NAME

re - Perl-like regular expressions for Erlang.

DESCRIPTION

This module contains regular expression matching functions for strings and binaries.

The regular expression syntax and semantics resemble that of Perl.

The matching algorithms of the library are based on the PCRE library, but not all of the PCRE library is interfaced and some parts of the library go beyond what PCRE offers. Currently PCRE version 8.40 (release date 2017-01-11) is used. The sections of the PCRE documentation that are relevant to this module are included here.

Note:

The Erlang literal syntax for strings uses the "\" (backslash) character as an escape code. You need to escape backslashes in literal strings, both in your code and in the shell, with an extra backslash, that is, "\\".

DATA TYPES

mp() = {re_pattern, term(), term(), term(), term()}

Opaque data type containing a compiled regular expression. *mp()* is guaranteed to be a tuple() having the atom *re_pattern* as its first element, to allow for matching in guards. The arity of the tuple or the content of the other fields can change in future Erlang/OTP releases.

nl_spec() = cr | crlf | lf | anycrlf | any

compile_option() =
unicode | anchored | caseless | dollar_endonly | dotall |
extended | firstline | multiline | no_auto_capture |
dupnames | ungreedy |
{newline, nl_spec()} |
bsr_anycrlf | bsr_unicode | no_start_optimize | ucp |
never_utf

EXPORTS

version() -> binary()

The return of this function is a string with the PCRE version of the system that was used in the Erlang/OTP compilation.

compile(Regexp) -> {ok, MP} | {error, ErrSpec}

Types:

```
Regexp = iodata()
MP = mp()
ErrSpec =
   {ErrString :: string(), Position :: integer() >= 0}
```

The same as *compile(Regexp,[])*

compile(Regexp, Options) -> {ok, MP} | {error, ErrSpec}

Types:

```
Regexp = iodata() | unicode:charlist()
Options = [Option]
Option = compile_option()
MP = mp()
ErrSpec =
   {ErrString :: string(), Position :: integer() >= 0}
```

Compiles a regular expression, with the syntax described below, into an internal format to be used later as a parameter to *run/2* and *run/3*.

Compiling the regular expression before matching is useful if the same expression is to be used in matching against multiple subjects during the lifetime of the program. Compiling once and executing many times is far more efficient than compiling each time one wants to match.

When option *unicode* is specified, the regular expression is to be specified as a valid Unicode *charlist()*, otherwise as any valid *iodata()*.

Options:

unicode:

The regular expression is specified as a Unicode *charlist()* and the resulting regular expression code is to be run against a valid Unicode *charlist()* subject. Also consider option *ucp* when using Unicode characters.

anchored:
The pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string that is searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself.

caseless:
Letters in the pattern match both uppercase and lowercase letters. It is equivalent to Perl option */i* and can be changed within a pattern by a *(?i)* option setting. Uppercase and lowercase letters are defined as in the ISO 8859-1 character set.

dollar_endonly:
A dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). This option is ignored if option *multiline* is specified. There is no equivalent option in Perl, and it cannot be set within a pattern.

dotall:
A dot in the pattern matches all characters, including those indicating newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl option */s* and it can be changed within a pattern by a *(?s)* option setting. A negative class, such as *[^a]*, always matches newline characters, independent of the setting of this option.

extended:
If this option is set, most white space characters in the pattern are totally ignored except when escaped or inside a character class. However, white space is not allowed within sequences such as *(?>* that introduce various parenthesized subpatterns, nor within a numerical quantifier such as *{1,3}*. However, ignorable white space is permitted between an item and a following quantifier and between a quantifier and a following + that indicates possessiveness.

White space did not used to include the VT character (code 11), because Perl did not treat this character as white space. However, Perl changed at release 5.18, so PCRE followed at release 8.34, and VT is now treated as white space.

This also causes characters between an unescaped # outside a character class and the next newline, inclusive, to be ignored. This is equivalent to Perl's */x* option, and it can be changed within a pattern by a *(?x)* option setting.

With this option, comments inside complicated patterns can be included. However, notice that this applies only to data characters. Whitespace characters can never appear within special character sequences in a pattern, for example within sequence *(?(* that introduces a conditional subpattern.

firstline:
An unanchored pattern is required to match before or at the first newline in the subject string, although the matched text can continue over the newline.

multiline:
By default, PCRE treats the subject string as consisting of a single line of characters (even if it contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter (*$*) matches only at the end of the string, or before a terminating newline (unless option *dollar_endonly* is specified). This is the same as in Perl.

When this option is specified, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl option */m* and can be changed within a pattern by a *(?m)* option setting. If there are no newlines in a subject string, or no occurrences of ^ or *$* in a pattern, setting *multiline* has no effect.

no_auto_capture:
Disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by *?* behaves as if it is followed by *?:*. Named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent option in Perl.

dupnames:
Names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched. More details of named subpatterns are provided below.

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

{newline, NLSpec}:
Overrides the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

cr:
Newline is indicated by a single character *cr* (ASCII 13).

lf:
Newline is indicated by a single character LF (ASCII 10), the default.

crlf:
Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence.

anycrlf:
Any of the three preceding sequences is to be recognized.

any:
Any of the newline sequences above, and the Unicode sequences VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

bsr_anycrlf:
Specifies specifically that \R is to match only the CR, LF, or CRLF sequences, not the Unicode-specific newline characters.

bsr_unicode:
Specifies specifically that \R is to match all the Unicode newline characters (including CRLF, and so on, the default).

no_start_optimize:
Disables optimization that can malfunction if "Special start-of-pattern items" are present in the regular expression. A typical example would be when matching "DEFABC" against "(*COMMIT)ABC", where the start optimization of PCRE would skip the subject up to "A" and never realize that the (*COMMIT) instruction is to have made the matching fail. This option is only relevant if you use "start-of-pattern items", as discussed in section PCRE Regular Expression Details.

ucp:
Specifies that Unicode character properties are to be used when resolving \B, \b, \D, \d, \S, \s, \W and \w. Without this flag, only ISO Latin-1 properties are used. Using Unicode properties hurts performance, but is semantically correct when working with Unicode characters beyond the ISO Latin-1 range.

never_utf:
Specifies that the (*UTF) and/or (*UTF8) "start-of-pattern items" are forbidden. This flag cannot be combined with option *unicode*. Useful if ISO Latin-1 patterns from an external source are to be compiled.

inspect(MP, Item) -> {namelist, [binary()]}

Types:

MP = mp()
Item = namelist

Takes a compiled regular expression and an item, and returns the relevant data from the regular expression. The only supported item is *namelist*, which returns the tuple *{namelist, [binary()]}*, containing the names of all (unique) named subpatterns in the regular expression. For example:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
                <<69,82,67,80,119,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
                  255,255,...>>}}
2> re:inspect(MP,namelist).
{namelist,[<<"A">>,<<"B">>,<<"C">>]}
3> {ok,MPD} = re:compile("(?<C>A)|(?<B>B)|(?<C>C)",[dupnames]).
{ok,{re_pattern,3,0,0,
                <<69,82,67,80,119,0,0,0,0,0,8,0,1,0,0,0,255,255,255,255,
                  255,255,...>>}}
4> re:inspect(MPD,namelist).
{namelist,[<<"B">>,<<"C">>]}
```

Notice in the second example that the duplicate name only occurs once in the returned list, and that the list is in alphabetical order regardless of where the names are positioned in the regular expression. The order of the names is the same as the order of captured subexpressions if *{capture, all_names}* is specified as an option to *run/3*. You can therefore create a name-to-value mapping from the result of *run/3* like this:

```
1> {ok,MP} = re:compile("(?<A>A)|(?<B>B)|(?<C>C)").
{ok,{re_pattern,3,0,0,
```

```
                <<69,82,67,80,119,0,0,0,0,0,0,0,1,0,0,0,255,255,255,255,
                  255,255,...>>}}
2> {namelist, N} = re:inspect(MP,namelist).
{namelist,[<<"A">>,<<"B">>,<<"C">>]}
3> {match,L} = re:run("AA",MP,[{capture,all_names,binary}]).
{match,[<<"A">>,<<>>,<<>>]}
4> NameMap = lists:zip(N,L).
[{<<"A">>,<<"A">>},{<<"B">>,<<>>},{<<"C">>,<<>>}]
```

replace(Subject, RE, Replacement) -> iodata() | unicode:charlist()

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata()
Replacement = iodata() | unicode:charlist()
```

Same as *replace(Subject, RE, Replacement, [])*.

replace(Subject, RE, Replacement, Options) ->
iodata() | unicode:charlist()

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata() | unicode:charlist()
Replacement = iodata() | unicode:charlist()
Options = [Option]
Option =
   anchored | global | notbol | noteol | notempty |
   notempty_atstart |
   {offset, integer() >= 0} |
   {newline, NLSpec} |
   bsr_anycrlf |
   {match_limit, integer() >= 0} |
   {match_limit_recursion, integer() >= 0} |
   bsr_unicode |
   {return, ReturnType} |
```

CompileOpt
ReturnType = iodata | list | binary
CompileOpt = compile_option()
NLSpec = cr | crlf | lf | anycrlf | any

Replaces the matched part of the *Subject* string with the contents of *Replacement*.

The permissible options are the same as for *run/3*, except that option *capture* is not allowed. Instead a *{return, ReturnType}* is present. The default return type is *iodata*, constructed in a way to minimize copying. The *iodata* result can be used directly in many I/O operations. If a flat *list()* is desired, specify *{return, list}*. If a binary is desired, specify *{return, binary}*.

As in function *run/3*, an *mp()* compiled with option *unicode* requires *Subject* to be a Unicode *charlist()*. If compilation is done implicitly and the *unicode* compilation option is specified to this function, both the regular expression and *Subject* are to specified as valid Unicode *charlist()*s.

The replacement string can contain the special character &, which inserts the whole matching expression in the result, and the special sequence \N (where N is an integer > 0), *g*N, or *g{*N*}*, resulting in the subexpression number N, is inserted in the result. If no subexpression with that number is generated by the regular expression, nothing is inserted.

To insert an & or a \ in the result, precede it with a \. Notice that Erlang already gives a special meaning to \ in literal strings, so a single \ must be written as ″\\″ and therefore a double \ as ″\\\\″.

Example:

re:replace("abcd","c","[&]",[{return,list}]).

gives

"ab[c]d"

while

re:replace("abcd","c","[\\&]",[{return,list}]).

gives

"ab[&]d"

As with *run/3*, compilation errors raise the *badarg* exception. *compile/2* can be used to get more information about the error.

run(Subject, RE) -> {match, Captured} | nomatch

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata()
Captured = [CaptureData]
CaptureData = {integer(), integer()}
```

Same as *run(Subject,RE,[])*.

run(Subject, RE, Options) ->
{match, Captured} | match | nomatch | {error, ErrType}

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata() | unicode:charlist()
Options = [Option]
Option =
   anchored | global | notbol | noteol | notempty |
   notempty_atstart | report_errors |
   {offset, integer() >= 0} |
   {match_limit, integer() >= 0} |
   {match_limit_recursion, integer() >= 0} |
   {newline, NLSpec :: nl_spec()} |
   bsr_anycrlf | bsr_unicode |
   {capture, ValueSpec} |
   {capture, ValueSpec, Type} |
   CompileOpt
Type = index | list | binary
ValueSpec =
   all | all_but_first | all_names | first | none | ValueList
```

```
ValueList = [ValueID]
ValueID = integer() | string() | atom()
CompileOpt = compile_option()
  See compile/2.
Captured = [CaptureData] | [[CaptureData]]
CaptureData =
  {integer(), integer()} | ListConversionData | binary()
ListConversionData =
  string() |
  {error, string(), binary()} |
  {incomplete, string(), binary()}
ErrType =
  match_limit | match_limit_recursion | {compile, CompileErr}
CompileErr =
  {ErrString :: string(), Position :: integer() >= 0}
```

Executes a regular expression matching, and returns *match/{match, Captured}* or *nomatch*. The regular expression can be specified either as *iodata()* in which case it is automatically compiled (as by *compile/2*) and executed, or as a precompiled *mp()* in which case it is executed against the subject directly.

When compilation is involved, exception *badarg* is thrown if a compilation error occurs. Call *compile/2* to get information about the location of the error in the regular expression.

If the regular expression is previously compiled, the option list can only contain the following options:

* *anchored*

* *{capture, ValueSpec}/{capture, ValueSpec, Type}*

* *global*

* *{match_limit, integer() >= 0}*

* *{match_limit_recursion, integer() >= 0}*

* *{newline, NLSpec}*

* *notbol*

* *notempty*
* *notempty_atstart*
* *noteol*
* *{offset, integer() >= 0}*
* *report_errors*

Otherwise all options valid for function *compile/2* are also allowed. Options allowed both for compilation and execution of a match, namely *anchored* and *{newline, NLSpec}*, affect both the compilation and execution if present together with a non-precompiled regular expression.

If the regular expression was previously compiled with option *unicode*, *Subject* is to be provided as a valid Unicode *charlist()*, otherwise any *iodata()* will do. If compilation is involved and option *unicode* is specified, both *Subject* and the regular expression are to be specified as valid Unicode *charlists()*.

{capture, ValueSpec}/*{capture, ValueSpec, Type}* defines what to return from the function upon successful matching. The *capture* tuple can contain both a value specification, telling which of the captured substrings are to be returned, and a type specification, telling how captured substrings are to be returned (as index tuples, lists, or binaries). The options are described in detail below.

If the capture options describe that no substring capturing is to be done (*{capture, none}*), the function returns the single atom *match* upon successful matching, otherwise the tuple *{match, ValueList}*. Disabling capturing can be done either by specifying *none* or an empty list as *ValueSpec*.

Option *report_errors* adds the possibility that an error tuple is returned. The tuple either indicates a matching error (*match_limit* or *match_limit_recursion*), or a compilation error, where the error tuple has the format *{error, {compile, CompileErr}}*. Notice that if option *report_errors* is not specified, the function never returns error tuples, but reports compilation errors as a *badarg* exception and failed matches because of exceeded match limits simply as *nomatch*.

The following options are relevant for execution:

anchored:
Limits *run/3* to matching at the first matching position. If a pattern was compiled with *anchored*, or turned out to be anchored by virtue of its contents, it cannot be made unanchored

at matching time, hence there is no *unanchored* option.

global:
Implements global (repetitive) search (flag *g* in Perl). Each match is returned as a separate *list()* containing the specific match and any matching subexpressions (or as specified by option *capture*. The *Captured* part of the return value is hence a *list()* of *list()*s when this option is specified.

The interaction of option *global* with a regular expression that matches an empty string surprises some users. When option *global* is specified, *run/3* handles empty matches in the same way as Perl: a zero-length match at any point is also retried with options *[anchored, notempty_atstart]*. If that search gives a result of length > 0, the result is included. Example:

```
re:run("cat","(|at)",[global]).
```

The following matchings are performed:

At offset *0*:
The regular expression *(|at)* first match at the initial position of string *cat*, giving the result set *[{0,0},{0,0}]* (the second *{0,0}* is because of the subexpression marked by the parentheses). As the length of the match is 0, we do not advance to the next position yet.

At offset *0* with *[anchored, notempty_atstart]*:
The search is retried with options *[anchored, notempty_atstart]* at the same position, which does not give any interesting result of longer length, so the search position is advanced to the next character (*a*).

At offset *1*:
The search results in *[{1,0},{1,0}]*, so this search is also repeated with the extra options.

At offset *1* with *[anchored, notempty_atstart]*:
Alternative *ab* is found and the result is [{1,2},{1,2}]. The result is added to the list of results and the position in the search string is advanced two steps.

At offset *3*:
The search once again matches the empty string, giving *[{3,0},{3,0}]*.

At offset *1* with *[anchored, notempty_atstart]*:
This gives no result of length > 0 and we are at the last position, so the global search is

complete.

The result of the call is:

```
{match,[[{0,0},{0,0}],[{1,0},{1,0}],[{1,2},{1,2}],[{3,0},{3,0}]]}
```

notempty:
An empty string is not considered to be a valid match if this option is specified. If alternatives in the pattern exist, they are tried. If all the alternatives match the empty string, the entire match fails.

Example:

If the following pattern is applied to a string not beginning with "a" or "b", it would normally match the empty string at the start of the subject:

```
a?b?
```

With option *notempty*, this match is invalid, so *run/3* searches further into the string for occurrences of "a" or "b".

notempty_atstart:
Like *notempty*, except that an empty string match that is not at the start of the subject is permitted. If the pattern is anchored, such a match can occur only if the pattern contains \K.

Perl has no direct equivalent of *notempty* or *notempty_atstart*, but it does make a special case of a pattern match of the empty string within its split() function, and when using modifier */g*. The Perl behavior can be emulated after matching a null string by first trying the match again at the same offset with *notempty_atstart* and *anchored*, and then, if that fails, by advancing the starting offset (see below) and trying an ordinary match again.

notbol:
Specifies that the first character of the subject string is not the beginning of a line, so the circumflex metacharacter is not to match before it. Setting this without *multiline* (at compile time) causes circumflex never to match. This option only affects the behavior of the circumflex metacharacter. It does not affect \A.

noteol:

Specifies that the end of the subject string is not the end of a line, so the dollar metacharacter is not to match it nor (except in multiline mode) a newline immediately before it. Setting this without *multiline* (at compile time) causes dollar never to match. This option affects only the behavior of the dollar metacharacter. It does not affect \Z or \z.

report_errors:
Gives better control of the error handling in *run/3*. When specified, compilation errors (if the regular expression is not already compiled) and runtime errors are explicitly returned as an error tuple.

The following are the possible runtime errors:

match_limit:
The PCRE library sets a limit on how many times the internal match function can be called. Defaults to 10,000,000 in the library compiled for Erlang. If *{error, match_limit}* is returned, the execution of the regular expression has reached this limit. This is normally to be regarded as a *nomatch*, which is the default return value when this occurs, but by specifying *report_errors*, you are informed when the match fails because of too many internal calls.

match_limit_recursion:
This error is very similar to *match_limit*, but occurs when the internal match function of PCRE is "recursively" called more times than the *match_limit_recursion* limit, which defaults to 10,000,000 as well. Notice that as long as the *match_limit* and *match_limit_default* values are kept at the default values, the *match_limit_recursion* error cannot occur, as the *match_limit* error occurs before that (each recursive call is also a call, but not conversely). Both limits can however be changed, either by setting limits directly in the regular expression string (see section PCRE Regular Eexpression Details) or by specifying options to *run/3*.

It is important to understand that what is referred to as "recursion" when limiting matches is not recursion on the C stack of the Erlang machine or on the Erlang process stack. The PCRE version compiled into the Erlang VM uses machine "heap" memory to store values that must be kept over recursion in regular expression matches.

{match_limit, integer() >= 0}:
Limits the execution time of a match in an implementation-specific way. It is described as follows by the PCRE documentation:

The match_limit field provides a means of preventing PCRE from using up a vast amount of resources when running patterns that are not going to match, but which have a very large number of possibilities in their search trees. The classic example is a pattern that uses nested unlimited repeats.

Internally, pcre_exec() uses a function called match(), which it calls repeatedly (sometimes recursively). The limit set by match_limit is imposed on the number of times this function is called during a match, which has the effect of limiting the amount of backtracking that can take place. For patterns that are not anchored, the count restarts from zero for each position in the subject string.

This means that runaway regular expression matches can fail faster if the limit is lowered using this option. The default value 10,000,000 is compiled into the Erlang VM.

Note:

This option does in no way affect the execution of the Erlang VM in terms of "long running BIFs". *run/3* always gives control back to the scheduler of Erlang processes at intervals that ensures the real-time properties of the Erlang system.

{match_limit_recursion, integer() >= 0}:

Limits the execution time and memory consumption of a match in an implementation-specific way, very similar to *match_limit*. It is described as follows by the PCRE documentation:

The match_limit_recursion field is similar to match_limit, but instead of limiting the total number of times that match() is called, it limits the depth of recursion. The recursion depth is a smaller number than the total number of calls, because not all calls to match() are recursive. This limit is of use only if it is set smaller than match_limit.

Limiting the recursion depth limits the amount of machine stack that can be used, or, when PCRE has been compiled to use memory on the heap instead of the stack, the amount of heap memory that can be used.

The Erlang VM uses a PCRE library where heap memory is used when regular expression match recursion occurs. This therefore limits the use of machine heap, not C stack.

Specifying a lower value can result in matches with deep recursion failing, when they should have matched:

```
1> re:run("aaaaaaaaaaaaaz","(a+)*z").
{match,[{0,14},{0,13}]}
2> re:run("aaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5}]).
nomatch
3> re:run("aaaaaaaaaaaaaz","(a+)*z",[{match_limit_recursion,5},report_errors]).
{error,match_limit_recursion}
```

This option and option *match_limit* are only to be used in rare cases. Understanding of the PCRE library internals is recommended before tampering with these limits.

{offset, integer() >= 0}:
Start matching at the offset (position) specified in the subject string. The offset is zero-based, so that the default is *{offset,0}* (all of the subject string).

{newline, NLSpec}:
Overrides the default definition of a newline in the subject string, which is LF (ASCII 10) in Erlang.

cr:
Newline is indicated by a single character CR (ASCII 13).

lf:
Newline is indicated by a single character LF (ASCII 10), the default.

crlf:
Newline is indicated by the two-character CRLF (ASCII 13 followed by ASCII 10) sequence.

anycrlf:
Any of the three preceding sequences is be recognized.

any:
Any of the newline sequences above, and the Unicode sequences VT (vertical tab, U+000B), FF (formfeed, U+000C), NEL (next line, U+0085), LS (line separator, U+2028), and PS (paragraph separator, U+2029).

bsr_anycrlf:
Specifies specifically that \R is to match only the CR LF, or CRLF sequences, not the Unicode-specific newline characters. (Overrides the compilation option.)

bsr_unicode:
Specifies specifically that \R is to match all the Unicode newline characters (including CRLF, and so on, the default). (Overrides the compilation option.)

{capture, ValueSpec}/{capture, ValueSpec, Type}:
Specifies which captured substrings are returned and in what format. By default, *run/3* captures all of the matching part of the substring and all capturing subpatterns (all of the pattern is automatically captured). The default return type is (zero-based) indexes of the captured parts of the string, specified as *{Offset,Length}* pairs (the *index Type* of capturing).

As an example of the default behavior, the following call returns, as first and only captured string, the matching part of the subject ("abcd" in the middle) as an index pair *{3,4}*, where character positions are zero-based, just as in offsets:

```
re:run("ABCabcdABC","abcd",[]).
```

The return value of this call is:

```
{match,[{3,4}]}
```

Another (and quite common) case is where the regular expression matches all of the subject:

```
re:run("ABCabcdABC",".*abcd.*",[]).
```

Here the return value correspondingly points out all of the string, beginning at index 0, and it is 10 characters long:

```
{match,[{0,10}]}
```

If the regular expression contains capturing subpatterns, like in:

```
re:run("ABCabcdABC",".*(abcd).*",[]).
```

all of the matched subject is captured, as well as the captured substrings:

```
{match,[{0,10},{3,4}]}
```

The complete matching pattern always gives the first return value in the list and the remaining subpatterns are added in the order they occurred in the regular expression.

The capture tuple is built up as follows:

ValueSpec:
Specifies which captured (sub)patterns are to be returned. *ValueSpec* can either be an atom describing a predefined set of return values, or a list containing the indexes or the names of specific subpatterns to return.

The following are the predefined sets of subpatterns:

all:
All captured subpatterns including the complete matching string. This is the default.

all_names:
All *named* subpatterns in the regular expression, as if a *list()* of all the names *in alphabetical order* was specified. The list of all names can also be retrieved with *inspect/2*.

first:
Only the first captured subpattern, which is always the complete matching part of the subject. All explicitly captured subpatterns are discarded.

all_but_first:
All but the first matching subpattern, that is, all explicitly captured subpatterns, but not the complete matching part of the subject string. This is useful if the regular expression as a whole matches a large part of the subject, but the part you are interested in is in an explicitly captured subpattern. If the return type is *list* or *binary*, not returning subpatterns you are not interested in is a good way to optimize.

none:
Returns no matching subpatterns, gives the single atom *match* as the return value of the

function when matching successfully instead of the *{match, list()}* return. Specifying an empty list gives the same behavior.

The value list is a list of indexes for the subpatterns to return, where index 0 is for all of the pattern, and 1 is for the first explicit capturing subpattern in the regular expression, and so on. When using named captured subpatterns (see below) in the regular expression, one can use *atom()*s or *string()*s to specify the subpatterns to be returned. For example, consider the regular expression:

```
".*(abcd).*"
```

matched against string "ABCabcdABC", capturing only the "abcd" part (the first explicit subpattern):

```
re:run("ABCabcdABC",".*(abcd).*",[{capture,[1]}]).
```

The call gives the following result, as the first explicitly captured subpattern is "(abcd)", matching "abcd" in the subject, at (zero-based) position 3, of length 4:

```
{match,[{3,4}]}
```

Consider the same regular expression, but with the subpattern explicitly named ’FOO’:

```
".*(?<FOO>abcd).*"
```

With this expression, we could still give the index of the subpattern with the following call:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,[1]}]).
```

giving the same result as before. But, as the subpattern is named, we can also specify its name in the value list:

```
re:run("ABCabcdABC",".*(?<FOO>abcd).*",[{capture,[’FOO’]}]).
```

This would give the same result as the earlier examples, namely:

```
{match,[{3,4}]}
```

The values list can specify indexes or names not present in the regular expression, in which case the return values vary depending on the type. If the type is *index*, the tuple *{-1,0}* is returned for values with no corresponding subpattern in the regular expression, but for the other types (*binary* and *list*), the values are the empty binary or list, respectively.

Type:
Optionally specifies how captured substrings are to be returned. If omitted, the default of *index* is used.

Type can be one of the following:

index:
Returns captured substrings as pairs of byte indexes into the subject string and length of the matching string in the subject (as if the subject string was flattened with *erlang:iolist_to_binary/1* or *unicode:characters_to_binary/2* before matching). Notice that option *unicode* results in *byte-oriented* indexes in a (possibly virtual) *UTF-8 encoded* binary. A byte index tuple *{0,2}* can therefore represent one or two characters when *unicode* is in effect. This can seem counter-intuitive, but has been deemed the most effective and useful way to do it. To return lists instead can result in simpler code if that is desired. This return type is the default.

list:
Returns matching substrings as lists of characters (Erlang *string()*s). It option *unicode* is used in combination with the \C sequence in the regular expression, a captured subpattern can contain bytes that are not valid UTF-8 (\C matches bytes regardless of character encoding). In that case the *list* capturing can result in the same types of tuples that *unicode:characters_to_list/2* can return, namely three-tuples with tag *incomplete* or *error*, the successfully converted characters and the invalid UTF-8 tail of the conversion as a binary. The best strategy is to avoid using the \C sequence when capturing lists.

binary:
Returns matching substrings as binaries. If option *unicode* is used, these binaries are in UTF-8. If the \C sequence is used together with *unicode*, the binaries can be invalid UTF-8.

In general, subpatterns that were not assigned a value in the match are returned as the tuple *{-1,0}* when *type* is *index*. Unassigned subpatterns are returned as the empty binary or list, respectively, for other return types. Consider the following regular expression:

```
".*((?<FOO>abdd)|a(..d)).*"
```

There are three explicitly capturing subpatterns, where the opening parenthesis position determines the order in the result, hence *((?<FOO>abdd)|a(..d))* is subpattern index 1, *(?<FOO>abdd)* is subpattern index 2, and *(..d)* is subpattern index 3. When matched against the following string:

```
"ABCabcdABC"
```

the subpattern at index 2 does not match, as "abdd" is not present in the string, but the complete pattern matches (because of the alternative *a(..d)*). The subpattern at index 2 is therefore unassigned and the default return value is:

```
{match,[{0,10},{3,4},{-1,0},{4,3}]}
```

Setting the capture *Type* to *binary* gives:

```
{match,[<<"ABCabcdABC">>,<<"abcd">>,<<>>,<<"bcd">>]}
```

Here the empty binary (<<>>) represents the unassigned subpattern. In the *binary* case, some information about the matching is therefore lost, as <<>> can also be an empty string captured.

If differentiation between empty matches and non-existing subpatterns is necessary, use the *type index* and do the conversion to the final type in Erlang code.

When option *global* is speciified, the *capture* specification affects each match separately, so that:

```
re:run("cacb","c(a|b)",[global,{capture,[1],list}]).
```

gives

```
{match,[["a"],["b"]]}
```

For a descriptions of options only affecting the compilation step, see *compile/2*.

split(Subject, RE) -> SplitList

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata()
SplitList = [iodata() | unicode:charlist()]
```

Same as *split(Subject, RE, [])*.

split(Subject, RE, Options) -> SplitList

Types:

```
Subject = iodata() | unicode:charlist()
RE = mp() | iodata() | unicode:charlist()
Options = [Option]
Option =
  anchored | notbol | noteol | notempty | notempty_atstart |
  {offset, integer() >= 0} |
  {newline, nl_spec()} |
  {match_limit, integer() >= 0} |
  {match_limit_recursion, integer() >= 0} |
  bsr_anycrlf | bsr_unicode |
  {return, ReturnType} |
  {parts, NumParts} |
  group | trim | CompileOpt
NumParts = integer() >= 0 | infinity
ReturnType = iodata | list | binary
CompileOpt = compile_option()
```

See *compile/2*.

SplitList = [RetData] | [GroupedRetData]
GroupedRetData = [RetData]
RetData = iodata() | unicode:charlist() | binary() | list()

Splits the input into parts by finding tokens according to the regular expression supplied. The splitting is basically done by running a global regular expression match and dividing the initial string wherever a match occurs. The matching part of the string is removed from the output.

As in *run/3*, an *mp()* compiled with option *unicode* requires *Subject* to be a Unicode *charlist()*. If compilation is done implicitly and the *unicode* compilation option is specified to this function, both the regular expression and *Subject* are to be specified as valid Unicode *charlist()*s.

The result is given as a list of "strings", the preferred data type specified in option *return* (default *iodata*).

If subexpressions are specified in the regular expression, the matching subexpressions are returned in the resulting list as well. For example:

re:split("Erlang","[ln]",[{return,list}]).

gives

["Er","a","g"]

while

re:split("Erlang","([ln])",[{return,list}]).

gives

["Er","l","a","n","g"]

The text matching the subexpression (marked by the parentheses in the regular expression) is inserted in the result list where it was found. This means that concatenating the result of a split where the whole regular expression is a single subexpression (as in the last example) always results in the original string.

As there is no matching subexpression for the last part in the example (the "g"), nothing is inserted after that. To make the group of strings and the parts matching the subexpressions more obvious, one can use option *group*, which groups together the part of the subject string with the parts matching the subexpressions when the string was split:

```
re:split("Erlang","([ln])",[{return,list},group]).
```

gives

```
[["Er","l"],["a","n"],["g"]]
```

Here the regular expression first matched the "l", causing "Er" to be the first part in the result. When the regular expression matched, the (only) subexpression was bound to the "l", so the "l" is inserted in the group together with "Er". The next match is of the "n", making "a" the next part to be returned. As the subexpression is bound to substring "n" in this case, the "n" is inserted into this group. The last group consists of the remaining string, as no more matches are found.

By default, all parts of the string, including the empty strings, are returned from the function, for example:

```
re:split("Erlang","[lg]",[{return,list}]).
```

gives

```
["Er","an",[]]
```

as the matching of the "g" in the end of the string leaves an empty rest, which is also returned. This behavior differs from the default behavior of the split function in Perl, where empty strings at the end are by default removed. To get the "trimming" default behavior of Perl, specify *trim* as an option:

```
re:split("Erlang","[lg]",[{return,list},trim]).
```

gives

["Er","an"]

The "trim" option says; "give me as many parts as possible except the empty ones", which sometimes can be useful. You can also specify how many parts you want, by specifying *{parts,*N*}*:

re:split("Erlang","[lg]",[{return,list},{parts,2}]).

gives

["Er","ang"]

Notice that the last part is "ang", not "an", as splitting was specified into two parts, and the splitting stops when enough parts are given, which is why the result differs from that of *trim*.

More than three parts are not possible with this indata, so

re:split("Erlang","[lg]",[{return,list},{parts,4}]).

gives the same result as the default, which is to be viewed as "an infinite number of parts".

Specifying *0* as the number of parts gives the same effect as option *trim*. If subexpressions are captured, empty subexpressions matched at the end are also stripped from the result if *trim* or *{parts,0}* is specified.

The *trim* behavior corresponds exactly to the Perl default. *{parts,N}*, where N is a positive integer, corresponds exactly to the Perl behavior with a positive numerical third parameter. The default behavior of *split/3* corresponds to the Perl behavior when a negative integer is specified as the third parameter for the Perl routine.

Summary of options not previously described for function *run/3*:

{return,ReturnType}:
Specifies how the parts of the original string are presented in the result list. Valid types:

iodata:
The variant of *iodata()* that gives the least copying of data with the current implementation (often a binary, but do not depend on it).

binary:
All parts returned as binaries.

list:
All parts returned as lists of characters ("strings").

group:
Groups together the part of the string with the parts of the string matching the subexpressions of the regular expression.

The return value from the function is in this case a *list()* of *list()*s. Each sublist begins with the string picked out of the subject string, followed by the parts matching each of the subexpressions in order of occurrence in the regular expression.

{parts,N}:
Specifies the number of parts the subject string is to be split into.

The number of parts is to be a positive integer for a specific maximum number of parts, and *infinity* for the maximum number of parts possible (the default). Specifying *{parts,0}* gives as many parts as possible disregarding empty parts at the end, the same as specifying *trim*.

trim:
Specifies that empty parts at the end of the result list are to be disregarded. The same as specifying *{parts,0}*. This corresponds to the default behavior of the *split* built-in function in Perl.

PERL-LIKE REGULAR EXPRESSION SYNTAX

The following sections contain reference material for the regular expressions used by this module. The information is based on the PCRE documentation, with changes where this module behaves differently to the PCRE library.

PCRE REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described in detail in the following sections. Perl’s regular expressions are described in its own documentation, and regular expressions in general are covered in many books, some with copious examples. Jeffrey Friedl’s "Mastering Regular Expressions", published by O’Reilly, covers regular expressions in great detail. This description of the PCRE regular expressions is intended as reference material.

The reference material is divided into the following sections:

* Special Start-of-Pattern Items
* Characters and Metacharacters
* Backslash
* Circumflex and Dollar
* Full Stop (Period, Dot) and \N
* Matching a Single Data Unit
* Square Brackets and Character Classes
* Posix Character Classes
* Vertical Bar
* Internal Option Setting
* Subpatterns
* Duplicate Subpattern Numbers
* Named Subpatterns
* Repetition
* Atomic Grouping and Possessive Quantifiers
* Back References
* Assertions
* Conditional Subpatterns
* Comments
* Recursive Patterns

* Subpatterns as Subroutines

* Oniguruma Subroutine Syntax

* Backtracking Control

SPECIAL START-OF-PATTERN ITEMS

Some options that can be passed to *compile/2* can also be set by special items at the start of a pattern. These are not Perl-compatible, but are provided to make these options accessible to pattern writers who are not able to change the program that processes the pattern. Any number of these items can appear, but they must all be together right at the start of the pattern string, and the letters must be in upper case.

UTF Support

Unicode support is basically UTF-8 based. To use Unicode characters, you either call *compile/2* or *run/3* with option *unicode*, or the pattern must start with one of these special sequences:

(*UTF8)
(*UTF)

Both options give the same effect, the input string is interpreted as UTF-8. Notice that with these instructions, the automatic conversion of lists to UTF-8 is not performed by the *re* functions. Therefore, using these sequences is not recommended. Add option *unicode* when running *compile/2* instead.

Some applications that allow their users to supply patterns can wish to restrict them to non-UTF data for security reasons. If option *never_utf* is set at compile time, (*UTF), and so on, are not allowed, and their appearance causes an error.

Unicode Property Support

The following is another special sequence that can appear at the start of a pattern:

(*UCP)

This has the same effect as setting option *ucp*: it causes sequences such as \d and \w to use Unicode properties to determine character types, instead of recognizing only characters with codes < 256 through a lookup table.

Disabling Startup Optimizations

If a pattern starts with *(*NO_START_OPT)*, it has the same effect as setting option *no_start_optimize* at compile time.

Newline Conventions

PCRE supports five conventions for indicating line breaks in strings: a single CR (carriage return) character, a single LF (line feed) character, the two-character sequence CRLF, any of the three preceding, and any Unicode newline sequence.

A newline convention can also be specified by starting a pattern string with one of the following five sequences:

(*CR):
Carriage return

(*LF):
Line feed

(*CRLF):
>Carriage return followed by line feed

(*ANYCRLF):
Any of the three above

(*ANY):
All Unicode newline sequences

These override the default and the options specified to *compile/2*. For example, the following pattern changes the convention to CR:

(*CR)a.b

This pattern matches *a\nb*, as LF is no longer a newline. If more than one of them is present, the last one is used.

The newline convention affects where the circumflex and dollar assertions are true. It also affects the interpretation of the dot metacharacter when *dotall* is not set, and the behavior of \N. However, it does

not affect what the \R escape sequence matches. By default, this is any Unicode newline sequence, for Perl compatibility. However, this can be changed; see the description of \R in section Newline Sequences. A change of the \R setting can be combined with a change of the newline convention.

Setting Match and Recursion Limits

The caller of *run/3* can set a limit on the number of times the internal match() function is called and on the maximum depth of recursive calls. These facilities are provided to catch runaway matches that are provoked by patterns with huge matching trees (a typical example is a pattern with nested unlimited repeats) and to avoid running out of system stack by too much recursion. When one of these limits is reached, *pcre_exec()* gives an error return. The limits can also be set by items at the start of the pattern of the following forms:

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

Here d is any number of decimal digits. However, the value of the setting must be less than the value set by the caller of *run/3* for it to have any effect. That is, the pattern writer can lower the limit set by the programmer, but not raise it. If there is more than one setting of one of these limits, the lower value is used.

The default value for both the limits is 10,000,000 in the Erlang VM. Notice that the recursion limit does not affect the stack depth of the VM, as PCRE for Erlang is compiled in such a way that the match function never does recursion on the C stack.

Note that *LIMIT_MATCH* and *LIMIT_RECURSION* can only reduce the value of the limits set by the caller, not increase them.

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 following pattern matches a portion of a subject string that is identical to itself:

```
The quick brown fox
```

When caseless matching is specified (option *caseless*), 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.

Two sets of metacharacters exist: 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 many 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 quantifiers

+:
1 or more quantifier, also "possessive quantifier"

{:
Start min/max quantifier

Part of a pattern within square brackets is called a "character class". The following are the only metacharacters in a character class:

\: 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 metacharacter.

BACKSLASH

The backslash character has many uses. First, if it is followed by a character that is not a number or a letter, it takes away any special meaning that a character can 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 if 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, write \\.

In *unicode* mode, only ASCII numbers and letters have any special meaning after a backslash. All other characters (in particular, those whose code points are > 127) are treated as literals.

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

To remove the special meaning from a sequence of characters, put them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, while $ and @ cause variable interpolation in Perl. Notice the following examples:

Pattern	PCRE matches	Perl matches
\Qabc$xyz\E	abc$xyz	abc followed by the contents of $xyz
\Qabc\$xyz\E	abc\$xyz	abc\$xyz
\Qabc\E\$\Qxyz\E	abc$xyz	abc$xyz

The \Q...\E sequence is recognized both inside and outside character classes. An isolated \E that is not preceded by \Q is ignored. If \Q is not followed by \E later in the pattern, the literal interpretation

continues to the end of the pattern (that is, \E is assumed at the end). If the isolated \Q is inside a character class, this causes an error, as the character class is not terminated.

Non-Printing Characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern. When a pattern is prepared by text editing, it is often 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 ASCII character

\e:
Escape (hex 1B)

\f:
Form feed (hex 0C)

\n:
Line feed (hex 0A)

\r:
Carriage return (hex 0D)

\t:
Tab (hex 09)

\0dd:
Character with octal code 0dd

\ddd:
Character with octal code ddd, or back reference

\o{ddd..}:
character with octal code ddd..

\xhh:
Character with hex code hh

\x{hhh..}:
Character with hex code hhh..

Note:
Note that \0dd is always an octal code, and that \8 and \9 are the literal characters "8" and "9".

The precise effect of \cx on ASCII characters is as follows: if x is a lowercase letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \cA to \cZ become hex 01 to hex 1A (A is 41, Z is 5A), but \c{ becomes hex 3B ({ is 7B), and \c; becomes hex 7B (; is 3B). If the data item (byte or 16-bit value) following \c has a value > 127, a compile-time error occurs. This locks out non-ASCII characters in all modes.

The \c facility was designed for use with ASCII characters, but with the extension to Unicode it is even less useful than it once was.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\015 specifies two binary zeros followed by a CR character (code value 13). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

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

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

The handling of a backslash followed by a digit other than 0 is complicated, and Perl has changed in recent releases, causing PCRE also to change. Outside a character class, PCRE reads the digit and any following digits as a decimal number. If the number is < 8, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is provided later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number following \ is > 7 and there have not been that many

capturing subpatterns, PCRE handles \8 and \9 as the literal characters "8" and "9", and otherwise re-reads up to three octal digits following the backslash, and using them to generate a data character. Any subsequent digits stand for themselves. For example:

\040:
Another way of writing an ASCII space

\40:
The same, provided there are < 40 previous capturing subpatterns

\7:
Always a back reference

\11:
Can be a back reference, or another way of writing a tab

\011:
Always a tab

\0113:
A tab followed by character "3"

\113:
Can be a back reference, otherwise the character with octal code 113

\377:
Can be a back reference, otherwise value 255 (decimal)

\81:
Either a back reference, or the two characters "8" and "1"

Notice that octal values >= 100 that are specified using this syntax must not be introduced by a leading zero, as no more than three octal digits are ever read.

By default, after \x that is not followed by {, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }. If a character other than a hexadecimal digit appears between \x{ and }, or if there is no terminating }, an error occurs.

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}.

Constraints on character values

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

8-bit non-UTF mode:
< 0x100

8-bit UTF-8 mode:
< 0x10ffff and a valid codepoint

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

Escape sequences in character classes

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

\N is not allowed in a character class. \B, \R, and \X are not special inside a character class. Like other unrecognized escape sequences, they are treated as the literal characters "B", "R", and "X". Outside a character class, these sequences have different meanings.

Unsupported Escape Sequences

In Perl, the sequences \l, \L, \u, and \U are recognized by its string handler and used to modify the case of following characters. PCRE does not support these escape sequences.

Absolute and Relative Back References

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

Absolute and Relative Subroutine Calls

For compatibility with Oniguruma, the non-Perl syntax \g followed by a name or a number enclosed either in angle brackets or single quotes, is alternative syntax for referencing a subpattern as a

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

Generic Character Types

Another use of backslash is for specifying generic character types:

\d:
Any decimal digit

\D:
Any character that is not a decimal digit

\h:
Any horizontal whitespace character

\H:
Any character that is not a horizontal whitespace character

\s:
Any whitespace character

\S:
Any character that is not a whitespace character

\v:
Any vertical whitespace character

\V:
Any character that is not a vertical whitespace character

\w:
Any "word" character

\W:
Any "non-word" character

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

Each pair of lowercase and uppercase escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair. The sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all fail, as there is no character to match.

For compatibility with Perl, \s did not used to match the VT character (code 11), which made it different from the the POSIX "space" class. However, Perl added VT at release 5.18, and PCRE followed suit at release 8.34. The default \s characters are now HT (9), LF (10), VT (11), FF (12), CR (13), and space (32), which are defined as white space in the "C" locale. This list may vary if locale-specific matching is taking place. For example, in some locales the "non-breaking space" character (\xA0) is recognized as white space, and in others the VT character is not.

A "word" character is an underscore or any character that is a letter or a digit. By default, the definition of letters and digits is controlled by the PCRE low-valued character tables, in Erlang’s case (and without option *unicode*), the ISO Latin-1 character set.

By default, in *unicode* mode, characters with values > 255, that is, all characters outside the ISO Latin-1 character set, never match \d, \s, or \w, and always match \D, \S, and \W. These sequences retain their original meanings from before UTF support was available, mainly for efficiency reasons. However, if option *ucp* is set, the behavior is changed so that Unicode properties are used to determine character types, as follows:

\d:
Any character that \p{Nd} matches (decimal digit)

\s:
Any character that \p{Z} or \h or \v

\w:
Any character that matches \p{L} or \p{N} matches, plus underscore

The uppercase escapes match the inverse sets of characters. Notice that \d matches only decimal digits, while \w matches any Unicode digit, any Unicode letter, and underscore. Notice also that *ucp* affects \b and \B, as they are defined in terms of \w and \W. Matching these sequences is noticeably slower when *ucp* is set.

The sequences \h, \H, \v, and \V are features that were added to Perl in release 5.10. In contrast to the other sequences, which match only ASCII characters by default, these always match certain high-valued code points, regardless if *ucp* is set.

The following are the horizontal space characters:

U+0009:
Horizontal tab (HT)

U+0020:
Space

U+00A0:
Non-break space

U+1680:
Ogham space mark

U+180E:
Mongolian vowel separator

U+2000:
En quad

U+2001:
Em quad

U+2002:
En space

U+2003:
Em space

U+2004:
Three-per-em space

U+2005:
Four-per-em space

U+2006:
Six-per-em space

U+2007:
Figure space

U+2008:
Punctuation space

U+2009:
Thin space

U+200A:
Hair space

U+202F:
Narrow no-break space

U+205F:
Medium mathematical space

U+3000:
Ideographic space

The following are the vertical space characters:

U+000A:
Line feed (LF)

U+000B:
Vertical tab (VT)

U+000C:
Form feed (FF)

U+000D:
Carriage return (CR)

U+0085:
Next line (NEL)

U+2028:
Line separator

U+2029:
Paragraph separator

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

Newline Sequences

Outside a character class, by default, the escape sequence \R matches any Unicode newline sequence. In non-UTF-8 mode, \R is equivalent to the following:

(?>\r\n|\n|\x0b|\f|\r|\x85)

This is an example of an "atomic group", details are provided below.

This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (line feed, U+000A), VT (vertical tab, U+000B), FF (form feed, U+000C), CR (carriage return, U+000D), or NEL (next line, U+0085). The two-character sequence is treated as a single unit that cannot be split.

In Unicode mode, two more characters whose code points are > 255 are added: LS (line separator, U+2028) and PS (paragraph separator, U+2029). Unicode character property support is not needed for these characters to be recognized.

\R can be restricted to match only CR, LF, or CRLF (instead of the complete set of Unicode line endings) by setting option *bsr_anycrlf* either at compile time or when the pattern is matched. (BSR is an acronym for "backslash R".) This can be made the default when PCRE is built; if so, the other behavior can be requested through option *bsr_unicode*. These settings can also be specified by starting a pattern string with one of the following sequences:

(*BSR_ANYCRLF):
CR, LF, or CRLF only

(*BSR_UNICODE):
Any Unicode newline sequence

These override the default and the options specified to the compiling function, but they can themselves be overridden by options specified to a matching function. Notice that these special settings, which are not Perl-compatible, are recognized only at the very start of a pattern, and that they must be in upper case. If more than one of them is present, the last one is used. They can be combined with a change of newline convention; for example, a pattern can start with:

(*ANY)(*BSR_ANYCRLF)

They can also be combined with the (*UTF8), (*UTF), or (*UCP) special sequences. Inside a character class, \R is treated as an unrecognized escape sequence, and so matches the letter "R" by default.

Unicode Character Properties

Three more escape sequences that match characters with specific properties are available. When in 8-bit non-UTF-8 mode, these sequences are limited to testing characters whose code points are < 256, but they do work in this mode. The following are the extra escape sequences:

\p{*xx***}**:
A character with property *xx*

\P{*xx***}**:
A character without property *xx*

\X:
A Unicode extended grapheme cluster

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

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

\p{Greek} \P{Han}

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

* Arabic
* Armenian
* Avestan

* Balinese
* Bamum
* Bassa_Vah
* Batak
* Bengali
* Bopomofo
* Braille
* Buginese
* Buhid
* Canadian_Aboriginal
* Carian
* Caucasian_Albanian
* Chakma
* Cham
* Cherokee
* Common
* Coptic
* Cuneiform
* Cypriot
* Cyrillic

* Deseret
* Devanagari
* Duployan
* Egyptian_Hieroglyphs
* Elbasan
* Ethiopic
* Georgian
* Glagolitic
* Gothic
* Grantha
* Greek
* Gujarati
* Gurmukhi
* Han
* Hangul
* Hanunoo
* Hebrew
* Hiragana
* Imperial_Aramaic
* Inherited

* Inscriptional_Pahlavi
* Inscriptional_Parthian
* Javanese
* Kaithi
* Kannada
* Katakana
* Kayah_Li
* Kharoshthi
* Khmer
* Khojki
* Khudawadi
* Lao
* Latin
* Lepcha
* Limbu
* Linear_A
* Linear_B
* Lisu
* Lycian
* Lydian

* Mahajani
* Malayalam
* Mandaic
* Manichaean
* Meetei_Mayek
* Mende_Kikakui
* Meroitic_Cursive
* Meroitic_Hieroglyphs
* Miao
* Modi
* Mongolian
* Mro
* Myanmar
* Nabataean
* New_Tai_Lue
* Nko
* Ogham
* Ol_Chiki
* Old_Italic
* Old_North_Arabian

* Old_Permic
* Old_Persian
* Oriya
* Old_South_Arabian
* Old_Turkic
* Osmanya
* Pahawh_Hmong
* Palmyrene
* Pau_Cin_Hau
* Phags_Pa
* Phoenician
* Psalter_Pahlavi
* Rejang
* Runic
* Samaritan
* Saurashtra
* Sharada
* Shavian
* Siddham
* Sinhala

* Sora_Sompeng
* Sundanese
* Syloti_Nagri
* Syriac
* Tagalog
* Tagbanwa
* Tai_Le
* Tai_Tham
* Tai_Viet
* Takri
* Tamil
* Telugu
* Thaana
* Thai
* Tibetan
* Tifinagh
* Tirhuta
* Ugaritic
* Vai
* Warang_Citi

* Yi

Each character has exactly one Unicode general category property, specified by a two-letter acronym. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \p{^Lu} is the same as \P{Lu}.

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional. The following two examples have the same effect:

```
\p{L}
\pL
```

The following general category property codes are supported:

C:
Other

Cc:
Control

Cf:
Format

Cn:
Unassigned

Co:
Private use

Cs:
Surrogate

L:
Letter

Ll:
Lowercase letter

Lm:
Modifier letter

Lo:
Other letter

Lt:
Title case letter

Lu:
Uppercase letter

M:
Mark

Mc:
Spacing mark

Me:
Enclosing mark

Mn:
Non-spacing mark

N:
Number

Nd:
Decimal number

Nl:
Letter number

No:
Other number

P:
Punctuation

Pc:

Connector punctuation

Pd:
Dash punctuation

Pe:
Close punctuation

Pf:
Final punctuation

Pi:
Initial punctuation

Po:
Other punctuation

Ps:
Open punctuation

S:
Symbol

Sc:
Currency symbol

Sk:
Modifier symbol

Sm:
Mathematical symbol

So:
Other symbol

Z:
Separator

Zl:
Line separator

Zp:
Paragraph separator

Zs:
Space separator

The special property L& is also supported. It matches a character that has the Lu, Ll, or Lt property, that is, a letter that is not classified as a modifier or "other".

The Cs (Surrogate) property applies only to characters in the range U+D800 to U+DFFF. Such characters are invalid in Unicode strings and so cannot be tested by PCRE. Perl does not support the Cs property.

The long synonyms for property names supported by Perl (such as \p{Letter}) are not supported by PCRE. It is not permitted to prefix any of these properties with "Is".

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

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

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

Extended Grapheme Clusters

The \X escape matches any number of Unicode characters that form an "extended grapheme cluster", and treats the sequence as an atomic group (see below). Up to and including release 8.31, PCRE matched an earlier, simpler definition that was equivalent to *(?>\PM\pM*)*. That is, it matched a character without the "mark" property, followed by zero or more characters with the "mark" property. Characters with the "mark" property are typically non-spacing accents that affect the preceding character.

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

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

* End at the end of the subject string.

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

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

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

* Do not end after prepend characters.

* Otherwise, end the cluster.

PCRE Additional Properties

In addition to the standard Unicode properties described earlier, PCRE supports four more that make it possible to convert traditional escape sequences, such as \w and \s to use Unicode properties. PCRE uses these non-standard, non-Perl properties internally when the *ucp* option is passed. However, they can also be used explicitly. The properties are as follows:

Xan:
Any alphanumeric character. Matches characters that have either the L (letter) or the N (number) property.

Xps:
Any Posix space character. Matches the characters tab, line feed, vertical tab, form feed, carriage return, and any other character that has the Z (separator) property.

Xsp:
Any Perl space character. Matches the same as Xps, except that vertical tab is excluded.

Xwd:
Any Perl "word" character. Matches the same characters as Xan, plus underscore.

Perl and POSIX space are now the same. Perl added VT to its space character set at release 5.18 and PCRE changed at release 8.34.

Xan matches characters that have either the L (letter) or the N (number) property. Xps matches the characters tab, linefeed, vertical tab, form feed, or carriage return, and any other character that has the Z (separator) property. Xsp is the same as Xps; it used to exclude vertical tab, for Perl compatibility, but Perl changed, and so PCRE followed at release 8.34. Xwd matches the same characters as Xan, plus underscore.

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

Resetting the Match Start

The escape sequence \K causes any previously matched characters not to be included in the final matched sequence. For example, the following pattern matches "foobar", but reports that it has matched "bar":

```
foo\Kbar
```

This feature is similar to a lookbehind assertion (described below). However, in this case, the part of the subject before the real match does not have to be of fixed length, as lookbehind assertions do. The use of \K does not interfere with the setting of captured substrings. For example, when the following pattern matches "foobar", the first substring is still set to "foo":

```
(foo)\Kbar
```

Perl documents that the use of \K within assertions is "not well defined". In PCRE, \K is acted upon when it occurs inside positive assertions, but is ignored in negative assertions. Note that when a pattern such as (?=ab\K) matches, the reported start of the match can be greater than the end of the match.

Simple Assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that must

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 following are the backslashed assertions:

\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, and before a newline at the end of the subject.

\z:
Matches only at the end of the subject.

\G:
Matches at the first matching position in the subject.

Inside a character class, \b has a different meaning; it matches the backspace character. If any other of these assertions appears in a character class, by default it matches the corresponding literal character (for example, \B matches the letter B).

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 (that is, one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively. In UTF mode, the meanings of \w and \W can be changed by setting option *ucp*. When this is done, it also affects \b and \B. PCRE and Perl do not have a separate "start of word" or "end of word" metasequence. However, whatever follows \b normally determines which it is. For example, the fragment \ba matches "a" at the start of a word.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by options *notbol* or *noteol*, which affect only the behavior of the circumflex and dollar metacharacters. However, if argument *startoffset* of *run/3* is non-zero, indicating that matching is to start at a point other than the beginning of the subject, \A can never match. The difference between \Z and \z is that \Z matches before a newline at the end of the string and at the very end, while \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by argument *startoffset* of *run/3*. It differs from \A when the value of *startoffset* is non-zero. By calling *run/3* multiple times with appropriate arguments, you can mimic the Perl option */g*, and it is in this kind of implementation where \G can be useful.

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

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

CIRCUMFLEX AND DOLLAR

The circumflex and dollar metacharacters are zero-width assertions. That is, they test for a particular condition to be true without consuming any characters from the subject string.

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. If argument *startoffset* of *run/3* is non-zero, circumflex can never match if option *multiline* is unset. Inside a character class, circumflex has an entirely different meaning (see below).

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

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

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting option *dollar_endonly* at compile time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if option *multiline* is set. When this is the case, a circumflex matches immediately after internal newlines and 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, and at the very end, when *multiline* is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /^abc$/ matches the subject string "def\nabc" (where \n represents a newline) in multiline mode, but not otherwise. So, patterns that are anchored in single-line mode because all branches start with ^ are not anchored in multiline mode, and a match for circumflex is possible when argument *startoffset* of *run/3* is non-zero. Option *dollar_endonly* is ignored if *multiline* is set.

Notice that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes. If all branches of a pattern start with \A, it is always anchored, regardless if *multiline* is set.

FULL STOP (PERIOD, DOT) AND \N

Outside a character class, a dot in the pattern matches any 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, otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are recognized, dot does not match CR, LF, or any of the other line-ending characters.

The behavior of dot regarding newlines can be changed. If option *dotall* is set, a dot matches any 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 is that both involve newlines. Dot has no special meaning in a character class.

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

MATCHING A SINGLE DATA UNIT

Outside a character class, the escape sequence \C matches any data unit, regardless if a UTF mode is set. One data unit is one byte. Unlike a dot, \C always matches line-ending characters. The feature is provided in Perl to match individual bytes in UTF-8 mode, but it is unclear how it can usefully be used. As \C breaks up characters into individual data units, matching one unit with \C in a UTF mode means that the remaining string can start with a malformed UTF character. This has undefined results, as PCRE assumes that it deals with valid UTF strings.

PCRE does not allow \C to appear in lookbehind assertions (described below) in a UTF mode, as this

would make it impossible to calculate the length of the lookbehind.

The \C escape sequence is best avoided. However, one way of using it that avoids the problem of malformed UTF characters is to use a lookahead to check the length of the next character, as in the following pattern, which can be used with a UTF-8 string (ignore whitespace and line breaks):

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

A group that starts with (?| resets the capturing parentheses numbers in each alternative (see section Duplicate Subpattern Numbers). The assertions at the start of each branch check the next UTF-8 character for values whose encoding uses 1, 2, 3, or 4 bytes, respectively. The individual bytes of the character are then captured by the appropriate number of groups.

SQUARE BRACKETS AND CHARACTER CLASSES

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special by default. However, if option *PCRE_JAVASCRIPT_COMPAT* is set, a lone closing square bracket causes a compile-time error. If a closing square bracket is required as a member of the class, it is to be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

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

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

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

When caseless matching is set, any letters in a class represent both their uppercase and lowercase

versions. For example, a caseless *[aeiou]* matches "A" and "a", and a caseless *[^aeiou]* does not match "A", but a caseful version would. In a UTF mode, PCRE always understands the concept of case for characters whose values are < 256, so caseless matching is always possible. For characters with higher values, the concept of case is supported only if PCRE is compiled with Unicode property support. If you want to use caseless matching in a UTF mode for characters >=, ensure that PCRE is compiled with Unicode property support and with UTF support.

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

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

The literal character "]" cannot be 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 "]" is escaped with a backslash, it is interpreted as the end of range, so [W-\]46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

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

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

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\\^_‘wxyzabc], matched caselessly. In a non-UTF mode, if character tables for a French locale are in use, [\xc8-\xcb] matches accented E characters in both cases. In UTF modes, PCRE supports the concept of case for characters with values > 255 only when it is compiled with Unicode property support.

The character escape sequences \d, \D, \h, \H, \p, \P, \s, \S, \v, \V, \w, and \W can appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. In UTF modes, option *ucp* affects the meanings of \d, \s, \w and their uppercase

partners, just as it does when they appear outside a character class, as described in section Generic Character Types earlier. The escape sequence \b has a different meaning inside a character class; it matches the backspace character. The sequences \B, \N, \R, and \X are not special inside a character class. Like any other unrecognized escape sequences, they are treated as the literal characters "B", "N", "R", and "X".

A circumflex can conveniently be used with the uppercase character types to specify a more restricted set of characters than the matching lowercase type. For example, class [^\\W_] matches any letter or digit, but not underscore, while [\w] includes underscore. A positive character class is to be read as "something OR something OR ..." and a negative class as "NOT something AND NOT something AND NOT ...".

Only the following metacharacters are recognized in character classes:

* Backslash

* Hyphen (only where it can be interpreted as specifying a range)

* Circumflex (only at the start)

* Opening square bracket (only when it can be interpreted as introducing a Posix class name, or for a special compatibility feature; see the next two sections)

* 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, the following matches "0", "1", any alphabetic character, or "%":

[01[:alpha:]%]

The following are the supported class names:

alnum:
Letters and digits

alpha:
Letters

blank:
Space or tab only

cntrl:
Control characters

digit:
Decimal digits (same as \d)

graph:
Printing characters, excluding space

lower:
Lowercase letters

print:
Printing characters, including space

punct:
Printing characters, excluding letters, digits, and space

space:
Whitespace (the same as \s from PCRE 8.34)

upper:
Uppercase letters

word:
"Word" characters (same as \w)

xdigit:
Hexadecimal digits

There is another character class, *ascii*, that erroneously matches Latin-1 characters instead of the 0-127 range specified by POSIX. This cannot be fixed without altering the behaviour of other classes, so we recommend matching the range with *[\\0-\x7f]* instead.

The default "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). If locale-specific matching is taking place, the list of space characters may be different; there may be fewer or more of them. "Space" used to be different to \s, which did not include VT, for Perl compatibility. However, Perl changed at release 5.18, and PCRE followed at release 8.34. "Space" and \s now match the same set of characters.

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example, the following matches "1", "2", or any non-digit:

[12[:^digit:]]

PCRE (and Perl) also recognize the Posix syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

By default, characters with values > 255 do not match any of the Posix character classes. However, if option *PCRE_UCP* is passed to *pcre_compile()*, some of the classes are changed so that Unicode character properties are used. This is achieved by replacing certain Posix classes by other sequences, as follows:

[:alnum:]:
Becomes *p{Xan}*

[:alpha:]:
Becomes *p{L}*

[:blank:]:
Becomes *h*

[:digit:]:
Becomes *p{Nd}*

[:lower:]:
Becomes *p{Ll}*

[:space:]:
Becomes *p{Xps}*

[:upper:]:

Becomes *\p{Lu}*

[:word:]:
Becomes *\p{Xwd}*

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

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

U+061C:
Arabic Letter Mark

U+180E:
Mongolian Vowel Separator

U+2066 - U+2069:
Various "isolate"s

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

[:punct:]:
This matches all characters that have the Unicode P (punctuation) property, plus those characters whose code points are less than 128 that have the S (Symbol) property.

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

Compatibility Feature for Word Boundaries

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

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

[[:>:]]:

is converted to \b(?<=\w)

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

VERTICAL BAR

Vertical bar characters are used to separate alternative patterns. For example, the following pattern matches either "gilbert" or "sullivan":

gilbert|sullivan

Any number of alternatives can 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 that succeeds is used. If the alternatives are within a subpattern (defined in section Subpatterns), "succeeds" means matching the remaining main pattern and the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the Perl-compatible options *caseless*, *multiline*, *dotall*, and *extended* can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are as follows:

i: For *caseless*

m:
For *multiline*

s: For *dotall*

x:
For *extended*

For example, *(?im)* sets caseless, multiline matching. These options can also be unset by preceding the letter with a hyphen. A combined setting and unsetting such as *(?im-sx)*, which sets *caseless* and *multiline*, while unsetting *dotall* and *extended*, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The PCRE-specific options *dupnames*, *ungreedy*, and *extra* can be changed in the same way as the Perl-compatible options by using the characters J, U, and X respectively.

When one of these option changes occurs at top-level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows.

An option change within a subpattern (see section Subpatterns) affects only that part of the subpattern that follows it. So, the following matches abc and aBc and no other strings (assuming *caseless* is not used):

(a(?i)b)c

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", although when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings occur at compile time. There would be some weird behavior otherwise.

Note:

Other PCRE-specific options can be set by the application when the compiling or matching functions are called. Sometimes the pattern can contain special leading sequences, such as (*CRLF), to override what the application has set or what has been defaulted. Details are provided in section Newline Sequences earlier.

The (*UTF8) and (*UCP) leading sequences can be used to set UTF and Unicode property modes. They are equivalent to setting options *unicode* and *ucp*, respectively. The (*UTF) sequence is a generic version that can be used with any of the libraries. However, the application can set option *never_utf*, which locks out the use of the (*UTF) sequences.

SUBPATTERNS

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

1.:

It localizes a set of alternatives. For example, the following pattern matches "cataract", "caterpillar", or "cat":

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

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

2.:
It sets up the subpattern as a capturing subpattern. That is, when the complete pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller through the return value of *run/3*.

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 following pattern, the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively:

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

It is not always helpful that plain parentheses fulfill two functions. Often 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 following pattern, the captured substrings are "white queen" and "queen", and are numbered 1 and 2:

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

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 can appear between "?" and ":". Thus, the following two patterns match the same set of strings:

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

As 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 both "SUNDAY" and "Saturday".

DUPLICATE SUBPATTERN NUMBERS

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

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

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

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

A back reference to a numbered subpattern uses the most recent value that is set for that number by any subpattern. The following pattern matches "abcabc" or "defdef":

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

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

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

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

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

NAMED SUBPATTERNS

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

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

Names consist of up to 32 alphanumeric characters and underscores, but must start with a non-digit. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. The *capture* specification to *run/3* can use named values if they are present in the regular expression.

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

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

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

For capturing named subpatterns which names are not unique, the first matching occurrence (counted from left to right in the subject) is returned from *run/3*, if the name is specified in the *values* part of the *capture* statement. The *all_names* capturing value matches all the names in the same way.

Note:

You cannot use different names to distinguish between two subpatterns with the same number, as PCRE uses only the numbers when matching. For this reason, an error is given at compile time if different names are specified to subpatterns with the same number. However, you can specify the same name to subpatterns with the same number, even when *dupnames* is not set.

REPETITION

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

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

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be < 65536, and the first must be less than or equal to the second. For example, the following matches "zz", "zzz", or "zzzz":

z{2,4}

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, the following matches at least three successive vowels, but can match many more:

```
[aeiou]{3,}
```

The following matches exactly eight digits:

```
\d{8}
```

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

In Unicode mode, quantifiers apply to characters rather than to individual data units. Thus, for example, \x{100}{2} matches two characters, each of which is represented by a 2-byte sequence in a UTF-8 string. Similarly, \X{3} matches three Unicode extended grapheme clusters, each of which can be many data units long (and they can be of different lengths).

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present. This can be useful for subpatterns that are referenced as subroutines from elsewhere in the pattern (but see also section Defining Subpatterns for Use by Reference Only). Items other than subpatterns that have a {0} quantifier are omitted from the compiled pattern.

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

***:**
Equivalent to {0,}

+:
Equivalent to {1,}

?: Equivalent to {0,1}

Infinite loops can be constructed 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, as there are cases where this can be useful, such patterns are now accepted. However, if any repetition of the subpattern matches 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 remaining pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */. Within the comment, individual * and / characters can appear. An attempt to match C comments by applying the pattern

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

to the string

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

fails, as 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 following pattern does the right thing with the C comments:

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

The meaning of the various quantifiers is not otherwise changed, only the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. As 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 remaining pattern matches.

If option *ungreedy* 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. That is, it inverts the default behavior.

When a parenthesized subpattern is quantified with a minimum repeat count that is > 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 option *dotall* (equivalent to Perl option */s*) is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows is 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 if it was preceded by \A.

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

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

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

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

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

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

It matches "ab" in the subject "aab". The use of the backtracking control verbs (*PRUNE) and (*SKIP) also disable this optimization.

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

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

has matched "tweedledum tweedledee", the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values can 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 remaining pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it to fail earlier than it otherwise might, when the author of the pattern knows that there is no point in carrying on.

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

123456bar

After matching all six digits and then failing to match "foo", the normal action of the matcher is to try again with only five digits matching item \d+, and then with four, 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 atomic grouping is used 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 the following 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 to make the remaining pattern match, *(?>\d+)* can only match an entire sequence of digits.

Atomic groups in general can contain any 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 extra + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo
```

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

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

Possessive quantifiers are always greedy; the setting of option *ungreedy* is ignored. They are a convenient notation for the simpler forms of an atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, but there can be a performance difference; possessive quantifiers are probably 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 the Sun 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, as 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 long time. The pattern

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

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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in many ways, and all must be tried. (The example uses [!?] rather than a single character at the end, as 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 the following, sequences of non-digits cannot be broken, and failure happens quickly:

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

BACK REFERENCES

Outside a character class, a backslash followed by a digit > 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 < 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. That is, the parentheses that are referenced do need not be to the left of the reference for numbers < 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, as a sequence such as \50 is interpreted as a character defined in octal. For more details of the handling of digits following a backslash, see section Non-Printing Characters earlier. 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 to avoid the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence. This escape must be followed by an unsigned number or a negative number, optionally enclosed in braces. The following examples are identical:

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

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

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

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

A back reference matches whatever matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (section Subpattern as Subroutines describes a way of doing that). So, the following pattern matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility":

(sens|respons)e and \1ibility

If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example, the following matches "rah rah" and "RAH RAH", but not "RAH rah", although the original capturing subpattern is matched caselessly:

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

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

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

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

There can be more than one back reference to the same subpattern. If a subpattern has not been used in a particular match, any back references to it always fails. For example, the following pattern always fails if it starts to match "a" rather than "bc":

(a|(bc))\2

As there can 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 option *extended* is set, this can be whitespace. Otherwise an empty comment (see section Comments) can be used.

Recursive Back References

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the following pattern matches any number of "a"s and also "aba", "ababbaa", and so on:

(a|b\1)+

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

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

ASSERTIONS

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

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

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

Warning:

If a positive assertion containing one or more capturing subpatterns succeeds, but failure to match later in the pattern causes backtracking over this assertion, the captures within the assertion are reset only if no higher numbered captures are already set. This is, unfortunately, a fundamental limitation of the current implementation, and as PCRE1 is now in maintenance-only status, it is unlikely ever to change.

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

* If the quantifier is {0}, the assertion is never obeyed during matching. However, it can contain internal capturing parenthesized groups that are called from elsewhere through the subroutine mechanism.

* If quantifier is {0,n}, where n > 0, it is treated as if it was {0,1}. At runtime, the remaining pattern match is tried with and without the assertion, the order depends on the greediness of the quantifier.

* If the minimum repetition is > 0, the quantifier is ignored. The assertion is obeyed only once when encountered during matching.

Lookahead Assertions

Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example, the following matches a word followed by a semicolon, but does not include the semicolon in the match:

\w+(?=;)

The following matches any occurrence of "foo" that is not followed by "bar":

foo(?!bar)

Notice 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, as 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 (?!), as an empty string always matches. So, an assertion that requires there is not to be an empty string must always fail. The backtracking control verb (*FAIL) or (*F) is a synonym for (?!).

Lookbehind Assertions

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example, the following finds an occurrence of "bar" that is not preceded by "foo":

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

The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are many top-level alternatives, they do not all have to have the same fixed length. Thus, the following is permitted:

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

The following causes an error at compile time:

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

Branches that match different length strings are permitted only at the top-level of a lookbehind assertion. This is an extension compared with Perl, which requires all branches to match the same length of string. An assertion such as the following is not permitted, as its single top-level branch can match two different lengths:

```
(?<=ab(c|de))
```

However, it is acceptable to PCRE if rewritten to use two top-level branches:

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

Sometimes the escape sequence \K (see above) can be used instead of a lookbehind assertion to get round the fixed-length restriction.

The implementation of lookbehind assertions is, for each alternative, to move the current position back temporarily by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

In a UTF mode, PCRE does not allow the \C escape (which matches a single data unit even in a UTF mode) to appear in lookbehind assertions, as it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of data units, are not permitted either.

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

Possessive quantifiers can be used with lookbehind assertions to specify efficient matching of fixed-length strings at the end of subject strings. Consider the following simple pattern when applied to a long string that does not match:

abcd$

As matching proceeds from left to right, PCRE looks for each "a" in the subject and then sees if what follows matches the remaining pattern. If the pattern is specified as

^.*abcd$

the initial .* matches the entire string at first. However, when this fails (as 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

Many assertions (of any sort) can occur in succession. For example, the following matches "foo" preceded by three digits that are not "999":

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

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 does not match "123abcfoo". A pattern to do that is the following:

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

This time the first assertion looks at the preceding six characters, checks 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, the following matches an occurrence of "baz" that is preceded by "bar", which in turn is not preceded by "foo":

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

The following pattern matches "foo" preceded by three digits and any three characters that are not "999":

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

CONDITIONAL SUBPATTERNS

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

```
(?(condition)yes-pattern)
```

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

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

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

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

Checking for a Used Subpattern By Number

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

Consider the following pattern, which contains non-significant whitespace to make it more readable (assume option *extended*) 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, as no-pattern is not present, the subpattern matches nothing. That is, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If this pattern is embedded in a larger one, a relative reference can be used:

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

Checking for a Used Subpattern By Name

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

Rewriting the previous example to use a named subpattern gives:

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

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

Checking for Pattern Recursion

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

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

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

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

Defining Subpatterns for Use By Reference Only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there can be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern. The idea of DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of subroutines is described below.) For example, a pattern to match an IPv4 address, such as "192.168.23.245", can 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 is a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number < 256). When matching takes place, this part of the pattern is skipped, as DEFINE acts like a false condition. The remaining 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 can be a positive or negative lookahead or lookbehind assertion. Consider the following pattern, containing non-significant whitespace, 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. That is, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative, otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

There are two ways to include comments in patterns that are processed by PCRE. In both cases, the start of the comment must not be in a character class, or in the middle of any other sequence of related characters such as (?: or a subpattern name or number. The characters that make up a comment play no part in the pattern matching.

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. If option PCRE_EXTENDED is set, an unescaped # character also introduces a comment, which in this case continues to immediately after the next newline character or character sequence in the pattern. Which characters are interpreted as newlines is controlled by the options passed to a compiling function or by a special sequence at the start of the pattern, as described in section Newline Conventions earlier.

Notice that the end of this type of comment is a literal newline sequence in the pattern; escape sequences that happen to represent a newline do not count. For example, consider the following pattern when *extended* is set, and the default newline convention is in force:

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

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

RECURSIVE PATTERNS

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

For some time, Perl has provided a facility that allows regular expressions to recurse (among other things). It does this by interpolating Perl code in the expression at runtime, 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;
```

Item (?p{...}) interpolates Perl code at runtime, 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 for individual subpattern recursion. After its introduction in PCRE and Python, this kind of recursion was later introduced into Perl at release 5.10.

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

This PCRE pattern solves the nested parentheses problem (assume that option *extended* is set so that whitespace 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. Notice the use of a possessive quantifier to avoid backtracking into sequences of non-parentheses.

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

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

The pattern is here within parentheses so that the recursion refers to them instead of the whole pattern.

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

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

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

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

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

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

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it gives "no match" quickly. However, if a possessive quantifier is not used, the match runs for a long

time, as there are so many different ways the + and * repeats can carve up the subject, and all must be tested before failure can be reported.

At the end of a match, the values of capturing parentheses are those from the outermost level. If the pattern above is matched against

(ab(cd)ef)

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

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

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

Here (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. Item (?R) is the actual recursive call.

Differences in Recursion Processing between PCRE and Perl

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

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

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

At the top level, the first character is matched, but as it is not at the end of the string, the first

alternative fails, the second alternative is taken, and the recursion kicks in. The recursive call to subpattern 1 successfully matches the next character ("b"). (Notice that the beginning and end of line tests are not part of the recursion.)

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

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

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

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

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

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

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

If you want to match typical palindromic phrases, the pattern must ignore all non-word characters, which can be done as follows:

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

If run with option *caseless*, this pattern matches phrases such as "A man, a plan, a canal: Panama!" and it works well in both PCRE and Perl. Notice the use of the possessive quantifier *+ to avoid backtracking into sequences of non-word characters. Without this, PCRE takes much longer (10 times or more) to match typical phrases, and Perl takes so long that you think it has gone into a loop.

Note:

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

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

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

In PCRE, it matches "bab". The first capturing parentheses match "b", then in the second group, when the back reference \1 fails to match "b", the second alternative matches "a", and then recurses. In the recursion, \1 does now match "b" and so the whole match succeeds. In Perl, the pattern fails to match because inside the recursive call \1 cannot access the externally set value.

SUBPATTERNS AS SUBROUTINES

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

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

An earlier example pointed out that the following pattern matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility":

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

If instead the following pattern is used, it matches "sense and responsibility" and the other two strings:

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

Another example is provided in the discussion of DEFINE earlier.

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

Processing options such as case-independence are fixed when a subpattern is defined, so if it is used as a subroutine, such options cannot be changed for different calls. For example, the following pattern matches "abcabc" but not "abcABC", as the change of processing option does not affect the called subpattern:

(abc)(?i:(?-1))

ONIGURUMA SUBROUTINE SYNTAX

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

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

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

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

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

BACKTRACKING CONTROL

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

same remarks apply to the PCRE features described in this section.

The new verbs make use of what was previously invalid syntax: an opening parenthesis followed by an asterisk. They are generally of the form (*VERB) or (*VERB:NAME). Some can take either form, possibly behaving differently depending on whether a name is present. A name is any sequence of characters that does not include a closing parenthesis. The maximum name length is 255 in the 8-bit library and 65535 in the 16-bit and 32-bit libraries. If the name is empty, that is, if the closing parenthesis immediately follows the colon, the effect is as if the colon was not there. Any number of these verbs can occur in a pattern.

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

Optimizations That Affect Backtracking Verbs

PCRE contains some optimizations that are used to speed up matching by running some checks at the start of each match attempt. For example, it can know the minimum length of matching subject, or that a particular character must be present. When one of these optimizations bypasses the running of a match, any included backtracking verbs are not processed. processed. You can suppress the start-of-match optimizations by setting option *no_start_optimize* when calling *compile/2* or *run/3*, or by starting the pattern with (*NO_START_OPT).

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

Verbs That Act Immediately

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

(*ACCEPT)

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

If (*ACCEPT) is inside capturing parentheses, the data so far is captured. For example, the following matches "AB", "AAD", or "ACD". When it matches "AB", "B" is captured by the outer parentheses.

A((?:A|B(*ACCEPT)|C)D)

The following verb causes a matching failure, forcing backtracking to occur. It is equivalent to (?!) but easier to read.

(*FAIL) or (*F)

The Perl documentation states that it is probably useful only when combined with (?{}) or (??{}). Those are Perl features that are not present in PCRE.

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

Recording Which Path Was Taken

The main purpose of this verb is to track how a match was arrived at, although it also has a secondary use in with advancing the match starting point (see (*SKIP) below).

Note:

In Erlang, there is no interface to retrieve a mark with *run/2,3*, so only the secondary purpose is relevant to the Erlang programmer.

The rest of this section is therefore deliberately not adapted for reading by the Erlang programmer, but the examples can help in understanding NAMES as they can be used by (*SKIP).

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

A name is always required with this verb. There can be as many instances of (*MARK) as you like in a pattern, and their names do not have to be unique.

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

```
  re> /X(*MARK:A)Y|X(*MARK:B)Z/K
data> XY
```

```
 0: XY
MK: A
XZ
 0: XZ
MK: B
```

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

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

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

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

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

Verbs That Act after Backtracking

The following verbs do nothing when they are encountered. Matching continues with what follows, but if there is no subsequent match, causing a backtrack to the verb, a failure is forced. That is, backtracking cannot pass to the left of the verb. However, when one of these verbs appears inside an atomic group or an assertion that is true, its effect is confined to that group, as once the group has been matched, there is never any backtracking into it. In this situation, backtracking can "jump back" to the left of the entire atomic group or assertion. (Remember also, as stated above, that this localization also applies in subroutine calls.)

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

The following verb, which must not be followed by a name, causes the whole match to fail outright if there is a later matching failure that causes backtracking to reach it. Even if the pattern is unanchored, no further attempts to find a match by advancing the starting point take place.

(*COMMIT)

If (*COMMIT) is the only backtracking verb that is encountered, once it has been passed, *run/2,3* is committed to find a match at the current starting point, or not at all, for example:

a+(*COMMIT)b

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

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

Notice that (*COMMIT) at the start of a pattern is not the same as an anchor, unless the PCRE start-of-match optimizations are turned off, as shown in the following example:

```
1> re:run("xyzabc","(*COMMIT)abc",[{capture,all,list}]).
{match,["abc"]}
2> re:run("xyzabc","(*COMMIT)abc",[{capture,all,list},no_start_optimize]).
nomatch
```

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

The following verb causes the match to fail at the current starting position in the subject if there is a later matching failure that causes backtracking to reach it:

(*PRUNE) or (*PRUNE:NAME)

If the pattern is unanchored, the normal "bumpalong" advance to the next starting character then occurs. Backtracking can occur as usual to the left of (*PRUNE), before it is reached, or when matching to the right of (*PRUNE), but if there is no match to the right, backtracking cannot cross (*PRUNE). In simple cases, the use of (*PRUNE) is just an alternative to an atomic group or possessive quantifier, but there are some uses of (*PRUNE) that cannot be expressed in any other way. In an anchored pattern, (*PRUNE) has the same effect as (*COMMIT).

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

Note:

The fact that (*PRUNE:NAME) remembers the name is useless to the Erlang programmer, as names cannot be retrieved.

The following verb, when specified without a name, is like (*PRUNE), except that if the pattern is unanchored, the "bumpalong" advance is not to the next character, but to the position in the subject where (*SKIP) was encountered.

(*SKIP)

(*SKIP) signifies that whatever text was matched leading up to it cannot be part of a successful match. Consider:

a+(*SKIP)b

If the subject is "aaaac...", after the first match attempt fails (starting at the first character in the string), the starting point skips on to start the next attempt at "c". Notice that a possessive quantifier does not have the same effect as this example; although it would suppress backtracking during the first match attempt, the second attempt would start at the second character instead of skipping on to "c".

When (*SKIP) has an associated name, its behavior is modified:

(*SKIP:NAME)

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

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

The following verb causes a skip to the next innermost alternative when backtracking reaches it. That is, it cancels any further backtracking within the current alternative.

(*THEN) or (*THEN:NAME)

The verb name comes from the observation that it can be used for a pattern-based if-then-else block:

(COND1 (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ) ...

If the COND1 pattern matches, FOO is tried (and possibly further items after the end of the group if FOO succeeds). On failure, the matcher skips to the second alternative and tries COND2, without backtracking into COND1. If that succeeds and BAR fails, COND3 is tried. If BAZ then fails, there are no more alternatives, so there is a backtrack to whatever came before the entire group. If (*THEN) is not inside an alternation, it acts like (*PRUNE).

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

Note:

The fact that (*THEN:NAME) remembers the name is useless to the Erlang programmer, as names cannot be retrieved.

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

A (B(*THEN)C) | D

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

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

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

Notice that a conditional subpattern is not considered as having two alternatives, as only one is ever used. That is, the | character in a conditional subpattern has a different meaning. Ignoring whitespace, consider:

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

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

The verbs described above provide four different "strengths" of control when subsequent matching fails:

* (*THEN) is the weakest, carrying on the match at the next alternative.

* (*PRUNE) comes next, fails the match at the current starting position, but allows an advance to the next character (for an unanchored pattern).

* (*SKIP) is similar, except that the advance can be more than one character.

* (*COMMIT) is the strongest, causing the entire match to fail.

More than One Backtracking Verb

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

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

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

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

Backtracking Verbs in Repeated Groups

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

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

If the subject is "abac", Perl matches, but PCRE fails because the (*COMMIT) in the second repeat of the group acts.

Backtracking Verbs in Assertions

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

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

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

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

Backtracking Verbs in Subroutines

These behaviors occur regardless if the subpattern is called recursively. The treatment of subroutines in Perl is different in some cases.

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

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

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

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