property, plus those characters with code points less than 256 that have the S (Symbol) property.

The other POSIX classes are unchanged, and match only characters with code points less than 256.

COMPATIBILITY FEATURE FOR WORD BOUNDARIES

In the POSIX.2 compliant library that was included in 4.4BSD Unix, the ugly syntax [[:<:]] and [[:>:]] is used for matching "start of word" and "end of word". PCRE2 treats these items as follows:

[[:<:]] is converted to \b(?=\w)
[[:>:]] is converted to \b(?<=\w)

Only these exact character sequences are recognized. A sequence such as [a[:<:]b] provokes error for an unrecognized POSIX class name. This support is not compatible with Perl. It is provided to help migrations from other environments, and is best not used in any new patterns. Note that \b matches at the start and the end of a word (see "Simple assertions" above), and in a Perl-style pattern the preceding or following character normally shows which is wanted, without the need for the assertions that are used above in order to give exactly the POSIX behaviour.

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 group (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the group.

INTERNAL OPTION SETTING

The settings of the PCRE2_CASELESS, PCRE2_MULTILINE, PCRE2_DOTALL, PCRE2_EXTENDED, PCRE2_EXTENDED_MORE, and PCRE2_NO_AUTO_CAPTURE options can be changed from within the pattern by a sequence of letters enclosed between "(?" and ")". These options are Perl-compatible, and are described in detail in the **pcre2api** documentation. The option letters are:

i for PCRE2_CASELESS
m for PCRE2_MULTILINE
n for PCRE2_NO_AUTO_CAPTURE
s for PCRE2_DOTALL
x for PCRE2_EXTENDED

xx for PCRE2_EXTENDED_MORE

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the relevant letters with a hyphen, for example (?-im). The two "extended" options are not independent; unsetting either one cancels the effects of both of them.

A combined setting and unsetting such as (?im-sx), which sets PCRE2_CASELESS and PCRE2_MULTILINE while unsetting PCRE2_DOTALL and PCRE2_EXTENDED, is also permitted. Only one hyphen may appear in the options string. If a letter appears both before and after the hyphen, the option is unset. An empty options setting "(?)" is allowed. Needless to say, it has no effect.

If the first character following (? is a circumflex, it causes all of the above options to be unset. Thus, (?^) is equivalent to (?-imnsx). Letters may follow the circumflex to cause some options to be re-instated, but a hyphen may not appear.

The PCRE2-specific options PCRE2_DUPNAMES and PCRE2_UNGREEDY can be changed in the same way as the Perl-compatible options by using the characters J and U respectively. However, these are not unset by (?^).

When one of these option changes occurs at top level (that is, not inside group parentheses), the change applies to the remainder of the pattern that follows. An option change within a group (see below for a description of groups) affects only that part of the group that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming PCRE2_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 group. 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.

As a convenient shorthand, if any option settings are required at the start of a non-capturing group (see the next section), 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.

Note: There are other PCRE2-specific options, applying to the whole pattern, which can be set by the application when the compiling function is called. In addition, 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 are also the (*UTF) and (*UCP) leading sequences that can be used to set UTF and Unicode property modes; they are equivalent to setting the PCRE2_UTF and PCRE2_UCP options, respectively. However, the application can set the PCRE2_NEVER_UTF and PCRE2_NEVER_UCP options, which lock out the use of the (*UTF) and (*UCP) sequences.

GROUPS

Groups are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a group does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches "cataract", "caterpillar", or "cat". Without the parentheses, it would match "cataract", "erpillar" or an empty string.

2. It creates a "capture group". This means that, when the whole pattern matches, the portion of the subject string that matched the group is passed back to the caller, separately from the portion that matched the whole pattern. (This applies only to the traditional matching function; the DFA matching function does not support capturing.)

Opening parentheses are counted from left to right (starting from 1) to obtain numbers for capture groups. 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 grouping is required without capturing. If an opening parenthesis is followed by a question mark and a colon, the group does not do any capturing, and is not counted when computing the number of any subsequent capture groups. 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 capture groups is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing group, 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 group is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

DUPLICATE GROUP NUMBERS

Perl 5.10 introduced a feature whereby each alternative in a group uses the same numbers for its capturing parentheses. Such a group starts with (?| and is itself a non-capturing group. 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 parentheses that follow the whole group 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 backreference to a capture group uses the most recent value that is set for the group. The following pattern matches "abcabc" or "defdef":

```
/(?|(abc)|(def))\1/
```

In contrast, a subroutine call to a capture group always refers to the first one in the pattern with the given number. The following pattern matches "abcabc" or "defabc":

```
/(?|(abc)|(def))(?1)/
```

A relative reference such as (?-1) is no different: it is just a convenient way of computing an absolute group number.

If a condition test for a group's having matched refers to a non-unique number, the test is true if any group with that number has matched.

An alternative approach to using this "branch reset" feature is to use duplicate named groups, as described in the next section.

NAMED CAPTURE GROUPS

Identifying capture groups by number is simple, but it can be very hard to keep track of the numbers in complicated patterns. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE2 supports the naming of capture groups. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE1 introduced it at release 4.0, using the Python syntax. PCRE2 supports both the Perl and the Python syntax.

In PCRE2, a capture group can be named in one of three ways: (?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. Names may be up to 32 code units long. When PCRE2_UTF is not set, they may contain only ASCII alphanumeric characters and underscores, but must start with a non-digit. When PCRE2_UTF is set, the syntax of group names is extended to allow any Unicode letter or Unicode decimal digit. In other words, group names must match one of these patterns:

```
^[_A-Za-z][_A-Za-z0-9]*\z   when PCRE2_UTF is not set
^[_\p{L}][_\p{L}\p{Nd}]*\z  when PCRE2_UTF is set
```

References to capture groups from other parts of the pattern, such as backreferences, recursion, and conditions, can all be made by name as well as by number.

Named capture groups are allocated numbers as well as names, exactly as if the names were not present. In both PCRE2 and Perl, capture groups are primarily identified by numbers; any names are just aliases for these numbers. The PCRE2 API provides function calls for extracting the complete name-to-number translation table from a compiled pattern, as well as convenience functions for extracting captured substrings by name.

Warning: When more than one capture group has the same number, as described in the previous section, a name given to one of them applies to all of them. Perl allows identically numbered groups to have different names. Consider this pattern, where there are two capture groups, both numbered 1:

```
(?|(?<AA>aa)|(?<BB>bb))
```

Perl allows this, with both names AA and BB as aliases of group 1. Thus, after a successful match, both names yield the same value (either "aa" or "bb").

In an attempt to reduce confusion, PCRE2 does not allow the same group number to be associated with more than one name. The example above provokes a compile-time error. However, there is still scope for confusion. Consider this pattern:

```
(?|(?<AA>aa)|(bb))
```

Although the second group number 1 is not explicitly named, the name AA is still an alias for any group 1. Whether the pattern matches "aa" or "bb", a reference by name to group AA yields the matched string.

By default, a name must be unique within a pattern, except that duplicate names are permitted for groups with the same number, for example:

```
(?|(?<AA>aa)|(?<AA>bb))
```

The duplicate name constraint can be disabled by setting the PCRE2_DUPNAMES option at compile time, or by the use of (?J) within the pattern, as described in the section entitled "Internal Option Setting" above.

Duplicate names can be useful for patterns where only one instance of the named capture group 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:

```
(?J)
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capture groups, but only one is ever set after a match. The convenience functions for extracting the data by name returns the substring for the first (and in this example, the only) group of that name that matched. This saves searching to find which numbered group it was. (An alternative way of solving this problem is to use a "branch reset" group, as described in the previous section.)

If you make a backreference to a non-unique named group from elsewhere in the pattern, the groups to which the name refers are checked in the order in which they appear in the overall pattern. The first one that is set is used for the reference. For example, this pattern matches both "foofoo" and "barbar" but not "foobar" or "barfoo":

```
(?J)(?:(?<n>foo)|(?<n>bar))\k<n>
```

If you make a subroutine call to a non-unique named group, the one that corresponds to the first occurrence of the name is used. In the absence of duplicate numbers 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 capture group has matched, or to check for recursion, all groups 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 capture groups, see the **pcre2api** documentation.

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 \R escape sequence
the \X escape sequence
an escape such as \d or \pL that matches a single character
a character class
a backreference
a parenthesized group (including lookaround assertions)
a subroutine call (recursive or otherwise)
```

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, whereas

```
\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 modes, quantifiers apply to characters rather than to individual code units. Thus, for example, \x{100}{2} matches two characters, each of which is represented by a two-byte sequence in a UTF-8 string. Similarly, \X{3} matches three Unicode extended grapheme clusters, each of which may be several code units 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 capture groups that are referenced as subroutines from elsewhere in the pattern (but see also the section entitled "Defining capture groups for use by reference only" below). Except for parenthesized groups, items 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 group that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE1 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 whenever an iteration of such a group matches no characters, matching moves on to the next item in the pattern instead of repeatedly matching an empty string. This does not prevent backtracking into any of the iterations if a subsequent item fails to match.

By default, 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 PCRE2_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 group 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 PCRE2_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. PCRE2 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 PCRE2_DOTALL in order to obtain this optimization, or alternatively, using ^ to indicate anchoring explicitly.

However, there are some cases where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference 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.

Another case where implicit anchoring is not applied is when the leading .* is inside an atomic group. Once again, a match at the start may fail where a later one succeeds. Consider this pattern:

```
(?>.*?a)b
```

It matches "ab" in the subject "aab". The use of the backtracking control verbs (*PRUNE) and (*SKIP) also disable this optimization, and there is an option, PCRE2_NO_DOTSTAR_ANCHOR, to do so explicitly.

When a capture group 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 capture groups, 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 group 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
```

Perl 5.28 introduced an experimental alphabetic form starting with (* which may be easier to remember:

```
(*atomic:\d+)foo
```

This kind of parenthesized group "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 group of this type matches exactly the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic groups are not capture groups. 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 expressions, and can be nested. However, when the contents of 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 PCRE2_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 PCRE1 copied it from there. It found its way into Perl at release 5.10.

PCRE2 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. This feature can be disabled by the PCRE2_NO_AUTOPOSSESS option, or starting the pattern with (*NO_AUTO_POSSESS).

When a pattern contains an unlimited repeat inside a group 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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

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 PCRE2 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.

BACKREFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a backreference to a capture group earlier (that is, to its left) in the pattern, provided there have been that many previous capture groups.

However, if the decimal number following the backslash is less than 8, it is always taken as a backreference, and causes an error only if there are not that many capture groups in the entire pattern. In other words, the group that is referenced need not be to the left of the reference for numbers less than 8. A "forward backreference" of this type can make sense when a repetition is involved and the group to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward backreference" to a group whose number is 8 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. Other forms of backreferencing do not suffer from this restriction. In particular, there is no problem when named capture groups are used (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence. This escape must be followed by a signed or unsigned 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 signed 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 capture group before \g, that is, is it equivalent to \2 in this example. 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.

The sequence \g{+1} is a reference to the next capture group. This kind of forward reference can be useful in patterns that repeat. Perl does not support the use of + in this way.

A backreference matches whatever actually most recently matched the capture group in the current subject string, rather than anything at all that matches the group (see "Groups as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the backreference, 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 capture group is matched caselessly.

There are several different ways of writing backreferences to named capture groups. 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 backreference 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 capture group that is referenced by name may appear in the pattern before or after the reference.

There may be more than one backreference to the same group. If a group has not actually been used in a particular match, backreferences 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 PCRE2_MATCH_UNSET_BACKREF option is set at compile time, a backreference to an unset value matches an empty string.

Because there may be many capture groups in a pattern, all digits following a backslash are taken as part of a potential backreference number. If the pattern continues with a digit character, some delimiter must be used to terminate the backreference. If the PCRE2_EXTENDED or PCRE2_EXTENDED_MORE option is set, this can be white space. Otherwise, the \g{} syntax or an empty comment (see "Comments" below) can be used.

Recursive backreferences

A backreference that occurs inside the group to which it refers fails when the group is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated groups. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the group, the backreference 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 backreference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

For versions of PCRE2 less than 10.25, backreferences of this type used to cause the group that they reference to be treated as an atomic group. This restriction no longer applies, and backtracking into such groups can occur as normal.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ^ and $ are described above.

More complicated assertions are coded as parenthesized groups. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it, and in each case an assertion may be positive (must match for the assertion to be true) or negative (must not match for the assertion to be true). An assertion group is matched in the normal way, and if it is true, matching continues after it, but with the matching position in the subject string reset to what it was before the assertion was processed.

The Perl-compatible lookaround assertions are atomic. If an assertion is true, but there is a subsequent matching failure, there is no backtracking into the assertion. However, there are some cases where non-atomic assertions can be useful. PCRE2 has some support for these, described in the section entitled "Non-atomic assertions" below, but they are not Perl-compatible.

A lookaround assertion may appear as the condition in a conditional group (see below). In this case, the result of matching the assertion determines which branch of the condition is followed.

Assertion groups are not capture groups. If an assertion contains capture groups within it, these are counted for the purposes of numbering the capture groups in the whole pattern. Within each branch of an assertion, locally captured substrings may be referenced in the usual way. For example, a sequence such as (.)\g{-1} can be used to check that two adjacent characters are the same.

When a branch within an assertion fails to match, any substrings that were captured are discarded (as happens with any pattern branch that fails to match). A negative assertion is true only when all its branches fail to match; this means that no captured substrings are ever retained after a successful negative assertion. When an assertion contains a matching branch, what happens depends on the type of assertion.

For a positive assertion, internally captured substrings in the successful branch are retained, and matching continues with the next pattern item after the assertion. For a negative assertion, a matching branch means that the assertion is not true. If such an assertion is being used as a condition in a conditional group (see below), captured substrings are retained, because matching continues with the "no" branch of the condition. For other failing negative assertions, control passes to the previous backtracking point, thus discarding any captured strings within the assertion.

Most assertion groups may be repeated; though it makes no sense to assert the same thing several times, the side effect of capturing in positive assertions may occasionally be useful. However, an assertion that forms the condition for a conditional group may not be quantified. PCRE2 used to restrict the repetition of assertions, but from release 10.35 the only restriction is that an unlimited maximum repetition is changed to be one more than the minimum. For example, {3,} is treated as {3,4}.

Alphabetic assertion names

Traditionally, symbolic sequences such as (?= and (?<= have been used to specify lookaround assertions. Perl 5.28 introduced some experimental alphabetic alternatives which might be easier to remember. They all start with (* instead of (? and must be written using lower case letters. PCRE2 supports the following synonyms:

(*positive_lookahead: or (*pla: is the same as (?=
(*negative_lookahead: or (*nla: is the same as (?!

(*positive_lookbehind: or (*plb: is the same as (?<=
(*negative_lookbehind: or (*nlb: is the same as (?<!

For example, (*pla:foo) is the same assertion as (?=foo). In the following sections, the various assertions are described using the original symbolic forms.

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 backtracking control verb (*FAIL) or (*F) is 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, 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 PCRE2 if rewritten to use two top-level branches:

(?<=abc|abde)

In some cases, the 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.

In UTF-8 and UTF-16 modes, PCRE2 does not allow the \C escape (which matches a single code unit even in a UTF 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 code units, are never permitted in lookbehinds.

"Subroutine" calls (see below) such as (?2) or (?&X) are permitted in lookbehinds, as long as the called capture group matches a fixed-length string. However, recursion, that is, a "subroutine" call into a group that is already active, is not supported.

Perl does not support backreferences in lookbehinds. PCRE2 does support them, but only if certain conditions are met. The PCRE2_MATCH_UNSET_BACKREF option must not be set, there must be no use of (?| in the pattern (it creates duplicate group numbers), and if the backreference is by name, the name must be unique. Of course, the referenced group must itself match a fixed length substring. The following pattern matches words containing at least two characters that begin and end with the same character:

```
\b(\w)\w++(?<=\1)
```

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, PCRE2 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 because of the possessive quantifier; 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 "123abcfoo". 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".

NON-ATOMIC ASSERTIONS

The traditional Perl-compatible lookaround assertions are atomic. That is, if an assertion is true, but there is a subsequent matching failure, there is no backtracking into the assertion. However, there are some cases where non-atomic positive assertions can be useful. PCRE2 provides these using the following syntax:

```
(*non_atomic_positive_lookahead:  or (*napla: or (?*
(*non_atomic_positive_lookbehind: or (*naplb: or (?<*
```

Consider the problem of finding the right-most word in a string that also appears earlier in the string, that is, it must appear at least twice in total. This pattern returns the required result as captured substring 1:

```
^(?x)(*napla: .* \b(\w++)) (?> .*? \b\1\b ){2}
```

For a subject such as "word1 word2 word3 word2 word3 word4" the result is "word3". How does it work? At the start, ^(?x) anchors the pattern and sets the "x" option, which causes white space (introduced for readability) to be ignored. Inside the assertion, the greedy .* at first consumes the entire string, but then has to backtrack until the rest of the assertion can match a word, which is captured by group 1. In other words, when the assertion first succeeds, it captures the right-most word in the string.

The current matching point is then reset to the start of the subject, and the rest of the pattern match checks for two occurrences of the captured word, using an ungreedy .*? to scan from the left. If this succeeds, we are done, but if the last word in the string does not occur twice, this part of the pattern fails. If a traditional atomic lookhead (?= or (*pla: had been used, the assertion could not be re-entered, and the whole match would fail. The pattern would succeed only if the very last word in the subject was found twice.

Using a non-atomic lookahead, however, means that when the last word does not occur twice in the string, the lookahead can backtrack and find the second-last word, and so on, until either the match succeeds, or all words have been tested.

Two conditions must be met for a non-atomic assertion to be useful: the contents of one or more capturing groups must change after a backtrack into the assertion, and there must be a backreference to a changed group later in the pattern. If this is not the case, the rest of the pattern match fails exactly as before because nothing has changed, so using a non-atomic assertion just wastes resources.

There is one exception to backtracking into a non-atomic assertion. If an (*ACCEPT) control verb is triggered, the assertion succeeds atomically. That is, a subsequent match failure cannot backtrack into the assertion.

Non-atomic assertions are not supported by the alternative matching function **pcre2_dfa_match()**. They are supported by JIT, but only if they do not contain any control verbs such as (*ACCEPT). (This may change in future). Note that assertions that appear as conditions for conditional groups (see below) must be atomic.

SCRIPT RUNS

In concept, a script run is a sequence of characters that are all from the same Unicode script such as Latin or Greek. However, because some scripts are commonly used together, and because some diacritical and other marks are used with multiple scripts, it is not that simple. There is a full description of the rules that PCRE2 uses in the section entitled "Script Runs" in the **pcre2unicode** documentation.

If part of a pattern is enclosed between (*script_run: or (*sr: and a closing parenthesis, it fails if the sequence of characters that it matches are not a script run. After a failure, normal backtracking occurs. Script runs can be used to detect spoofing attacks using characters that look the same, but are from different scripts. The string "paypal.com" is an infamous example, where the letters could be a mixture of Latin and Cyrillic. This pattern ensures that the matched characters in a sequence of non-spaces that follow white space are a script run:

```
\s+(*sr:\S+)
```

To be sure that they are all from the Latin script (for example), a lookahead can be used:

```
\s+(?=\p{Latin})(*sr:\S+)
```

This works as long as the first character is expected to be a character in that script, and not (for example) punctuation, which is allowed with any script. If this is not the case, a more creative lookahead is needed. For example, if digits, underscore, and dots are permitted at the start:

```
\s+(?=[0-9_.]*\p{Latin})(*sr:\S+)
```

In many cases, backtracking into a script run pattern fragment is not desirable. The script run can employ an atomic group to prevent this. Because this is a common requirement, a shorthand notation is provided by (*atomic_script_run: or (*asr:

```
(*asr:...) is the same as (*sr:(?>...))
```

Note that the atomic group is inside the script run. Putting it outside would not prevent backtracking into the script run pattern.

Support for script runs is not available if PCRE2 is compiled without Unicode support. A compile-time error is given if any of the above constructs is encountered. Script runs are not supported by the alternate matching function, **pcre2_dfa_match()** because they use the same mechanism as capturing parentheses.

Warning: The (*ACCEPT) control verb (see below) should not be used within a script run group, because it causes an immediate exit from the group, bypassing the script run checking.

CONDITIONAL GROUPS

It is possible to cause the matching process to obey a pattern fragment conditionally or to choose between two alternative fragments, depending on the result of an assertion, or whether a specific capture group has already been matched. The two possible forms of conditional group 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. An absent no-pattern is equivalent to an empty string (it always matches). If there are more than two alternatives in the group, a compile-time error occurs. Each of the two alternatives may itself contain nested groups of any form, including conditional groups; the restriction to two alternatives applies only at the level of the condition itself. This pattern fragment is an example where the alternatives are complex:

```
(?(1) (A|B|C) | (D | (?(2)E|F) | E) )
```

There are five kinds of condition: references to capture groups, references to recursion, two pseudo-conditions called DEFINE and VERSION, and assertions.

Checking for a used capture group by number

If the text between the parentheses consists of a sequence of digits, the condition is true if a capture group of that number has previously matched. If there is more than one capture group with the same number (see the earlier section about duplicate group numbers), the condition is true if any of them have matched. An alternative notation is to precede the digits with a plus or minus sign. In this case, the group number is relative rather than absolute. The most recently opened capture group can be referenced by (?(-1), the next most recent by (?(-2), and so on. Inside loops it can also make sense to refer to subsequent groups. The next capture group can be referenced as (?(+1), and so on. (The value zero in any of these forms is not used; it provokes a compile-time error.)

Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE2_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 group that tests whether or not the first capture group matched. If it 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 conditional group 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 capture group by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used capture group by name. For compatibility with earlier versions of PCRE1, which had this facility before Perl, the syntax (?(name)...) is also recognized. Note, however, that undelimited names consisting of the letter R followed by digits are ambiguous (see the following section). Rewriting the above example to use a named group gives this:

```
(?<OPEN> \( )?    [^()]+    (?(<OPEN>) \) )
```

If the name used in a condition of this kind is a duplicate, the test is applied to all groups of the same name, and is true if any one of them has matched.

Checking for pattern recursion

"Recursion" in this sense refers to any subroutine-like call from one part of the pattern to another, whether or not it is actually recursive. See the sections entitled "Recursive patterns" and "Groups as subroutines" below for details of recursion and subroutine calls.

If a condition is the string (R), and there is no capture group with the name R, the condition is true if matching is currently in a recursion or subroutine call to the whole pattern or any capture group. If digits follow the letter R, and there is no group with that name, the condition is true if the most recent call is into a group with the given number, which must exist somewhere in the overall pattern. This is a contrived example that is equivalent to a+b:

```
((?(R1)a+|(?1)b))
```

However, in both cases, if there is a capture group with a matching name, the condition tests for its being set, as described in the section above, instead of testing for recursion. For example, creating a group with the name R1 by adding (?<R1>) to the above pattern completely changes its meaning.

If a name preceded by ampersand follows the letter R, for example:

```
(?(R&name)...)
```

the condition is true if the most recent recursion is into a group of that name (which must exist within the pattern).

This condition does not check the entire recursion stack. It tests only the current level. If the name used in a condition of this kind is a duplicate, the test is applied to all groups 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.

Defining capture groups for use by reference only

If the condition is the string (DEFINE), the condition is always false, even if there is a group with the name DEFINE. In this case, there may be only one alternative in the rest of the conditional group. 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 such as "192.168.23.245" could be written like this (ignore white space 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.

Checking the PCRE2 version

Programs that link with a PCRE2 library can check the version by calling **pcre2_config()** with appropriate arguments. Users of applications that do not have access to the underlying code cannot do this. A special "condition" called VERSION exists to allow such users to discover which version of PCRE2 they are dealing with by using this condition to match a string such as "yesno". VERSION must be followed either by "=" or ">=" and a version number. For example:

```
(?(VERSION>=10.4)yes|no)
```

This pattern matches "yes" if the PCRE2 version is greater or equal to 10.4, or "no" otherwise. The fractional part of the version number may not contain more than two digits.

Assertion conditions

If the condition is not in any of the above formats, it must be a parenthesized assertion. This may be a positive or negative lookahead or lookbehind assertion. However, it must be a traditional atomic assertion, not one of the PCRE2-specific non-atomic assertions.

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.

When an assertion that is a condition contains capture groups, any capturing that occurs in a matching branch is retained afterwards, for both positive and negative assertions, because matching always continues after the assertion, whether it succeeds or fails. (Compare non-conditional assertions, for which captures are retained only for positive assertions that succeed.)

COMMENTS

There are two ways of including comments in patterns that are processed by PCRE2. In both cases, the start of the comment must not be in a character class, nor in the middle of any other sequence of related characters such as (?: or a group name or number. The characters that make up a comment play no part in the pattern matching.

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. If the PCRE2_EXTENDED or PCRE2_EXTENDED_MORE option is set, an unescaped # character also introduces a comment, which in this case continues to immediately after the next newline character or character sequence in the pattern. Which characters are interpreted as newlines is controlled by an option passed to the compiling function or by a special sequence at the start of the pattern, as described in the section entitled "Newline conventions" above. Note that the end of this type of comment is a literal newline sequence in the pattern; escape sequences that happen to represent a newline do not count. For example, consider this pattern when PCRE2_EXTENDED is set, and the default newline convention (a single linefeed character) is in force:

```
abc #comment \n still comment
```

On encountering the # character, **pcre2_compile()** skips along, looking for a newline in the pattern. The sequence \n is still literal at this stage, so it does not terminate the comment. Only an actual character with the code value 0x0a (the default newline) does so.

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, PCRE2 cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual capture group recursion. After its introduction in PCRE1 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 subroutine call of the capture group of the given number, provided that it occurs inside that group. (If not, it is a non-recursive 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 PCRE2 pattern solves the nested parentheses problem (assume the PCRE2_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. 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.

Be aware however, that if duplicate capture group numbers are in use, relative references refer to the earliest group with the appropriate number. Consider, for example:

```
(?|(a)|(b)) (c) (?-2)
```

The first two capture groups (a) and (b) are both numbered 1, and group (c) is number 2. When the reference (?-2) is encountered, the second most recently opened parentheses has the number 1, but it is the first such group (the (a) group) to which the recursion refers. This would be the same if an absolute reference (?1) was used. In other words, relative references are just a shorthand for computing a group number.

It is also possible to refer to subsequent capture groups, 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 non-recursive subroutine calls, as described in the next section.

An alternative approach is to use named parentheses. The Perl syntax for this is (?&name); PCRE1's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

```
(?<pn> \( ( [^()]++ | (?&pn) )* \) )
```

If there is more than one group with the same name, the earliest one is used.

The 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

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

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 **pcre2callout** 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 capture group is not matched at the top level, its final captured value is unset, even if it was (temporarily) set at a deeper level during the matching process.

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 group, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

Differences in recursion processing between PCRE2 and Perl

Some former differences between PCRE2 and Perl no longer exist.

Before release 10.30, recursion processing in PCRE2 differed from Perl in that a recursive subroutine call was always treated as an atomic group. That is, once it had matched some of the subject string, it was never re-entered, even if it contained untried alternatives and there was a subsequent matching failure. (Historical note: PCRE implemented recursion before Perl did.)

Starting with release 10.30, recursive subroutine calls are no longer treated as atomic. That is, they can be re-entered to try unused alternatives if there is a matching failure later in the pattern. This is now compatible with the way Perl works. If you want a subroutine call to be atomic, you must explicitly enclose it in an atomic group.

Supporting backtracking into recursions simplifies certain types of recursive pattern. For example, this pattern matches palindromic strings:

```
^((.)(?1)\2|.?)$
```

The second branch in the group matches a single central character in the palindrome when there are an odd number of characters, or nothing when there are an even number of characters, but in order to work it has to be able to try the second case when the rest of the pattern match fails. 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*+.?)\W*+$
```

If run with the PCRE2_CASELESS option, this pattern matches phrases such as "A man, a plan, a canal: Panama!". Note the use of the possessive quantifier *+ to avoid backtracking into sequences of non-word characters. Without this, PCRE2 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.

Another way in which PCRE2 and Perl used to differ in their recursion processing is in the handling of captured values. Formerly in Perl, when a group was called recursively or as a subroutine (see the next section), it had no access to any values that were captured outside the recursion, whereas in PCRE2 these values can be referenced. Consider this pattern:

```
^(.)(\1|a(?2))
```

This pattern matches "bab". The first capturing parentheses match "b", then in the second group, when the backreference \1 fails to match "b", the second alternative matches "a" and then recurses. In the recursion, \1 does now match "b" and so the whole match succeeds. This match used to fail in Perl, but in later versions (I tried 5.024) it now works.

GROUPS AS SUBROUTINES

If the syntax for a recursive group call (either by number or by name) is used outside the parentheses to which it refers, it operates a bit like a subroutine in a programming language. More accurately, PCRE2 treats the referenced group as an independent subpattern which it tries to match at the current matching position. The called group may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

```
(...(absolute)...)...(?2)...
(...(relative)...)...(?-1)...
(...(?+1)...(relative)...
```

An earlier example pointed out that the pattern

```
(sens|respons)e and \1ibility
```

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 recursions, subroutine calls used to be treated as atomic, but this changed at PCRE2 release 10.30, so backtracking into subroutine calls can now occur. However, any capturing parentheses that are set during the subroutine call revert to their previous values afterwards.

Processing options such as case-independence are fixed when a group is defined, so if it is used as a subroutine, such options 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 group.

The behaviour of backtracking control verbs in groups when called as subroutines is described in the section entitled "Backtracking verbs in subroutines" below.

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 calling a group as a subroutine, possibly recursively. Here are two of the examples used above, rewritten using this syntax:

```
(?<pn> \( ( (?>[^()]+) | \g<pn> )* \) )
(sens|respons)e and \g'1'ibility
```

PCRE2 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 backreference; 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.

PCRE2 provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE2 provides an external function by putting its entry point in a match context using the function **pcre2_set_callout()**, and then passing that context to **pcre2_match()** or **pcre2_dfa_match()**. If no match context is passed, or if the callout entry point is set to NULL, callouts are disabled.

Within a regular expression, (?C<arg>) indicates a point at which the external function is to be called. There are two kinds of callout: those with a numerical argument and those with a string argument. (?C) on its own with no argument is treated as (?C0). A numerical argument allows the application to distinguish between different callouts. String arguments were added for release 10.20 to make it possible for script languages that use PCRE2 to embed short scripts within patterns in a similar way to Perl.

During matching, when PCRE2 reaches a callout point, the external function is called. It is provided with the number or string argument of the callout, the position in the pattern, and one item of data that is also set in the match block. The callout function may cause matching to proceed, to backtrack, or to fail.

By default, PCRE2 implements a number of optimizations at matching time, and one side-effect is that sometimes callouts are skipped. If you need all possible callouts to happen, you need to set options that disable the relevant optimizations. More details, including a complete description of the programming interface to the callout function, are given in the **pcre2callout** documentation.

Callouts with numerical arguments

If you just want to have a means of identifying different callout points, put a number less than 256 after the letter C. For example, this pattern has two callout points:

```
(?C1)abc(?C2)def
```

If the PCRE2_AUTO_CALLOUT flag is passed to **pcre2_compile()**, numerical callouts are automatically installed before each item in the pattern. They are all numbered 255. If there is a conditional group in the pattern whose condition is an assertion, an additional callout is inserted just before the condition. An explicit callout may also be set at this position, as in this example:

```
(?(?C9)(?=a)abc|def)
```

Note that this applies only to assertion conditions, not to other types of condition.

Callouts with string arguments

A delimited string may be used instead of a number as a callout argument. The starting delimiter must be one of ‘ ’ " ^ % # $ { and the ending delimiter is the same as the start, except for {, where the ending delimiter is }. If the ending delimiter is needed within the string, it must be doubled. For example:

```
(?C’ab ’’c’’ d’)xyz(?C{any text})pqr
```

The doubling is removed before the string is passed to the callout function.

BACKTRACKING CONTROL

There are a number of special "Backtracking Control Verbs" (to use Perl’s terminology) that modify the behaviour of backtracking during matching. They are generally of the form (*VERB) or (*VERB:NAME). Some verbs take either form, and may behave differently depending on whether or not a name argument is present. The names are not required to be unique within the pattern.

By default, for compatibility with Perl, a name is any sequence of characters that does not include a closing parenthesis. The name is not processed in any way, and it is not possible to include a closing parenthesis in the name. This can be changed by setting the PCRE2_ALT_VERBNAMES option, but the result is no longer Perl-compatible.

When PCRE2_ALT_VERBNAMES is set, backslash processing is applied to verb names and only an unescaped closing parenthesis terminates the name. However, the only backslash items that are permitted are \Q, \E, and sequences such as \x{100} that define character code points. Character type escapes such as \d are faulted.

A closing parenthesis can be included in a name either as \) or between \Q and \E. In addition to backslash processing, if the PCRE2_EXTENDED or PCRE2_EXTENDED_MORE option is also set, unescaped whitespace in verb names is skipped, and #-comments are recognized, exactly as in the rest of the pattern. PCRE2_EXTENDED and PCRE2_EXTENDED_MORE do not affect verb names unless PCRE2_ALT_VERBNAMES is also set.

The maximum length of a name is 255 in the 8-bit library and 65535 in the 16-bit and 32-bit libraries. If the name is empty, that is, if the closing parenthesis immediately follows the colon, the effect is as if the colon were not there. Any number of these verbs may occur in a pattern. Except for (*ACCEPT), they may not be quantified.

Since these verbs are specifically related to backtracking, most of them can be used only when the pattern is to be matched using the traditional matching function, because that uses a backtracking algorithm. With the exception of (*FAIL), which behaves like a failing negative assertion, the backtracking control verbs cause an error if encountered by the DFA matching function.

The behaviour of these verbs in repeated groups, assertions, and in capture groups called as subroutines (whether or not recursively) is documented below.

Optimizations that affect backtracking verbs

PCRE2 contains some optimizations that are used to speed up matching by running some checks at the start of each match attempt. For example, it may know the minimum length of matching subject, or that a particular character must be present. When one of these optimizations bypasses the running of a match, any included backtracking verbs will not, of course, be processed. You can suppress the start-of-match optimizations by setting the PCRE2_NO_START_OPTIMIZE option when calling **pcre2_compile()**, or by starting the pattern with (*NO_START_OPT). There is more discussion of this option in the section entitled "Compiling a pattern" in the **pcre2api** documentation.

Experiments with Perl suggest that it too has similar optimizations, and like PCRE2, turning them off can change the result of a match.

Verbs that act immediately

The following verbs act as soon as they are encountered.

(*ACCEPT) or (*ACCEPT:NAME)

This verb causes the match to end successfully, skipping the remainder of the pattern. However, when it is inside a capture group that is called as a subroutine, only that group is ended successfully. Matching then continues at the outer level. If (*ACCEPT) in triggered in a positive assertion, the assertion succeeds; in a negative assertion, the assertion fails.

If (*ACCEPT) is inside capturing parentheses, the data so far is captured. 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.

(*ACCEPT) is the only backtracking verb that is allowed to be quantified because an ungreedy

quantification with a minimum of zero acts only when a backtrack happens. Consider, for example,

```
(A(*ACCEPT)??B)C
```

where A, B, and C may be complex expressions. After matching "A", the matcher processes "BC"; if that fails, causing a backtrack, (*ACCEPT) is triggered and the match succeeds. In both cases, all but C is captured. Whereas (*COMMIT) (see below) means "fail on backtrack", a repeated (*ACCEPT) of this type means "succeed on backtrack".

Warning: (*ACCEPT) should not be used within a script run group, because it causes an immediate exit from the group, bypassing the script run checking.

```
(*FAIL) or (*FAIL:NAME)
```

This verb causes a matching failure, forcing backtracking to occur. It may be abbreviated to (*F). 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 PCRE2. 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).

(*ACCEPT:NAME) and (*FAIL:NAME) behave the same as (*MARK:NAME)(*ACCEPT) and (*MARK:NAME)(*FAIL), respectively, that is, a (*MARK) is recorded just before the verb acts.

Recording which path was taken

There is one verb whose main purpose is to track how a match was arrived at, though it also has a secondary use in conjunction with advancing the match starting point (see (*SKIP) below).

```
(*MARK:NAME) or (*:NAME)
```

A name is always required with this verb. For all the other backtracking control verbs, a NAME argument is optional.

When a match succeeds, the name of the last-encountered mark name on the matching path is passed back to the caller as described in the section entitled "Other information about the match" in the **pcre2api** documentation. This applies to all instances of (*MARK) and other verbs, including those inside assertions and atomic groups. However, there are differences in those cases when (*MARK) is used in conjunction with (*SKIP) as described below.

The mark name that was last encountered on the matching path is passed back. A verb without a NAME argument is ignored for this purpose. Here is an example of **pcre2test** output, where the "mark" modifier requests the retrieval and outputting of (*MARK) data:

```
  re> /X(*MARK:A)Y|X(*MARK:B)Z/mark
data> XY
 0: XY
MK: A
XZ
 0: XZ
MK: B
```

The (*MARK) name is tagged with "MK:" in this output, and in this example it indicates which of the two alternatives matched. This is a more efficient way of obtaining this information than putting each alternative

in its own capturing parentheses.

If a verb with a name is encountered in a positive assertion that is true, the name is recorded and passed back if it is the last-encountered. This does not happen for negative assertions or failing positive assertions.

After a partial match or a failed match, the last encountered name in the entire match process is returned. For example:

```
  re> /X(*MARK:A)Y|X(*MARK:B)Z/mark
data> XP
No match, mark = B
```

Note that in this unanchored example the mark is retained from the match attempt that started at the letter "X" in the subject. Subsequent match attempts starting at "P" and then with an empty string do not get as far as the (*MARK) item, but nevertheless do not reset it.

If you are interested in (*MARK) values after failed matches, you should probably set the PCRE2_NO_START_OPTIMIZE option (see above) to ensure that the match is always attempted.

Verbs that act after backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is a subsequent match failure, causing a backtrack to the verb, a failure is forced. That is, backtracking cannot pass to the left of the verb. However, when one of these verbs appears inside an atomic group or in a lookaround assertion that is true, its effect is confined to that group, because once the group has been matched, there is never any backtracking into it. Backtracking from beyond an assertion or an atomic group ignores the entire group, and seeks a preceding backtracking point.

These verbs differ in exactly what kind of failure occurs when backtracking reaches them. The behaviour described below is what happens when the verb is not in a subroutine or an assertion. Subsequent sections cover these special cases.

(*COMMIT) or (*COMMIT:NAME)

This verb causes the whole match to fail outright if there is a later matching failure that causes backtracking to reach it. Even if the pattern is unanchored, no further attempts to find a match by advancing the starting point take place. If (*COMMIT) is the only backtracking verb that is encountered, once it has been passed **pcre2_match()** 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."

The behaviour of (*COMMIT:NAME) is not the same as (*MARK:NAME)(*COMMIT). It is like (*MARK:NAME) in that the name is remembered for passing back to the caller. However, (*SKIP:NAME) searches only for names that are set with (*MARK), ignoring those set by any of the other backtracking verbs.

If there is more than one backtracking verb in a pattern, a different one that follows (*COMMIT) may be triggered first, so merely passing (*COMMIT) during a match does not always guarantee that a match must be at this starting point.

Note that (*COMMIT) at the start of a pattern is not the same as an anchor, unless PCRE2’s start-of-match optimizations are turned off, as shown in this output from **pcre2test**:

```
  re> /(*COMMIT)abc/
data> xyzabc
 0: abc
```

```
data>
re> /(*COMMIT)abc/no_start_optimize
data> xyzabc
No match
```

For the first pattern, PCRE2 knows that any match must start with "a", so the optimization skips along the subject to "a" before applying the pattern to the first set of data. The match attempt then succeeds. The second pattern disables the optimization that skips along to the first character. The pattern is now applied starting at "x", and so the (*COMMIT) causes the match to fail without trying any other starting points.

(*PRUNE) or (*PRUNE:NAME)

This verb causes the match to fail at the current starting position in the subject if there is a later matching failure that causes backtracking to reach it. 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), before it is reached, 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. In an anchored pattern (*PRUNE) has the same effect as (*COMMIT).

The behaviour of (*PRUNE:NAME) is not the same as (*MARK:NAME)(*PRUNE). It is like (*MARK:NAME) in that the name is remembered for passing back to the caller. However, (*SKIP:NAME) searches only for names set with (*MARK), ignoring those set by other backtracking verbs.

(*SKIP)

This verb, when given without a name, 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 if there is a later mismatch. 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".

If (*SKIP) is used to specify a new starting position that is the same as the starting position of the current match, or (by being inside a lookbehind) earlier, the position specified by (*SKIP) is ignored, and instead the normal "bumpalong" occurs.

(*SKIP:NAME)

When (*SKIP) has an associated name, its behaviour is modified. When such a (*SKIP) is triggered, the previous path through the pattern is searched for the most recent (*MARK) that has the same name. If one is found, the "bumpalong" advance is to the subject position that corresponds to that (*MARK) instead of to where (*SKIP) was encountered. If no (*MARK) with a matching name is found, the (*SKIP) is ignored.

The search for a (*MARK) name uses the normal backtracking mechanism, which means that it does not see (*MARK) settings that are inside atomic groups or assertions, because they are never re-entered by backtracking. Compare the following **pcre2test** examples:

```
  re> /a(?>(*MARK:X))(*SKIP:X)(*F)|(.)/
data: abc
```

```
 0: a
 1: a
data:
  re> /a(?:(*MARK:X))(*SKIP:X)(*F)|(.)/
data: abc
 0: b
 1: b
```

In the first example, the (*MARK) setting is in an atomic group, so it is not seen when (*SKIP:X) triggers, causing the (*SKIP) to be ignored. This allows the second branch of the pattern to be tried at the first character position. In the second example, the (*MARK) setting is not in an atomic group. This allows (*SKIP:X) to find the (*MARK) when it backtracks, and this causes a new matching attempt to start at the second character. This time, the (*MARK) is never seen because "a" does not match "b", so the matcher immediately jumps to the second branch of the pattern.

Note that (*SKIP:NAME) searches only for names set by (*MARK:NAME). It ignores names that are set by other backtracking verbs.

(*THEN) or (*THEN:NAME)

This verb causes a skip to the next innermost alternative when backtracking reaches it. That is, it cancels any further backtracking within the current alternative. 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 that succeeds and BAR fails, COND3 is tried. If subsequently BAZ fails, there are no more alternatives, so there is a backtrack to whatever came before the entire group. If (*THEN) is not inside an alternation, it acts like (*PRUNE).

The behaviour of (*THEN:NAME) is not the same as (*MARK:NAME)(*THEN). It is like (*MARK:NAME) in that the name is remembered for passing back to the caller. However, (*SKIP:NAME) searches only for names set with (*MARK), ignoring those set by other backtracking verbs.

A group that does not contain a | character is just a part of the enclosing alternative; it is not a nested alternation with only one alternative. The effect of (*THEN) extends beyond such a group to the enclosing alternative. Consider this pattern, where A, B, etc. are complex pattern fragments that do not contain any | characters at this level:

```
A (B(*THEN)C) | D
```

If A and B are matched, but there is a failure in C, matching does not backtrack into A; instead it moves to the next alternative, that is, D. However, if the group containing (*THEN) is given an alternative, it behaves differently:

```
A (B(*THEN)C | (*FAIL)) | D
```

The effect of (*THEN) is now confined to the inner group. After a failure in C, matching moves to (*FAIL), which causes the whole group to fail because there are no more alternatives to try. In this case, matching does backtrack into A.

Note that a conditional group is not considered as having two alternatives, because only one is ever used. In other words, the | character in a conditional group has a different meaning. Ignoring white space, consider:

```
^.*? (?(?=a) a | b(*THEN)c )
```

If the subject is "ba", this pattern does not match. Because .*? is ungreedy, it initially matches zero characters. The condition (?=a) then fails, the character "b" is matched, but "c" is not. At this point, matching does not backtrack to .*? as might perhaps be expected from the presence of the | character. The conditional group is part of the single alternative that comprises the whole pattern, and so the match fails. (If there was a backtrack into .*?, allowing it to match "b", the match would succeed.)

The verbs just described provide four different "strengths" of control when subsequent matching fails. (*THEN) is the weakest, carrying on the match at the next alternative. (*PRUNE) comes next, failing the match at the current starting position, but allowing an advance to the next character (for an unanchored pattern). (*SKIP) is similar, except that the advance may be more than one character. (*COMMIT) is the strongest, causing the entire match to fail.

More than one backtracking verb

If more than one backtracking verb is present in a pattern, the one that is backtracked onto first acts. For example, consider this pattern, where A, B, etc. are complex pattern fragments:

```
(A(*COMMIT)B(*THEN)C|ABD)
```

If A matches but B fails, the backtrack to (*COMMIT) causes the entire match to fail. However, if A and B match, but C fails, the backtrack to (*THEN) causes the next alternative (ABD) to be tried. This behaviour is consistent, but is not always the same as Perl's. It means that if two or more backtracking verbs appear in succession, all the the last of them has no effect. Consider this example:

```
...(*COMMIT)(*PRUNE)...
```

If there is a matching failure to the right, backtracking onto (*PRUNE) causes it to be triggered, and its action is taken. There can never be a backtrack onto (*COMMIT).

Backtracking verbs in repeated groups

PCRE2 sometimes differs from Perl in its handling of backtracking verbs in repeated groups. For example, consider:

```
/(a(*COMMIT)b)+ac/
```

If the subject is "abac", Perl matches unless its optimizations are disabled, but PCRE2 always fails because the (*COMMIT) in the second repeat of the group acts.

Backtracking verbs in assertions

(*FAIL) in any assertion has its normal effect: it forces an immediate backtrack. The behaviour of the other backtracking verbs depends on whether or not the assertion is standalone or acting as the condition in a conditional group.

(*ACCEPT) in a standalone positive assertion causes the assertion to succeed without any further processing; captured strings and a mark name (if set) are retained. In a standalone negative assertion, (*ACCEPT) causes the assertion to fail without any further processing; captured substrings and any mark name are discarded.

If the assertion is a condition, (*ACCEPT) causes the condition to be true for a positive assertion and false for a negative one; captured substrings are retained in both cases.

The remaining verbs act only when a later failure causes a backtrack to reach them. This means that, for the Perl-compatible assertions, their effect is confined to the assertion, because Perl lookaround assertions are atomic. A backtrack that occurs after such an assertion is complete does not jump back into the assertion. Note in particular that a (*MARK) name that is set in an assertion is not "seen" by an instance of (*SKIP:NAME) later in the pattern.

PCRE2 now supports non-atomic positive assertions, as described in the section entitled "Non-atomic assertions" above. These assertions must be standalone (not used as conditions). They are not Perl-compatible. For these assertions, a later backtrack does jump back into the assertion, and therefore verbs such as (*COMMIT) can be triggered by backtracks from later in the pattern.

The effect of (*THEN) is not allowed to escape beyond an assertion. If there are no more branches to try, (*THEN) causes a positive assertion to be false, and a negative assertion to be true.

The other backtracking verbs are not treated specially if they appear in a standalone positive assertion. In a conditional positive assertion, backtracking (from within the assertion) into (*COMMIT), (*SKIP), or (*PRUNE) causes the condition to be false. However, for both standalone and conditional negative assertions, backtracking into (*COMMIT), (*SKIP), or (*PRUNE) causes the assertion to be true, without considering any further alternative branches.

Backtracking verbs in subroutines

These behaviours occur whether or not the group is called recursively.

(*ACCEPT) in a group called as a subroutine causes the subroutine match to succeed without any further processing. Matching then continues after the subroutine call. Perl documents this behaviour. Perl's treatment of the other verbs in subroutines is different in some cases.

(*FAIL) in a group called as a subroutine has its normal effect: it forces an immediate backtrack.

(*COMMIT), (*SKIP), and (*PRUNE) cause the subroutine match to fail when triggered by being backtracked to in a group called as a subroutine. There is then a backtrack at the outer level.

(*THEN), when triggered, skips to the next alternative in the innermost enclosing group that has alternatives (its normal behaviour). However, if there is no such group within the subroutine's group, the subroutine match fails and there is a backtrack at the outer level.

SEE ALSO

pcre2api(3), **pcre2callout**(3), **pcre2matching**(3), **pcre2syntax**(3), **pcre2**(3).

AUTHOR

Philip Hazel
Retired from University Computing Service
Cambridge, England.

REVISION

Last updated: 30 August 2021