NAME

pcregrep - a grep with Perl-compatible regular expressions.

SYNOPSIS

pcregrep [-Vcfhilnrsvx] pattern [file] ...

DESCRIPTION

pcregrep searches files for character patterns, in the same way as other grep commands do, but it uses the PCRE regular expression library to support patterns that are compatible with the regular expressions of Perl 5. See **pcre(3)** for a full description of syntax and semantics.

If no files are specified, **pcregrep** reads the standard input. By default, each line that matches the pattern is copied to the standard output, and if there is more than one file, the file name is printed before each line of output. However, there are options that can change how **pcregrep** behaves.

Lines are limited to BUFSIZ characters. BUFSIZ is defined in **<stdio.h>**. The newline character is removed from the end of each line before it is matched against the pattern.

OPTIONS

-V Write the version number of the PCRE library being used to the standard error stream.

-c Do not print individual lines; instead just print a count of the number of lines that would otherwise have been printed. If several files are given, a count is printed for each of them.

versity of Cambridge for use on Unix systems connected to
the Internet. It is freely available under the terms of
the GNU General Public Licence. In style it is similar to
Smail 3, but its facilities are more extensive, and in
particular it has some defences against mail bombs and
unsolicited junk mail, in the form of options for refusing
messages from particular hosts, networks, or senders.

Exim's command line takes the standard Unix form of a
sequence of options, each starting with a hyphen charac-**f***filename* Read patterns from the file, one per line, and match all patterns against each line. There is a maximum of 100 patterns. Trailing white space is removed, and blank lines are ignored. An empty file contains no patterns and therefore matches nothing.

-h Suppress printing of filenames when searching multiple files.

-i Ignore upper/lower case distinctions during comparisons.

-l Instead of printing lines from the files, just print the names of the files containing lines that would have been printed. Each file name is printed once, on a separate line.

-n Precede each line by its line number in the file.

-r If any file is a directory, recursively scan the files it contains. Without **-r** a directory is scanned as a normal file.

-s Work silently, that is, display nothing except error messages. The exit status indicates whether any matches were found.

-v Invert the sense of the match, so that lines which do *not* match the pattern are now the ones that are found.

-x Force the pattern to be anchored (it must start matching at the beginning of the line) and in addition, require it to match the entire line. This is equivalent to having ^ and $ characters at the start and end of each alternative branch in the regular expression.

SEE ALSO

pcre(3), Perl 5 documentation

DIAGNOSTICS

Exit status is 0 if any matches were found, 1 if no matches were found, and 2 for syntax errors or inaccessible fi les (even if matches were found).

AUTHOR

Philip Hazel <ph10@cam.ac.uk>

Last updated: 15 August 2001

NAME

pcretest - a program for testing Perl-compatible regular expressions.

SYNOPSIS

pcretest [-d] [-i] [-m] [-o osize] [-p] [-t] [source] [destination]

pcretest was written as a test program for the PCRE regular expression library itself, but it can also be used for experimenting with regular expressions. This man page describes the features of the test program; for details of the regular expressions themselves, see the **pcre** man page.

OPTIONS

-d Behave as if each regex had the **/D** modifier (see below); the internal form is output after compilation.

-i Behave as if each regex had the **/I** modifier; information about the compiled pattern is given after compilation.

-m Output the size of each compiled pattern after it has been compiled. This is equivalent to adding /M to each regular expression. For compatibility with earlier versions of pcretest, **-s** is a synonym for **-m**.

-o *osize* Set the number of elements in the output vector that is used when calling PCRE to be *osize*. The default value is 45, which is enough for 14 capturing subexpressions. The vector size can be changed for individual matching calls by including \O in the data line (see below).

-p Behave as if each regex has **/P** modifier; the POSIX wrapper API is used to call PCRE. None of the other options has any effect when **-p** is set.

-t Run each compile, study, and match 20000 times with a timer, and output resulting time per compile or match (in milliseconds). Do not set **-t** with **-m**, because you will then get the size output 20000 times and the timing will be distorted.

DESCRIPTION

If **pcretest** is given two filename arguments, it reads from the first and writes to the second. If it is given only one filename argument, it reads from that file and writes to stdout. Otherwise, it reads from stdin and writes to stdout, and prompts for each line of input, using "re>" to prompt for regular expressions, and "data>" to prompt for data lines.

The program handles any number of sets of input on a single input file. Each set starts with a regular expression, and continues with any number of data lines to be matched against the pattern. An empty line signals the end of the data lines, at which point a new regular expression is read. The regular expressions are given enclosed in any non-alphameric delimiters other than backslash, for example

```
/(a|bc)x+yz/
```

White space before the initial delimiter is ignored. A regular expression may be continued over several input lines, in which case the newline characters are included within it. It is possible to include the delimiter within the pattern by escaping it, for example

```
/abc\/def/
```

If you do so, the escape and the delimiter form part of the pattern, but since delimiters are always non-alphameric, this does not affect its interpretation. If the terminating delimiter is immediately followed by a backslash, for example,

```
/abc/\
```

then a backslash is added to the end of the pattern. This is done to provide a way of testing the error condition that arises if a pattern finishes with a backslash, because

```
/abc\/
```

is interpreted as the first line of a pattern that starts with "abc/", causing pcretest to read the next line as a continuation of the regular expression.

PATTERN MODIFIERS

The pattern may be followed by **i**, **m**, **s**, or **x** to set the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, or PCRE_EXTENDED options, respectively. For example:

```
/caseless/i
```

These modifier letters have the same effect as they do in Perl. There are others which set PCRE options that do not correspond to anything in Perl: **/A**, **/E**, and **/X** set PCRE_ANCHORED, PCRE_DOLLAR_ENDONLY, and PCRE_EXTRA respectively.

Searching for all possible matches within each subject string can be requested by the **/g** or **/G** modifier. After finding a match, PCRE is called again to search the remainder of the subject string. The difference between **/g** and **/G** is that the former uses the *startoffset* argument to **pcre_exec()** to start searching at a new point within the entire string (which is in effect what Perl does), whereas the latter passes over a shortened substring. This makes a difference to the matching process if the pattern begins with a lookbehind assertion (including \b or \B).

If any call to **pcre_exec()** in a **/g** or **/G** sequence matches an empty string, the next call is done with the PCRE_NOTEMPTY and PCRE_ANCHORED flags set in order to search for another, non-empty, match at the same point. If this second match fails, the start offset is advanced by one, and the normal match is retried. This imitates the way Perl handles such cases when using the **/g** modifier or the **split()** function.

There are a number of other modifiers for controlling the way **pcretest** operates.

The **/+** modifier requests that as well as outputting the substring that matched the entire pattern, pcretest should in addition output the remainder of the subject string. This is useful for tests where the subject contains multiple copies of the same substring.

The **/L** modifier must be followed directly by the name of a locale, for example,

```
/pattern/Lfr
```

For this reason, it must be the last modifier letter. The given locale is set, **pcre_maketables()** is called to build a set of character tables for the locale, and this is then passed to **pcre_compile()** when compiling the regular expression. Without an **/L** modifier, NULL is passed as the tables pointer; that is, **/L** applies only to the expression on which it appears.

The **/I** modifier requests that **pcretest** output information about the compiled expression (whether it is anchored, has a fixed first character, and so on). It does this by calling **pcre_fullinfo()** after compiling an expression, and outputting the information it gets back. If the pattern is studied, the results of that are also output.

The **/D** modifier is a PCRE debugging feature, which also assumes **/I**. It causes the internal form of compiled regular expressions to be output after compilation.

The **/S** modifier causes **pcre_study()** to be called after the expression has been compiled, and the results used when the expression is matched.

The **/M** modifier causes the size of memory block used to hold the compiled pattern to be output.

The **/P** modifier causes **pcretest** to call PCRE via the POSIX wrapper API rather than its native API.

When this is done, all other modifiers except **/i**, **/m**, and **/+** are ignored. REG_ICASE is set if **/i** is present, and REG_NEWLINE is set if **/m** is present. The wrapper functions force PCRE_DOLLAR_ENDONLY always, and PCRE_DOTALL unless REG_NEWLINE is set.

The **/8** modifier causes **pcretest** to call PCRE with the PCRE_UTF8 option set. This turns on the (currently incomplete) support for UTF-8 character handling in PCRE, provided that it was compiled with this support enabled. This modifier also causes any non-printing characters in output strings to be printed using the \x{hh...} notation if they are valid UTF-8 sequences.

DATA LINES

Before each data line is passed to **pcre_exec()**, leading and trailing whitespace is removed, and it is then scanned for \ escapes. The following are recognized:

```
\a         alarm (= BEL)
\b         backspace
\e         escape
\f         formfeed
\n         newline
\r         carriage return
\t         tab
\v         vertical tab
\nnn       octal character (up to 3 octal digits)
\xhh       hexadecimal character (up to 2 hex digits)
\x{hh...}  hexadecimal UTF-8 character

\A         pass the PCRE_ANCHORED option to pcre_exec()
\B         pass the PCRE_NOTBOL option to pcre_exec()
\Cdd       call pcre_copy_substring() for substring dd
           after a successful match (any decimal number
           less than 32)
\Gdd       call pcre_get_substring() for substring dd
           after a successful match (any decimal number
           less than 32)
\L         call pcre_get_substringlist() after a
           successful match
\N         pass the PCRE_NOTEMPTY option to pcre_exec()
\Odd       set the size of the output vector passed to
           pcre_exec() to dd (any number of decimal
           digits)
\Z         pass the PCRE_NOTEOL option to pcre_exec()
```

When \O is used, it may be higher or lower than the size set by the **-O** option (or defaulted to 45); \O applies only to the call of **pcre_exec()** for the line in which it appears.

A backslash followed by anything else just escapes the anything else. If the very last character is a backslash, it is ignored. This gives a way of passing an empty line as data, since a real empty line terminates the data input.

If **/P** was present on the regex, causing the POSIX wrapper API to be used, only 0 causing REG_NOTBOL and REG_NOTEOL to be passed to **regexec()** respectively.

The use of \x{hh...} to represent UTF-8 characters is not dependent on the use of the **/8** modifier on the pattern. It is recognized always. There may be any number of hexadecimal digits inside the braces. The result is from one to six bytes, encoded according to the UTF-8 rules.

OUTPUT FROM PCRETEST

When a match succeeds, pcretest outputs the list of captured substrings that **pcre_exec()** returns, starting with number 0 for the string that matched the whole pattern. Here is an example of an interactive pcretest run.

```
$ pcretest
PCRE version 2.06 08-Jun-1999

  re> /^abc(\d+)/
data> abc123
 0: abc123
 1: 123
data> xyz
No match
```

If the strings contain any non-printing characters, they are output as \0x escapes, or as \x{...} escapes if the **/8** modifier was present on the pattern. If the pattern has the **/+** modifier, then the output for substring 0 is followed by the the rest of the subject string, identified by "0+" like this:

```
  re> /cat/+
data> cataract
 0: cat
 0+ aract
```

If the pattern has the **/g** or **/G** modifier, the results of successive matching attempts are output in sequence, like this:

```
  re> /\Bi(\w\w)/g
data> Mississippi
 0: iss
 1: ss
 0: iss
 1: ss
 0: ipp
 1: pp
```

"No match" is output only if the first match attempt fails.

If any of the sequences **\C**, **\G**, or **\L** are present in a data line that is successfully matched, the substrings extracted by the convenience functions are output with C, G, or L after the string number instead of a colon. This is in addition to the normal full list. The string length (that is, the return from the extraction function) is given in parentheses after each string for **\C** and **\G**.

Note that while patterns can be continued over several lines (a plain ">" prompt is used for continuations), data lines may not. However newlines can be included in data by means of the \n escape.

AUTHOR

Philip Hazel <ph10@cam.ac.uk>
University Computing Service,
New Museums Site,
Cambridge CB2 3QG, England.
Phone: +44 1223 334714

Last updated: 15 August 2001

NAME

pcre - Perl-compatible regular expressions.

SYNOPSIS

#include <pcre.h>

**pcre *pcre_compile(const char ****pattern*, int *options*,
**const char ****errptr*, int **erroffset*,
**const unsigned char ***tableptr*);

**pcre_extra *pcre_study(const pcre ***code*, int *options*,
**const char ****errptr*);

**int pcre_exec(const pcre ***code*, const pcre_extra **extra*,
**const char ***subject*, int *length*, int *startoffset*,
int *options*, int **ovector*, int *ovecsize*);

**int pcre_copy_substring(const char ***subject*, int **ovector*,
int *stringcount*, int *stringnumber*, char **buffer*,
int *buffersize*);

**int pcre_get_substring(const char ***subject*, int **ovector*,
int *stringcount*, int *stringnumber*,
**const char ****stringptr*);

**int pcre_get_substring_list(const char ***subject*,
**int ***ovector*, int *stringcount*, const char ****listptr*);

**void pcre_free_substring(const char ***stringptr*);

**void pcre_free_substring_list(const char ****stringptr*);

const unsigned char *pcre_maketables(void);

**int pcre_fullinfo(const pcre ***code*, const pcre_extra **extra*,
int *what*, void **where*);

**int pcre_info(const pcre ***code*, int **optptr*, int ***firstcharptr*);

char *pcre_version(void);

void *(*pcre_malloc)(size_t);

void (*pcre_free)(void *);

DESCRIPTION

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5, with just a few differences (see below). The current implementation corresponds to Perl 5.005, with some additional features from later versions. This includes some experimental, incomplete support for UTF-8 encoded strings. Details of exactly what is and what is not supported are given below.

PCRE has its own native API, which is described in this document. There is also a set of wrapper functions that correspond to the POSIX regular expression API. These are described in the **pcreposix** documentation.

The native API function prototypes are defined in the header file **pcre.h**, and on Unix systems the library itself is called **libpcre.a**, so can be accessed by adding **-lpcre** to the command for linking an application which calls it. The header file defines the macros PCRE_MAJOR and PCRE_MINOR to contain the major and minor release numbers for the library. Applications can use these to include support for different releases.

The functions **pcre_compile()**, **pcre_study()**, and **pcre_exec()** are used for compiling and matching regular expressions. A sample program that demonstrates the simplest way of using them is given in the file *pcredemo.c*. The last section of this man page describes how to run it.

The functions **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are convenience functions for extracting captured substrings from a matched subject string; **pcre_free_substring()** and **pcre_free_substring_list()** are also provided, to free the memory used for extracted strings.

The function **pcre_maketables()** is used (optionally) to build a set of character tables in the current locale for passing to **pcre_compile()**.

The function **pcre_fullinfo()** is used to find out information about a compiled pattern; **pcre_info()** is an obsolete version which returns only some of the available information, but is retained for backwards compatibility. The function **pcre_version()** returns a pointer to a string containing the version of PCRE and its date of release.

The global variables **pcre_malloc** and **pcre_free** initially contain the entry points of the standard **malloc()** and **free()** functions respectively. PCRE calls the memory management functions via these variables, so a calling program can replace them if it wishes to intercept the calls. This should be done before calling any PCRE functions.

MULTI-THREADING

The PCRE functions can be used in multi-threading applications, with the proviso that the memory management functions pointed to by **pcre_malloc** and **pcre_free** are shared by all threads.

The compiled form of a regular expression is not altered during matching, so the same compiled pattern can safely be used by several threads at once.

COMPILING A PATTERN

The function **pcre_compile()** is called to compile a pattern into an internal form. The pattern is a C string terminated by a binary zero, and is passed in the argument *pattern*. A pointer to a single block of memory that is obtained via **pcre_malloc** is returned. This contains the compiled code and related data. The **pcre** type is defined for the returned block; this is a typedef for a structure whose contents are not externally defined. It is up to the caller to free the memory when it is no longer required.

Although the compiled code of a PCRE regex is relocatable, that is, it does not depend on memory location, the complete **pcre** data block is not fully relocatable, because it contains a copy of the *tableptr* argument, which is an address (see below).

The size of a compiled pattern is roughly proportional to the length of the pattern string, except that each character class (other than those containing just a single character, negated or not) requires 33 bytes, and repeat quantifiers with a minimum greater than one or a bounded maximum cause the relevant portions of the compiled pattern to be replicated.

The *options* argument contains independent bits that affect the compilation. It should be zero if no options are required. Some of the options, in particular, those that are compatible with Perl, can also be set and unset from within the pattern (see the detailed description of regular expressions below). For these options, the contents of the *options* argument specifies their initial settings at the start of compilation and execution. The PCRE_ANCHORED option can be set at the time of matching as well as at compile time.

If *errptr* is NULL, **pcre_compile()** returns NULL immediately. Otherwise, if compilation of a pattern fails, **pcre_compile()** returns NULL, and sets the variable pointed to by *errptr* to point to a textual error message. The offset from the start of the pattern to the character where the error was discovered is placed in the variable pointed to by *erroffset*, which must not be NULL. If it is, an immediate error is given.

If the final argument, *tableptr*, is NULL, PCRE uses a default set of character tables which are built when it is compiled, using the default C locale. Otherwise, *tableptr* must be the result of a call to

pcre_maketables(). See the section on locale support below.

This code fragment shows a typical straightforward call to **pcre_compile()**:

```
pcre *re;
const char *error;
int erroffset;
re = pcre_compile(
  "^A.*Z",          /* the pattern */
  0,                /* default options */
  &error,           /* for error message */
  &erroffset,       /* for error offset */
  NULL);            /* use default character tables */
```

The following option bits are defined in the header file:

PCRE_ANCHORED

If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the start of the string which is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

PCRE_CASELESS

If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option.

PCRE_DOLLAR_ENDONLY

If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before the final character if it is a newline (but not before any other newlines). The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set. There is no equivalent to this option in Perl.

PCRE_DOTALL

If this bit is set, a dot metacharater in the pattern matches all characters, including newlines. Without it, newlines are excluded. This option is equivalent to Perl's /s option. A negative class such as [^a] always matches a newline character, independent of the setting of this option.

PCRE_EXTENDED

If this bit is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class, and characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This is equivalent to Perl's /x option, and makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

PCRE_EXTRA

This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

PCRE_MULTILINE

By default, PCRE treats the subject string as consisting of a single "line" of characters (even if it actually contains several newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless PCRE_DOLLAR_ENDONLY is set). This is the same as Perl.

When PCRE_MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option. If there are no "\n" characters in a subject string, or no occurrences of ^ or $ in a pattern, setting PCRE_MULTILINE has no effect.

PCRE_UNGREEDY

This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

PCRE_UTF8

This option causes PCRE to regard both the pattern and the subject as strings of UTF-8 characters instead of just byte strings. However, it is available only if PCRE has been built to include UTF-8 support. If not, the use of this option provokes an error. Support for UTF-8 is new, experimental, and incomplete. Details of exactly what it entails are given below.

STUDYING A PATTERN

When a pattern is going to be used several times, it is worth spending more time analyzing it in order to speed up the time taken for matching. The function **pcre_study()** takes a pointer to a compiled pattern as its first argument, and returns a pointer to a **pcre_extra** block (another typedef for a structure with hidden contents) containing additional information about the pattern; this can be passed to **pcre_exec()**. If no additional information is available, NULL is returned.

The second argument contains option bits. At present, no options are defined for **pcre_study()**, and this argument should always be zero.

The third argument for **pcre_study()** is a pointer to an error message. If studying succeeds (even if no data is returned), the variable it points to is set to NULL. Otherwise it points to a textual error message.

This is a typical call to **pcre_study**():

```
pcre_extra *pe;
pe = pcre_study(
  re,           /* result of pcre_compile() */
  0,            /* no options exist */
  &error);      /* set to NULL or points to a message */
```

At present, studying a pattern is useful only for non-anchored patterns that do not have a single fixed starting character. A bitmap of possible starting characters is created.

LOCALE SUPPORT

PCRE handles caseless matching, and determines whether characters are letters, digits, or whatever, by reference to a set of tables. The library contains a default set of tables which is created in the default C locale when PCRE is compiled. This is used when the final argument of **pcre_compile()** is NULL, and is sufficient for many applications.

An alternative set of tables can, however, be supplied. Such tables are built by calling the **pcre_maketables()** function, which has no arguments, in the relevant locale. The result can then be passed to **pcre_compile()** as often as necessary. For example, to build and use tables that are

appropriate for the French locale (where accented characters with codes greater than 128 are treated as letters), the following code could be used:

```
setlocale(LC_CTYPE, "fr");
tables = pcre_maketables();
re = pcre_compile(..., tables);
```

The tables are built in memory that is obtained via **pcre_malloc**. The pointer that is passed to **pcre_compile** is saved with the compiled pattern, and the same tables are used via this pointer by **pcre_study()** and **pcre_exec()**. Thus for any single pattern, compilation, studying and matching all happen in the same locale, but different patterns can be compiled in different locales. It is the caller's responsibility to ensure that the memory containing the tables remains available for as long as it is needed.

INFORMATION ABOUT A PATTERN

The **pcre_fullinfo()** function returns information about a compiled pattern. It replaces the obsolete **pcre_info()** function, which is nevertheless retained for backwards compability (and is documented below).

The first argument for **pcre_fullinfo()** is a pointer to the compiled pattern. The second argument is the result of **pcre_study()**, or NULL if the pattern was not studied. The third argument specifies which piece of information is required, while the fourth argument is a pointer to a variable to receive the data. The yield of the function is zero for success, or one of the following negative numbers:

```
PCRE_ERROR_NULL       the argument code was NULL
                      the argument where was NULL
PCRE_ERROR_BADMAGIC   the "magic number" was not found
PCRE_ERROR_BADOPTION  the value of what was invalid
```

Here is a typical call of **pcre_fullinfo()**, to obtain the length of the compiled pattern:

```
int rc;
unsigned long int length;
rc = pcre_fullinfo(
  re,             /* result of pcre_compile() */
  pe,             /* result of pcre_study(), or NULL */
  PCRE_INFO_SIZE,   /* what is required */
  &length);       /* where to put the data */
```

The possible values for the third argument are defined in **pcre.h**, and are as follows:

```
PCRE_INFO_OPTIONS
```

Return a copy of the options with which the pattern was compiled. The fourth argument should point to an **unsigned long int** variable. These option bits are those specified in the call to **pcre_compile()**, modified by any top-level option settings within the pattern itself, and with the PCRE_ANCHORED bit forcibly set if the form of the pattern implies that it can match only at the start of a subject string.

```
PCRE_INFO_SIZE
```

Return the size of the compiled pattern, that is, the value that was passed as the argument to **pcre_malloc()** when PCRE was getting memory in which to place the compiled data. The fourth argument should point to a **size_t** variable.

```
PCRE_INFO_CAPTURECOUNT
```

Return the number of capturing subpatterns in the pattern. The fourth argument should point to an int

variable.

PCRE_INFO_BACKREFMAX

Return the number of the highest back reference in the pattern. The fourth argument should point to an **int** variable. Zero is returned if there are no back references.

PCRE_INFO_FIRSTCHAR

Return information about the first character of any matched string, for a non-anchored pattern. If there is a fixed first character, e.g. from a pattern such as (cat|cow|coyote), it is returned in the integer pointed to by *where*. Otherwise, if either

(a) the pattern was compiled with the PCRE_MULTILINE option, and every branch starts with "^ ", or

(b) every branch of the pattern starts with ".*" and PCRE_DOTALL is not set (if it were set, the pattern would be anchored),

-1 is returned, indicating that the pattern matches only at the start of a subject string or after any "\n" within the string. Otherwise -2 is returned. For anchored patterns, -2 is returned.

PCRE_INFO_FIRSTTABLE

If the pattern was studied, and this resulted in the construction of a 256-bit table indicating a fixed set of characters for the first character in any matching string, a pointer to the table is returned. Otherwise NULL is returned. The fourth argument should point to an **unsigned char *** variable.

PCRE_INFO_LASTLITERAL

For a non-anchored pattern, return the value of the rightmost literal character which must exist in any matched string, other than at its start. The fourth argument should point to an **int** variable. If there is no such character, or if the pattern is anchored, -1 is returned. For example, for the pattern /a\d+z\d+/ the returned value is 'z'.

The **pcre_info()** function is now obsolete because its interface is too restrictive to return all the available data about a compiled pattern. New programs should use **pcre_fullinfo()** instead. The yield of **pcre_info()** is the number of capturing subpatterns, or one of the following negative numbers:

PCRE_ERROR_NULL the argument *code* was NULL
PCRE_ERROR_BADMAGIC the "magic number" was not found

If the *optptr* argument is not NULL, a copy of the options with which the pattern was compiled is placed in the integer it points to (see PCRE_INFO_OPTIONS above).

If the pattern is not anchored and the *firstcharptr* argument is not NULL, it is used to pass back information about the first character of any matched string (see PCRE_INFO_FIRSTCHAR above).

MATCHING A PATTERN

The function **pcre_exec()** is called to match a subject string against a pre-compiled pattern, which is passed in the *code* argument. If the pattern has been studied, the result of the study should be passed in the *extra* argument. Otherwise this must be NULL.

Here is an example of a simple call to **pcre_exec()**:

```
int rc;
int ovector[30];
rc = pcre_exec(
```

```
re,           /* result of pcre_compile() */
NULL,         /* we didn't study the pattern */
"some string",  /* the subject string */
11,           /* the length of the subject string */
0,            /* start at offset 0 in the subject */
0,            /* default options */
ovector,      /* vector for substring information */
30);          /* number of elements in the vector */
```

The PCRE_ANCHORED option can be passed in the *options* argument, whose unused bits must be zero. However, if a pattern was compiled with PCRE_ANCHORED, or turned out to be anchored by virtue of its contents, it cannot be made unachored at matching time.

There are also three further options that can be set only at matching time:

PCRE_NOTBOL

The first character of the string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without PCRE_MULTILINE (at compile time) causes circumflex never to match.

PCRE_NOTEOL

The end of the string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without PCRE_MULTILINE (at compile time) causes dollar never to match.

PCRE_NOTEMPTY

An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern

```
a?b?
```

is applied to a string not beginning with "a" or "b", it matches the empty string at the start of the subject. With PCRE_NOTEMPTY set, this match is not valid, so PCRE searches further into the string for occurrences of "a" or "b".

Perl has no direct equivalent of PCRE_NOTEMPTY, but it does make a special case of a pattern match of the empty string within its **split()** function, and when using the /g modifier. It is possible to emulate Perl's behaviour after matching a null string by first trying the match again at the same offset with PCRE_NOTEMPTY set, and then if that fails by advancing the starting offset (see below) and trying an ordinary match again.

The subject string is passed as a pointer in *subject*, a length in *length*, and a starting offset in *startoffset*. Unlike the pattern string, the subject may contain binary zero characters. When the starting offset is zero, the search for a match starts at the beginning of the subject, and this is by far the most common case.

A non-zero starting offset is useful when searching for another match in the same subject by calling **pcre_exec()** again after a previous success. Setting *startoffset* differs from just passing over a shortened string and setting PCRE_NOTBOL in the case of a pattern that begins with any kind of lookbehind. For example, consider the pattern

```
\Biss\B
```

which finds occurrences of "iss" in the middle of words. (\B matches only if the current position in the subject is not a word boundary.) When applied to the string "Mississipi" the first call to **pcre_exec()**

finds the first occurrence. If **pcre_exec()** is called again with just the remainder of the subject, namely "issipi", it does not match, because \B is always false at the start of the subject, which is deemed to be a word boundary. However, if **pcre_exec()** is passed the entire string again, but with *startoffset* set to 4, it finds the second occurrence of "iss" because it is able to look behind the starting point to discover that it is preceded by a letter.

If a non-zero starting offset is passed when the pattern is anchored, one attempt to match at the given offset is tried. This can only succeed if the pattern does not require the match to be at the start of the subject.

In general, a pattern matches a certain portion of the subject, and in addition, further substrings from the subject may be picked out by parts of the pattern. Following the usage in Jeffrey Friedl's book, this is called "capturing" in what follows, and the phrase "capturing subpattern" is used for a fragment of a pattern that picks out a substring. PCRE supports several other kinds of parenthesized subpattern that do not cause substrings to be captured.

Captured substrings are returned to the caller via a vector of integer offsets whose address is passed in *ovector*. The number of elements in the vector is passed in *ovecsize*. The first two-thirds of the vector is used to pass back captured substrings, each substring using a pair of integers. The remaining third of the vector is used as workspace by **pcre_exec()** while matching capturing subpatterns, and is not available for passing back information. The length passed in *ovecsize* should always be a multiple of three. If it is not, it is rounded down.

When a match has been successful, information about captured substrings is returned in pairs of integers, starting at the beginning of *ovector*, and continuing up to two-thirds of its length at the most. The first element of a pair is set to the offset of the first character in a substring, and the second is set to the offset of the first character after the end of a substring. The first pair, *ovector[0]* and *ovector[1]*, identify the portion of the subject string matched by the entire pattern. The next pair is used for the first capturing subpattern, and so on. The value returned by **pcre_exec()** is the number of pairs that have been set. If there are no capturing subpatterns, the return value from a successful match is 1, indicating that just the first pair of offsets has been set.

Some convenience functions are provided for extracting the captured substrings as separate strings. These are described in the following section.

It is possible for an capturing subpattern number *n+1* to match some part of the subject when subpattern *n* has not been used at all. For example, if the string "abc" is matched against the pattern (a|(z))(bc) subpatterns 1 and 3 are matched, but 2 is not. When this happens, both offset values corresponding to the unused subpattern are set to -1.

If a capturing subpattern is matched repeatedly, it is the last portion of the string that it matched that gets returned.

If the vector is too small to hold all the captured substrings, it is used as far as possible (up to two-thirds of its length), and the function returns a value of zero. In particular, if the substring offsets are not of interest, **pcre_exec()** may be called with *ovector* passed as NULL and *ovecsize* as zero. However, if the pattern contains back references and the *ovector* isn't big enough to remember the related substrings, PCRE has to get additional memory for use during matching. Thus it is usually advisable to supply an *ovector*.

Note that **pcre_info()** can be used to find out how many capturing subpatterns there are in a compiled pattern. The smallest size for *ovector* that will allow for *n* captured substrings in addition to the offsets of the substring matched by the whole pattern is (*n*+1)*3.

If **pcre_exec()** fails, it returns a negative number. The following are defined in the header file:

PCRE_ERROR_NOMATCH (-1)

The subject string did not match the pattern.

PCRE_ERROR_NULL (-2)

Either *code* or *subject* was passed as NULL, or *ovector* was NULL and *ovecsize* was not zero.

PCRE_ERROR_BADOPTION (-3)

An unrecognized bit was set in the *options* argument.

PCRE_ERROR_BADMAGIC (-4)

PCRE stores a 4-byte "magic number" at the start of the compiled code, to catch the case when it is passed a junk pointer. This is the error it gives when the magic number isn't present.

PCRE_ERROR_UNKNOWN_NODE (-5)

While running the pattern match, an unknown item was encountered in the compiled pattern. This error could be caused by a bug in PCRE or by overwriting of the compiled pattern.

PCRE_ERROR_NOMEMORY (-6)

If a pattern contains back references, but the *ovector* that is passed to **pcre_exec()** is not big enough to remember the referenced substrings, PCRE gets a block of memory at the start of matching to use for this purpose. If the call via **pcre_malloc()** fails, this error is given. The memory is freed at the end of matching.

EXTRACTING CAPTURED SUBSTRINGS

Captured substrings can be accessed directly by using the offsets returned by **pcre_exec()** in *ovector*. For convenience, the functions **pcre_copy_substring()**, **pcre_get_substring()**, and **pcre_get_substring_list()** are provided for extracting captured substrings as new, separate, zero-terminated strings. A substring that contains a binary zero is correctly extracted and has a further zero added on the end, but the result does not, of course, function as a C string.

The first three arguments are the same for all three functions: *subject* is the subject string which has just been successfully matched, *ovector* is a pointer to the vector of integer offsets that was passed to **pcre_exec()**, and *stringcount* is the number of substrings that were captured by the match, including the substring that matched the entire regular expression. This is the value returned by **pcre_exec** if it is greater than zero. If **pcre_exec()** returned zero, indicating that it ran out of space in *ovector*, the value passed as *stringcount* should be the size of the vector divided by three.

The functions **pcre_copy_substring()** and **pcre_get_substring()** extract a single substring, whose number is given as *stringnumber*. A value of zero extracts the substring that matched the entire pattern, while higher values extract the captured substrings. For **pcre_copy_substring()**, the string is placed in *buffer*, whose length is given by *buffersize*, while for **pcre_get_substring()** a new block of memory is obtained via **pcre_malloc**, and its address is returned via *stringptr*. The yield of the function is the length of the string, not including the terminating zero, or one of

PCRE_ERROR_NOMEMORY (-6)

The buffer was too small for **pcre_copy_substring()**, or the attempt to get memory failed for **pcre_get_substring()**.

PCRE_ERROR_NOSUBSTRING (-7)

There is no substring whose number is *stringnumber*.

The **pcre_get_substring_list()** function extracts all available substrings and builds a list of pointers to them. All this is done in a single block of memory which is obtained via **pcre_malloc**. The address of

the memory block is returned via *listptr*, which is also the start of the list of string pointers. The end of the list is marked by a NULL pointer. The yield of the function is zero if all went well, or

PCRE_ERROR_NOMEMORY (-6)

if the attempt to get the memory block failed.

When any of these functions encounter a substring that is unset, which can happen when capturing subpattern number *n+1* matches some part of the subject, but subpattern *n* has not been used at all, they return an empty string. This can be distinguished from a genuine zero-length substring by inspecting the appropriate offset in *ovector*, which is negative for unset substrings.

The two convenience functions **pcre_free_substring()** and **pcre_free_substring_list()** can be used to free the memory returned by a previous call of **pcre_get_substring()** or **pcre_get_substring_list()**, respectively. They do nothing more than call the function pointed to by **pcre_free**, which of course could be called directly from a C program. However, PCRE is used in some situations where it is linked via a special interface to another programming language which cannot use **pcre_free** directly; it is for these cases that the functions are provided.

LIMITATIONS

There are some size limitations in PCRE but it is hoped that they will never in practice be relevant. The maximum length of a compiled pattern is 65539 (sic) bytes. All values in repeating quantifiers must be less than 65536. There maximum number of capturing subpatterns is 65535. There is no limit to the number of non-capturing subpatterns, but the maximum depth of nesting of all kinds of parenthesized subpattern, including capturing subpatterns, assertions, and other types of subpattern, is 200.

The maximum length of a subject string is the largest positive number that an integer variable can hold. However, PCRE uses recursion to handle subpatterns and indefinite repetition. This means that the available stack space may limit the size of a subject string that can be processed by certain patterns.

DIFFERENCES FROM PERL

The differences described here are with respect to Perl 5.005.

1. By default, a whitespace character is any character that the C library function **isspace()** recognizes, though it is possible to compile PCRE with alternative character type tables. Normally **isspace()** matches space, formfeed, newline, carriage return, horizontal tab, and vertical tab. Perl 5 no longer includes vertical tab in its set of whitespace characters. The \v escape that was in the Perl documentation for a long time was never in fact recognized. However, the character itself was treated as whitespace at least up to 5.002. In 5.004 and 5.005 it does not match \s.

2. PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do not mean what you might think. For example, (?!a){3} does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.

3. Capturing subpatterns that occur inside negative lookahead assertions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeeding), but only if the negative lookahead assertion contains just one branch.

4. Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence "\0" can be used in the pattern to represent a binary zero.

5. The following Perl escape sequences are not supported: \l, \u, \L, \U, \E, \Q. In fact these are implemented by Perl's general string-handling and are not part of its pattern matching engine.

6. The Perl \G assertion is not supported as it is not relevant to single pattern matches.

7. Fairly obviously, PCRE does not support the (?{code}) and (?p{code}) constructions. However, there is some experimental support for recursive patterns using the non-Perl item (?R).

8. There are at the time of writing some oddities in Perl 5.005_02 concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern /^ (a(b)?)+$/ sets $2 to the value "b", but matching "aabbaa" against /^ (aa(bb)?)+$/ leaves $2 unset. However, if the pattern is changed to /^ (aa(b(b))?)+$/ then $2 (and $3) are set.

In Perl 5.004 $2 is set in both cases, and that is also true of PCRE. If in the future Perl changes to a consistent state that is different, PCRE may change to follow.

9. Another as yet unresolved discrepancy is that in Perl 5.005_02 the pattern /^ (a)?(?(1)a|b)+$/ matches the string "a", whereas in PCRE it does not. However, in both Perl and PCRE /^ (a)?a/ matched against "a" leaves $1 unset.

10. PCRE provides some extensions to the Perl regular expression facilities:

(a) Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl 5.005 requires them all to have the same length.

(b) If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the $ meta- character matches only at the very end of the string.

(c) If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted.

(d) If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

(e) PCRE_ANCHORED can be used to force a pattern to be tried only at the start of the subject.

(f) The PCRE_NOTBOL, PCRE_NOTEOL, and PCRE_NOTEMPTY options for **pcre_exec()** have no Perl equivalents.

(g) The (?R) construct allows for recursive pattern matching (Perl 5.6 can do this using the (?p{code}) construct, which PCRE cannot of course support.)

REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1-56592-257), covers them in great detail.

The description here is intended as reference documentation. The basic operation of PCRE is on strings of bytes. However, there is the beginnings of some support for UTF-8 character strings. To use this support you must configure PCRE to include it, and then call **pcre_compile()** with the PCRE_UTF8 option. How this affects the pattern matching is described in the final section of this document.

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. 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 *meta-characters*, which do not stand for themselves but instead are interpreted in some special way.

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

```
\      general escape character with several uses
^       assert start of subject (or line, in multiline mode)
$      assert end of subject (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
{      start min/max quantifier
```

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

```
\      general escape character
^       negate the class, but only if the first character
-      indicates character range
]      terminates the character class
```

The following sections describe the use of each of the meta-characters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a "#" outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or "#" character as part of the pattern.

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 usually easier to use one of the following escape sequences than the binary character it represents:

```
\a     alarm, that is, the BEL character (hex 07)
\cx    "control-x", where x is any character
\e     escape (hex 1B)
\f     formfeed (hex 0C)
```

```
\n     newline (hex 0A)
\r     carriage return (hex 0D)
\t     tab (hex 09)
\xhh   character with hex code hh
\ddd   character with octal code ddd, or backreference
```

The precise effect of "\cx" is as follows: if "x" is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus "\cz" becomes hex 1A, but "\c{" becomes hex 3B, while "\c;" becomes hex 7B.

After "\x", up to two hexadecimal digits are read (letters can be in upper or lower case).

After "\0" up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence "\0\x\07" specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

```
\040   is another way of writing a space
\40    is the same, provided there are fewer than 40
         previous capturing subpatterns
\7     is always a back reference
\11    might be a back reference, or another way of
         writing a tab
\011   is always a tab
\0113  is a tab followed by the character "3"
\113   is the character with octal code 113 (since there
         can be no more than 99 back references)
\377   is a byte consisting entirely of 1 bits
\81    is either a back reference, or a binary zero
         followed by the two characters "8" and "1"
```

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

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence "\b" is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

```
\d     any decimal digit
\D     any character that is not a decimal digit
\s     any whitespace character
\S     any character that is not a whitespace character
\w     any "word" character
\W     any "non-word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given

character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale- specific matching is taking place (see "Locale support" above). For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by \w.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth 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 word boundary
\B not a word boundary
\A start of subject (independent of multiline mode)
\Z end of subject or newline at end (independent of multiline mode)
\z end of subject (independent of multiline mode)

These assertions may not appear in character classes (but note that "\b" has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options. If the *startoffset* argument of **pcre_exec()** is non-zero, \A can never match. The difference between \Z and \z is that \Z matches before a newline that is the last character of the string as well as at the end of the string, whereas \z matches only at the end.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion which 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. Inside a character class, circumflex has an entirely different meaning (see below).

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

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile or matching 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, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^ abc$/ matches the subject string "def\nabc" 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 PCRE_MULTILINE is set or not.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the PCRE_DOTALL option is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

SQUARE BRACKETS

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. 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; the character must be in the set of characters defined by the class, unless the first character in the class 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 which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

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.

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE_DOTALL or PCRE_MULTILINE options is. A class such as [^ a] will always match a newline.

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

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000-\037]. 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 if character tables for the "fr" locale are in use, [\xc8-\xcb] matches accented E characters in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^ \W_] matches any letter or digit, but not underscore.

All non-alphameric characters other than \, -, ^ (at the start) and the terminating] are non-special in character classes, but it does no harm if they are escaped.

POSIX CHARACTER CLASSES

Perl 5.6 (not yet released at the time of writing) is going to support the POSIX notation for character classes, which uses names enclosed by [: and :] within the enclosing square brackets. PCRE 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
cntrl    control characters
digit    decimal digits (same as \d)
graph    printing characters, excluding space
lower    lower case letters
print    printing characters, including space
punct    printing characters, excluding letters and digits
space    white space (same as \s)
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The names "ascii" and "word" are Perl extensions. 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.

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 PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED 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 scope of these option changes depends on where in the pattern the setting occurs. For settings that are outside any subpattern (defined below), the effect is the same as if the options were set or unset at the start of matching. The following patterns all behave in exactly the same way:

```
(?i)abc
a(?i)bc
ab(?i)c
abc(?i)
```

which in turn is the same as compiling the pattern abc with PCRE_CASELESS set. In other words, such "top level" settings apply to the whole pattern (unless there are other changes inside subpatterns). If there is more than one setting of the same option at top level, the rightmost setting is used.

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern 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.

The PCRE-specific options PCRE_UNGREEDY and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

SUBPATTERNS

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

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

cat(aract|erpillar|)

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

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the *ovector* argument of **pcre_exec()**. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of 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 "?:", 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 captured substrings is 99, and the maximum number of all subpatterns, both capturing and non-capturing, is 200.

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".

REPETITION

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

a single character, possibly escaped
the . metacharacter
a character class
a back reference (see next section)
a parenthesized subpattern (unless it is an assertion - see below)

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.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) 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 the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern

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

to the string

```
/* first command */  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 which 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 store 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 . 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 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 when the pattern begins with .* in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

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".

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 (i.e. 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. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. 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 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. For example, the pattern

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

always fails if it starts to match "a" rather than "bc". Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

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.

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, \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. 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 this effect.

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 alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, 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 if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

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 which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

ONCE-ONLY SUBPATTERNS

With both maximizing and minimizing 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. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re-evaluated in this way, so the matcher would give up immediately on failing to match "foo" the first time. The notation is another kind of special parenthesis, starting with (?> as in this example:

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

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.

Once-only 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.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string which 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.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once-only subpattern 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 two repeats in a large number of ways, and all have to be tried. (The example used [!?] 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 to

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

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

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 previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

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

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

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

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

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

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

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

COMMENTS

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a

comment that continues up to the next newline character in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 has provided an experimental 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 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, the special item (?R) is provided for the specific case of recursion. This PCRE pattern solves the 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 (i.e. a correctly parenthesized substring). Finally there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once-only subpattern for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when it is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if a once-only subpattern 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.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against

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

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

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

the string they capture is "ab(cd)ef", the contents of the top level parentheses. 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, it saves data for the first 15 capturing parentheses only, as there is no way to give an out-of-memory error from within a recursion.

PERFORMANCE

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like [aeiou] than a set of alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of discussion about optimizing regular expressions for efficient performance.

When a pattern begins with .* and the PCRE_DOTALL option is set, the pattern is implicitly anchored

by PCRE, since it can match only at the start of a subject string. However, if PCRE_DOTALL is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

(.*) second

matches the subject "first\nand second" (where \n stands for a newline character) with the first captured substring being "and". In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE_DOTALL, or starting the pattern with ^ .* to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

(a+)*

This can match "aaaa" in 33 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as

(a+)*b

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

(a+)*\d

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

UTF-8 SUPPORT

Starting at release 3.3, PCRE has some support for character strings encoded in the UTF-8 format. This is incomplete, and is regarded as experimental. In order to use it, you must configure PCRE to include UTF-8 support in the code, and, in addition, you must call **pcre_compile()** with the PCRE_UTF8 option flag. When you do this, both the pattern and any subject strings that are matched against it are treated as UTF-8 strings instead of just strings of bytes, but only in the cases that are mentioned below.

If you compile PCRE with UTF-8 support, but do not use it at run time, the library will be a bit bigger, but the additional run time overhead is limited to testing the PCRE_UTF8 flag in several places, so should not be very large.

PCRE assumes that the strings it is given contain valid UTF-8 codes. It does not diagnose invalid UTF-8 strings. If you pass invalid UTF-8 strings to PCRE, the results are undefined.

Running with PCRE_UTF8 set causes these changes in the way PCRE works:

1. In a pattern, the escape sequence \x{...}, where the contents of the braces is a string of hexadecimal

digits, is interpreted as a UTF-8 character whose code number is the given hexadecimal number, for example: \x{1234}. This inserts from one to six literal bytes into the pattern, using the UTF-8 encoding. If a non-hexadecimal digit appears between the braces, the item is not recognized.

2. The original hexadecimal escape sequence, \xhh, generates a two-byte UTF-8 character if its value is greater than 127.

3. Repeat quantifiers are NOT correctly handled if they follow a multibyte character. For example, \x{100}* and \xc3+ do not work. If you want to repeat such characters, you must enclose them in non-capturing parentheses, for example (?:\x{100}), at present.

4. The dot metacharacter matches one UTF-8 character instead of a single byte.

5. Unlike literal UTF-8 characters, the dot metacharacter followed by a repeat quantifier does operate correctly on UTF-8 characters instead of single bytes.

4. Although the \x{...} escape is permitted in a character class, characters whose values are greater than 255 cannot be included in a class.

5. A class is matched against a UTF-8 character instead of just a single byte, but it can match only characters whose values are less than 256. Characters with greater values always fail to match a class.

6. Repeated classes work correctly on multiple characters.

7. Classes containing just a single character whose value is greater than 127 (but less than 256), for example, [\x80] or [^ \x{93}], do not work because these are optimized into single byte matches. In the first case, of course, the class brackets are just redundant.

8. Lookbehind assertions move backwards in the subject by a fixed number of characters instead of a fixed number of bytes. Simple cases have been tested to work correctly, but there may be hidden gotchas herein.

9. The character types such as \d and \w do not work correctly with UTF-8 characters. They continue to test a single byte.

10. Anything not explicitly mentioned here continues to work in bytes rather than in characters.

The following UTF-8 features of Perl 5.6 are not implemented:

1. The escape sequence \C to match a single byte.

2. The use of Unicode tables and properties and escapes \p, \P, and \X.

SAMPLE PROGRAM

The code below is a simple, complete demonstration program, to get you started with using PCRE. This code is also supplied in the file *pcredemo.c* in the PCRE distribution.

The program compiles the regular expression that is its first argument, and matches it against the subject string in its second argument. No options are set, and default character tables are used. If matching succeeds, the program outputs the portion of the subject that matched, together with the contents of any captured substrings.

On a Unix system that has PCRE installed in */usr/local*, you can compile the demonstration program using a command like this:

```
gcc -o pcredemo pcredemo.c -I/usr/local/include -L/usr/local/lib -lpcre
```

Then you can run simple tests like this:

```
./pcredemo 'cat|dog' 'the cat sat on the mat'
```

Note that there is a much more comprehensive test program, called **pcretest**, which supports many more facilities for testing regular expressions. The **pcredemo** program is provided as a simple coding example.

On some operating systems (e.g. Solaris) you may get an error like this when you try to run **pcredemo**:

```
ld.so.1: a.out: fatal: libpcre.so.0: open failed: No such fi le or directory
```

This is caused by the way shared library support works on those systems. You need to add

```
-R/usr/local/lib
```

to the compile command to get round this problem. Here's the code:

```
#include <stdio.h>
#include <string.h>
#include <pcre.h>

#defi ne OVECCOUNT 30    /* should be a multiple of 3 */

int main(int argc, char **argv)
{
pcre *re;
const char *error;
int erroffset;
int ovector[OVECCOUNT];
int rc, i;

if (argc != 3)
  {
  printf("Two arguments required: a regex and a "
    "subject string\n");
  return 1;
  }

/* Compile the regular expression in the fi rst argument */

re = pcre_compile(
  argv[1],     /* the pattern */
  0,           /* default options */
  &error,      /* for error message */
  &erroffset,  /* for error offset */
  NULL);       /* use default character tables */

/* Compilation failed: print the error message and exit */

if (re == NULL)
  {
  printf("PCRE compilation failed at offset %d: %s\n",
    erroffset, error);
  return 1;
  }

/* Compilation succeeded: match the subject in the second
   argument */
```

```
rc = pcre_exec(
  re,         /* the compiled pattern */
  NULL,       /* we didn't study the pattern */
  argv[2],    /* the subject string */
  (int)strlen(argv[2]), /* the length of the subject */
  0,          /* start at offset 0 in the subject */
  0,          /* default options */
  ovector,    /* vector for substring information */
  OVECCOUNT);  /* number of elements in the vector */

/* Matching failed: handle error cases */

if (rc < 0)
  {
  switch(rc)
    {
    case PCRE_ERROR_NOMATCH: printf("No match\n"); break;
    /*
    Handle other special cases if you like
    */
    default: printf("Matching error %d\n", rc); break;
    }
  return 1;
  }

/* Match succeeded */

printf("Match succeeded\n");

/* The output vector wasn't big enough */

if (rc == 0)
  {
  rc = OVECCOUNT/3;
  printf("ovector only has room for %d captured "
    substrings\n", rc - 1);
  }

/* Show substrings stored in the output vector */

for (i = 0; i < rc; i++)
  {
  char *substring_start = argv[2] + ovector[2*i];
  int substring_length = ovector[2*i+1] - ovector[2*i];
  printf("%2d: %.*s\n", i, substring_length,
    substring_start);
  }

return 0;
}
```

AUTHOR

Philip Hazel <ph10@cam.ac.uk>
University Computing Service,
New Museums Site,
Cambridge CB2 3QG, England.
Phone: +44 1223 334714

Last updated: 15 August 2001

NAME

pcreposix - POSIX API for Perl-compatible regular expressions.

SYNOPSIS

#include <pcreposix.h>

**int regcomp(regex_t ***_preg_, const char *_pattern_,
int _cflags_);

**int regexec(regex_t ***_preg_, const char *_string_,
size_t _nmatch_, regmatch_t _pmatch_[], int _eflags_);

size_t regerror(int _errcode_, const regex_t *_preg_,
**char ***_errbuf_, size_t _errbuf_size_);

**void regfree(regex_t ***_preg_);

DESCRIPTION

This set of functions provides a POSIX-style API to the PCRE regular expression package. See the **pcre** documentation for a description of the native API, which contains additional functionality.

The functions described here are just wrapper functions that ultimately call the native API. Their prototypes are defined in the **pcreposix.h** header file, and on Unix systems the library itself is called **pcreposix.a**, so can be accessed by adding **-lpcreposix** to the command for linking an application which uses them. Because the POSIX functions call the native ones, it is also necessary to add -lpcre.

I have implemented only those option bits that can be reasonably mapped to PCRE native options. In addition, the options REG_EXTENDED and REG_NOSUB are defined with the value zero. They have no effect, but since programs that are written to the POSIX interface often use them, this makes it easier to slot in PCRE as a replacement library. Other POSIX options are not even defined.

When PCRE is called via these functions, it is only the API that is POSIX-like in style. The syntax and semantics of the regular expressions themselves are still those of Perl, subject to the setting of various PCRE options, as described below.

The header for these functions is supplied as **pcreposix.h** to avoid any potential clash with other POSIX libraries. It can, of course, be renamed or aliased as **regex.h**, which is the "correct" name. It provides two structure types, *regex_t* for compiled internal forms, and *regmatch_t* for returning captured substrings. It also defines some constants whose names start with "REG_"; these are used for setting options and identifying error codes.

COMPILING A PATTERN

The function **regcomp()** is called to compile a pattern into an internal form. The pattern is a C string terminated by a binary zero, and is passed in the argument *pattern*. The *preg* argument is a pointer to a regex_t structure which is used as a base for storing information about the compiled expression.

The argument *cflags* is either zero, or contains one or more of the bits defined by the following macros:

REG_ICASE

The PCRE_CASELESS option is set when the expression is passed for compilation to the native function.

REG_NEWLINE

The PCRE_MULTILINE option is set when the expression is passed for compilation to the native function.

In the absence of these flags, no options are passed to the native function. This means the the regex is compiled with PCRE default semantics. In particular, the way it handles newline characters in the subject string is the Perl way, not the POSIX way. Note that setting PCRE_MULTILINE has only *some* of the effects specified for REG_NEWLINE. It does not affect the way newlines are matched by . (they aren't) or a negative class such as [^a] (they are).

The yield of **regcomp()** is zero on success, and non-zero otherwise. The *preg* structure is filled in on success, and one member of the structure is publicized: *re_nsub* contains the number of capturing subpatterns in the regular expression. Various error codes are defined in the header file.

MATCHING A PATTERN

The function **regexec()** is called to match a pre-compiled pattern *preg* against a given *string*, which is terminated by a zero byte, subject to the options in *eflags*. These can be:

REG_NOTBOL

The PCRE_NOTBOL option is set when calling the underlying PCRE matching function.

REG_NOTEOL

The PCRE_NOTEOL option is set when calling the underlying PCRE matching function.

The portion of the string that was matched, and also any captured substrings, are returned via the *pmatch* argument, which points to an array of *nmatch* structures of type *regmatch_t*, containing the members *rm_so* and *rm_eo*. These contain the offset to the first character of each substring and the offset to the first character after the end of each substring, respectively. The 0th element of the vector relates to the entire portion of *string* that was matched; subsequent elements relate to the capturing subpatterns of the regular expression. Unused entries in the array have both structure members set to -1.

A successful match yields a zero return; various error codes are defined in the header file, of which REG_NOMATCH is the "expected" failure code.

ERROR MESSAGES

The **regerror()** function maps a non-zero errorcode from either **regcomp** or **regexec** to a printable message. If *preg* is not NULL, the error should have arisen from the use of that structure. A message terminated by a binary zero is placed in *errbuf*. The length of the message, including the zero, is limited to *errbuf_size*. The yield of the function is the size of buffer needed to hold the whole message.

STORAGE

Compiling a regular expression causes memory to be allocated and associated with the *preg* structure. The function **regfree()** frees all such memory, after which *preg* may no longer be used as a compiled expression.

AUTHOR

Philip Hazel <ph10@cam.ac.uk>
University Computing Service,
New Museums Site,
Cambridge CB2 3QG, England.
Phone: +44 1223 334714