Extreme Search™ Manual

Version 2024.04

https://lewis-rhodes.com

support@lewis-rhodes.com

6 Expression Syntax

6.1 NPU*search* subset of PCRE documentation

Compiler version: 0.4.3

The text of this documentation is derived from the PCRE documentation at https://www.pcre.org/current/doc/html/pcre2pattern.html. Please see the "PCRE_LICENSE.txt" file for authorship.

NPU*search* supports a subset of PCRE. This document details what from the above link is supported and what is not.

6.1.1 Characters and Metacharacters

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

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. When caseless matching is specified (`(?i)` within the pattern), letters are matched independently of case.

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

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

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

Brace characters { and } are also used to enclose data for constructions such as \o{23} or \x{BA}. In almost all uses of braces, space and/or horizontal tab characters that follow { or precede } are allowed and are ignored. In the case of quantifiers, they may also appear before or after the comma.

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

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

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

6.1.2 Backslash

The backslash character has several uses. Firstly, if it is followed by a character that is not a digit or a letter, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

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

Only ASCII digits and letters have any special meaning after a backslash. All other characters (in particular, those whose code points are greater than 127) are treated as literals.

If you want to treat all characters in a sequence as literals, you can do so by putting them between `\Q` and `\E`. The `\Q...\E` sequence is recognized both inside and outside character classes. An isolated `\E` that is not preceded by `\Q` is ignored. If `\Q` is not followed by `\E` later in the pattern, the literal interpretation continues to the end of the pattern (that is, `\E` is assumed at the end). If the isolated `\Q` is inside a character class, this causes an error, because the character class is not terminated by a closing square bracket.

6.1.2.1 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 in a pattern, but when a pattern is being prepared by text editing, it is often easier to use one of the following escape sequences instead of the binary character it represents. These escapes are as follows:

- `\a` alarm, that is, the BEL character (hex 07)
- `\cx` "control-x", where x is any printable ASCII character
- `\e` escape (hex 1B)
- `\f` form feed (hex 0C)
- `\n` linefeed (hex 0A)
- `\r` carriage return (hex 0D) (but see below)
- `\t` tab (hex 09)
- `\0dd` character with octal code 0dd
- `\ddd` character with octal code ddd
- `\o{ddd..}` character with octal code ddd..
- `\xhh` character with hex code hh
- `\x{hhh..}` character with hex code hhh..

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 can be defined by either of the two syntaxes for `\x` or by an octal sequence. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}` or `\334`. However, using the braced versions does make such sequences easier to read.

The precise effect of `\cx` on ASCII characters is as follows: if "x" is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cA` to `\cZ` become hex 01 to hex 1A ("A" is 41, "Z" is 5A), but `\c{` becomes hex 3B ("{" is 7B), and `\c;` becomes hex 7B (";" is 3B). If the code unit following `\c` has a value less than 32 or greater than 126, a compile-time error occurs.

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.

6.1.2.2 Constraints on character values

Characters that are specified using octal or hexadecimal numbers are limited to be no greater than `\xff` or `\377`.

6.1.2.3 Escape sequences in character classes

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

`\N` is not allowed in a character class.

6.1.2.4 Generic character types

Another use of backslash is for specifying generic character types:

- `\d` any decimal digit
- `\D` any character that is not a decimal digit
- `\h` any horizontal white space character
- `\H` any character that is not a horizontal white space character
- `\N` any character that is not a newline
- `\s` any white space character
- `\S` any character that is not a white space character
- `\v` any vertical white space character
- `\V` any character that is not a vertical white space character
- `\w` any "word" character
- `\W` any "non-word" character

The `\N` escape sequence has the same meaning as the `.` metacharacter when DOTALL (`(?s)` within the pattern) is not set, but setting DOTALL does not change the meaning of `\N`.

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

The `\s` characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32).

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

Characters whose code points are greater than 127 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`. The upper case escapes match the inverse sets of characters. The horizontal space characters are:

- hex 09 Horizontal tab (HT)
- hex 20 Space
- hex A0 Non-break space

The vertical space characters are:

- hex 0A Linefeed (LF)
- hex 0B Vertical tab (VT)
- hex 0C Form feed (FF)
- hex 0D Carriage return (CR)
- hex 85 Next line (NEL)

6.1.2.5 Simple assertions

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

- `\b` matches at a word boundary
- `\B` matches when not at a word boundary

- `\A` matches at the start of the subject
- `\Z` matches at the end of the subject (also matches before a newline at the end of the subject)
- `\z` matches only at the end of the subject

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, an "invalid escape sequence" error is generated. A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively. NPU*search* does 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. The difference between `\Z` and `\z` is that `\Z` matches before a newline at the end of the string as well as at the very end, whereas `\z` matches only at the end.

6.1.3 Circumflex and Dollar

The circumflex and dollar metacharacters are zero-width assertions. That is, they test for a particular condition being true without consuming any characters from the subject string. These two metacharacters are concerned with matching the starts and ends of lines.

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true immediately after internal newlines as well as at the start of the subject string. It does match after a newline that ends the string. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative (or after a newline) in which it appears if the pattern is ever to match that branch.

The dollar character is an assertion that is true before any newlines in the string, as well as at the very end (by default). Note, however, that it does not actually match the newline. Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch (or before a newline) in which it appears. Dollar has no special meaning in a character class.

The meanings of the circumflex and dollar metacharacters are changed if the MULTILINE option is NOT set. When this is the case, a dollar character matches 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, and a circumflex matches only if the current matching point is at the start of the subject string. If all possible alternatives start with a circumflex and MULTILINE is NOT set, 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 MULTILINE option can be turned off using the `(?-m)` flag; see the section on INTERNAL OPTION SETTING.

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

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

6.1.4 Full Stop (Period, Dot) and `\N`

Outside a character class, a dot in the pattern matches any one character in the subject string except (by default) "\n".

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

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

The escape sequence `\N` when not followed by an opening brace behaves like a dot, except that it is not affected by the DOTALL option. In other words, it matches any character except one that signifies the end of a line.

6.1.5 Matching a Single Code Unit

The shorthand `\C` can be used to match a single byte. Since all code units are single bytes, `\C` will match any single character. This is the same as dot when DOTALL is set.

6.1.6 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. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash. This means that an empty class cannot be defined.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

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

Characters in a class may be specified by their code points using `\o` or `\x` in the usual way. When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `[^aeiou]` does not match "A", whereas a caseful version would.

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

The generic character type escape sequences `\d`, `\D`, `\h`, `\H`, `\s`, `\S`, `\v`, `\V`, `\w`, and `\W` may appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. The escape sequence `\b` has a different meaning inside a character class; it matches the backspace character.

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.

If a hyphen appears before or after a POSIX class (see below) or before or after a character type escape such as as `\d` or `\H`, an error is given.

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

Ranges include all code points between the start and end characters, inclusive. They can also be used for code points specified numerically, for example `[\000-\037]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[][\\^_`wxyzabc]`, matched caselessly, and `[\xc8-\xcb]` matches accented E characters in both cases.

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

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

6.1.7 Posix Character Classes

NPU*search* supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing square brackets. For example,

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

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

- `alnum` letters and digits
- `alpha` letters
- `ascii` character codes 0 - 127
- `blank` space or tab only
- `cntrl` control characters
- `digit` decimal digits (same as `\d`)
- `graph` printing characters, excluding space
- `lower` lower case letters
- `print` printing characters, including space
- `punct` printing characters, excluding letters and digits and space
- `space` white space (the same as `\s` from PCRE2 8.34)
- `upper` upper case letters
- `word` "word" characters (same as `\w`)
- `xdigit` hexadecimal digits

The `space` characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). `space` and `\s` match the same set of characters. The name `word` is a Perl extension, and `blank` is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example,

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

matches "1", "2", or any non-digit. NPU*search* also recognizes 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. Characters with values greater than 127 do not match any of the POSIX character classes, unless the class is negated.

6.1.8 Vertical Bar

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

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The ordering of alternatives does not matter.

6.1.9 Internal Option Setting

The settings of the CASELESS, MULTILINE, DOTALL, EXTENDED, EXTENDED_MORE, and NO_AUTO_CAPTURE options can be changed from within the pattern by a sequence of letters enclosed between `(?` and `)`. These options are Perl-compatible, and are described in detail in the pcre2api documentation. The option letters are:

- `i` for CASELESS
- `m` for MULTILINE
- `n` for NO_AUTO_CAPTURE (currently does nothing)
- `s` for DOTALL
- `x` for EXTENDED
- `xx` for EXTENDED_MORE

For example, `(?im)` sets caseless, multiline matching. It is also possible to unset these options by preceding the relevant letters with a hyphen, for example `(?-im)`. The two "extended" options are not independent; unsetting either one cancels the effects of both of them.

A combined setting and unsetting such as `(?im-sx)`, which sets CASELESS and MULTILINE while unsetting DOTALL and EXTENDED, is also permitted. Only one hyphen may appear in the options string. If a letter appears both before and after the hyphen, the option is unset. An empty options setting `(?)` is allowed. Needless to say, it has no effect.

The MULTILINE option is turned on by default.

If the first character following `(?` is a circumflex, it causes all of the above options to be unset. Thus, `(?^)` is equivalent to `(?-imnsx)`. Letters may follow the circumflex to cause some options to be re-instated, but a hyphen may not appear.

When one of these option changes occurs at top level (that is, not inside group parentheses), the change applies to the remainder of the pattern that follows. An option change within a group (see below for a description of groups) affects only that part of the group that follows it, so

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

matches abc and aBc and no other strings (assuming CASELESS is not set earlier in the pattern). 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 group. For example,

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

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise. As a convenient shorthand, if any option settings are required at the start of a non-capturing group (see the next section), the option letters may appear between the `?` and the `:`. Thus the two patterns

```
(?i:saturday|sunday)

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

match exactly the same set of strings.

The characters affected by the `(?i)` setting are `[a-z\xe0-\xfe]` except \xf7 and their corresponding equivalence. They are deemed equivalent to the character which has a code point 32 less. For example, "f" is code point \x66 and it is deemed equivalent to \x46, which is "F".

The character \xdf, which is known as "ß" or "sharp-s" officially has "SS" as it's corresponding caseless character. NPU*search* does not support matching `(?i)ß` with "SS" and treats "ß" as if it had no caseless equivalent.

6.1.10 Groups

Groups are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a group localizes a set of alternatives. For example, the pattern

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

matches "cataract", "caterpillar", or "cat". Without the parentheses, it would match "cataract", "erpillar" or an empty string. Accessing capture groups is not supported. The characters `?:` at the start of a group will be parsed and ignored.

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

```
(?i:saturday|sunday)

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

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

6.1.11 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
- an escape such as `\d` that matches a single character
- a character class
- a parenthesized group (including lookaround assertions)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 1000, and the first must be less than or equal to the second. For example,

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

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

matches at least 3 successive vowels, but may match many more, whereas

```
\d{8}
```

matches exactly 8 digits. If the first number is omitted, the lower limit is taken as zero; in this case the upper limit must be present.

```
X{,4} is interpreted as X{0,4}
```

This is a change in behaviour that happened in Perl 5.34.0 and PCRE2 10.43. NPUsearch followed for all releases in 2024. In earlier versions such a sequence was not interpreted as a quantifier. Other regular expression engines may behave either way.

If the characters that follow an opening brace do not match the syntax of a quantifier, the brace is taken as a literal character. In particular, this means that {,} is a literal string of three characters.

Note that not every opening brace is potentially the start of a quantifier because braces are used in other items such as \o{345} or \x{80}.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present. Items that have a `{0}` quantifier are omitted from the compiled pattern.

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

- `*` is equivalent to `{0,}`

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

NPU*search* supports the EXNET subset of regular expressions, which means any "infinite repetition" (quantifiers `*`, `+`, and `{n,}`) can only be applied to a single character or character class. Anything else will throw an error.

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

6.1.12 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 above.

More complicated assertions are coded as parenthesized groups. NPU*search* only supports one of the are two kinds: those that look ahead of the current position in the subject string are not supported, and those that look behind it are supported, and in each case an assertion may be positive (must match for the assertion to be true) or negative (must not match for the assertion to be true). An assertion group is matched in the normal way, and if it is true, matching continues after it, but with the matching position in the subject string reset to what it was before the assertion was processed.

NPU*search* assertions are all non-atomic; that is if an assertion is true, but there is a subsequent matching failure, there will be backtracking into the assertion.

For a positive assertion, matching continues with the next pattern item after the assertion. For a negative assertion, a matching branch means that the assertion is not true.

Most assertion groups may be repeated; though it makes no sense to assert the same thing several times.

6.1.12.1 Alphabetic assertion names

Traditionally, the symbolic sequences `(?<=` and `(?<!` have been used to specify lookbehind assertions. PCRE2 supports a set of synonyms such as `(*plb`, however NPU*search* does not support those synonyms.

6.1.12.2 Lookahead assertions

NPU*search* does not support lookahead assertions.

6.1.12.3 Lookbehind assertions

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

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

does find an occurrence of "bar" that is not preceded by "foo".

The contents of a lookbehind assertion are not restricted in any way. Any expression that can be used outside a lookbehind can be used inside a lookbehind.

The implementation of lookbehind assertions is to calculate if the lookbehind matches in parallel with the rest of the regex, and then AND the results of the lookbehind and if the engine is currently at the location for the lookbehind in the main regex branch. For example, the expression `[bc]at(?<=ca[rt])s` will attempt to match `[bc]at` and `ca[rt]` in parallel and only attempt to match the last `s` if both of the two expressions returned a match on the same byte.

6.1.12.4 Using multiple assertions

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

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

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

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

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999". Assertions can be nested in any combination. For example,

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

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

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

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

6.1.13 Non-atomic Assertions

All NPU*search* assertions are non-atomic.

6.1.14 PCRE Syntax not supported by NPU*search*

6.1.14.1 Special Start-of-pattern Items

Special start-of-pattern items are not supported and will cause an error.

6.1.14.2 EBCDIC Character Codes

EBCDIC character codes are not supported by NPU search. NPU search only supports Latin1 encoding.

6.1.14.3 Compatibility Feature for Word Boundaries

`[[:<:]]` and `[[:>:]]` are not supported and will cause an error.

6.1.14.4 Duplicate Group Numbers

`(?|...)` syntax is not supported and will cause an error.

6.1.14.5 Named Capture Groups

Named capture groups are not supported and will cause an error. This includes `(?<name>...)`, `(?'name'...)`, and `(?P<name>...)` syntax.

6.1.14.6 Atomic Grouping and Possessive Quantifiers

Atomic grouping will be parsed and a warning will be thrown. Possessive quantifiers will be parsed and a warning will be thrown. Matching will proceede as if the grouping was not atomic or as if the quantifier was not possessive, which may cause false positives.

6.1.14.7 Backreferences

Backreferences are not supported. Any numeric backreference will be parsed as an octal escape and a warning will be thrown. `\g` will cause an error.

6.1.14.8 Script Runs

Script runs are not supported and will cause an error.

6.1.14.9 Conditional Groups

Conditional groups are not supported and will cause an error.

6.1.14.10 Comments

Comments are not supported. `?#` will cause an error. PCRE2_EXTENDED and PCRE2_EXTENDED_MORE options are not supported.

6.1.14.11 Recursive Patterns

Recursive patterns are not supported and will cause an error.

6.1.14.12 Groups as Subroutines

Groups as subroutines are not supported and will cause an error.

6.1.14.13 Oniguruma Subroutine Syntax

Oniguruma subroutine syntax is not supported and will cause an error.

6.1.14.14 Callouts

Callouts are not supported and will cause an error.

6.1.14.15 Backtracking Control

Backtracking control is not supported and will cause an error.

6.2 Complex Input Guide

6.2.1 Overview

In addition to supporting standard regular expression syntax, the NPU*search* compiler can compile combinations of expressions in non-standard ways. The two currently supported additional ways are

1. **At least *n***: Files are only returned if at least *n* of the selected regular expressions match anywhere in the file. If fewer than *n* expressions match in a file, that file is not returned.

2. **Combinations**: Files are only returned if a specific combination of regular expressions match or do not match anywhere in the file.

A user must follow a specific syntax to use this functionality.

In certain cases, a user should not use this functionality, and instead search for all the expressions normally, doing any post-results analysis in software. If all of the following conditions hold, then a user should do the analysis in software. If any of the following conditions do not hold, then a user should let the NPU evaluate "at least *n*" or "combinations".

- The number of files which match at least one expression is of a managable size.
- The maximum number of expressions matched by a single file is less than or equal to the number of match gaps (currently 16).
- When "at least *n*" is desired, *n* must be less than or equal to the number of match gaps (currently 16).

6.2.2 Overall syntax

When inputing expressions into `NPUGlusterClient.scan()` or `check_expressions`, the simple syntax is to input an iterable of expressions. The indicies of expressions which matched a given file are returned.

The complex input syntax is to input an iterable of dictionaries, each of which represents one expression. The dictionaries must be formatted as such for different types:

```
{'type': 'expression',
 'expr': str,
 'id': int, #optional
 'ref_id': int, #optional
 'output_id': int} #optional

 {'type': 'at_least_n',
 'refs': list of ints,
 'n': int,
 'id': int, #optional
 'ref_id': int, #optional
 'output_id': int, #optional
 'maybe_match_id': int} #optional

 {'type': 'combination',
 'formula': str,
 'id': int, #optional
 'ref_id': int, #optional
 'output_id': int, #optional
 'simplify': bool} #optional
```

Example:

```
>>> from npusearch import NPUGlusterClient
>>> client = NPUGlusterClient(gluster_hosts='localhost', gluster_volname='gv0')
>>> exprs = [
...     {'type': 'expression', 'expr': 'Hello', 'ref_id': 101},
...     {'type': 'expression', 'expr': 'World', 'ref_id': 102},
...     {'type': 'expression', 'expr': 'hello', 'ref_id': 103},
...     {'type': 'expression', 'expr': 'world', 'ref_id': 104},
...     {'type': 'at_least_n', 'refs': [101,102,103,104], 'n': 2, 'output_id': 201},
...     {
...         'type': 'combination',
...         'formula': '(101 AND 102) OR (103 AND 104)',
...         'output_id': 301,
...     },
...     {'type': 'expression', 'expr': 'Hello World', 'output_id': 1},
... ]
>>> client.scan(exprs, ['examples/*'])
{'pe_usage': 47,
 'matches': [{'matches': [1, 201, 301],
   'overflow': False,
   'path': 'examples/Hello_World.txt',
   'brick': 'server01:/mnt/npusearch_10/gv0/examples/Hello_World.txt'}],
 'errors': [],
 'exceptions': {}}
```

All types have the fields `'type'`, `'id'`, `'ref_id'`, and `'output_id'`.

- The `'type'` field tells the compiler what type of syntax is inputted.
- The `'id'` field is an alias for `'ref_id'` and `'output_id'`. Instead of passing in the same value to both `'ref_id'` and `'output_id'`, a user can pass that value to the `'id'` field instead and it will propagate to the other two fields.
- The `'ref_id'` field is so that other expressions can reference each other.
- The `'output_id'` field is the number which is returned in the match result as opposed to the index of the dictionary inside the iterable.

Both `'ref_id'` and `'output_id'` are optional, but if a user doesn't include either the expression is ignored. If an expression is not referenced anywhere else, no `'ref_id'` is needed. If a user does not want to include which files an expression matched in the match result, no `'output_id'` is needed. In general, a user should only include the `'output_id'` if they are interested in which files an expression matched, since including it currently uses additional PEs and can clutter up the match result with many additional matched files. All 'ref_id's must be unique and all 'output_id's must be unique among the passed in expressions, however a number can be used as both a 'ref_id' and an 'output_id'.

Section 5.2.6 contains a few recipies for converting common expression input formats into complex input syntax.

6.2.3 `expression`

The 'expression' type compiles the same as a simple regular expression string. It is used as the building block for 'at_least_n' and 'combination' expressions, or if a user wants to have a specific output id for an expression.

6.2.3.1 Example

Note how the last expression has a 'ref_id' but no 'output_id', so there are no match results for that expression. In this example, no other expressions reference that expression, so it is ignored.

```
>>> exprs = [
...     {'type': 'expression', 'expr': r'(?i)NPUsearch', 'output_id': 27},
...     {'type': 'expression', 'expr': r'Hello', 'output_id': 42},
...     {'type': 'expression', 'expr': r'\w+', 'ref_id': 10},
... ]
>>> client.scan(exprs, ['examples/*'], return_type='gluster')
{'pe_usage': 5,
 'matches': [{'matches': [27],
   'overflow': False,
   'path': 'examples/NPUsearch.txt'},
  {'matches': [27], 'overflow': False, 'path': 'examples/LRL.txt'},
  {'matches': [42], 'overflow': False, 'path': 'examples/Hello_World.txt'}],
 'errors': [],
 'exceptions': {}}
```

6.2.4 `at_least_n`

WARNING: Omitting 'maybe_match_id' can lead to false negative results.

The 'at_least_n' syntax includes 'refs', 'n', and 'maybe_match_id'. A match can happen when at least 'n' of the expressions referenced in 'refs' match on a given file. However, there are cases where a match which should be reported will not be reported due to hardware limitations. This can occur when, for all *b*, at least *b* of the expressions referenced match in the last *b* bytes of the file. In these cases a match is returned with the id of 'maybe_match_id'. We strongly recommend always including 'maybe_match_id' as not including it can lead to false negative results.

There are cases where a match is returned with 'maybe_match_id' when a match should not be reported. Same as above, this can occur when, for all *b*, at least *b* of the expressions referenced match in the last *b* bytes of the file. For this reason, we strongly recommend avoiding '$', '\\z', and '\\Z' in expressions which are referenced by an 'at_least_n' expression, since these cause matches to occur at the end of the file.

'refs' is a list of 'ref_id's of other expressions. Currently, all of the expressions referenced must be of type 'expression'.

6.2.4.1 Example

Consider the expressions:

```
exprs = [
 {'type': 'expression', 'expr': 'L(ewis )?R(hodes )?L(abs)?', 'ref_id': 0},
 {'type': 'expression', 'expr': '(?i)neuromorphic\s*computing', 'ref_id': 1},
 {'type': 'expression', 'expr': '(?i)extremesearch', 'ref_id': 2},
 {'type': 'expression', 'expr': '(?i)datalake', 'ref_id': 3},
 {'type': 'at_least_n', 'refs': [0,1,2,3], 'n': 2, 'output_id': 0, 'maybe_match_id': 4}
]
```

Then the following text fragments would match:

- "Lewis Rhodes Labs uses ExtremeSearch to search large datalakes"
- "Neuromorphic computing allows many datalakes to be searched."
- "If you have a large datalake, you need ExtremeSearch."

The following text fragments would not match:

- "Only neuromorphic computing appears in this fragment."
- "Even though Lewis Rhodes Labs (LRL) is repeated in two forms, it only counts as one expression."
- "None of the expressions appear here."

The following text fragment would return a maybe_match:

- "Since the only expression matches at the end, a maybe_match would be returned by ExtremeSearch"

6.2.4.2 Formatting Recipe

```python
def iterable_to_at_least_n(iterable, n, output_id=0, maybe_match_id=1):
    """
    When you have an iterable of expressions and a "n" value for "at_least_n",
    this function converts it to a valid complex input format.

    Parameters
    ----------
    iterable : iterable of strings
        Each string represents one expression.

    n : int
        The "n" in "at_least_n".
        The minimum number of expressions that a file needs to match.

    output_id : int, optional
        output_id for final "at_least_n" expression. The default is 0.

    maybe_match_id : int, optional
        maybe_match_id for final "at_least_n" expression. The default is 1.

    Returns
    -------
    exprs : dict
        Use as input into client.scan().


    Example:
    >>> iterable = ['expr0', 'expr1', 'expr2', 'expr3']
    >>> n = 2
    >>> iterable_to_at_least_n(iterable, n)
    [{'type': 'expression', 'expr': 'expr0', 'ref_id': 2},
     {'type': 'expression', 'expr': 'expr1', 'ref_id': 3},
     {'type': 'expression', 'expr': 'expr2', 'ref_id': 4},
     {'type': 'expression', 'expr': 'expr3', 'ref_id': 5},
     {'type': 'at_least_n',
      'refs': [2, 3, 4, 5],
      'n': 2,
      'output_id': 0,
      'maybe_match_id': 1}]

    """
    expr_list = list(iterable)
    if n > len(expr_list):
        raise SyntaxError(f'"n" ({n}) cannot be larger than the number of '
                          f'expressions ({len(expr_list)}).')

    ref_ids = []
    i = 0
    while len(ref_ids) < len(expr_list):
        if i not in {output_id, maybe_match_id}:
            ref_ids.append(i)
        i += 1

    exprs = [{'type': 'expression', 'expr': expr, 'ref_id': ref_id}
             for ref_id, expr in zip(ref_ids, expr_list)]
    match_n = {'type': 'at_least_n', 'refs': ref_ids, 'n': n,
               'output_id': output_id, 'maybe_match_id': maybe_match_id}
    exprs.append(match_n)
    return exprs
```

6.2.5 `combination`

The `'combination'` syntax includes `'formula'` and `'simplify'`. `'formula'` is a string which represents a boolean combination of expressions using `'ref_id'`s. Currently, all of the expressions referenced must be of type `'expression'` or `'combination'`. Logical ands, ors, and nots are supported by multiple syntax formats:

- Logical not: `'NOT'`, `'!'`, `'~'`
- Logical and: `'AND'`, `'&'`
- Logical or : `'OR'` , `'|'`

The priority order is: not, and, or. As an example, the expression `NOT 0 AND 1 OR 2 AND NOT 3 OR 4` is equivalent to `((NOT 0) AND 1) OR (2 AND (NOT 3)) OR 4`.

Passing in `'simplify': True` causes the compiler to attempt to simplify the `'formula'` using `sympy.simplify_logic`. No simplification is made if `sympy` is not installed or `'simplify'` is left blank. The potential advantage of simplification is that it could cause the result to use fewer PEs.

Currently, the compiler combines the expressions given into one large expression which represents the boolean combination. This can be done manually as well, which often results in better processing element utilization. To attempt this manually, please see the document 'logical_combinations_guide.pdf'.

6.2.5.1 Example

This example also demonstrates how NPUsearch evaluates the entire file before determining whether the logical combination is True. Consider the expressions:

```
exprs = [
 {'type': 'expression', 'expr': 'opq', 'ref_id': 1},
 {'type': 'expression', 'expr': 'rst', 'ref_id': 2},
 {'type': 'expression', 'expr': 'baz.*uv', 'ref_id': 3},
 {'type': 'expression', 'expr': 'neuromorphic{3,}', 'ref_id': 4},
 {'type': 'expression', 'expr': 'wxy[Zz]', 'ref_id': 5},
 {'type': 'combination', 'formula': '((2 | 3) & 1) & (4 | ! 5)', 'output_id': 0}
]
```

Consider the data: `"-------opq-baz-uv"`. Observe that expressions 1 and 3 are the only expressions which match, which causes the combination to evaluate to `True`. A match would then be returned in this scenario.

However, consider the data: `"-------opq-baz-uv-----wxyZ"`, which is the data from above with additional text added on. Observe that expressions 1, 3, and 5 are the only expressions which match, which causes the combination to evaluate to `False`. A match would not be returned in this scenario.

Note also that only `0` would ever be returned in the `'matches'` list as it is the only output id.