NAME

PCRE - Perl-compatible regular expressions

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions that are supported by PCRE are described in detail below. There is a quick-reference syntax summary in the **pcresyntax** page. PCRE tries to match Perl syntax and semantics as closely as it can. PCRE also supports some alternative regular expression syntax (which does not conflict with the Perl syntax) in order to provide some compatibility with regular expressions in Python, .NET, and Oniguruma.

Perl's regular expressions are described in its own documentation, and regular expressions in general are covered in a number of books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

This document discusses the patterns that are supported by PCRE when one its main matching functions, **pcre_exec()** (8-bit) or **pcre[16|32]_exec()** (16- or 32-bit), is used. PCRE also has alternative matching functions, **pcre_dfa_exec()** and **pcre[16|32_dfa_exec()**, which match 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 functions, and how they differ from the normal functions, are discussed in the **pcrematching** page.

SPECIAL START-OF-PATTERN ITEMS

A number of options that can be passed to **pcre_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

The original operation of PCRE was on strings of one-byte characters. However, there is now also support for UTF-8 strings in the original library, an extra library that supports 16-bit and UTF-16 character strings, and a third library that supports 32-bit and UTF-32 character strings. To use these features, PCRE must be built to include appropriate support. When using UTF strings you must either call the compiling function with the PCRE_UTF8, PCRE_UTF16, or PCRE_UTF32 option, or the pattern must start with one of these special sequences:

```
(*UTF8)
(*UTF16)
(*UTF32)
(*UTF)
```

(*UTF) is a generic sequence that can be used with any of the libraries. Starting a pattern with such a sequence is equivalent to setting the relevant option. How setting a UTF mode affects pattern matching is mentioned in several places below. There is also a summary of features in the **pcreunicode** page.

Some applications that allow their users to supply patterns may wish to restrict them to non-UTF data for security reasons. If the PCRE_NEVER_UTF option is set at compile time, (*UTF) etc. are not allowed, and their appearance 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 PCRE_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 128 via a lookup table.

Disabling auto-possessification

If a pattern starts with (*NO_AUTO_POSSESS), it has the same effect as setting the PCRE_NO_AUTO_POSSESS option at compile time. This stops PCRE 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 **pcreapi** documentation.

Disabling start-up optimizations

If a pattern starts with (*NO_START_OPT), it has the same effect as setting the PCRE_NO_START_OPTIMIZE option either at compile or matching time. This disables several optimizations for quickly reaching "no match" results. For more details, see the **pcreapi** documentation.

Newline conventions

PCRE supports five 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, or any Unicode newline sequence. The **pcreapi** page has further discussion about newlines, and shows how to set the newline convention in the *options* arguments for the compiling and matching functions.

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

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

These override the default and the options given to 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 PCRE_DOTALL is not set, and the behaviour of \N. 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 description of \R in the section entitled "Newline sequences" below. A change of \R setting can be combined with a change of newline convention.

Setting match and recursion limits

The caller of **pcre_exec()** can set a limit on the number of times the internal **match()** function is called and on the maximum depth of recursive calls. These facilities are provided to catch runaway matches that are provoked by patterns with huge matching trees (a typical example is a pattern with nested unlimited repeats) and to avoid running out of system stack by too much recursion. When one of these limits is reached, **pcre_exec()** gives an error return. The limits can also be set by items at the start of the pattern of the form

```
(*LIMIT_MATCH=d)
(*LIMIT_RECURSION=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 **pcre_exec()** 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.

EBCDIC CHARACTER CODES

PCRE can be compiled to run in an environment that uses EBCDIC as its character code rather than 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 PCRE_CASELESS option), letters are matched independently of case. In a UTF mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF support.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized within square brackets. Outside square brackets, the metacharacters are as follows:

```
\      general escape character with several uses
^      assert start of string (or line, in multiline mode)
$      assert end of string (or line, in multiline mode)
.      match any character except newline (by default)
[      start character class definition
|      start of alternative branch
(      start subpattern
)      end subpattern
?      extends the meaning of (
       also 0 or 1 quantifier
       also quantifier minimizer
*      0 or more quantifier
+      1 or more quantifier
       also "possessive quantifier"
{      start min/max quantifier
```

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

```
\      general escape character
^      negate the class, but only if the first character
-      indicates character range
[      POSIX character class (only if followed by POSIX
         syntax)
]      terminates the character class
```

The following sections describe the use of each of the metacharacters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a character that is not a number 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 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 \\.

In a UTF mode, only ASCII numbers and letters have any special meaning after a backslash. All other characters (in particular, those whose codepoints are greater than 127) are treated as literals.

If a pattern is compiled with the PCRE_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 # character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
\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

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.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences than the binary character it represents. 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 ASCII character
\e	escape (hex 1B)
\f	form feed (hex 0C)
\n	linefeed (hex 0A)
\r	carriage return (hex 0D)
\t	tab (hex 09)
\0dd	character with octal code 0dd
\ddd	character with octal code ddd, or back reference
\o{ddd..}	character with octal code ddd..
\xhh	character with hex code hh
\x{hhh..}	character with hex code hhh.. (non-JavaScript mode)
\uhhhh	character with hex code hhhh (JavaScript mode only)

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 data item (byte or 16-bit value) following \c has a value greater than 127, a compile-time error occurs. This locks out non-ASCII characters in all modes.

When PCRE is compiled in EBCDIC mode, \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 \@ encodes character code 0; 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 \? becomes either 255 (hex FF) or 95 (hex 5F).

Thus, apart from \?, 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, \G always generates code value 7, which is BEL in ASCII but DEL in EBCDIC.

The sequence \? 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, PCRE makes \? 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 back references 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 character numbers, and \g{} to specify back references. 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 in recent releases, causing PCRE also to change. Outside a character class, PCRE reads the digit and any following digits as a decimal number. If the number is less than 8, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number following \ is greater than 7 and there have not been that many capturing subpatterns, PCRE handles \8 and \9 as the literal characters "8" and "9", and otherwise re-reads up to three octal digits following the backslash, using them to generate a data character. Any subsequent digits stand for themselves. For example:

```
\040   is another way of writing an ASCII space
\40    is the same, provided there are fewer than 40
         previous capturing subpatterns
\7     is always a back reference
\11    might be a back reference, or another way of
         writing a tab
\011   is always a tab
\0113  is a tab followed by the character "3"
\113   might be a back reference, otherwise the
         character with octal code 113
\377   might be a back reference, otherwise
```

```
      the value 255 (decimal)
\81   is either a back reference, or the two
      characters "8" and "1"
```

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.

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.

If the PCRE_JAVASCRIPT_COMPAT option is set, the interpretation of \x is as just described only when it is followed by two hexadecimal digits. Otherwise, it matches a literal "x" character. In JavaScript mode, support for code points greater than 256 is provided by \u, which must be followed by four hexadecimal digits; otherwise it matches a literal "u" character.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x (or by \u in JavaScript mode). There is no difference in the way they are handled. For example, \xdc is exactly the same as \x{dc} (or \u00dc in JavaScript mode).

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    less than 0x100
8-bit UTF-8 mode      less than 0x10ffff and a valid codepoint
16-bit non-UTF mode   less than 0x10000
16-bit UTF-16 mode    less than 0x10ffff and a valid codepoint
32-bit non-UTF mode   less than 0x100000000
32-bit UTF-32 mode    less than 0x10ffff and a valid codepoint
```

Invalid Unicode codepoints are the range 0xd800 to 0xdfff (the so-called "surrogate" codepoints), and 0xffef.

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).

\N is not allowed in a character class. \B, \R, and \X are not special inside a character class. Like other unrecognized escape sequences, they are treated as the literal characters "B", "R", and "X" by default, but cause an error if the PCRE_EXTRA option is set. Outside a character class, these sequences have different meanings.

Unsupported escape sequences

In Perl, the sequences \l, \L, \u, and \U are recognized by its string handler and used to modify the case of following characters. By default, PCRE does not support these escape sequences. However, if the PCRE_JAVASCRIPT_COMPAT option is set, \U matches a "U" character, and \u can be used to define a character by code point, as described in the previous section.

Absolute and relative back references

The sequence \g followed by an unsigned or a negative number, optionally enclosed in braces, is an absolute or relative back reference. A named back reference can be coded as \g{name}. Back references are discussed later, following the discussion of parenthesized subpatterns.

Absolute and relative subroutine calls

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either

in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a "subroutine". Details are discussed later. Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are*not* synon y-mous. The former is a back reference; the latter is a subroutine call.

Generic character types

Another use of backslash is for specifying generic character types:

```
\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
\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
```

There is also the single sequence \N, which matches a non-newline character. This is the same as the "." metacharacter when PCRE_DOTALL is not set. Perl also uses \N to match characters by name; PCRE 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.

For compatibility with Perl, \s did not used to match the VT character (code 11), which made it different from the the POSIX "space" class. However, Perl added VT at release 5.18, and PCRE followed suit at release 8.34. The default \s characters are now 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 PCRE's low-valued character tables, and may vary if locale-specific matching is taking place (see "Locale support" in the **pcreapi** page). For example, in a French locale such as "fr_FR" in Unix-like systems, or "french" in Windows, some character codes greater than 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 vary 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 PCRE is compiled with Unicode property support, and the PCRE_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 PCRE_UCP affects \b, and \B because they are defined in terms of \w and \W. Matching these sequences is noticeably slower when PCRE_UCP is set.

The sequences \h, \H, \v, and \V are features that were added to Perl at release 5.10. In contrast to the other

sequences, which match only ASCII characters by default, these always match certain high-valued code points, whether or not PCRE_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 codepoints 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). The two-character sequence is treated as a single unit that cannot be split.

In other modes, two additional characters whose codepoints are greater than 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

It is possible to restrict \R to match only CR, LF, or CRLF (instead of the complete set of Unicode line endings) by setting the option PCRE_BSR_ANYCRLF either at compile time or when the pattern is matched. (BSR is an abbrevation for "backslash R".) This can be made the default when PCRE is built; if this is the case, the other behaviour can be requested via the PCRE_BSR_UNICODE option. It is also possible to specify these settings by starting a pattern string with one of the following sequences:

(*BSR_ANYCRLF) CR, LF, or CRLF only
(*BSR_UNICODE) any Unicode newline sequence

These override the default and the options given to the compiling function, but they can themselves be overridden by options given to a matching 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 (*UTF8), (*UTF16), (*UTF32), (*UTF) or (*UCP) special sequences. Inside a character class, \R is treated as an unrecognized escape sequence, and so matches the letter "R" by default, but causes an error if PCRE_EXTRA is set.

Unicode character properties

When PCRE is built with Unicode character property support, three additional escape sequences that match characters with specific properties are available. When in 8-bit non-UTF-8 mode, these sequences are of course limited to testing characters whose codepoints are less than 256, but they do work in this mode. The extra escape sequences are:

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

The property names represented by *xx* above are limited to the Unicode script names, the general category properties, "Any", which matches any character (including newline), and some special PCRE properties (described in the next section). Other Perl properties such as "InMusicalSymbols" are not currently supported by PCRE. Note that \P{Any} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example:

\p{Greek}
\P{Han}

Those that are not part of an identified script are lumped together as "Common". The current list of scripts is:

Arabic, Armenian, Avestan, Balinese, Bamum, Bassa_Vah, Batak, Bengali, Bopomofo, Brahmi, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Caucasian_Albanian, Chakma, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Duployan, Egyptian_Hieroglyphs, Elbasan, Ethiopic, Georgian, Glagolitic, Gothic, Grantha, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Imperial_Aramaic, Inherited, Inscriptional_Pahlavi, Inscriptional_Parthian, Javanese, Kaithi, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Khojki, Khudawadi, Lao, Latin, Lepcha, Limbu, Linear_A, Linear_B, Lisu, Lycian, Lydian, Mahajani, Malayalam, Mandaic, Manichaean, Meetei_Mayek, Mende_Kikakui, Meroitic_Cursive, Meroitic_Hieroglyphs, Miao, Modi, Mongolian, Mro, Myanmar, Nabataean, New_Tai_Lue, Nko, Ogham, Ol_Chiki, Old_Italic, Old_North_Arabian, Old_Permic, Old_Persian, Old_South_Arabian, Old_Turkic, Oriya, Osmanya, Pahawh_Hmong, Palmyrene, Pau_Cin_Hau, Phags_Pa, Phoenician, Psalter_Pahlavi, Rejang, Runic, Samaritan, Saurashtra, Sharada, Shavian, Siddham, Sinhala, Sora_Sompeng, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tai_Tham, Tai_Viet, Takri, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Tirhuta, Ugaritic, Vai, Warang_Citi, Yi.

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 in the range U+D800 to U+DFFF. Such characters are not valid in Unicode strings and so cannot be tested by PCRE, unless UTF validity checking has been turned off (see the discussion of PCRE_NO_UTF8_CHECK, PCRE_NO_UTF16_CHECK and PCRE_NO_UTF32_CHECK in the **pcreapi** page). Perl does not support the Cs property.

The long synonyms for property names that Perl supports (such as \p{Letter}) are not supported by PCRE, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the Cn (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, \p{Lu} always matches only upper case letters. This is different from the behaviour of current versions of Perl.

Matching characters by Unicode property is not fast, because PCRE 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 PCRE by default, though you can make them do so by setting the PCRE_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). Up to and including release 8.31, PCRE matched an earlier, simpler definition that was equivalent to

```
(?>\PM\pM*)
```

That is, it matched a character without the "mark" property, followed by zero or more characters with the "mark" property. Characters with the "mark" property are typically non-spacing accents that affect the preceding character.

This simple definition was extended in Unicode to include more complicated kinds of composite character by giving each character a grapheme breaking property, and creating rules that use these properties to define the boundaries of extended grapheme clusters. In releases of PCRE later than 8.31, \X matches one of these clusters.

\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 follwed only by a T character.

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

5. Do not end after prepend characters.

6. Otherwise, end the cluster.

PCRE's additional properties

As well as the standard Unicode properties described above, PCRE supports four more that make it possible to convert traditional escape sequences such as \w and \s to use Unicode properties. PCRE uses these non-standard, non-Perl properties internally when PCRE_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; it used to exclude vertical tab, for Perl compatibility, but Perl changed, and so PCRE followed at release 8.34. 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

The escape sequence \K causes any previously matched characters not to be included in the final matched sequence. For example, the pattern:

```
foo\Kbar
```

matches "foobar", but reports that it has matched "bar". This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of \K does not interfere with the setting of captured substrings. For example, when the pattern

```
(foo)\Kbar
```

matches "foobar", the first substring is still set to "foo".

Perl documents that the use of \K within assertions is "not well defined". In PCRE, \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.

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

```
\b    matches at a word boundary
\B    matches when not at a word boundary
\A    matches at the start of the subject
\Z    matches at the end of the subject
      also matches before a newline at the end of the subject
\z    matches only at the end of the subject
\G    matches at the first matching position in the subject
```

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, by default it matches the corresponding literal character (for example, \B matches the letter B). However, if the PCRE_EXTRA option is set, an "invalid escape sequence" error is generated instead.

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. In a UTF mode, the meanings of \w and \W can be changed by setting the PCRE_UCP option. When this is done, it also affects \b and \B. Neither PCRE nor Perl has a separate "start of word" or "end of word" metasequence. However, whatever follows \b normally determines which it is. For example, the fragment \ba matches "a" at the start of a word.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the *startoffset* argument of **pcre_exec()** is non-zero, indicating that matching is to start at a point other than the beginning of the subject, \A can never match. The difference between \Z and \z is that \Z matches before a newline at the end of the string as well as at the very end, whereas \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the *startoffset* argument of **pcre_exec()**. It differs from \A when the value of *startoffset* is non-zero. By calling **pcre_exec()** multiple times with appropriate arguments, you can mimic Perl's /g option, and it is in this kind of implementation where \G can be useful.

Note, however, that PCRE's interpretation of \G, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behaviour.

If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR

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.

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If the *startoffset* argument of **pcre_exec()** is non-zero, circumflex can never match if the PCRE_MULTILINE option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

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). 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 PCRE_DOLLAR_ENDONLY option at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when PCRE_MULTILINE is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /^abc$/ matches the subject string "def\nabc" (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with ^ are not anchored in multiline mode, and a match for circumflex is possible when the

startoffset argument of **pcre_exec()** is non-zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A it is always anchored, whether or not PCRE_MULTILINE is set.

FULL STOP (PERIOD, DOT) 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 PCRE_DOTALL option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

The escape sequence \N behaves like a dot, except that it is not affected by the PCRE_DOTALL option. In other words, it matches any character except one that signifies the end of a line. Perl also uses \N to match characters by name; PCRE does not support this.

MATCHING A SINGLE DATA UNIT

Outside a character class, the escape sequence \C matches any one data unit, whether or not a UTF mode is set. In the 8-bit library, one data 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 data units, matching one unit with \C in a UTF mode means that the rest of the string may start with a malformed UTF character. This has undefined results, because PCRE assumes that it is dealing with valid UTF strings (and by default it checks this at the start of processing unless the PCRE_NO_UTF8_CHECK, PCRE_NO_UTF16_CHECK or PCRE_NO_UTF32_CHECK option is used).

PCRE does not allow \C to appear in lookbehind assertions (described below) in a UTF mode, because this would make it impossible to calculate the length of the lookbehind.

In general, the \C escape sequence is best avoided. However, one way of using it that avoids the problem of malformed UTF 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))
```

A group that starts with (?| resets the capturing parentheses numbers in each alternative (see "Duplicate Subpattern 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 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. However, if the PCRE_JAVASCRIPT_COMPAT option is set, a lone closing square bracket causes a compile-time error. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In a UTF mode, the character may be more than one data unit long. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion; it still consumes a character from the subject string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 (UTF-16, UTF-32) mode, characters with values greater than 255 (0xffff) can be included in a class as a literal string of data units, or by using the \x{ escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would. In a UTF mode, PCRE always understands the concept of case for characters whose values are less than 128, so caseless matching is always possible. For characters with higher values, the concept of case is supported if PCRE is compiled with Unicode property support, but not otherwise. If you want to use caseless matching in a UTF mode for characters 128 and above, you must ensure that PCRE is compiled with Unicode property support as well as with UTF support.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the PCRE_DOTALL and PCRE_MULTILINE options is used. A class such as [^a] always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class, or immediately after a range. For example, [b-d-z] matches letters in the range b to d, a hyphen character, or z.

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.

An error is generated if a POSIX character class (see below) or an escape sequence other than one that defines a single character appears at a point where a range ending character is expected. For example, [z-\xff] is valid, but [A-\d] and [A-[:digit:]] are not.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037]. Ranges can include any characters that are valid for the current mode.

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. In UTF modes, PCRE supports the concept of case for characters with values greater than 128 only when it is compiled with Unicode property support.

The character 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 PCRE_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, \N, \R, and \X are not special inside a character class. Like any other unrecognized escape sequences, they are treated as the literal characters "B", "N", "R", and "X" by default, but cause an error if the PCRE_EXTRA option is set.

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. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are:

```
alnum    letters and digits
alpha    letters
ascii    character codes 0 - 127
blank    space or tab only
cntrl    control characters
digit    decimal digits (same as \d)
graph    printing characters, excluding space
lower    lower case letters
print    printing characters, including space
punct    printing characters, excluding letters and digits and space
space    white space (the same as \s from PCRE 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" used to be different to \s, which did not include VT, for Perl compatibility. However, Perl changed at release 5.18, and PCRE followed at release 8.34. "Space" and \s now 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. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

By default, characters with values greater than 128 do not match any of the POSIX character classes.

However, if the PCRE_UCP option is passed to **pcre_compile()**, some of the classes are changed so that Unicode character properties are used. This is achieved by replacing certain POSIX classes by other sequences, as follows:

```
[:alnum:]  becomes  \p{Xan}
[:alpha:]  becomes  \p{L}
[:blank:]  becomes  \h
[: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 whose code points are less than 128 that have the S (Symbol) property.

The other POSIX classes are unchanged, and match only characters with code points less than 128.

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

INTERNAL OPTION SETTING

The settings of the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED options (which are Perl-compatible) can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

```
i  for PCRE_CASELESS
m  for PCRE_MULTILINE
s  for PCRE_DOTALL
x  for PCRE_EXTENDED
```

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options PCRE_DUPNAMES, PCRE_UNGREEDY, and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters J, U and X respectively.

When one of these option changes occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options (and it will therefore show up in data extracted by the **pcre_fullinfo()** function).

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the subpattern that follows it, so

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

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

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

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

Note: There are other PCRE-specific options that can be set by the application when the compiling or matching functions are called. In some cases the pattern can contain special leading sequences such as (*CRLF) to override what the application has set or what has been defaulted. Details are given in the section entitled "Newline sequences" above. There are also the (*UTF8), (*UTF16),(*UTF32), and (*UCP) leading sequences that can be used to set UTF and Unicode property modes; they are equivalent to setting the PCRE_UTF8, PCRE_UTF16, PCRE_UTF32 and the PCRE_UCP options, respectively. The (*UTF) sequence is a generic version that can be used with any of the libraries. However, the application can set the PCRE_NEVER_UTF option, which locks out the use of the (*UTF) sequences.

SUBPATTERNS

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

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

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

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

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of the matching function. (This applies only to the traditional matching functions; the DFA matching functions do not support capturing.)

Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns. For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

DUPLICATE SUBPATTERN NUMBERS

Perl 5.10 introduced a feature whereby each alternative in a subpattern uses the same numbers for its capturing parentheses. Such a subpattern starts with (?| and is itself a non-capturing subpattern. For example, consider this pattern:

```
(?|(Sat)ur|(Sun))day
```

Because the two alternatives are inside a (?| group, both sets of capturing parentheses are numbered one. Thus, when the pattern matches, you can look at captured substring number one, whichever alternative matched. This construct is useful when you want to capture part, but not all, of one of a number of alternatives. Inside a (?| group, parentheses are numbered as usual, but the number is reset at the start of each branch. The numbers of any capturing parentheses that follow the subpattern start after the highest number used in any branch. The following example is taken from the Perl documentation. The numbers underneath show in which buffer the captured content will be stored.

```
# before  ----------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1           2         2  3        2     3     4
```

A back reference to a numbered subpattern uses the most recent value that is set for that number by any

subpattern. The following pattern matches "abcabc" or "defdef":

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

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

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

If a condition test for a subpattern's having matched refers to a non-unique number, the test is true if any of the subpatterns of that number have matched.

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

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns. This feature was not added to Perl until release 5.10. Python had the feature earlier, and PCRE introduced it at release 4.0, using the Python syntax. PCRE now supports both the Perl and the Python syntax. Perl allows identically numbered subpatterns to have different names, but PCRE does not.

In PCRE, a subpattern can be named in one of three ways: (?<name>...) or (?'name'...) as in Perl, or (?P<name>...) as in Python. References to capturing parentheses from other parts of the pattern, such as back references, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores, but must start with a non-digit. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The PCRE API provides function calls for extracting the name-to-number translation table from a compiled pattern. There is also a convenience function for extracting a captured substring by name.

By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the PCRE_DUPNAMES option at compile time. (Duplicate names are also always permitted for subpatterns with the same number, set up as described in the previous section.) Duplicate names can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

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

There are five capturing substrings, but only one is ever set after a match. (An alternative way of solving this problem is to use a "branch reset" subpattern, as described in the previous section.)

The convenience function for extracting the data by name returns the substring for the first (and in this example, the only) subpattern of that name that matched. This saves searching to find which numbered subpattern it was.

If you make a back reference to a non-unique named subpattern from elsewhere in the pattern, the subpatterns 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":

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

If you make a subroutine call to a non-unique named subpattern, the one that corresponds to the first occurrence of the name is used. In the absence of duplicate numbers (see the previous section) this is the one with the lowest number.

If you use a named reference in a condition test (see the section about conditions below), either to check whether a subpattern has matched, or to check for recursion, all subpatterns with the same name are tested. If the condition is true for any one of them, the overall condition is true. This is the same behaviour as testing by number. For further details of the interfaces for handling named subpatterns, see the **pcreapi** documentation.

Warning: You cannot use different names to distinguish between two subpatterns with the same number because PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are given to subpatterns with the same number. However, you can always give the same name to subpatterns with the same number, even when PCRE_DUPNAMES is not set.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

a literal data character
the dot metacharacter
the \C escape sequence
the \X escape sequence
the \R escape sequence
an escape such as \d or \pL that matches a single character
a character class
a back reference (see next section)
a parenthesized subpattern (including assertions)
a subroutine call to a subpattern (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, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

In UTF modes, quantifiers apply to characters rather than to individual data 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 data 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 subpatterns that are referenced as subroutines from elsewhere in

the pattern (but see also the section entitled "Defining subpatterns for use by reference only" below). Items other than subpatterns that have a {0} quantifier are omitted from the compiled pattern.

For convenience, the three most common quantifiers have single-character abbreviations:

```
*    is equivalent to {0,}
+    is equivalent to {1,}
?    is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first comment */  not comment  /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

However, there are some cases where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a back reference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

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.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl’s book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher gives up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with (?> as in this example:

```
(?>\d+)foo
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

Note that a possessive quantifier can be used with an entire group, for example:

```
(abc|xyz){2,3}+
```

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl 5.8 syntax. Jeffrey Friedl originated the idea (and the name) in the first edition of his book. Mike McCloskey liked it, so implemented it when he built Sun's Java package, and PCRE copied it from there. It ultimately found its way into Perl at release 5.10.

PCRE has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A's when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
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 PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been

that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to a subpattern whose number is 10 or more using this syntax because a sequence such as \50 is interpreted as a character defined in octal. See the subsection entitled "Non-printing characters" above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. These examples are all identical:

```
(ring), \1
(ring), \g1
(ring), \g{1}
```

An unsigned number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider this example:

```
(abc(def)ghi)\g{-1}
```

The sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2 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.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \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 back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There are several different ways of writing back references to named subpatterns. The .NET syntax \k{name} and the Perl syntax \k<name> or \k’name’ are supported, as is the Python syntax (?P=name). Perl 5.10’s unified back reference syntax, in which \g can be used for both numeric and named references, is also supported. We could rewrite the above example in any of the following ways:

```
(?<p1>(?i)rah)\s+\k<p1>
(?’p1’(?i)rah)\s+\k{p1}
(?P<p1>(?i)rah)\s+(?P=p1)
(?<p1>(?i)rah)\s+\g{p1}
```

A subpattern that is referenced by name may appear in the pattern before or after the reference.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail by default. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". However, if the PCRE_JAVASCRIPT_COMPAT option is set at compile time, a back reference to an unset value matches an empty string.

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

Recursive back references

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

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

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Back references of this type cause the group that they reference to be treated as an atomic group. Once the whole group has been matched, a subsequent matching failure cannot cause backtracking into the middle of the group.

ASSERTIONS

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

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns. If such an assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions. (Perl sometimes, but not always, does do capturing in negative assertions.)

For compatibility with Perl, assertion subpatterns may be repeated; though it makes no sense to assert the same thing several times, the side effect of capturing parentheses may occasionally be useful. In practice, there only three cases:

(1) If the quantifier is {0}, the assertion is never obeyed during matching. However, it may contain internal capturing parenthesized groups that are called from elsewhere via the subroutine mechanism.

(2) If quantifier is {0,n} where n is greater than zero, it is treated as if it were {0,1}. At run time, the rest of the pattern match is tried with and without the assertion, the order depending on the greediness of the quantifier.

(3) If the minimum repetition is greater than zero, the quantifier is ignored. The assertion is obeyed just

once when encountered during matching.

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 PCRE 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 a UTF mode, PCRE does not allow the \C escape (which matches a single data 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 data units, are also not permitted.

"Subroutine" calls (see below) such as (?2) or (?&X) are permitted in lookbehinds, as long as the subpattern matches a fixed-length string. Recursion, however, is not supported.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching of fixed-length strings at the end of subject strings. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^.*+(?<=abcd)
```

there can be no backtracking for the .*+ item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "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".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a specific capturing subpattern has already been matched. The two possible forms of conditional subpattern are:

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs. Each of the two alternatives may itself contain nested subpatterns of any form, including conditional subpatterns; the restriction to two alternatives applies only at the level of the condition. This pattern fragment is an example where the alternatives are complex:

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

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if a capturing subpattern of that number has previously matched. If there is more than one capturing subpattern with the same number (see the earlier section about duplicate subpattern numbers), the condition is true if any of them have matched. An alternative notation is to precede the digits with a plus or minus sign. In this case, the subpattern number is relative rather than absolute. The most recently opened parentheses can be referenced by (?(-1), the next most recent by (?(-2), and so on. Inside loops it can also make sense to refer to subsequent groups. The next parentheses to be opened 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 PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether or not the first set of parentheses matched. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If you were embedding this pattern in a larger one, you could use a relative reference:

```
...other stuff... ( \( )?    [^()]+    (?(-1) \) ) ...
```

This makes the fragment independent of the parentheses in the larger pattern.

Checking for a used subpattern by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax (?(name)...) is also recognized.

Rewriting the above example to use a named subpattern gives this:

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

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

Checking for pattern recursion

If the condition is the string (R), and there is no subpattern with the name R, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter R, for example:

```
(?(R3)...) or (?(R&name)...)
```

the condition is true if the most recent recursion is into a subpattern whose number or name is given. This condition does not check the entire recursion stack. If the name used in a condition of this kind is a duplicate, the test is applied to all subpatterns of the same name, and is true if any one of them is the most recent recursion.

At "top level", all these recursion test conditions are false. The syntax for recursive patterns is described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define subroutines that can be referenced from elsewhere. (The use of subroutines is described below.) For example, a pattern to match an IPv4 address 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 a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition. The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

There are two ways of including comments in patterns that are processed by PCRE. 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 subpattern 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 PCRE_EXTENDED 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 the options passed to a 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 PCRE_EXTENDED is set, and the default newline convention is in force:

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

On encountering the # character, **pcre_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, PCRE cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was subsequently introduced into Perl at release 5.10.

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive subroutine call of the subpattern of the given number, provided that it occurs inside that subpattern. (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 PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
\( ( [^()]++ | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis. Note the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( [^()]++ | (?1) )* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern.

In a larger pattern, keeping track of parenthesis numbers can be tricky. This is made easier by the use of relative references. Instead of (?1) in the pattern above you can write (?-2) to refer to the second most recently opened parentheses preceding the recursion. In other words, a negative number counts capturing parentheses leftwards from the point at which it is encountered.

It is also possible to refer to subsequently opened parentheses, by writing references such as (?+2). However, these cannot be recursive because the reference is not inside the parentheses that are referenced. They are always non-recursive subroutine calls, as described in the next section.

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

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

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

This particular example pattern that we have been looking at contains nested unlimited repeats, and so the use of a possessive quantifier for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if a possessive quantifier is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values of capturing parentheses are those from the outermost level. If you want to obtain intermediate values, a callout function can be used (see below and the **pcrecallout** documentation). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the inner capturing parentheses (numbered 2) is "ef", which is the last value taken on at the top level. If a capturing subpattern is not matched at the top level, its final captured value is unset, even if it was (temporarily) set at a deeper level during the matching process.

If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using **pcre_malloc**, freeing it via **pcre_free** afterwards. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the (?R) item with the condition (R), which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

In this pattern, (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

Differences in recursion processing between PCRE and Perl

Recursion processing in PCRE differs from Perl in two important ways. In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. This can be illustrated by the following pattern, which purports to match a palindromic string that contains an odd number of characters (for example, "a", "aba", "abcba", "abcdcba"):

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

The idea is that it either matches a single character, or two identical characters surrounding a sub-palindrome. In Perl, this pattern works; in PCRE it does not if the pattern is longer than three characters. Consider the subject string "abcba":

At the top level, the first character is matched, but as it is not at the end of the string, the first alternative fails; the second alternative is taken and the recursion kicks in. The recursive call to subpattern 1 successfully matches the next character ("b"). (Note that the beginning and end of line tests are not part of the recursion).

Back at the top level, the next character ("c") is compared with what subpattern 2 matched, which was "a". This fails. Because the recursion is treated as an atomic group, there are now no backtracking points, and so the entire match fails. (Perl is able, at this point, to re-enter the recursion and try the second alternative.) However, if the pattern is written with the alternatives in the other order, things are different:

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

This time, the recursing alternative is tried first, and continues to recurse until it runs out of characters, at which point the recursion fails. But this time we do have another alternative to try at the higher level. That is the big difference: in the previous case the remaining alternative is at a deeper recursion level, which PCRE cannot use.

To change the pattern so that it matches all palindromic strings, not just those with an odd number of characters, it is tempting to change the pattern to this:

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

Again, this works in Perl, but not in PCRE, and for the same reason. When a deeper recursion has matched a single character, it cannot be entered again in order to match an empty string. The solution is to separate the two cases, and write out the odd and even cases as alternatives at the higher level:

```
^(?:((.)(?1)\2|)|((.)(?3)\4|.))
```

If you want to match typical palindromic phrases, the pattern has to ignore all non-word characters, which can be done like this:

```
^\W*+(?:((.)\W*+(?1)\W*+\2|)|((.)\W*+(?3)\W*+\4|\W*+.\W*+))\W*+$
```

If run with the PCRE_CASELESS option, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works well in both PCRE and Perl. Note the use of the possessive quantifier *+ to avoid backtracking into sequences of non-word characters. Without this, PCRE takes a great deal longer (ten times or more) to match typical phrases, and Perl takes so long that you think it has gone into a loop.

WARNING: The palindrome-matching patterns above work only if the subject string does not start with a palindrome that is shorter than the entire string. For example, although "abcba" is correctly matched, if the subject is "ababa", PCRE finds the palindrome "aba" at the start, then fails at top level because the end of the string does not follow. Once again, it cannot jump back into the recursion to try other alternatives, so the entire match fails.

The second way in which PCRE and Perl differ in their recursion processing is in the handling of captured values. In Perl, when a subpattern is called recursively or as a subpattern (see the next section), it has no access to any values that were captured outside the recursion, whereas in PCRE these values can be referenced. Consider this pattern:

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

In PCRE, this pattern matches "bab". The first capturing parentheses match "b", then in the second group, when the back reference \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. In Perl, the pattern fails to match because inside the recursive call \1 cannot access the externally set value.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern call (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The called subpattern may be defined before or after the reference. A numbered reference can be absolute or relative, as in these examples:

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

All subroutine calls, whether recursive or not, are always treated as atomic groups. That is, once a subroutine has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure. Any capturing parentheses that are set during the subroutine call revert to their previous values afterwards.

Processing options such as case-independence are fixed when a subpattern 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 subpattern.

ONIGURUMA SUBROUTINE SYNTAX

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is an alternative syntax for referencing a subpattern as a subroutine, possibly recursively. Here are two of the examples used above, rewritten using this syntax:

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

PCRE supports an extension to Oniguruma: if a number is preceded by a plus or a minus sign it is taken as a relative reference. For example:

```
(abc)(?i:\g<-1>)
```

Note that \g{...} (Perl syntax) and \g<...> (Oniguruma syntax) are *not* synonymous. The former is a back reference; the latter is a subroutine call.

CALLOUTS

Perl has a feature whereby using the sequence (?{...}) causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its entry point in the global variable *pcre_callout* (8-bit library) or *pcre[16|32]_callout* (16-bit or 32-bit library). By default, this variable contains NULL, which disables all calling out.

Within a regular expression, (?C) indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

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

If the PCRE_AUTO_CALLOUT flag is passed to a compiling function, 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.

During matching, when PCRE reaches a callout point, the external function is called. It is provided with the number of the callout, the position in the pattern, and, optionally, one item of data originally supplied by the caller of the matching function. The callout function may cause matching to proceed, to backtrack, or to fail altogether.

By default, PCRE implements a number of optimizations at compile time and 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, and a complete description of the interface to the callout function, are given in the **pcrecallout** documentation.

BACKTRACKING CONTROL

Perl 5.10 introduced a number of "Special Backtracking Control Verbs", which are still described in the Perl documentation as "experimental and subject to change or removal in a future version of Perl". It goes on to say: "Their usage in production code should be noted to avoid problems during upgrades." The same remarks apply to the PCRE features described in this section.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. They are generally of the form (*VERB) or (*VERB:NAME). Some may take either form, possibly behaving differently depending on whether or not a name is present. A name is any sequence of characters that does not include a closing parenthesis. The maximum length of 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.

Since these verbs are specifically related to backtracking, most of them can be used only when the pattern is to be matched using one of the traditional matching functions, because these use 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 a DFA matching function.

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

Optimizations that affect backtracking verbs

PCRE 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 PCRE_NO_START_OPTIMIZE option when calling **pcre_compile()** or **pcre_exec()**, or by starting the pattern with (*NO_START_OPT). There is more discussion of this option in the section entitled "Option bits for **pcre_exec()**" in the **pcreapi** documentation.

Experiments with Perl suggest that it too has similar optimizations, sometimes leading to anomalous results.

Verbs that act immediately

The following verbs act as soon as they are encountered. They may not be followed by a name.

(*ACCEPT)

This verb causes the match to end successfully, skipping the remainder of the pattern. However, when it is inside a subpattern that is called as a subroutine, only that subpattern 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.

(*FAIL) or (*F)

This verb causes a matching failure, forcing backtracking to occur. It is equivalent to (?!) but easier to read. The Perl documentation notes that it is probably useful only when combined with (?{}) or (??{}). Those are, of course, Perl features that are not present in PCRE. The nearest equivalent is the callout feature, as for example in this pattern:

a+(?C)(*FAIL)

A match with the string "aaaa" always fails, but the callout is taken before each backtrack happens (in this example, 10 times).

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. There may be as many instances of (*MARK) as you like in a pattern, and their names do not have to be unique.

When a match succeeds, the name of the last-encountered (*MARK:NAME), (*PRUNE:NAME), or (*THEN:NAME) on the matching path is passed back to the caller as described in the section entitled "Extra data for **pcre_exec()**" in the **pcreapi** documentation. Here is an example of **pcretest** output, where the /K modifier requests the retrieval and outputting of (*MARK) data:

```
  re> /X(*MARK:A)Y|X(*MARK:B)Z/K
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/K
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 PCRE_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 no subsequent match, 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 an 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. In this situation, backtracking can "jump back" to the left of the entire atomic group or assertion. (Remember also, as stated above, that this localization also applies in subroutine calls.)

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)

This verb, which may not be followed by a name, 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 **pcre_exec()** is committed to finding a match at the current starting point, or not at all. For example:

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

This matches "xxaab" but not "aacaab". It can be thought of as a kind of dynamic anchor, or "I've started, so I must finish." The name of the most recently passed (*MARK) in the path is passed back when (*COMMIT) forces a match failure.

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 PCRE's start-of-match optimizations are turned off, as shown in this output from **pcretest**:

```
  re> /(*COMMIT)abc/
data> xyzabc
 0: abc
data> xyzabc\Y
No match
```

For this pattern, PCRE 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. In the second set of data, the escape sequence \Y is interpreted by the **pcretest** program. It causes the PCRE_NO_START_OPTIMIZE option to be set when **pcre_exec()** is called. This 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 the 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).

(*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. 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 quantifer 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".

(*SKIP:NAME)

When (*SKIP) has an associated name, its behaviour is modified. When it 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.

Note that (*SKIP:NAME) searches only for names set by (*MARK:NAME). It ignores names that are set by (*PRUNE:NAME) or (*THEN:NAME).

(*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 the 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).

A subpattern 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 subpattern 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 subpattern 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 subpattern. After a failure in C, matching moves to (*FAIL), which causes the whole subpattern to fail because there are no more alternatives to try. In this case, matching does now backtrack into A.

Note that a conditional subpattern is not considered as having two alternatives, because only one is ever used. In other words, the | character in a conditional subpattern 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 subpattern 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

PCRE 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, but PCRE fails because the (*COMMIT) in the second repeat of the group acts.

Backtracking verbs in assertions

(*FAIL) in an assertion has its normal effect: it forces an immediate backtrack.

(*ACCEPT) in a positive assertion causes the assertion to succeed without any further processing. In a negative assertion, (*ACCEPT) causes the assertion to fail without any further processing.

The other backtracking verbs are not treated specially if they appear in a positive assertion. In particular, (*THEN) skips to the next alternative in the innermost enclosing group that has alternations, whether or not this is within the assertion.

Negative assertions are, however, different, in order to ensure that changing a positive assertion into a negative assertion changes its result. Backtracking into (*COMMIT), (*SKIP), or (*PRUNE) causes a negative assertion to be true, without considering any further alternative branches in the assertion. Backtracking into (*THEN) causes it to skip to the next enclosing alternative within the assertion (the normal behaviour), but if the assertion does not have such an alternative, (*THEN) behaves like (*PRUNE).

Backtracking verbs in subroutines

These behaviours occur whether or not the subpattern is called recursively. Perl's treatment of subroutines is different in some cases.

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

(*ACCEPT) in a subpattern called as a subroutine causes the subroutine match to succeed without any further processing. Matching then continues after the subroutine call.

(*COMMIT), (*SKIP), and (*PRUNE) in a subpattern called as a subroutine cause the subroutine match to fail.

(*THEN) skips to the next alternative in the innermost enclosing group within the subpattern that has alternatives. If there is no such group within the subpattern, (*THEN) causes the subroutine match to fail.

SEE ALSO

pcreapi(3), **pcrecallout**(3), **pcrematching**(3), **pcresyntax**(3), **pcre**(3), **pcre16(3)**, **pcre32(3)**.

AUTHOR

Philip Hazel
University Computing Service

Cambridge CB2 3QH, England.

REVISION

Last updated: 14 June 2015