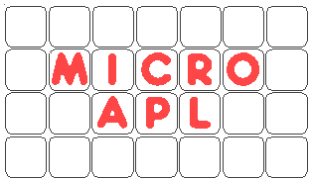
Regular Expressions in APLX



MicroAPL welcomes your comments and suggestions.
Please visit our website: http://www.microapl.co.uk/apl

APLX Version 4.0.0 December 2007

Regular Expressions in APLX

APLX includes facilities for string search using regular expressions, based on version 7.1 of Perl Compatible Regular Expressions (PCRE), an open-source product written by Philip Hazel of the University of Cambridge. Regular expressions can be used in the Find/Change dialog in an editor or session window, in the Search Workspace dialog, and in your APL code using the `⎕SS 'String' 'Search/Replace'` system function.

For an introduction to Regular Expressions, please consult one of the many books or web-sites on the subject, for example www.regular-expressions.info

The remainder of this section, which is an excerpt from the full PCRE documentation (Copyright © Philip Hazel), describes the syntax of regular-expression searches in PCRE, and thus in APLX. Some features which are not relevant to the use of PCRE in APLX have been edited out for clarity.

PCRE 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 books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions",

published by O'Reilly, covers regular expressions in great detail. This description of PCRE's regular expressions is intended as reference material.

CHARACTERS AND METACHARACTERS

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified, letters are matched independently of case.

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

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

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

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

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

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

BACKSLASH

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

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

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

```
Pattern            PCRE matches   Perl matches

\Qabc$xyz\E        abc$xyz        abc followed by the contents of $xyz
\Qabc\$xyz\E       abc\$xyz       abc\$xyz
\Qabc\E\$\Qxyz\E   abc$xyz        abc$xyz
```

The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is 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)
\ddd      character with octal code ddd, or backreference
\xhh      character with hex code hh
\x{hhh..} character with hex code hhh..
```

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

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the value of the character code must be less than 256 in non-UTF-8 mode, and less than 2**31 in UTF-8 mode (that is, the maximum hexadecimal value is 7FFFFFFF). If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of

escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

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

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

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

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. In non-UTF-8 mode, the value of a character specified in octal must be less than \400. In UTF-8 mode, values up to \777 are permitted. For example:

```
\040   is another way of writing a space
\40    is the same, provided there are fewer than 40
        previous capturing subpatterns
\7     is always a back reference
\11    might be a back reference, or another way of writing a tab
\011   is always a tab
\0113  is a tab followed by the character "3"
\113   might be a back reference, otherwise the character with
       octal code 113
\377   might be a back reference, otherwise the byte consisting
       entirely of 1 bits
\81    is either a back reference, or a binary zero followed by
       the two characters "8" and "1"
```

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

All the sequences that define a single character value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

Absolute and relative back references

The sequence \g followed by a positive or negative number, optionally enclosed in braces, is an absolute or relative back reference. Back references are discussed later, following the discussion of parenthesized subpatterns.

Generic character types

Another use of backslash is for specifying generic character types. The following are always recognized:

```
\d     any decimal digit
\D     any character that is not a decimal digit
\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.

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

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32). (If "use locale;" is included in a Perl script, \s may match the VT character. In PCRE, it never does.)

A "word" character is an underscore or any character that is a letter or digit.

Newline sequences

Outside a character class, the escape sequence \R matches any newline sequence. This is an extension to Perl. \R is equivalent to the following:

```
(?>\r\n|\n|\x0b|\f|\r)
```

This is an example of an "atomic group", details of which are given below. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D). The two-character sequence is treated as a single unit that cannot be split.

Inside a character class, \R matches the letter "R".

Simple assertions

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

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

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 in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. The difference between \Z and \z is that \Z matches before a newline at the end of the string as well as at the very end, whereas \z matches only at the end.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see below).

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

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

The meanings of the circumflex and dollar characters are changed if the multi-line search option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the subject string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when

multi-line search is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

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

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

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) a character that signifies the end of a line.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs).

The behaviour of dot with regard to newlines can be changed. If the PCRE_DOTALL option is set, a dot matches any one character, without exception. If the two-character sequence CRLF is present in the subject string, it takes two dots to match it.

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

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence \C matches any one byte. Unlike a dot, it always matches any line-ending characters.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. 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. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

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

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.

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

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

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

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example [\000-\037].

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 [\xc8-\xcb] matches accented E characters in both cases.

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

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes. This uses names enclosed by [: and :] within the enclosing square brackets. PCRE also supports this notation. For example,

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

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

```
alnum    letters and digits
alpha    letters
ascii    character codes 0 - 127
blank    space or tab only
cntrl    control characters
digit    decimal digits (same as \d)
graph    printing characters, excluding space
lower    lower case letters
print    printing characters, including space
punct    printing characters, excluding letters and digits
space    white space (not quite the same as \s)
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|Sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED options 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.

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

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

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

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

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

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

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

SUBPATTERNS

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

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

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

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

2. It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the subject string that matched the

subpattern is available as \1, \2 etc. Opening parentheses are counted from left to right (starting from 1) to obtain numbers for the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

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

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

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

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

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

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

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

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

NAMED SUBPATTERNS

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

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

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present.

REPETITION

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

```
a literal data character
the dot metacharacter
the \C escape sequence
the \R escape sequence
an escape such as \d that matches a single character
a character class
a back reference (see next section)
a parenthesized subpattern (unless it is an assertion)
```

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

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

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

```
(a?)*
```

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

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs.

These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

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

to the string

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

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

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

```
123456bar
```

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

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

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

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

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

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

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as

in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

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

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

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

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

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

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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

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

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

BACK REFERENCES

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

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

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

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence, which is a feature introduced in Perl 5.10. This escape must be followed by a positive or a negative number, optionally enclosed in braces. These examples are all identical:

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

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

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

The sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

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

```
(sens|respons)e and \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.

Back references to named subpatterns use the Perl syntax \k<name> or \k'name' or the Python syntax (?P=name). We could rewrite the above example in either of the following ways:

```
(?<p1>(?i)rah)\s+\k<p1>
(?P<p1>(?i)rah)\s+(?P=p1)
```

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

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

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

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, 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 (see "Comments" below) 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, \G, \Z, \z, ^ and $ are described above.

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

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

Lookahead assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?!) because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several top-level alternatives, they do not all have to have the same fixed length. Thus

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

is permitted, but

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

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), 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 length and then try to match. If there are insufficient characters before the current position, the assertion fails.

PCRE does not allow the \C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of bytes, are also not permitted.

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

```
^.*abcd$
```

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

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

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

Using multiple assertions

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

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

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

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

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

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

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a 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 four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched.

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.

Checking for a used subpattern by name

Perl uses the syntax (?(<name>)...) or (?('name')...) to test for a used subpattern by name. For compatibility with earlier versions of PCRE, which had this facility before Perl, the syntax (?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. PCRE looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, PCRE looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

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

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

Checking for pattern recursion

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

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

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At "top level", all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]?\d) )
\b (?&byte) (\.(?&byte)){3} \b
```

The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions

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

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

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

COMMENTS

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

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

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

A special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?R) or (?0) is a recursive call of the entire regular expression.

In PCRE (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This PCRE pattern solves the nested parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

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

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

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

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

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. The Perl syntax for this is (?&name); PCRE's earlier syntax (?P>name) is also supported. We could rewrite the above example as follows:

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

If there is more than one subpattern with the same name, the earliest one is used. This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

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

At the end of a match, the values 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. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

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

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

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

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. An earlier example pointed out that the pattern

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

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

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

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a "subroutine" call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:

```
(abc)(?i:(?1))
```

It matches "abcabc". It does not match "abcABC" because the change of processing option does not affect the called subpattern.

AUTHOR

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

REVISION

Last updated: 06 March 2007

Copyright and Licence Details

The following legal notice applies to the PCRE code used in APLX: