# Notes on the regexpr routines

**Declared In:** misckit/regexpr.h

## Module Description

*This is a second attempt at a description of the regexpr package. The Emacs documentation (sort of forgot about that the first time, oops) had information on most of the things that I missed in the first attempt, so I think that it is now fairly complete. Still, there well may be things I missed or got wrong, so PLEASE tell me if you come across any. Portions of the text were taken from UNIX man pages, the Emacs manual, and comments inside of the regexpr code.*

The regexpr package implements a set of regular expression handling routines. It imitates the syntax of GNU regular expressions, which are themselves an extension of standard Unix regular expressions (as defined by ed(1), sed(1), grep(1), etc.). A few extra syntax options are also provided.

From the ed(1) man page, a regular expression "specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression." For example, take the regular expression "`a*`". This will `match' any consecutive series of 'a' characters, since the '`*`' stands for `match zero or more occurrences of the previous regular expression', which in this case is just the letter 'a'. Matched strings would include "" (0 a's), "a", "aa", "aaa", "aaaaaaaa", and so on. So, in a sense, "`a*`" stands for the entire set of those matched strings.

In other words, regular expressions allow you to make generalized searches for strings instead of searching for each possible string in turn. In general, regular expressions have a syntax where a few characters have special meaning and the rest are "ordinary". An ordinary character simply matches itself; for example, "foo" can be considered to be a simple regular expression matching only the string "foo". In the syntax used by this package, the special characters are `` `*' ``, `` `+' ``, `` `?' ``, `` `[' ``, `` `]' ``, `` `$' ``, `` `^' ``, `` `.' ``, and `` `\' ``.

These are the basic rules for building regular expressions handled by this package. In the following descriptions, `character' excludes newline; that is, if it says "matches any character", it really means "matches any character except for the newline character".

- Two regular expressions concatenated match a match of the first followed by a match of the second.
- A single character not otherwise endowed with special meaning matches that character.
- A `.` (period) matches any character (except for a newline). So, the regular expression "`a.b`" matches any three-character string that starts with `a' and ends with `b'.
- A regular expression followed by an `*` (asterisk) matches a sequence of 0 or more matches of the regular expression. A regular expression followed by a `+` (plus) matches a sequence of 1 or more matches of the regular expression. A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression. Therefore, the expression "`ca+r`" matches the strings "car", "caar", "caaaar", etc., while "`ca*r`" matches the same set of strings as well as "cr", since the * allows for zero `a' characters between the `c' and `r'. The expression "`ca?r`" matches only the strings "cr" and "car".

  If one of these three characters is placed in a position that does not make sense for a postfix operator, it is matched as a literal character. Thus, "`*foo`" actually matches the string "*foo", because there is no previous regular expression for * to operate on. This "feature" is only included for back compatibility with Unix regular expressions. It is considered bad form to use this construct, and it is not a good idea to count on it. If you want to turn this feature off, meaning that these cases will return errors instead of matching the literal character, you can use the syntax option `RE_CONTEXT_INDEP_OPS`.

On a more technical note, the matcher processes a * (or +) construct by matching as many repetitions of the previous expression as can be found. Then, it continues with trying to match the rest of the pattern. If this fails, backtracking occurs, discarding some of the matches of the *'d construct if that makes it possible to match the rest of the pattern. For example, while matching "`c[ad]*ar`" against the string "caddaar", the "`[ad]*`" first matches "addaa", but this does not allow the next "a" in the pattern to match. Therefore, the last match of a [ad] character is undone and the following "a" tried again, this time succeeding and matching the entirety of "caddaar".

- A string enclosed in brackets `[...]` is called a "character set", which in the simplest case matches any single character that is in the set. For example, "`[ad]`" matches either "a"or "d", and "`c[ad]+r`" matches "car", "cdr", "cadr", "caddar", and so on.

  The usual special characters have no special meaning inside of a character set. Thus, "`[ab*]`" matches either the character `a', the character `b', or the character `*'. Instead, there is a different set of `special' characters inside a character set: `` `–' ``, `` `]' ``, and `` `^' ``.

  Ranges of ASCII character codes may be abbreviated, as in `[a-z0-9]`, which would match any lowercase letter or digit. If you want the character `-' to be one of the characters matched it must be placed where it can't be mistaken as a range indicator, such as the first or last character in the set or between ranges. For similar reasons, a `]' may occur only as the first character of the set.

If the first character of the set is `^`, it is a "complemented character set". In other words, it matches any character *not* specified in the set. Therefore, "`[^0-9a-zA-Z]`" matches any character that is not a letter or a digit (the `^' is not part of the set). Again, if you want `^' to actually be in the set, it can't be the first character, to avoid confusion.

A complemented character set *can* match a newline character, unless newline is in the set of characters not to match. This is in contrast to the way some programs -- such as grep(1) -- handle regular expressions.

- The character `^` matches the beginning of a line. Thus, "`^blah`" matches the string "blah" only if it is at the beginning of a line.
- The character `$` is like `^`, except it matches the end of a line. "`h+$`" will match a string of `h' characters at the end of a line.
- For the most part, `\` followed by a single character other than newline matches that character. There are a few cases -- most of them the GNU extensions -- where a character quoted in this manner has special meaning, and therefore does not match itself. In all of these cases, the character is normally not special, so it wouldn't be necessary to quote it in the first place. Typically, this construct is used to quote a special character to indicate that you want a literal match of that character.

  **NOTE:** Since the C compiler interprets \ in constant strings itself, it is actually a little difficult to use this method. Anytime you want a \ character in your regular expression, you need to put \\ in a constant string to tell the compiler you really want the character \ there. Eventually, this can get a bit confusing. For example, if you really want to match the literal character \ in your regular expression, the regexpr routines need to be given \\ to follow the above rule. To get that, you need to put \\\\ in your quoted string, since the compiler will reduce that to \\ in the string the regexpr routine gets. I've found that it is easier to just enclose a special character in square brackets [ ] for a literal match of that character. It doesn't always look very pretty, but it's better than trying to keep track of lots of `\' characters through the various reductions.

- `\|` is an OR construct. In other words, two regular expressions with `\|` in between them matches either what the first expression matches or what the second matches, but not both. For example, "`foo\|bar`" matches either the string "foo" or the string "bar", but no other string.

  You might have noticed that unlike *, +, and ? (which only apply to the shortest possible preceding subexpression), `\|` applies to the *largest* possible surrounding expressions. The only way to limit this scope is to use the "`\(` ... `\)`" construct as described below. Full backtracking capability exists to handle multiple uses of `\|`, so if your regular expression is "`abc\|def\|ghi`", it can match the string "ghi".

- `\(` ... `\)` is a grouping construct that can serve different purposes. First, as noted above, it can be used to limit the scope of the `\|` construct. For example, "`restricti\(ve\|on\)`" will match either the string "restrictive" or the string "restriction". Second, it can be used to specify the subexpression that the postfix operators *, +, and ? apply to. For example, "`ba\(na\)*`" will match "ba", "bana", "banana", "bananana", and so on.

Lastly, `\(` ... `\)` can be used to mark a portion of a matched string for future reference. This "future reference" is either by use of the `\{digit}` construct, as described below, or by using registers in the searching and matching functions (After a search or match, these registers contain the indexes and lengths of the portions of the matched string marked in this way). This is actually a different functionality altogether, since the first two purposes mark parts of the expression for use by other operators, while this marks portions of the actually matched text.

- `\{digit}` matches the same text that the `digit`'th `\(` ... `\)` grouping matched. For example, "`\([a-z]+\)_\1`" would match "find_find", "spoon_spoon", "buzz_buzz", etc. Note, again, that this refers to the actual text that was matched inside a `\(` ... `\)` grouping, NOT the expression contained inside. The groupings are numbered in order of their beginnings, and start at 1. Thus, you can use `\1` through `\9` to refer to the text matched by the corresponding grouping. If you need to match more than nine groupings, see the `RE_ANSI_HEX` syntax option.
- The order of precedence of operators at the same parenthesis level is `[]` then `*+?` then concatenation then `|` and newline.

That is the extent of the syntax for normal Unix regular expressions; the rest of the rules are GNU extensions.

- `` \` `` matches the beginning of the text buffer.
- `\'` matches the end of the text buffer.
- `\b` matches the beginning or end of a word. For example, "`\bplay\b`" matches "play" if it is a separate word, and "`\bplays?\b`" matches either "play" or "plays" as a separate word.
- `\B` matches the empty string, provided it is NOT at the beginning or end of a word.

  Actually, I'm not sure this construct is being handled correctly. Sometimes it seems to work, but other times it matches the same way that `\b` does or it does not match when it seems it should. If anyone can confirm that this feature is in fact broken, or can point out that I'm misunderstanding it, I'd appreciate it...
- `\<` matches the beginning of a word.
- `\>` matches the end of a word.
- `\w` matches any character that is part of a word. This seems to mean letters and digits, but not punctuation characters.
- `\W` matches any character that is not part of a word; basically, anything that `\w` does not match.

These syntax rules can be modified by setting the syntax variable with **re_set_syntax()**. The syntax variable is one of the following constants or a bitwise OR'ed (using `|') combination thereof:

`RE_NO_BK_PARENS`
: With this option set, the meanings of `( ... )` and `\( ... \)` are switched; normal parentheses will be treated as the grouping construct and quoted parentheses will match the literal characters.

`RE_NO_BK_VBAR`
: The meanings of **|** and **\|** are switched; a normal `|` will mean the OR construct and `\|` will match the literal character..

`RE_BK_PLUS_QM`
: The meanings of `+ ?` and `\+ \?` are switched; the normal characters will now match literally and the quoted ones will be the postfix operators.

`RE_TIGHT_VBAR`
: Changes the precedence rules slightly; now `|` binds tighter than `^` and `$`.

`RE_NEWLINE_OR`
: A newline character will be treated as an OR, just like the `\|` construct.

`RE_CONTEXT_INDEP_OPS`
: `^$?*+` are special in all contexts. Normally, if any of these operators are placed in a position that

doesn't make sense (such as the first character of the expression, where there is no previous subexpression to operate on), it will match the literal character. With this option set, these cases will return errors instead.

`RE_ANSI_HEX`

Allows ANSI sequences, e.g. `\n` (newline), `\t` (tab), `\r` (carriage return), `\x`*`hh`*, etc. Also means `\v`*`dd`*, where *`dd`* is a number from 10 to 99, will refer to the *`dd`*'th `\(` ... `\)` grouping in the same way that `\1` .. `\9` refer to the first nine groupings. You must #define `RE_NREGS` to be at least one more than the number of groupings you intend to have in your regex to use this feature The default is 10, so normally you can only refer to the first nine groupings.

Note, again, that the C compiler intercepts the `\` character in constant strings and will translate things like `\n` normally for you, so you don't necessarily need this option set to get that behavior. To get `\n` translated by the regexpr routines, you need to put `\\n` in a constant string. This will pass `\n` to the regexpr routines, which will then translate to a newline.

`RE_NO_GNU_EXTENSIONS`

Disallows GNU extensions; specifically, takes out the special meaning of `` \` ``, `\'`, `\b`, `\B`, `\<`, `\>`, `\w`, and `\W`.

The following are predefined OR'ed combinations of the above constants for some common regular expression styles that you may also use:

`RE_SYNTAX_AWK`
: The syntax used by awk(1), I guess. This is short for (`RE_NO_BK_PARENS` | `RE_NO_BK_VBAR` | `RE_CONTEXT_INDEP_OPS`)

`RE_SYNTAX_EGREP`
: The syntax used by egrep(1). This is short for (`RE_SYNTAX_AWK` | `RE_NEWLINE_OR`)

`RE_SYNTAX_GREP`
: The syntax used by grep(1). This is short for (`RE_BK_PLUS_QM` | `RE_NEWLINE_OR`)

`RE_SYNTAX_EMACS`
: The syntax used by emacs, i.e. the GNU syntax. This is basically the default, since it defines none of the extra syntax options.

There are actually a couple of other options normally in Emacs that are supported in this package if you `#define emacs`; these mostly have to do with matching character types you define in a syntax table. I am not listing them here because by default `emacs` is not defined. Look in the Emacs documentation for more information (a good idea anyways).

## Examples

Some usage examples:

`abc*`
Matches an 'a', then a 'b', then any number of 'c's.
Ex: "ab", "abc", "abcc", "abccccc", etc.

`(abc)*` [with `RE_NO_BK_PARENS`]
Matches any number of consecutive "abc" strings, including 0 of them.
Ex: "", "abc", "abcabc", "abcabcabc", etc.

`(abc)*` [without `RE_NO_BK_PARENS`]
Matches the string "(abc", then any number of ')' characters.
Ex: "(abc", "(abc)", "(abc))", "(abc))))", etc.

`[a-zA-Z]+`
Matches any string of only alphabetic characters. There has to be at least one character.

`[0-9]+`

Matches any string of only numeric characters. Basically, this means an integer.

`[0-9]+([.][0-9]*)?` [with `RE_NO_BK_PARENS`]

Matches a series of digits, then, optionally, a period (decimal point) followed by another string of digits. Basically, this matches some of the ways you can specify a floating point number.
Ex: "34", "34.2", "0.44"

`[0-9]+[.][0-9]+([eE][+-]?[0-9]+)?` [with `RE_NO_BK_PARENS`]

Matches other types of floating point numbers. Matches a series of digits, decimal point, another series of digits, then an optional exponent section, which contains an optional sign character.
Ex: "0.6", "2.37e-22", "9.1E23"

`[(][^)]*[)]`

Matches a pair of parentheses with whatever is inside of them. Literally, matches a left-paren, then any number of non-right-paren characters, then a right-paren.
Ex: "()", "(abc)", "(const char *)", "(aa((a(aa)"

`.*`

Matches a string of anything (except a newline character). This is a way of matching an entire line no matter what is there, or, when tacked on the end of a regular expression, matching the rest of the line.

`(abc)|(def)` [with `RE_NO_BK_VBAR` and `RE_NO_BK_PARENS`]
Matches either "abc" or "def".

## Types and Constants

**regexp_t**, **struct re_pattern_buffer**

```
typedef struct re_pattern_buffer {
        char *buffer;                  Compiled pattern
        int allocated;                 Allocated size of compiled pattern
        int used;                      Actual length of compiled pattern
        char *fastmap;                 fastmap[ch] is true if ch can start pattern
        char *translate;               Translation to apply during compilation/matching
        char fastmap_accurate;         True if fastmap is valid
        char can_be_null;              True if can match empty string
        char uses_registers;           Registers are used and need to be initialized
        char anchor;                   anchor: 0=none 1=begline 2=begbuf
} *regexp_t;
```

**regexp_registers_t**, **struct re_registers**

```
typedef struct re_registers {
        int start[RE_NREGS];        Start offset of region
        int end[RE_NREGS];          End offset of region
} *regexp_registers_t;
```

**RE_NREGS**

```
#define RE_NREGS   10               Number of registers available
```

## Functions

One thing to note is that these routines will return the longest possible match it can. For example, if you match a regular expression of "a*" in the string "aaaaaaaa", it will return a length of eight, since that is the longest possible match, even though a shorter portion would have technically qualified as well. When searching, the routines will return a match at the first possible index.

If you want to make your searches case-insensitive, you can modify the translation table. Simply map the positions of the upper-case characters to their lower case equivalents, or vice versa. You can actually map any characters to any others this way (such as perhaps mapping all whitespace characters to space), but case-insensitivity is probably the most common use.

```
struct re_pattern_buffer pat;
char tr[256]; int i;

for (i=0;i<256;i++) tr[i] = i; //sets up basic table
if (!case) for (i='A';i<='Z';i++) tr[i] = i- 'A' + 'a';
pat.translate = tr;
```

If you want to find out exactly where your \( ... \) groupings matched, you can pass a **regexp_registers_t** (a pointer to a registers struct) to the searching and matching functions. Upon return, the fields of the struct contain the indexes and lengths of all the \( ... \) subgroupings. If `regs` is your regexp_registers_t, then `regs->start[0]` and `regs->end[0]` are the index and length of the entire expression, and `regs->start[1]`, `regs->end[1]` through `regs->start[9]`, `regs->end[9]` are the indexes and lengths of the first nine groupings. If you wish to allocate more registers, increase `RE_NREGS`. You must allocate the space for `regs` before use. If you do not want to use this feature, just pass NULL for the registers argument in the functions.

**re_compile_fastmap()**

void **re_compile_fastmap(**regexp_t *compiled***)**

This computes the fastmap for the regexp. The fastmap is an array of 256 characters, where fastmap[ch] is nonzero if the character ch can possibly start the regular expression. The matching routines can then use this to skip more quickly over implausible characters. Use of the fastmap is recommended for searching large blocks of text, though it may not be an advantage for searching smaller blocks. To use the fastmap, first compile the regular expression with **re_compile_pattern()**, then set the fastmap field to point to an allocated and initialized (zeroed) block of 256 characters and call **re_compile_fastmap()**. If you do not want to use the fastmap, make sure the fastmap field is NULL.

**re_compile_pattern()**

char ***re_compile_pattern(**char **regex*, int *regex_size*, regexp_t *compiled***)**

Compiles the regular expression (given in *regex* and length in *regex_size*). Returns NULL if the regexp compiled successfully, and an error message if an error was encountered. The buffer field must be initialized to a memory area allocated by malloc (or to NULL) before use, and the allocated field must be set to its length (or 0 if buffer is NULL). Also, the translate field must be set to point to a valid translation table, or NULL if it is not used. The compiled expression will be place in *compiled*, the space for which must be allocated ahead of time.

If you do not allocate the buffer field, it will be automatically allocated for you, but you must remember to free it when you are finished with the regexp_t.

**re_match_pattern(), re_match_pattern_2()**

int **re_match_pattern(**regexp_t *compiled*, char **string*, int *size*, int *pos*, regexp_registers_t *regs***)**
int **re_match_pattern2(**regexp_t *compiled*, char **string1*, int *size1*, char **string2*, int *size2*, int *pos*,
regexp_registers_t *regs***)**

**re_match_pattern()** tries to match the regexp against *string*, where *pos* is the index to try matching at. **re_match_pattern_2()** does the same thing, except it matches against the concatenation of *string1* and *string2*. *Size*, *size1*, and *size2* are the lengths of their corresponding strings. These functions return the length of the matched portion, -1 if the pattern could not be matched, or -2 if an error (such as failure stack overflow) is encountered.

**re_search_pattern(), re_search_pattern_2()**

int **re_search_pattern(**regexp_t *compiled*, char **string*, int *size*, int *startpos*, int *range*, regexp_registers_t
*regs***)**
int **re_search_pattern2(**regexp_t *compiled*, char **string1*, int *size1*, char **string2*, int *size2*, int *startpos*, int

*range*, regexp_registers_t *regs***)**

The **re_search_pattern()** function searches for a substring matching the regexp. *range* specifies at how many positions to try matching; positive values indicate searching forwards, and negative values indicate searching backwards. *startpos* specifies the index to start searching at. **re_search_pattern_2()** does the same thing, except it searches the concatenation of *string1* and *string2*. These functions return the first index at which a match is found, -1 if no match is found, or -2 if an error (such as failure stack overflow) is encountered. Use **re_match_pattern()** or **re_match_pattern2()** at the returned index to find the length of the matched portion.

**re_set_syntax()**
int **re_set_syntax(**int *syntax***)**

**re_set_syntax()** sets the regular expression syntax to be used by all the routines in the regexpr package. The value of the previous syntax variable is returned. The syntax may be any of the following constants, or a bitwise OR (using `|') combination thereof:

| | |
|---|---|
| `RE_NO_BK_PARENS` | No quoting for parentheses |
| `RE_NO_BK_VBAR` | No quoting for vertical bar |
| `RE_BK_PLUS_QM` | Quoting needed for + and ? |

| | |
|---|---|
| `RE_TIGHT_VBAR` | \| binds tighter than ^ and $ |
| `RE_NEWLINE_OR` | Treat newline as or |
| `RE_CONTEXT_INDEP_OPS` | ^$?*+ are special in all contexts |
| `RE_ANSI_HEX` | ANSI sequences (`\n` etc) and `\x`*`hh`* |
| `RE_NO_GNU_EXTENSIONS` | No GNU extensions |

See the overview section for a further description of these constants.

The following are predefined definitions for some common regexp styles that you may also use:

| | |
|---|---|
| `RE_SYNTAX_AWK` | `RE_NO_BK_PARENS\|RE_NO_BK_VBAR\|RE_CONTEXT_INDEP_OPS` |
| `RE_SYNTAX_EGREP` | `RE_SYNTAX_AWK\|RE_NEWLINE_OR` |
| `RE_SYNTAX_GREP` | `RE_BK_PLUS_QM\|RE_NEWLINE_OR` |
| `RE_SYNTAX_EMACS` | <default>, none of above constants set |