1 *regex* regular expression matching library.

1.1 Overview

Regular expression matching allows you to test whether a string fits into a specific syntactic shape. You can also search a string for a substring that fits a pattern.

A regular expression describes a set of strings. The simplest case is one that describes a particular string; for example, the string '`foo`' when regarded as a regular expression matches '`foo`' and nothing else. Nontrivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression '`foo\|bar`' matches either the string '`foo`' or the string '`bar`'; the regular expression '`c[ad]*r`' matches any of the strings '`cr`', '`car`', '`cdr`', '`caar`', '`cadddar`' and all other such strings with any number of '`a`''s and '`d`''s.

The first step in matching a regular expression is to compile it. You must supply the pattern string and also a pattern buffer to hold the compiled result. That result contains the pattern in an internal format that is easier to use in matching.

Having compiled a pattern, you can match it against strings. You can match the compiled pattern any number of times against different strings.

1.2 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`+`', '`?`', '`[`', '`]`' and '`\`'. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial.

Note: for Unix compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '`*foo`' treats '`*`' as ordinary since there is no preceding expression on which the '`*`' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

'`.`' is a special character that matches anything except a newline. Using concatenation, we can make regular expressions like '`a.b`' which matches any three-character string which begins with '`a`' and ends with '`b`'.

'`*`' is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches '`f`' followed by any number of '`o`''s.

The case of zero '`o`''s is allowed: '`fo*`' does match '`f`'.

'`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'.

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`''d construct in case that makes it possible to match the rest of the pattern. For example, 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. So the last of the matches of '`[ad]`' is undone and the following '`a`' is tried again. Now it succeeds.

'`+`' '`+`' is like '`*`' except that at least one match for the preceding pattern is required for '`+`'. Thus, '`c[ad]+r`' does not match '`cr`' but does match anything else that '`c[ad]*r`' would match.

'`?`' '`?`' is like '`*`' except that it allows either zero or one match for the preceding pattern. Thus, '`c[ad]?r`' matches '`cr`' or '`car`' or '`cdr`', and nothing else.

'`[ ... ]`' '`[`' begins a *character set*, which is terminated by a '`]`'. In the simplest case, the characters between the two form the set. Thus, '`[ad]`' matches either '`a`' or '`d`', and '`[ad]*`' matches any string of '`a`''s and '`d`''s (including the empty string), from which it follows that '`c[ad]*r`' matches '`car`', etc.

Character ranges can also be included in a character set, by writing two characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case letter or '`$`', '`%`' or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: '`]`', '`-`' and '`^`'.

To include a '`]`' in a character set, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

'`[^ ... ]`' '`[^`' begins a *complement character set*, which matches any character except the ones specified. Thus, '`[^a-z0-9A-Z]`' matches all characters *except* letters and digits.

'`^`' is not special in a character set unless it is the first character. The character following the '`^`' is treated as if it were first (it may be a '`-`' or a '`]`').

'`^`' is a special character that matches the empty string – but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, '`^foo`' matches a '`foo`' which occurs at the beginning of a line.

'`$`' is similar to '`^`' but matches only at the end of a line. Thus, '`xx*$`' matches a string of one or more '`x`''s at the end of a line.

'`\`' has two functions: it quotes the above special characters (including '`\`'), and it introduces additional special constructs.

Because '`\`' quotes special characters, '`\$`' is a regular expression which matches only '`$`', and '`\[`' is a regular expression which matches only '`[`', and so on.

For the most part, '`\`' followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by '`\`', are special constructs. Such characters are always ordinary when encountered on their own.

No new special characters will ever be defined. All extensions to the regular expression syntax are made by defining new two-character constructs that begin with '`\`'.

'`\|`' specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* will match.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

Full backtracking capability exists when multiple '`\|`''s are used.

'`\( ... \)`'

is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix '`*`' to operate on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number of '`na`''s.
3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same '`\( ... \)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

'`\`*`digit`*' After the end of a '`\( ... \)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by *digit* to mean "match the same text matched the *digit*'th time by the '`\( ... \)`' construct." The '`\( ... \)`' constructs are numbered in order of commencement in the regexp.

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in order of their beginnings. '`\1`' through '`\9`' may be used to refer to the text matched by the corresponding '`\( ... \)`' construct.

For example, '`\(.*\)\1`' matches any string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

'`\b`' matches the empty string, but only if it is at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bball\(s\|\)\b`' matches '`ball`' or '`balls`' as a separate word.

'`\B`'	matches the empty string, provided it is *not* at the beginning or end of a word.
'`\<`'	matches the empty string, but only if it is at the beginning of a word.
'`\>`'	matches the empty string, but only if it is at the end of a word.
'`\w`'	matches any word-constituent character.
'`\W`'	matches any character that is not a word-constituent.

There are a number of additional '\' regexp directives available for use within Emacs only.

1.2.1 Constructs Available in Emacs Only

'`` \` ``'	matches the empty string, but only if it is at the beginning of the buffer.
'`\'`'	matches the empty string, but only if it is at the end of the buffer.
'`\s`*code*'	matches any character whose syntax is *code*. *code* is a letter which represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open-parenthesis, etc. See the documentation for the Emacs function '`modify-syntax-entry`' for further details. Thus, '`\s(`' matches any character with open-parenthesis syntax.
'`\S`*code*'	matches any character whose syntax is not *code*.

1.3 Programming using the `regex` library

1.3.1 Compiling a Regular Expression

To compile a regular expression, you must supply a pattern buffer. This is a structure defined, in the include file `regex.h`, as follows:

```
struct re_pattern_buffer
  {
    char *buffer   /* Space holding the compiled pattern commands. */
    int allocated  /* Size of space that  buffer  points to */
    int used       /* Length of portion of buffer actually occupied */
    char *fastmap; /* Pointer to fastmap, if any, or zero if none. */
                   /* re_search uses the fastmap, if there is one,
                      to skip quickly over totally implausible
                      characters */
    char *translate;
                   /* Translate table to apply to characters before
                      comparing, or zero for no translation.
                      The translation is applied to a pattern when
                      it is compiled and to data when it is matched. */
    char fastmap_accurate;
                   /* Set to zero when a new pattern is stored,
                      set to one when the fastmap is updated from it. */
  };
```

Before compiling a pattern, you must initialize the `buffer` field to point to a block of memory obtained with `malloc`, and the `allocated` field to the size of that block, in bytes. The pattern compiler will replace this block with a larger one if necessary.

You must also initialize the `translate` field to point to the translate table that you will use when you match the compiled pattern, or to zero if you will use no translate table when you match. See Section 1.3.4 [translation], page 6.

Then call `re_compile_pattern` to compile a regular expression into the buffer:

```
re_compile_pattern (regex, regex_size, buf)
```

regex is the address of the regular expression (`char *`), *regex_size* is its length (`int`), *buf* is the address of the buffer (`struct re_pattern_buffer *`).

`re_compile_pattern` returns zero if it succeeds in compiling the regular expression. In that case, `*buf` now contains the results. Otherwise, `re_compile_pattern` returns a string which serves as an error message.

After compiling, if you wish to search for the pattern, you must initialize the `fastmap` component of the pattern buffer. See Section 1.3.3 [searching], page 5.

1.3.2 Matching a Compiled Pattern

Once a regular expression has been compiled into a pattern buffer, you can match the pattern buffer against a string with `re_match`.

```
re_match (buf, string, size, pos, regs)
```

buf is, once again, the address of the buffer (`struct re_pattern_buffer *`). *string* is the string to be matched (`char *`). *size* is the length of that string (`int`). *pos* is the position within the string at which to begin matching (`int`). The beginning of the string is position 0. *regs* is described below. Normally it is zero. See Section 1.3.5 [registers], page 6.

`re_match` returns `-1` if the pattern does not match; otherwise, it returns the length of the portion of `string` which was matched.

For example, suppose that *buf* points to a buffer containing the result of compiling `x*`, *string* points to `xxxxxy`, and *size* is `6`. Suppose that *pos* is `2`. Then the last three `x`'s will be matched, so `re_match` will return `3`. If *pos* is zero, the value will be `5`. If *pos* is `5` or `6`, the value will be zero, meaning that the null string was successfully matched. Note that since `x*` matches the empty string, it will never entirely fail.

It is up to the caller to avoid passing a value of *pos* that results in matching outside the specified string. *pos* must not be negative and must not be greater than *size*.

1.3.3 Searching for a Match

Searching means trying successive starting positions for a match until a match is found. To search, you supply a compiled pattern buffer. Before searching you must initialize the `fastmap` field of the pattern buffer (see below).

```
re_search (buf, string, size, startpos, range, regs)
```

is called like `re_match` except that the *pos* argument is replaced by two arguments *startpos* and *range*. `re_search` tests for a match starting at index *startpos*, then at `startpos + 1`, and so on. It tries *range* consecutive positions before giving up and returning `-1`. If a match is found, `re_search` returns the index at which the match was found.

If *range* is negative, *re_search* tries starting positions *startpos*, `startpos - 1`, ... in that order. `|range|` is the number of tries made.

It is up to the caller to avoid passing value of *startpos* and *range* that result in matching outside the specified string. *startpos* must be between zero and *size*, inclusive, and so must `startpos + range - 1` (if *range* is positive) or `startpos + range + 1` (if *range* is negative).

If you may be searching over a long distance (that is, trying many different match starting points) with a compiled pattern, you should use a *fastmap* in it. This is a block of 256 bytes, whose address is placed in the `fastmap` component of the pattern buffer. The first time you search for a particular compiled pattern, the fastmap is set so that `fastmap[ch]` is nonzero if the character *ch* might possibly start a match for this pattern. `re_search` checks each character against the fastmap so that it can skip more quickly over non-matches.

If you do not want a fastmap, store zero in the `fastmap` component of the pattern buffer before calling `re_search`.

In either case, you must initialize this component in a pattern buffer before you can use that buffer in a search; but you can choose as an initial value either zero or the address of a suitable block of memory.

If you compile a new pattern in an existing pattern buffer, it is not necessary to reinitialize the `fastmap` component (unless you wish to override your previous choice).

1.3.4 Translate Tables

With a translate table, you can apply a transformation to all characters before they are compared. For example, a table that maps lower case letters into upper case (or vice versa) causes differences in case to be ignored by matching.

A translate table is a block of 256 bytes. Each character of raw data is used as an index in the translate table. The value found there is used instead of the original character. Each character in a regular expression, except for the syntactic constructs, is translated when the expression is compiled. Each character of a string being matched is translated whenever it is compared or tested.

A suitable translate table to ignore differences in case maps all characters into themselves, except for lower case letters, which are mapped into the corresponding upper case letters. It could be initialized by:

```
for (i = 0; i < 0400; i++)
  table[i] = i;
for (i = 'a'; i <= 'z'; i++)
  table[i] = i - 040;
```

You specify the use of a translate table by putting its address in the *translate* component of the compiled pattern buffer. If this component is zero, no translation is done. Since both compilation and matching use the translate table, you must use the same table contents for both operations or confusing things will happen.

1.3.5 Registers: or “What Did the ‘`\( ... \)`’ Groupings Actually Match?”

If you want to find out, after the match, what each of the first nine ‘`\( ... \)`’ groupings actually matched, you can pass the *regs* argument to the match or search function. Pass the address of a structure of this type:

```
struct re_registers
  {
    int start[RE_NREGS];
    int end[RE_NREGS];
  };
```

`re_match` and `re_search` will store into this structure the data you want. `*regs*->start[*reg*]` will be the index in *string* of the beginning of the data matched by the *reg*'th '`\( ... \)`' grouping, and `*regs*->end[*reg*]` will be the index of the end of that data (the index of the first character beyond those matched). The values in the start and end arrays at indexes greater than the number of '`\( ... \)`' groupings present in the regular expression will be set to the value -1. Register numbers start at 1 and run to `RE_NREGS - 1` (normally 9). `*regs*->start[0]` and `*regs*->end[0]` are similar but describe the extent of the substring matched by the entire pattern.

Both `struct re_registers` and `RE_NREGS` are defined in `regex.h`.

1.3.6 Matching against Split Data

The functions `re_match_2` and `re_search_2` allow one to match in or search data which is divided into two strings.

`re_match_2` works like `re_match` except that two data strings and sizes must be given.

```
re_match_2 (buf, string1, size1, string2, size2, pos, regs)
```

The matcher regards the contents of *string1* as effectively followed by the contents of *string2*, and matches the combined string against the pattern in *buf*.

`re_search_2` is likewise similar to `re_search`:

```
re_search_2 (buf, string1, size1, string2, size2, startpos, range, regs)
```

The value returned by *re_search_2* is an index into the combined data made up of *string1* and *string2*. It never exceeds `size1 + size2`. The values returned in the *regs* structure (if there is one) are likewise indices in the combined data.

1.3.7 Unix-Compatible Entry Points

The standard Berkeley Unix way to compile a regular expression is to call `re_comp`. This function takes a single argument, the address of the regular expression, which is assumed to be terminated by a null character.

`re_comp` does not ask you to specify a pattern buffer because it has its own pattern buffer — just one. Using `re_comp`, one may match only the most recently compiled regular expression.

The value of `re_comp` is zero for success or else an error message string, as for `re_compile_pattern`.

Calling `re_comp` with the null string as argument it has no effect; the contents of the buffer remain unchanged.

The standard Berkeley Unix way to match the last regular expression compiled is to call `re_exec`. This takes a single argument, the address of the string to be matched. This string is assumed to be terminated by a null character. Matching is tried starting at each position in the string. `re_exec` returns `1` for success or `0` for failure. One cannot find out how long a substring was matched, nor what the '`\( ... \)`' groupings matched.