

Inexpensive protection of your source code

Snob

Simple Name Obfuscator

Regular Expressions Reference

Version 1.0

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright the University of Cambridge, England. See <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>



MacroExpressions

<http://www.macroexpressions.com>

Table of Contents

0. SNOB REGULAR EXPRESSIONS.....	1
1. PCRE REGULAR EXPRESSION DETAILS.....	2
2. BACKSLASH.....	3
3. CIRCUMFLEX AND DOLLAR.....	6
4. FULL STOP (PERIOD, DOT).....	6
5. MATCHING A SINGLE BYTE	6
6. SQUARE BRACKETS	7
7. VERTICAL BAR.....	8
8. MATCH OPTIONS SETTING.....	8
9. SUBPATTERNS	9
10. NAMED SUBPATTERNS.....	10
11. REPETITION.....	10
12. ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS.....	12
13. BACK REFERENCES	13
14. ASSERTIONS.....	14
15. CONDITIONAL SUBPATTERNS.....	17
16. COMMENTS.....	18
17. RECURSIVE PATTERNS	18
18. SUBPATTERNS AS SUBROUTINES	19

0. Snob Regular Expressions

Snob uses regular expressions in its configuration files specific to programming languages.

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright the University of Cambridge, England. See <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>

With the understanding that “plagiarism is the sincerest form of flattery,” this reference is an adaptation of the PCRE regular expressions reference to cover regular expressions to the extent they are used in Snob configuration files. When reasonable, the text of the original reference has been preserved verbatim; however, references to programmer’s interfaces have been

removed or reformulated in terms of regular expressions themselves since the interfaces are not visible to a Snob user.

1. PCRE Regular Expression Details

The syntax and semantics of the regular expressions supported by Snob configuration files 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, covers them in great detail. The description here is intended as reference documentation.

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 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 meta-characters 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 meta-characters.

2. 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 meta-character, 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
<code>\Qabc\$xyz\E</code>	<code>abc\$xyz</code>	<code>abc</code> followed by the contents of <code>\$xyz</code>
<code>\Qabc\ \$xyz\E</code>	<code>abc\ \$xyz</code>	<code>abc\ \$xyz</code>
<code>\Qabc\E\ \$\Qxyz\E</code>	<code>abc\$xyz</code>	<code>abc\$xyz</code>

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

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:

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	formfeed (hex 0C)
<code>\n</code>	newline (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or backreference
<code>\xhh</code>	character with hex code hh

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). 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 byte whose value is zero.

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 (code value 7). 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    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 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.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32).

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 correspond to default C locale, as one would expect dealing with programming languages.

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

<code>\b</code>	matches at a word boundary
<code>\B</code>	matches when not at a word boundary
<code>\A</code>	matches at start of subject
<code>\Z</code>	matches at end of subject or before newline at end
<code>\z</code>	matches at end of subject
<code>\G</code>	matches at first matching position in 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 below) in that they only ever match at the very start and end of the subject file's text, whatever options are set. Thus, they are independent of multiline mode.

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.

The `\G` assertion is true only when the current matching position is at the start point of the match, which is the beginning of the file's text or the end of the previous match accepted by Snob. It differs from `\A` after the first match in the file's text. By matching patterns multiple times, you can mimic Perl's `/g` option, and it is in this kind of implementation where `\G` can be useful.

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

3. Circumflex and Dollar

Outside a character class, the circumflex character (^) is an assertion which is true only if the current matching point is at the start of a line. By default, the start of a line is the start of the subject text. This can be overridden by (?m) option which makes what follows any newline character also a start of a line. (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 meanings of the circumflex and dollar characters are changed if the (?m) option is in effect. When this is the case, they match immediately after and immediately before an internal newline 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.

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 (?m) is in effect or not.

4. 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 (?s) option is in effect, 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.

5. Matching A Single Byte

Outside a character class, the escape sequence \C matches any one byte. Unlike a dot, it always matches a newline. It is best avoided.

PCRE (and thus Snob) does not allow \C to appear in lookbehind assertions (see below).

6. 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. 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 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 careful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the dot matching or multiline options are (see below). 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 the ASCII sequence of character values (corresponding to default C locale). 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.

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

7. 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.

8. Match Options Setting

The default settings for match search can be changed from within the pattern by a sequence of Perl option letters enclosed between “(?” and “)”. The option letters are

```
i  for caseless (case-insensitive) matching
m  for multiline matching ('$' asserts any \n, not only the final)
s  for dot-all (. matches \n)
x  for extended syntax
```

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 caseless and multiline while unsetting dot-all and extended syntax, 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.

An option change within a subpattern affects only that part of the current pattern that follows it, so

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

matches `abc` and `aBc` and no other strings. 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.

Snob uses the underlying regular expressions matching engine provided by PCRE. In particular, Snob would recognize the PCRE-specific options equivalent PCRE_UNGREEDY and PCRE_EXTRA, which can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. It is recommended that U and X options not be used.

9. 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 available for use in other subpatterns. 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 fulfill 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, and the maximum depth of nesting 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".

10. 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 the difficulty, PCRE (and therefore Snob) supports the naming of subpatterns, something that Perl does not provide. The Python syntax (`?P<name>...`) is used. Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names

11. Repetition

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

- a literal data character
- the `.` metacharacter
- the `\C` escape sequence
- escapes such as `\d` that match single characters
- 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 (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 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.

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".

12. Atomic Grouping and Possessive Quantifiers

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. "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 would give 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++bar
```

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 or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun's Java package.

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

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

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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the 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 of 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.

13. 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. 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 (see ["Subpatterns as subroutines"](#) below for a way of doing that). So the pattern

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

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

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

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

Back references to named subpatterns use the Python syntax (`?P=name`). We could rewrite the above example as follows:

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

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 extended syntax option is in effect, 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.

14. 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. 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.

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 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. 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 width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

PCRE does not allow the \C escape to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind.

Atomic groups 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)
```

or, equivalently,

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

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.

15. 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 three 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 extended syntax option in effect) 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 the string (R), it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or (R), 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.

16. 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 extended syntax option is in effect, an unescaped `#` character outside a character class introduces a comment that continues up to the next newline character in the pattern.

17. 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 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, the interpolation of Perl code cannot be supported outside Perl realm. Instead, PCRE supports some special syntax for recursion of the entire pattern, and also for individual subpattern recursion.

The 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)` is a recursive call of the entire regular expression.

For example, this PCRE pattern solves the nested parentheses problem (assume the extended syntax option is in effect 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. For this, PCRE uses `(?P>name)`,

which is an extension to the Python syntax that PCRE uses for named parentheses (Perl does not provide named parentheses). We could rewrite the above example as follows:

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

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

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ( )
```

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.

```
(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.

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.

18. 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. An earlier example pointed out that the pattern

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

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

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

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

Adapted from: PCRE documentation:

[pcrepattern.html](#).

Last updated: 03 February 2003

Copyright © 1997-2003 University of Cambridge.

Adaptation Copyright © 2004 MacroExpressions