Regex

edition 0.12a
19 September 1992

Kathryn A. Hargreaves
Karl Berry

1 Overview

A *regular expression* (or *regexp*, or *pattern*) is a text string that describes some (mathematical) set of strings. A regexp r *matches* a string s if s is in the set of strings described by r.

Using the Regex library, you can:

- see if a string matches a specified pattern as a whole, and
- search within a string for a substring matching a specified pattern.

Some regular expressions match only one string, i.e., the set they describe has only one member. For example, the regular expression '`foo`' matches the string '`foo`' and no others. Other regular expressions match more than one string, i.e., the set they describe has more than one member. For example, the regular expression '`f*`' matches the set of strings made up of any number (including zero) of '`f`'s. As you can see, some characters in regular expressions match themselves (such as '`f`') and some don't (such as '`*`'); the ones that don't match themselves instead let you specify patterns that describe many different strings.

To either match or search for a regular expression with the Regex library functions, you must first compile it with a Regex pattern compiling function. A *compiled pattern* is a regular expression converted to the internal format used by the library functions. Once you've compiled a pattern, you can use it for matching or searching any number of times.

The Regex library consists of two source files: '`regex.h`' and '`regex.c`'. Regex provides three groups of functions with which you can operate on regular expressions. One group—the GNU group—is more powerful but not completely compatible with the other two, namely the POSIX and Berkeley UNIX groups; its interface was designed specifically for GNU. The other groups have the same interfaces as do the regular expression functions in POSIX and Berkeley UNIX.

We wrote this chapter with programmers in mind, not users of programs—such as Emacs—that use Regex. We describe the Regex library in its entirety, not how to write regular expressions that a particular program understands.

2 Regular Expression Syntax

Characters are things you can type. *Operators* are things in a regular expression that match one or more characters. You compose regular expressions from operators, which in turn you specify using one or more characters.

Most characters represent what we call the match-self operator, i.e., they match themselves; we call these characters *ordinary*. Other characters represent either all or parts of fancier operators; e.g., '`.`' represents what we call the match-any-character operator (which, no surprise, matches (almost) any character); we call these characters *special*. Two different things determine what characters represent what operators:

1. the regular expression syntax your program has told the Regex library to recognize, and
2. the context of the character in the regular expression.

In the following sections, we describe these things in more detail.

2.1 Syntax Bits

In any particular syntax for regular expressions, some characters are always special, others are sometimes special, and others are never special. The particular syntax that Regex recognizes for a given regular expression depends on the value in the `syntax` field of the pattern buffer of that regular expression.

You get a pattern buffer by compiling a regular expression. See Section 7.1.1 [GNU Pattern Buffers], page 24, and Section 7.2.1 [POSIX Pattern Buffers], page 35, for more information on pattern buffers. See Section 7.1.2 [GNU Regular Expression Compiling], page 26, Section 7.2.2 [POSIX Regular Expression Compiling], page 35, and Section 7.3.1 [BSD Regular Expression Compiling], page 40, for more information on compiling.

Regex considers the value of the `syntax` field to be a collection of bits; we refer to these bits as *syntax bits*. In most cases, they affect what characters represent what operators. We describe the meanings of the operators to which we refer in Chapter 3 [Common Operators], page 9, Chapter 4 [GNU Operators], page 20, and Chapter 5 [GNU Emacs Operators], page 22.

For reference, here is the complete list of syntax bits, in alphabetical order:

RE_BACKSLASH_ESCAPE_IN_LISTS

If this bit is set, then '\' inside a list (see Section 3.6 [List Operators], page 13 quotes (makes ordinary, if it's special) the following character; if this bit isn't set, then '\' is an ordinary character inside lists. (See Section 2.4 [The Backslash Character], page 7, for what '\' does outside of lists.)

RE_BK_PLUS_QM

If this bit is set, then '\+' represents the match-one-or-more operator and '\?' represents the match-zero-or-more operator; if this bit isn't set, then '+' represents the match-one-or-more operator and '?' represents the match-zero-or-one operator. This bit is irrelevant if RE_LIMITED_OPS is set.

RE_CHAR_CLASSES

If this bit is set, then you can use character classes in lists; if this bit isn't set, then you can't.

RE_CONTEXT_INDEP_ANCHORS

If this bit is set, then '^' and '$' are special anywhere outside a list; if this bit isn't set, then these characters are special only in certain contexts. See Section 3.9.1 [Match-beginning-of-line Operator], page 18, and Section 3.9.2 [Match-end-of-line Operator], page 18.

RE_CONTEXT_INDEP_OPS

If this bit is set, then certain characters are special anywhere outside a list; if this bit isn't set, then those characters are special only in some contexts and are ordinary elsewhere. Specifically, if this bit isn't set then '*', and (if the syntax bit RE_LIMITED_OPS isn't set) '+' and '?' (or '\+' and '\?', depending on the syntax bit RE_BK_PLUS_QM) represent repetition operators only if they're not first in a regular expression or just after an open-group or alternation operator. The same holds for '{' (or '\{', depending on the syntax bit RE_NO_BK_BRACES) if it is the beginning of a valid interval and the syntax bit RE_INTERVALS is set.

RE_CONTEXT_INVALID_OPS

If this bit is set, then repetition and alternation operators can't be in certain positions within a regular expression. Specifically, the regular expression is invalid if it has:

- a repetition operator first in the regular expression or just after a match-beginning-of-line, open-group, or alternation operator; or
- an alternation operator first or last in the regular expression, just before a match-end-of-line operator, or just after an alternation or open-group operator.

If this bit isn't set, then you can put the characters representing the repetition and alternation characters anywhere in a regular expression. Whether or not they will in fact be operators in certain positions depends on other syntax bits.

`RE_DOT_NEWLINE`

If this bit is set, then the match-any-character operator matches a newline; if this bit isn't set, then it doesn't.

`RE_DOT_NOT_NULL`

If this bit is set, then the match-any-character operator doesn't match a null character; if this bit isn't set, then it does.

`RE_INTERVALS`

If this bit is set, then Regex recognizes interval operators; if this bit isn't set, then it doesn't.

`RE_LIMITED_OPS`

If this bit is set, then Regex doesn't recognize the match-one-or-more, match-zero-or-one or alternation operators; if this bit isn't set, then it does.

`RE_NEWLINE_ALT`

If this bit is set, then newline represents the alternation operator; if this bit isn't set, then newline is ordinary.

`RE_NO_BK_BRACES`

If this bit is set, then '`{`' represents the open-interval operator and '`}`' represents the close-interval operator; if this bit isn't set, then '`\{`' represents the open-interval operator and '`\}`' represents the close-interval operator. This bit is relevant only if `RE_INTERVALS` is set.

`RE_NO_BK_PARENS`

If this bit is set, then '`(`' represents the open-group operator and '`)`' represents the close-group operator; if this bit isn't set, then '`\(`' represents the open-group operator and '`\)`' represents the close-group operator.

`RE_NO_BK_REFS`

If this bit is set, then Regex doesn't recognize '`\`'*digit* as the back reference operator; if this bit isn't set, then it does.

`RE_NO_BK_VBAR`

If this bit is set, then '`|`' represents the alternation operator; if this bit isn't set, then '`\|`' represents the alternation operator. This bit is irrelevant if `RE_LIMITED_OPS` is set.

`RE_NO_EMPTY_RANGES`

If this bit is set, then a regular expression with a range whose ending point collates lower than its starting point is invalid; if this bit isn't set, then Regex considers such a range to be empty.

`RE_UNMATCHED_RIGHT_PAREN_ORD`

If this bit is set and the regular expression has no matching open-group operator, then Regex considers what would otherwise be a close-group operator (based on how `RE_NO_BK_PARENS` is set) to match '`)`'.

2.2 Predefined Syntaxes

If you're programming with Regex, you can set a pattern buffer's (see Section 7.1.1 [GNU Pattern Buffers], page 24, and Section 7.2.1 [POSIX Pattern Buffers], page 35) `syntax` field either to an arbitrary combination of syntax bits (see Section 2.1 [Syntax Bits], page 2) or else to the configurations defined by Regex. These configurations define the syntaxes used by certain programs—GNU Emacs, POSIX Awk, traditional Awk, Grep, Egrep—in addition to syntaxes for POSIX basic and extended regular expressions.

The predefined syntaxes–taken directly from '`regex.h`'—are:

```
#define RE_SYNTAX_EMACS 0

#define RE_SYNTAX_AWK                                                   \
  (RE_BACKSLASH_ESCAPE_IN_LISTS | RE_DOT_NOT_NULL                       \
   | RE_NO_BK_PARENS            | RE_NO_BK_REFS                         \
   | RE_NO_BK_VBAR               | RE_NO_EMPTY_RANGES                   \
   | RE_UNMATCHED_RIGHT_PAREN_ORD)

#define RE_SYNTAX_POSIX_AWK                                             \
  (RE_SYNTAX_POSIX_EXTENDED | RE_BACKSLASH_ESCAPE_IN_LISTS)

#define RE_SYNTAX_GREP                                                  \
  (RE_BK_PLUS_QM              | RE_CHAR_CLASSES                         \
   | RE_HAT_LISTS_NOT_NEWLINE | RE_INTERVALS                            \
   | RE_NEWLINE_ALT)

#define RE_SYNTAX_EGREP                                                 \
  (RE_CHAR_CLASSES        | RE_CONTEXT_INDEP_ANCHORS                    \
   | RE_CONTEXT_INDEP_OPS | RE_HAT_LISTS_NOT_NEWLINE                    \
   | RE_NEWLINE_ALT       | RE_NO_BK_PARENS                             \
   | RE_NO_BK_VBAR)

#define RE_SYNTAX_POSIX_EGREP                                           \
  (RE_SYNTAX_EGREP | RE_INTERVALS | RE_NO_BK_BRACES)

/* P1003.2/D11.2, section 4.20.7.1, lines 5078ff.  */
#define RE_SYNTAX_ED RE_SYNTAX_POSIX_BASIC
```

```
#define RE_SYNTAX_SED RE_SYNTAX_POSIX_BASIC

/* Syntax bits common to both basic and extended POSIX regex syntax.  */
#define _RE_SYNTAX_POSIX_COMMON                                         \
  (RE_CHAR_CLASSES | RE_DOT_NEWLINE      | RE_DOT_NOT_NULL              \
   | RE_INTERVALS  | RE_NO_EMPTY_RANGES)

#define RE_SYNTAX_POSIX_BASIC                                           \
  (_RE_SYNTAX_POSIX_COMMON | RE_BK_PLUS_QM)

/* Differs from ..._POSIX_BASIC only in that RE_BK_PLUS_QM becomes
   RE_LIMITED_OPS, i.e., \? \+ \| are not recognized.  Actually, this
   isn't minimal, since other operators, such as \`, aren't disabled.  */
#define RE_SYNTAX_POSIX_MINIMAL_BASIC                                   \
  (_RE_SYNTAX_POSIX_COMMON | RE_LIMITED_OPS)

#define RE_SYNTAX_POSIX_EXTENDED                                        \
  (_RE_SYNTAX_POSIX_COMMON | RE_CONTEXT_INDEP_ANCHORS                   \
   | RE_CONTEXT_INDEP_OPS  | RE_NO_BK_BRACES                            \
   | RE_NO_BK_PARENS       | RE_NO_BK_VBAR                              \
   | RE_UNMATCHED_RIGHT_PAREN_ORD)

/* Differs from ..._POSIX_EXTENDED in that RE_CONTEXT_INVALID_OPS
   replaces RE_CONTEXT_INDEP_OPS and RE_NO_BK_REFS is added.  */
#define RE_SYNTAX_POSIX_MINIMAL_EXTENDED                                \
  (_RE_SYNTAX_POSIX_COMMON  | RE_CONTEXT_INDEP_ANCHORS                  \
   | RE_CONTEXT_INVALID_OPS | RE_NO_BK_BRACES                           \
   | RE_NO_BK_PARENS        | RE_NO_BK_REFS                             \
   | RE_NO_BK_VBAR          | RE_UNMATCHED_RIGHT_PAREN_ORD)
```

2.3 Collating Elements vs. Characters

POSIX generalizes the notion of a character to that of a collating element. It defines a *collating element* to be "a sequence of one or more bytes defined in the current collating sequence as a unit of collation."

This generalizes the notion of a character in two ways. First, a single character can map into two or more collating elements. For example, the German 'ß' collates as the collating element '`s`' followed by another collating element '`s`'. Second, two or more characters can map into one collating element. For example, the Spanish '`ll`' collates after '`l`' and before '`m`'.

Since POSIX's "collating element" preserves the essential idea of a "character," we use the latter, more familiar, term in this document.

2.4 The Backslash Character

The ‘`\`’ character has one of four different meanings, depending on the context in which you use it and what syntax bits are set (see Section 2.1 [Syntax Bits], page 2). It can: 1) stand for itself, 2) quote the next character, 3) introduce an operator, or 4) do nothing.

1. It stands for itself inside a list (see Section 3.6 [List Operators], page 13) if the syntax bit `RE_BACKSLASH_ESCAPE_IN_LISTS` is not set. For example, ‘`[\]`’ would match ‘`\`’.
2. It quotes (makes ordinary, if it's special) the next character when you use it either:
   - outside a list,[1] or
   - inside a list and the syntax bit `RE_BACKSLASH_ESCAPE_IN_LISTS` is set.
3. It introduces an operator when followed by certain ordinary characters—sometimes only when certain syntax bits are set. See the cases `RE_BK_PLUS_QM`, `RE_NO_BK_BRACES`, `RE_NO_BK_VAR`, `RE_NO_BK_PARENS`, `RE_NO_BK_REF` in Section 2.1 [Syntax Bits], page 2. Also:
   - ‘`\b`’ represents the match-word-boundary operator (see Section 4.1.2 [Match-word-boundary Operator], page 20).
   - ‘`\B`’ represents the match-within-word operator (see Section 4.1.3 [Match-within-word Operator], page 20).
   - ‘`\<`’ represents the match-beginning-of-word operator (see Section 4.1.4 [Match-beginning-of-word Operator], page 21).
   - ‘`\>`’ represents the match-end-of-word operator (see Section 4.1.5 [Match-end-of-word Operator], page 21).
   - ‘`\w`’ represents the match-word-constituent operator (see Section 4.1.6 [Match-word-constituent Operator], page 21).
   - ‘`\W`’ represents the match-non-word-constituent operator (see Section 4.1.7 [Match-non-word-constituent Operator], page 21).
   - ‘`` \` ``’ represents the match-beginning-of-buffer operator and ‘`\'`’ represents the match-end-of-buffer operator (see Section 4.2 [Buffer Operators], page 21).

[1] Sometimes you don't have to explicitly quote special characters to make them ordinary. For instance, most characters lose any special meaning inside a list (see Section 3.6 [List Operators], page 13). In addition, if the syntax bits `RE_CONTEXT_INVALID_OPS` and `RE_CONTEXT_INDEP_OPS` aren't set, then (for historical reasons) the matcher considers special characters ordinary if they are in contexts where the operations they represent make no sense; for example, then the match-zero-or-more operator (represented by ‘`*`’) matches itself in the regular expression ‘`*foo`’ because there is no preceding expression on which it can operate. It is poor practice, however, to depend on this behavior; if you want a special character to be ordinary outside a list, it's better to always quote it, regardless.

- If Regex was compiled with the C preprocessor symbol `emacs` defined, then '`\s`*class*' represents the match-syntactic-class operator and '`\S`*class*' represents the match-not-syntactic-class operator (see Section 5.1 [Syntactic Class Operators], page 22).

4. In all other cases, Regex ignores '`\`'. For example, '`\n`' matches '`n`'.

3 Common Operators

You compose regular expressions from operators. In the following sections, we describe the regular expression operators specified by POSIX; GNU also uses these. Most operators have more than one representation as characters. See Chapter 2 [Regular Expression Syntax], page 2, for what characters represent what operators under what circumstances.

For most operators that can be represented in two ways, one representation is a single character and the other is that character preceded by '`\`'. For example, either '`(`' or '`\(`' represents the open-group operator. Which one does depends on the setting of a syntax bit, in this case `RE_NO_BK_PARENS`. Why is this so? Historical reasons dictate some of the varying representations, while POSIX dictates others.

Finally, almost all characters lose any special meaning inside a list (see Section 3.6 [List Operators], page 13).

3.1 The Match-self Operator (*ordinary character*)

This operator matches the character itself. All ordinary characters (see Chapter 2 [Regular Expression Syntax], page 2) represent this operator. For example, '`f`' is always an ordinary character, so the regular expression '`f`' matches only the string '`f`'. In particular, it does *not* match the string '`ff`'.

3.2 The Match-any-character Operator (`.`)

This operator matches any single printing or nonprinting character except it won't match a:

newline — if the syntax bit `RE_DOT_NEWLINE` isn't set.

null — if the syntax bit `RE_DOT_NOT_NULL` is set.

The '`.`' (period) character represents this operator. For example, '`a.b`' matches any three-character string beginning with '`a`' and ending with '`b`'.

3.3 The Concatenation Operator

This operator concatenates two regular expressions *a* and *b*. No character represents this operator; you simply put *b* after *a*. The result is a regular expression that will match a string if *a* matches its first part and *b* matches the rest. For example, ‘`xy`’ (two match-self operators) matches ‘`xy`’.

3.4 Repetition Operators

Repetition operators repeat the preceding regular expression a specified number of times.

3.4.1 The Match-zero-or-more Operator (*)

This operator repeats the smallest possible preceding regular expression as many times as necessary (including zero) to match the pattern. ‘`*`’ represents this operator. For example, ‘`o*`’ matches any string made up of zero or more ‘`o`’s. Since this operator operates on the smallest preceding regular expression, ‘`fo*`’ has a repeating ‘`o`’, not a repeating ‘`fo`’. So, ‘`fo*`’ matches ‘`f`’, ‘`fo`’, ‘`foo`’, and so on.

Since the match-zero-or-more operator is a suffix operator, it may be useless as such when no regular expression precedes it. This is the case when it:

- is first in a regular expression, or
- follows a match-beginning-of-line, open-group, or alternation operator.

Three different things can happen in these cases:

1. If the syntax bit `RE_CONTEXT_INVALID_OPS` is set, then the regular expression is invalid.
2. If `RE_CONTEXT_INVALID_OPS` isn't set, but `RE_CONTEXT_INDEP_OPS` is, then ‘`*`’ represents the match-zero-or-more operator (which then operates on the empty string).
3. Otherwise, ‘`*`’ is ordinary.

The matcher processes a match-zero-or-more operator by first matching as many repetitions of the smallest preceding regular expression as it can. Then it continues to match the rest of the pattern.

If it can't match the rest of the pattern, it backtracks (as many times as necessary), each time discarding one of the matches until it can either match the entire pattern or be certain that it cannot get a match. For example, when matching 'ca*ar' against 'caaar', the matcher first matches all three 'a's of the string with the 'a*' of the regular expression. However, it cannot then match the final 'ar' of the regular expression against the final 'r' of the string. So it backtracks, discarding the match of the last 'a' in the string. It can then match the remaining 'ar'.

3.4.2 The Match-one-or-more Operator (+ or \+)

If the syntax bit RE_LIMITED_OPS is set, then Regex doesn't recognize this operator. Otherwise, if the syntax bit RE_BK_PLUS_QM isn't set, then '+' represents this operator; if it is, then '\+' does.

This operator is similar to the match-zero-or-more operator except that it repeats the preceding regular expression at least once; see Section 3.4.1 [Match-zero-or-more Operator], page 10, for what it operates on, how some syntax bits affect it, and how Regex backtracks to match it.

For example, supposing that '+' represents the match-one-or-more operator; then 'ca+r' matches, e.g., 'car' and 'caaaar', but not 'cr'.

3.4.3 The Match-zero-or-one Operator (? or \?)

If the syntax bit RE_LIMITED_OPS is set, then Regex doesn't recognize this operator. Otherwise, if the syntax bit RE_BK_PLUS_QM isn't set, then '?' represents this operator; if it is, then '\?' does.

This operator is similar to the match-zero-or-more operator except that it repeats the preceding regular expression once or not at all; see Section 3.4.1 [Match-zero-or-more Operator], page 10, to see what it operates on, how some syntax bits affect it, and how Regex backtracks to match it.

For example, supposing that '?' represents the match-zero-or-one operator; then 'ca?r' matches both 'car' and 'cr', but nothing else.

3.4.4 Interval Operators ({ ... } or \{ ... \})

If the syntax bit `RE_INTERVALS` is set, then Regex recognizes *interval expressions*. They repeat the smallest possible preceding regular expression a specified number of times.

If the syntax bit `RE_NO_BK_BRACES` is set, '`{`' represents the *open-interval operator* and '`}`' represents the *close-interval operator* ; otherwise, '`\{`' and '`\}`' do.

Specifically, supposing that '`{`' and '`}`' represent the open-interval and close-interval operators; then:

`{`*count*`}` matches exactly *count* occurrences of the preceding regular expression.

`{`*min*`,}` matches *min* or more occurrences of the preceding regular expression.

`{`*min, max*`}`
: matches at least *min* but no more than *max* occurrences of the preceding regular expression.

The interval expression (but not necessarily the regular expression that contains it) is invalid if:

- *min* is greater than *max*, or
- any of *count*, *min*, or *max* are outside the range zero to `RE_DUP_MAX` (which symbol '`regex.h`' defines).

If the interval expression is invalid and the syntax bit `RE_NO_BK_BRACES` is set, then Regex considers all the characters in the would-be interval to be ordinary. If that bit isn't set, then the regular expression is invalid.

If the interval expression is valid but there is no preceding regular expression on which to operate, then if the syntax bit `RE_CONTEXT_INVALID_OPS` is set, the regular expression is invalid. If that bit isn't set, then Regex considers all the characters—other than backslashes, which it ignores—in the would-be interval to be ordinary.

3.5 The Alternation Operator (| or \|)

If the syntax bit RE_LIMITED_OPS is set, then Regex doesn't recognize this operator. Otherwise, if the syntax bit RE_NO_BK_VBAR is set, then '|' represents this operator; otherwise, '\|' does.

Alternatives match one of a choice of regular expressions: if you put the character(s) representing the alternation operator between any two regular expressions *a* and *b*, the result matches the union of the strings that *a* and *b* match. For example, supposing that '|' is the alternation operator, then 'foo|bar|quux' would match any of 'foo', 'bar' or 'quux'.

The alternation operator operates on the *largest* possible surrounding regular expressions. (Put another way, it has the lowest precedence of any regular expression operator.) Thus, the only way you can delimit its arguments is to use grouping. For example, if '(' and ')' are the open and close-group operators, then 'fo(o|b)ar' would match either 'fooar' or 'fobar'. ('foo|bar' would match 'foo' or 'bar'.)

The matcher usually tries all combinations of alternatives so as to match the longest possible string. For example, when matching '(fooq|foo)*(qbarquux|bar)' against 'fooqbarquux', it cannot take, say, the first ("depth-first") combination it could match, since then it would be content to match just 'fooqbar'.

3.6 List Operators ([...] and [^ ...])

Lists, also called *bracket expressions*, are a set of one or more items. An *item* is a character, a character class expression, or a range expression. The syntax bits affect which kinds of items you can put in a list. We explain the last two items in subsections below. Empty lists are invalid.

A *matching list* matches a single character represented by one of the list items. You form a matching list by enclosing one or more items within an *open-matching-list operator* (represented by '[') and a *close-list operator* (represented by ']').

For example, '[ab]' matches either 'a' or 'b'. '[ad]*' matches the empty string and any string composed of just 'a's and 'd's in any order. Regex considers invalid a regular expression with a '[' but no matching ']'.

Nonmatching lists are similar to matching lists except that they match a single character *not* represented by one of the list items. You use an *open-nonmatching-list operator* (represented by '[^'[1]) instead of an open-matching-list operator to start a nonmatching list.

For example, '[^ab]' matches any character except 'a' or 'b'.

If the posix_newline field in the pattern buffer (see Section 7.1.1 [GNU Pattern Buffers], page 24 is set, then nonmatching lists do not match a newline.

Most characters lose any special meaning inside a list. The special characters inside a list follow.

']' ends the list if it's not the first list item. So, if you want to make the ']' character a list item, you must put it first.

'\' quotes the next character if the syntax bit RE_BACKSLASH_ESCAPE_IN_LISTS is set.

'[:' represents the open-character-class operator (see Section 3.6.1 [Character Class Operators], page 14) if the syntax bit RE_CHAR_CLASSES is set and what follows is a valid character class expression.

':]' represents the close-character-class operator if the syntax bit RE_CHAR_CLASSES is set and what precedes it is an open-character-class operator followed by a valid character class name.

'-' represents the range operator (see Section 3.6.2 [Range Operator], page 15) if it's not first or last in a list or the ending point of a range.

All other characters are ordinary. For example, '[.*]' matches '.' and '*'.

3.6.1 Character Class Operators ([: ... :])

If the syntax bit RE_CHARACTER_CLASSES is set, then Regex recognizes character class expressions inside lists. A *character class expression* matches one character from a given class. You form a character class expression by putting a character class name between an *open-character-class operator* (represented by '[:') and a *close-character-class operator* (represented by ':]'). The character class names and their meanings are:

alnum letters and digits

[1] Regex therefore doesn't consider the '^' to be the first character in the list. If you put a '^' character first in (what you think is) a matching list, you'll turn it into a nonmatching list.

`alpha`	letters
`blank`	system-dependent; for GNU, a space or tab
`cntrl`	control characters (in the ASCII encoding, code 0177 and codes less than 040)
`digit`	digits
`graph`	same as `print` except omits space
`lower`	lowercase letters
`print`	printable characters (in the ASCII encoding, space tilde—codes 040 through 0176)
`punct`	neither control nor alphanumeric characters
`space`	space, carriage return, newline, vertical tab, and form feed
`upper`	uppercase letters
`xdigit`	hexadecimal digits: `0`–`9`, `a`–`f`, `A`–`F`

These correspond to the definitions in the C library's '`<ctype.h>`' facility. For example, '`[:alpha:]`' corresponds to the standard facility `isalpha`. Regex recognizes character class expressions only inside of lists; so '`[[:alpha:]]`' matches any letter, but '`[:alpha:]`' outside of a bracket expression and not followed by a repetition operator matches just itself.

3.6.2 The Range Operator (-)

Regex recognizes *range expressions* inside a list. They represent those characters that fall between two elements in the current collating sequence. You form a range expression by putting a *range operator* between two characters.[2] '`-`' represents the range operator. For example, '`a-f`' within a list represents all the characters from '`a`' through '`f`' inclusively.

If the syntax bit `RE_NO_EMPTY_RANGES` is set, then if the range's ending point collates less than its starting point, the range (and the regular expression containing it) is invalid. For example, the regular expression '`[z-a]`' would be invalid. If this bit isn't set, then Regex considers such a range to be empty.

Since '`-`' represents the range operator, if you want to make a '`-`' character itself a list item, you must do one of the following:

- Put the '`-`' either first or last in the list.

[2] You can't use a character class for the starting or ending point of a range, since a character class is not a single character.

- Include a range whose starting point collates strictly lower than '-' and whose ending point collates equal or higher. Unless a range is the first item in a list, a '-' can't be its starting point, but *can* be its ending point. That is because Regex considers '-' to be the range operator unless it is preceded by another '-'. For example, in the ASCII encoding, ')', '*', '+', ',', '-', '.', and '/' are contiguous characters in the collating sequence. You might think that '[)-+--/]' has two ranges: ')-+' and '--/'. Rather, it has the ranges ')-+' and '+--', plus the character '/', so it matches, e.g., ',', not '.'.
- Put a range whose starting point is '-' first in the list.

For example, '[-a-z]' matches a lowercase letter or a hyphen (in English, in ASCII).

3.7 Grouping Operators ((...) or \(... \))

A *group*, also known as a *subexpression*, consists of an *open-group operator*, any number of other operators, and a *close-group operator*. Regex treats this sequence as a unit, just as mathematics and programming languages treat a parenthesized expression as a unit.

Therefore, using *groups*, you can:

- delimit the argument(s) to an alternation operator (see Section 3.5 [Alternation Operator], page 13) or a repetition operator (see Section 3.4 [Repetition Operators], page 10).
- keep track of the indices of the substring that matched a given group. See Section 7.1.8 [Using Registers], page 32, for a precise explanation. This lets you:
  - use the back-reference operator (see Section 3.8 [Back-reference Operator], page 17).
  - use registers (see Section 7.1.8 [Using Registers], page 32).

If the syntax bit `RE_NO_BK_PARENS` is set, then '(' represents the open-group operator and ')' represents the close-group operator; otherwise, '\(' and '\)' do.

If the syntax bit `RE_UNMATCHED_RIGHT_PAREN_ORD` is set and a close-group operator has no matching open-group operator, then Regex considers it to match ')'.

3.8 The Back-reference Operator (*digit*)

If the syntax bit `RE_NO_BK_REF` isn't set, then Regex recognizes back references. A back reference matches a specified preceding group. The back reference operator is represented by '`\`*digit*' anywhere after the end of a regular expression's *digit*-th group (see Section 3.7 [Grouping Operators], page 16).

digit must be between '`1`' and '`9`'. The matcher assigns numbers 1 through 9 to the first nine groups it encounters. By using one of '`\1`' through '`\9`' after the corresponding group's close-group operator, you can match a substring identical to the one that the group does.

Back references match according to the following (in all examples below, '`(`' represents the open-group, '`)`' the close-group, '`{`' the open-interval and '`}`' the close-interval operator):

- If the group matches a substring, the back reference matches an identical substring. For example, '`(a)\1`' matches '`aa`' and '`(bana)na\1bo\1`' matches '`bananabanabobana`'. Likewise, '`(.*)\1`' matches any (newline-free if the syntax bit `RE_DOT_NEWLINE` isn't set) string that is composed of two identical halves; the '`(.*)`' matches the first half and the '`\1`' matches the second half.
- If the group matches more than once (as it might if followed by, e.g., a repetition operator), then the back reference matches the substring the group *last* matched. For example, '`((a*)b)*\1\2`' matches '`aabababa`'; first group 1 (the outer one) matches '`aab`' and group 2 (the inner one) matches '`aa`'. Then group 1 matches '`ab`' and group 2 matches '`a`'. So, '`\1`' matches '`ab`' and '`\2`' matches '`a`'.
- If the group doesn't participate in a match, i.e., it is part of an alternative not taken or a repetition operator allows zero repetitions of it, then the back reference makes the whole match fail. For example, '`(one()|two())-and-(three\2|four\3)`' matches '`one-and-three`' and '`two-and-four`', but not '`one-and-four`' or '`two-and-three`'. For example, if the pattern matches '`one-and-`', then its group 2 matches the empty string and its group 3 doesn't participate in the match. So, if it then matches '`four`', then when it tries to back reference group 3—which it will attempt to do because '`\3`' follows the '`four`'—the match will fail because group 3 didn't participate in the match.

You can use a back reference as an argument to a repetition operator. For example, '`(a(b))\2*`' matches '`a`' followed by two or more '`b`'s. Similarly, '`(a(b))\2{3}`' matches '`abbbb`'.

If there is no preceding *digit*-th subexpression, the regular expression is invalid.

3.9 Anchoring Operators

These operators can constrain a pattern to match only at the beginning or end of the entire string or at the beginning or end of a line.

3.9.1 The Match-beginning-of-line Operator (^)

This operator can match the empty string either at the beginning of the string or after a newline character. Thus, it is said to *anchor* the pattern to the beginning of a line.

In the cases following, '`^`' represents this operator. (Otherwise, '`^`' is ordinary.)

- It (the '`^`') is first in the pattern, as in '`^foo`'.
- The syntax bit `RE_CONTEXT_INDEP_ANCHORS` is set, and it is outside a bracket expression.
- It follows an open-group or alternation operator, as in '`a\(^b\)`' and '`a\|^b`'. See Section 3.7 [Grouping Operators], page 16, and Section 3.5 [Alternation Operator], page 13.

These rules imply that some valid patterns containing '`^`' cannot be matched; for example, '`foo^bar`' if `RE_CONTEXT_INDEP_ANCHORS` is set.

If the `not_bol` field is set in the pattern buffer (see Section 7.1.1 [GNU Pattern Buffers], page 24), then '`^`' fails to match at the beginning of the string. See Section 7.2.3 [POSIX Matching], page 37, for when you might find this useful.

If the `newline_anchor` field is set in the pattern buffer, then '`^`' fails to match after a newline. This is useful when you do not regard the string to be matched as broken into lines.

3.9.2 The Match-end-of-line Operator ($)

This operator can match the empty string either at the end of the string or before a newline character in the string. Thus, it is said to *anchor* the pattern to the end of a line.

It is always represented by '`$`'. For example, '`foo$`' usually matches, e.g., '`foo`' and, e.g., the first three characters of '`foo\nbar`'.

Its interaction with the syntax bits and pattern buffer fields is exactly the dual of '^''s; see the previous section. (That is, "beginning" becomes "end", "next" becomes "previous", and "after" becomes "before".)

4 GNU Operators

Following are operators that GNU defines (and POSIX doesn't).

4.1 Word Operators

The operators in this section require Regex to recognize parts of words. Regex uses a syntax table to determine whether or not a character is part of a word, i.e., whether or not it is *word-constituent*.

4.1.1 Non-Emacs Syntax Tables

A *syntax table* is an array indexed by the characters in your character set. In the ASCII encoding, therefore, a syntax table has 256 elements. Regex always uses a `char *` variable `re_syntax_table` as its syntax table. In some cases, it initializes this variable and in others it expects you to initialize it.

- If Regex is compiled with the preprocessor symbols `emacs` and `SYNTAX_TABLE` both undefined, then Regex allocates `re_syntax_table` and initializes an element *i* either to `Sword` (which it defines) if *i* is a letter, number, or '`_`', or to zero if it's not.
- If Regex is compiled with `emacs` undefined but `SYNTAX_TABLE` defined, then Regex expects you to define a `char *` variable `re_syntax_table` to be a valid syntax table.
- See Section 5.1.1 [Emacs Syntax Tables], page 22, for what happens when Regex is compiled with the preprocessor symbol `emacs` defined.

4.1.2 The Match-word-boundary Operator (`\b`)

This operator (represented by '`\b`') matches the empty string at either the beginning or the end of a word. For example, '`\brat\b`' matches the separate word '`rat`'.

4.1.3 The Match-within-word Operator (`\B`)

This operator (represented by '`\B`') matches the empty string within a word. For example, '`c\Brat\Be`' matches '`crate`', but '`dirty \Brat`' doesn't match '`dirty rat`'.

4.1.4 The Match-beginning-of-word Operator (\<)

This operator (represented by ‘\<’) matches the empty string at the beginning of a word.

4.1.5 The Match-end-of-word Operator (\>)

This operator (represented by ‘\>’) matches the empty string at the end of a word.

4.1.6 The Match-word-constituent Operator (\w)

This operator (represented by ‘\w’) matches any word-constituent character.

4.1.7 The Match-non-word-constituent Operator (\W)

This operator (represented by ‘\W’) matches any character that is not word-constituent.

4.2 Buffer Operators

Following are operators which work on buffers. In Emacs, a *buffer* is, naturally, an Emacs buffer. For other programs, Regex considers the entire string to be matched as the buffer.

4.2.1 The Match-beginning-of-buffer Operator (\`)

This operator (represented by ‘\`’) matches the empty string at the beginning of the buffer.

4.2.2 The Match-end-of-buffer Operator (\')

This operator (represented by ‘\'’) matches the empty string at the end of the buffer.

5 GNU Emacs Operators

Following are operators that GNU defines (and POSIX doesn't) that you can use only when Regex is compiled with the preprocessor symbol `emacs` defined.

5.1 Syntactic Class Operators

The operators in this section require Regex to recognize the syntactic classes of characters. Regex uses a syntax table to determine this.

5.1.1 Emacs Syntax Tables

A *syntax table* is an array indexed by the characters in your character set. In the ASCII encoding, therefore, a syntax table has 256 elements.

If Regex is compiled with the preprocessor symbol `emacs` defined, then Regex expects you to define and initialize the variable `re_syntax_table` to be an Emacs syntax table. Emacs' syntax tables are more complicated than Regex's own (see Section 4.1.1 [Non-Emacs Syntax Tables], page 20). See section "Syntax" in *The GNU Emacs User's Manual*, for a description of Emacs' syntax tables.

5.1.2 The Match-syntactic-class Operator (`\s`*class*)

This operator matches any character whose syntactic class is represented by a specified character. '`\s`*class*' represents this operator where *class* is the character representing the syntactic class you want. For example, '`w`' represents the syntactic class of word-constituent characters, so '`\sw`' matches any word-constituent character.

5.1.3 The Match-not-syntactic-class Operator (`\S`*class*)

This operator is similar to the match-syntactic-class operator except that it matches any character whose syntactic class is *not* represented by the specified character. '`\S`*class*' represents this operator. For example, '`w`' represents the syntactic class of word-constituent characters, so '`\Sw`' matches any character that is not word-constituent.

6 What Gets Matched?

Regex usually matches strings according to the "leftmost longest" rule; that is, it chooses the longest of the leftmost matches. This does not mean that for a regular expression containing subexpressions that it simply chooses the longest match for each subexpression, left to right; the overall match must also be the longest possible one.

For example, '`(ac*)(c*d[ac]*)\1`' matches '`acdacaaa`', not '`acdac`', as it would if it were to choose the longest match for the first subexpression.

7 Programming with Regex

Here we describe how you use the Regex data structures and functions in C programs. Regex has three interfaces: one designed for GNU, one compatible with POSIX and one compatible with Berkeley UNIX.

7.1 GNU Regex Functions

If you're writing code that doesn't need to be compatible with either POSIX or Berkeley UNIX, you can use these functions. They provide more options than the other interfaces.

7.1.1 GNU Pattern Buffers

To compile, match, or search for a given regular expression, you must supply a pattern buffer. A *pattern buffer* holds one compiled regular expression.[1]

You can have several different pattern buffers simultaneously, each holding a compiled pattern for a different regular expression.

'`regex.h`' defines the pattern buffer `struct` as follows:

```
        /* Space that holds the compiled pattern.  It is declared as
          `unsigned char *' because its elements are
           sometimes used as array indexes.  */
  unsigned char *buffer;

        /* Number of bytes to which `buffer' points.  */
  unsigned long allocated;

        /* Number of bytes actually used in `buffer'.  */
  unsigned long used;

        /* Syntax setting with which the pattern was compiled.  */
  reg_syntax_t syntax;
```

[1] Regular expressions are also referred to as "patterns," hence the name "pattern buffer."

```
        /* Pointer to a fastmap, if any, otherwise zero.  re_search uses
           the fastmap, if there is one, to skip over impossible
           starting points for matches.  */
  char *fastmap;

        /* Either a translate table to apply to all characters before
           comparing them, or zero for no translation.  The translation
           is applied to a pattern when it is compiled and to a string
           when it is matched.  */
  char *translate;

        /* Number of subexpressions found by the compiler.  */
  size_t re_nsub;

        /* Zero if this pattern cannot match the empty string, one else.
           Well, in truth it's used only in `re_search_2', to see
           whether or not we should use the fastmap, so we don't set
           this absolutely perfectly; see `re_compile_fastmap' (the
           `duplicate' case).  */
  unsigned can_be_null : 1;

        /* If REGS_UNALLOCATED, allocate space in the `regs' structure
             for `max (RE_NREGS, re_nsub + 1)' groups.
           If REGS_REALLOCATE, reallocate space if necessary.
           If REGS_FIXED, use what's there.  */
#define REGS_UNALLOCATED 0
#define REGS_REALLOCATE 1
#define REGS_FIXED 2
  unsigned regs_allocated : 2;

        /* Set to zero when `regex_compile' compiles a pattern; set to one
           by `re_compile_fastmap' if it updates the fastmap.  */
  unsigned fastmap_accurate : 1;

        /* If set, `re_match_2' does not return information about
           subexpressions.  */
  unsigned no_sub : 1;

        /* If set, a beginning-of-line anchor doesn't match at the
           beginning of the string.  */
  unsigned not_bol : 1;

        /* Similarly for an end-of-line anchor.  */
  unsigned not_eol : 1;

        /* If true, an anchor at a newline matches.  */
  unsigned newline_anchor : 1;
```

7.1.2 GNU Regular Expression Compiling

In GNU, you can both match and search for a given regular expression. To do either, you must first compile it in a pattern buffer (see Section 7.1.1 [GNU Pattern Buffers], page 24).

Regular expressions match according to the syntax with which they were compiled; with GNU, you indicate what syntax you want by setting the variable `re_syntax_options` (declared in '`regex.h`' and defined in '`regex.c`') before calling the compiling function, `re_compile_pattern` (see below). See Section 2.1 [Syntax Bits], page 2, and Section 2.2 [Predefined Syntaxes], page 5.

You can change the value of `re_syntax_options` at any time. Usually, however, you set its value once and then never change it.

`re_compile_pattern` takes a pattern buffer as an argument. You must initialize the following fields:

`translate` initialization

`translate`

: Initialize this to point to a translate table if you want one, or to zero if you don't. We explain translate tables in Section 7.1.7 [GNU Translate Tables], page 31.

`fastmap`
: Initialize this to nonzero if you want a fastmap, or to zero if you don't.

`buffer`
`allocated`

: If you want `re_compile_pattern` to allocate memory for the compiled pattern, set both of these to zero. If you have an existing block of memory (allocated with `malloc`) you want Regex to use, set `buffer` to its address and `allocated` to its size (in bytes).

: `re_compile_pattern` uses `realloc` to extend the space for the compiled pattern as necessary.

To compile a pattern buffer, use:

```
char *
re_compile_pattern (const char *regex, const int regex_size,
                    struct re_pattern_buffer *pattern_buffer)
```

regex is the regular expression's address, *regex_size* is its length, and *pattern_buffer* is the pattern buffer's address.

If `re_compile_pattern` successfully compiles the regular expression, it returns zero and sets `*`*pattern_buffer* to the compiled pattern. It sets the pattern buffer's fields as follows:

`buffer`
: to the compiled pattern.

`used`
: to the number of bytes the compiled pattern in `buffer` occupies.

`syntax`
: to the current value of `re_syntax_options`.

`re_nsub`
: to the number of subexpressions in *regex*.

`fastmap_accurate`
: to zero on the theory that the pattern you're compiling is different than the one previously compiled into `buffer`; in that case (since you can't make a fastmap without a compiled pattern), `fastmap` would either contain an incompatible fastmap, or nothing at all.

If `re_compile_pattern` can't compile *regex*, it returns an error string corresponding to one of the errors listed in Section 7.2.2 [POSIX Regular Expression Compiling], page 35.

7.1.3 GNU Matching

Matching the GNU way means trying to match as much of a string as possible starting at a position within it you specify. Once you've compiled a pattern into a pattern buffer (see Section 7.1.2 [GNU Regular Expression Compiling], page 26), you can ask the matcher to match that pattern against a string using:

```
int
re_match (struct re_pattern_buffer *pattern_buffer,
          const char *string, const int size,
          const int start, struct re_registers *regs)
```

pattern_buffer is the address of a pattern buffer containing a compiled pattern. *string* is the string you want to match; it can contain newline and null characters. *size* is the length of that string. *start* is the string index at which you want to begin matching; the first character of *string* is at index zero. See Section 7.1.8 [Using Registers], page 32, for a explanation of *regs*; you can safely pass zero.

`re_match` matches the regular expression in *pattern_buffer* against the string *string* according to the syntax in *pattern_buffers*'s `syntax` field. (See Section 7.1.2 [GNU Regular Expression Compiling], page 26, for how to set it.) The function returns −1 if the compiled pattern does not match any part of *string* and −2 if an internal error happens; otherwise, it returns how many (possibly zero) characters of *string* the pattern matched.

An example: suppose *pattern_buffer* points to a pattern buffer containing the compiled pattern for '`a*`', and *string* points to '`aaaaab`' (whereupon *size* should be 6). Then if *start* is 2, `re_match` returns 3, i.e., '`a*`' would have matched the last three '`a`'s in *string*. If *start* is 0, `re_match` returns 5, i.e., '`a*`' would have matched all the '`a`'s in *string*. If *start* is either 5 or 6, it returns zero.

If *start* is not between zero and *size*, then `re_match` returns −1.

7.1.4 GNU Searching

Searching means trying to match starting at successive positions within a string. The function `re_search` does this.

Before calling `re_search`, you must compile your regular expression. See Section 7.1.2 [GNU Regular Expression Compiling], page 26.

Here is the function declaration:

```
int
re_search (struct re_pattern_buffer *pattern_buffer,
           const char *string, const int size,
           const int start, const int range,
           struct re_registers *regs)
```

whose arguments are the same as those to `re_match` (see Section 7.1.3 [GNU Matching], page 27) except that the two arguments *start* and *range* replace `re_match`'s argument *start*.

If *range* is positive, then `re_search` attempts a match starting first at index *start*, then at *start*+1 if that fails, and so on, up to *start*+*range*; if *range* is negative, then it attempts a match starting first at index *start*, then at *start* − 1 if that fails, and so on.

If *start* is not between zero and *size*, then `re_search` returns −1. When *range* is positive, `re_search` adjusts *range* so that *start*+*range* − 1 is between zero and *size*, if necessary; that way it won't search outside of *string*. Similarly, when *range* is negative, `re_search` adjusts *range* so that *start*+*range*+1 is between zero and *size*, if necessary.

If the `fastmap` field of *pattern_buffer* is zero, `re_search` matches starting at consecutive positions; otherwise, it uses `fastmap` to make the search more efficient. See Section 7.1.6 [Searching with Fastmaps], page 30.

If no match is found, `re_search` returns −1. If a match is found, it returns the index where the match began. If an internal error happens, it returns −2.

7.1.5 Matching and Searching with Split Data

Using the functions `re_match_2` and `re_search_2`, you can match or search in data that is divided into two strings.

The function:

```
int
re_match_2 (struct re_pattern_buffer *buffer,
            const char *string1, const int size1,
            const char *string2, const int size2,
            const int start,
            struct re_registers *regs,
            const int stop)
```

is similar to `re_match` (see Section 7.1.3 [GNU Matching], page 27) except that you pass *two* data strings and sizes, and an index *stop* beyond which you don't want the matcher to try matching. As with `re_match`, if it succeeds, `re_match_2` returns how many characters of *string* it matched. Regard *string1* and *string2* as concatenated when you set the arguments *start* and *stop* and use the contents of *regs*; `re_match_2` never returns a value larger than *size1*+*size2*.

The function:

```
int
```

```
re_search_2 (struct re_pattern_buffer *buffer,
             const char *string1, const int size1,
             const char *string2, const int size2,
             const int start, const int range,
             struct re_registers *regs,
             const int stop)
```

is similarly related to `re_search`.

7.1.6 Searching with Fastmaps

If you're searching through a long string, you should use a fastmap. Without one, the searcher tries to match at consecutive positions in the string. Generally, most of the characters in the string could not start a match. It takes much longer to try matching at a given position in the string than it does to check in a table whether or not the character at that position could start a match. A *fastmap* is such a table.

More specifically, a fastmap is an array indexed by the characters in your character set. Under the ASCII encoding, therefore, a fastmap has 256 elements. If you want the searcher to use a fastmap with a given pattern buffer, you must allocate the array and assign the array's address to the pattern buffer's `fastmap` field. You either can compile the fastmap yourself or have `re_search` do it for you; when `fastmap` is nonzero, it automatically compiles a fastmap the first time you search using a particular compiled pattern.

To compile a fastmap yourself, use:

```
int
re_compile_fastmap (struct re_pattern_buffer *pattern_buffer)
```

pattern_buffer is the address of a pattern buffer. If the character *c* could start a match for the pattern, `re_compile_fastmap` makes *pattern_buffer*`->fastmap[`*c*`]` nonzero. It returns 0 if it can compile a fastmap and −2 if there is an internal error. For example, if '`|`' is the alternation operator and *pattern_buffer* holds the compiled pattern for '`a|b`', then `re_compile_fastmap` sets `fastmap['a']` and `fastmap['b']` (and no others).

`re_search` uses a fastmap as it moves along in the string: it checks the string's characters until it finds one that's in the fastmap. Then it tries matching at that character. If the match fails, it repeats the process. So, by using a fastmap, `re_search` doesn't waste time trying to match at positions in the string that couldn't start a match.

If you don't want `re_search` to use a fastmap, store zero in the `fastmap` field of the pattern buffer before calling `re_search`.

Once you've initialized a pattern buffer's `fastmap` field, you need never do so again—even if you compile a new pattern in it—provided the way the field is set still reflects whether or not you want a fastmap. `re_search` will still either do nothing if `fastmap` is null or, if it isn't, compile a new fastmap for the new pattern.

7.1.7 GNU Translate Tables

If you set the `translate` field of a pattern buffer to a translate table, then the GNU Regex functions to which you've passed that pattern buffer use it to apply a simple transformation to all the regular expression and string characters at which they look.

A *translate table* is an array indexed by the characters in your character set. Under the ASCII encoding, therefore, a translate table has 256 elements. The array's elements are also characters in your character set. When the Regex functions see a character *c*, they use `translate[c]` in its place, with one exception: the character after a '`\`' is not translated. (This ensures that, the operators, e.g., '`\B`' and '`\b`', are always distinguishable.)

For example, a table that maps all lowercase letters to the corresponding uppercase ones would cause the matcher to ignore differences in case.[2] Such a table would map all characters except lowercase letters to themselves, and lowercase letters to the corresponding uppercase ones. Under the ASCII encoding, here's how you could initialize such a table (we'll call it `case_fold`):

```
for (i = 0; i < 256; i++)
  case_fold[i] = i;
for (i = 'a'; i <= 'z'; i++)
  case_fold[i] = i - ('a' - 'A');
```

You tell Regex to use a translate table on a given pattern buffer by assigning that table's address to the `translate` field of that buffer. If you don't want Regex to do any translation, put zero into

[2] A table that maps all uppercase letters to the corresponding lowercase ones would work just as well for this purpose.

this field. You'll get weird results if you change the table's contents anytime between compiling the pattern buffer, compiling its fastmap, and matching or searching with the pattern buffer.

7.1.8 Using Registers

A group in a regular expression can match a (posssibly empty) substring of the string that regular expression as a whole matched. The matcher remembers the beginning and end of the substring matched by each group.

To find out what they matched, pass a nonzero *regs* argument to a GNU matching or searching function (see Section 7.1.3 [GNU Matching], page 27 and Section 7.1.4 [GNU Searching], page 28), i.e., the address of a structure of this type, as defined in '`regex.h`':

```
struct re_registers
{
  unsigned num_regs;
  regoff_t *start;
  regoff_t *end;
};
```

Except for (possibly) the *num_regs*'th element (see below), the ith element of the `start` and `end` arrays records information about the ith group in the pattern. (They're declared as C pointers, but this is only because not all C compilers accept zero-length arrays; conceptually, it is simplest to think of them as arrays.)

The `start` and `end` arrays are allocated in various ways, depending on the value of the `regs_allocated` field in the pattern buffer passed to the matcher.

The simplest and perhaps most useful is to let the matcher (re)allocate enough space to record information for all the groups in the regular expression. If `regs_allocated` is `REGS_UNALLOCATED`, the matcher allocates 1+*re_nsub* (another field in the pattern buffer; see Section 7.1.1 [GNU Pattern Buffers], page 24). The extra element is set to -1, and sets `regs_allocated` to `REGS_REALLOCATE`. Then on subsequent calls with the same pattern buffer and *regs* arguments, the matcher reallocates more space if necessary.

It would perhaps be more logical to make the `regs_allocated` field part of the `re_registers` structure, instead of part of the pattern buffer. But in that case the caller would be forced to initialize the structure before passing it. Much existing code doesn't do this initialization, and it's arguably better to avoid it anyway.

`re_compile_pattern` sets `regs_allocated` to `REGS_UNALLOCATED`, so if you use the GNU regular expression functions, you get this behavior by default.

xx document re_set_registers

POSIX, on the other hand, requires a different interface: the caller is supposed to pass in a fixed-length array which the matcher fills. Therefore, if `regs_allocated` is `REGS_FIXED` the matcher simply fills that array.

The following examples illustrate the information recorded in the `re_registers` structure. (In all of them, '`(`' represents the open-group and '`)`' the close-group operator. The first character in the string *string* is at index 0.)

- If the regular expression has an *i*-th group not contained within another group that matches a substring of *string*, then the function sets *regs*`->start[`*i*`]` to the index in *string* where the substring matched by the *i*-th group begins, and *regs*`->end[`*i*`]` to the index just beyond that substring's end. The function sets *regs*`->start[0]` and *regs*`->end[0]` to analogous information about the entire pattern.

  For example, when you match '`((a)(b))`' against '`ab`', you get:

  0 in *regs*`->start[0]` and 2 in *regs*`->end[0]`

  0 in *regs*`->start[1]` and 2 in *regs*`->end[1]`

  0 in *regs*`->start[2]` and 1 in *regs*`->end[2]`

  1 in *regs*`->start[3]` and 2 in *regs*`->end[3]`

- If a group matches more than once (as it might if followed by, e.g., a repetition operator), then the function reports the information about what the group *last* matched.

  For example, when you match the pattern '`(a)*`' against the string '`aa`', you get:

  0 in *regs*`->start[0]` and 2 in *regs*`->end[0]`

  1 in *regs*`->start[1]` and 2 in *regs*`->end[1]`

- If the *i*-th group does not participate in a successful match, e.g., it is an alternative not taken or a repetition operator allows zero repetitions of it, then the function sets *regs*`->start[`*i*`]` and *regs*`->end[`*i*`]` to -1.

  For example, when you match the pattern '`(a)*b`' against the string '`b`', you get:

  0 in *regs*`->start[0]` and 1 in *regs*`->end[0]`

  -1 in *regs*`->start[1]` and -1 in *regs*`->end[1]`

- If the *i*-th group matches a zero-length string, then the function sets *regs*`->start[`*i*`]` and *regs*`->end[`*i*`]` to the index just beyond that zero-length string.

  For example, when you match the pattern '`(a*)b`' against the string '`b`', you get:

  0 in *regs*`->start[0]` and 1 in *regs*`->end[0]`

  0 in *regs*`->start[1]` and 0 in *regs*`->end[1]`

- If an *i*-th group contains a *j*-th group in turn not contained within any other group within group *i* and the function reports a match of the *i*-th group, then it records in *regs*`->start[`*j*`]` and *regs*`->end[`*j*`]` the last match (if it matched) of the *j*-th group.

  For example, when you match the pattern '`((a*)b)*`' against the string '`abb`', group 2 last matches the empty string, so you get what it previously matched:

  0 in *regs*`->start[0]` and 3 in *regs*`->end[0]`

  2 in *regs*`->start[1]` and 3 in *regs*`->end[1]`

  2 in *regs*`->start[2]` and 2 in *regs*`->end[2]`

  When you match the pattern '`((a)*b)*`' against the string '`abb`', group 2 doesn't participate in the last match, so you get:

  0 in *regs*`->start[0]` and 3 in *regs*`->end[0]`

  2 in *regs*`->start[1]` and 3 in *regs*`->end[1]`

  0 in *regs*`->start[2]` and 1 in *regs*`->end[2]`

- If an *i*-th group contains a *j*-th group in turn not contained within any other group within group *i* and the function sets *regs*`->start[`*i*`]` and *regs*`->end[`*i*`]` to -1, then it also sets *regs*`->start[`*j*`]` and *regs*`->end[`*j*`]` to -1.

  For example, when you match the pattern '`((a)*b)*c`' against the string '`c`', you get:

  0 in *regs*`->start[0]` and 1 in *regs*`->end[0]`

  -1 in *regs*`->start[1]` and -1 in *regs*`->end[1]`

  -1 in *regs*`->start[2]` and -1 in *regs*`->end[2]`

7.1.9 Freeing GNU Pattern Buffers

To free any allocated fields of a pattern buffer, you can use the POSIX function described in Section 7.2.6 [Freeing POSIX Pattern Buffers], page 39, since the type `regex_t`—the type for POSIX pattern buffers—is equivalent to the type `re_pattern_buffer`. After freeing a pattern buffer, you need to again compile a regular expression in it (see Section 7.1.2 [GNU Regular Expression Compiling], page 26) before passing it to a matching or searching function.

7.2 POSIX Regex Functions

If you're writing code that has to be POSIX compatible, you'll need to use these functions. Their interfaces are as specified by POSIX, draft 1003.2/D11.2.

7.2.1 POSIX Pattern Buffers

To compile or match a given regular expression the POSIX way, you must supply a pattern buffer exactly the way you do for GNU (see Section 7.1.1 [GNU Pattern Buffers], page 24). POSIX pattern buffers have type `regex_t`, which is equivalent to the GNU pattern buffer type `re_pattern_buffer`.

7.2.2 POSIX Regular Expression Compiling

With POSIX, you can only search for a given regular expression; you can't match it. To do this, you must first compile it in a pattern buffer, using `regcomp`.

To compile a pattern buffer, use:

```
int
regcomp (regex_t *preg, const char *regex, int cflags)
```

preg is the initialized pattern buffer's address, *regex* is the regular expression's address, and *cflags* is the compilation flags, which Regex considers as a collection of bits. Here are the valid bits, as defined in '`regex.h`':

`REG_EXTENDED`
: says to use POSIX Extended Regular Expression syntax; if this isn't set, then says to use POSIX Basic Regular Expression syntax. `regcomp` sets *preg*'s `syntax` field accordingly.

`REG_ICASE`
: says to ignore case; `regcomp` sets *preg*'s `translate` field to a translate table which ignores case, replacing anything you've put there before.

`REG_NOSUB`
: says to set *preg*'s `no_sub` field; see Section 7.2.3 [POSIX Matching], page 37, for what this means.

`REG_NEWLINE`

says that a:

- match-any-character operator (see Section 3.2 [Match-any-character Operator], page 9) doesn't match a newline.
- nonmatching list not containing a newline (see Section 3.6 [List Operators], page 13) matches a newline.
- match-beginning-of-line operator (see Section 3.9.1 [Match-beginning-of-line Operator], page 18) matches the empty string immediately after a newline, regardless of how `REG_NOTBOL` is set (see Section 7.2.3 [POSIX Matching], page 37, for an explanation of `REG_NOTBOL`).
- match-end-of-line operator (see Section 3.9.1 [Match-beginning-of-line Operator], page 18) matches the empty string immediately before a newline, regardless of how `REG_NOTEOL` is set (see Section 7.2.3 [POSIX Matching], page 37, for an explanation of `REG_NOTEOL`).

If `regcomp` successfully compiles the regular expression, it returns zero and sets `*`*pattern_buffer* to the compiled pattern. Except for `syntax` (which it sets as explained above), it also sets the same fields the same way as does the GNU compiling function (see Section 7.1.2 [GNU Regular Expression Compiling], page 26).

If `regcomp` can't compile the regular expression, it returns one of the error codes listed here. (Except when noted differently, the syntax of in all examples below is basic regular expression syntax.)

`REG_BADRPT`

For example, the consecutive repetition operators '`**`' in '`a**`' are invalid. As another example, if the syntax is extended regular expression syntax, then the repetition operator '`*`' with nothing on which to operate in '`*`' is invalid.

`REG_BADBR`

For example, the *count* '`-1`' in '`a\{-1`' is invalid.

`REG_EBRACE`

For example, '`a\{1`' is missing a close-interval operator.

`REG_EBRACK`

For example, '`[a`' is missing a close-list operator.

`REG_ERANGE`
: For example, the range ending point '`z`' that collates lower than does its starting point '`a`' in '`[z-a]`' is invalid. Also, the range with the character class '`[:alpha:]`' as its starting point in '`[[:alpha:]-|]`'.

`REG_ECTYPE`
: For example, the character class name '`foo`' in '`[[:foo:]`' is invalid.

`REG_EPAREN`
: For example, '`a\)`' is missing an open-group operator and '`\(a`' is missing a close-group operator.

`REG_ESUBREG`
: For example, the back reference '`\2`' that refers to a nonexistent subexpression in '`\(a\)\2`' is invalid.

`REG_EEND`
: Returned when a regular expression causes no other more specific error.

`REG_EESCAPE`
: For example, the trailing backslash '`\`' in '`a\`' is invalid, as is the one in '`\`'.

`REG_BADPAT`
: For example, in the extended regular expression syntax, the empty group '`()`' in '`a()b`' is invalid.

`REG_ESIZE`
: Returned when a regular expression needs a pattern buffer larger than 65536 bytes.

`REG_ESPACE`
: Returned when a regular expression makes Regex to run out of memory.

7.2.3 POSIX Matching

Matching the POSIX way means trying to match a null-terminated string starting at its first character. Once you've compiled a pattern into a pattern buffer (see Section 7.2.2 [POSIX Regular Expression Compiling], page 35), you can ask the matcher to match that pattern against a string using:

```
int
regexec (const regex_t *preg, const char *string,
         size_t nmatch, regmatch_t pmatch[], int eflags)
```

preg is the address of a pattern buffer for a compiled pattern. *string* is the string you want to match.

See Section 7.2.5 [Using Byte Offsets], page 39, for an explanation of *pmatch*. If you pass zero for *nmatch* or you compiled *preg* with the compilation flag `REG_NOSUB` set, then `regexec` will ignore *pmatch*; otherwise, you must allocate it to have at least *nmatch* elements. `regexec` will record *nmatch* byte offsets in *pmatch*, and set to -1 any unused elements up to $pmatch[nmatch] - 1$.

eflags specifies *execution flags*—namely, the two bits `REG_NOTBOL` and `REG_NOTEOL` (defined in '`regex.h`'). If you set `REG_NOTBOL`, then the match-beginning-of-line operator (see Section 3.9.1 [Match-beginning-of-line Operator], page 18) always fails to match. This lets you match against pieces of a line, as you would need to if, say, searching for repeated instances of a given pattern in a line; it would work correctly for patterns both with and without match-beginning-of-line operators. `REG_NOTEOL` works analogously for the match-end-of-line operator (see Section 3.9.2 [Match-end-of-line Operator], page 18); it exists for symmetry.

`regexec` tries to find a match for *preg* in *string* according to the syntax in *preg*'s `syntax` field. (See Section 7.2.2 [POSIX Regular Expression Compiling], page 35, for how to set it.) The function returns zero if the compiled pattern matches *string* and `REG_NOMATCH` (defined in '`regex.h`') if it doesn't.

7.2.4 Reporting Errors

If either `regcomp` or `regexec` fail, they return a nonzero error code, the possibilities for which are defined in '`regex.h`'. See Section 7.2.2 [POSIX Regular Expression Compiling], page 35, and Section 7.2.3 [POSIX Matching], page 37, for what these codes mean. To get an error string corresponding to these codes, you can use:

```
size_t
regerror (int errcode,
          const regex_t *preg,
          char *errbuf,
          size_t errbuf_size)
```

errcode is an error code, *preg* is the address of the pattern buffer which provoked the error, *errbuf* is the error buffer, and *errbuf_size* is *errbuf*'s size.

`regerror` returns the size in bytes of the error string corresponding to *errcode* (including its terminating null). If *errbuf* and *errbuf_size* are nonzero, it also returns in *errbuf* the first $errbuf\_size - 1$ characters of the error string, followed by a null. *errbuf_size* must be a nonnegative number less than or equal to the size in bytes of *errbuf*.

You can call `regerror` with a null *errbuf* and a zero *errbuf_size* to determine how large *errbuf* need be to accommodate `regerror`'s error string.

7.2.5 Using Byte Offsets

In POSIX, variables of type `regmatch_t` hold analogous information, but are not identical to, GNU's registers (see Section 7.1.8 [Using Registers], page 32). To get information about registers in POSIX, pass to `regexec` a nonzero *pmatch* of type `regmatch_t`, i.e., the address of a structure of this type, defined in '`regex.h`':

```
typedef struct
{
  regoff_t rm_so;
  regoff_t rm_eo;
} regmatch_t;
```

When reading in Section 7.1.8 [Using Registers], page 32, about how the matching function stores the information into the registers, substitute *pmatch* for *regs*, *pmatch*`[`*i*`]->rm_so` for *regs*`->start[`*i*`]` and *pmatch*`[`*i*`]->rm_eo` for *regs*`->end[`*i*`]`.

7.2.6 Freeing POSIX Pattern Buffers

To free any allocated fields of a pattern buffer, use:

```
void
regfree (regex_t *preg)
```

preg is the pattern buffer whose allocated fields you want freed. `regfree` also sets *preg*'s `allocated` and `used` fields to zero. After freeing a pattern buffer, you need to again compile a regular expression in it (see Section 7.2.2 [POSIX Regular Expression Compiling], page 35) before passing it to the matching function (see Section 7.2.3 [POSIX Matching], page 37).

7.3 BSD Regex Functions

If you're writing code that has to be Berkeley UNIX compatible, you'll need to use these functions whose interfaces are the same as those in Berkeley UNIX.

7.3.1 BSD Regular Expression Compiling

With Berkeley UNIX, you can only search for a given regular expression; you can't match one. To search for it, you must first compile it. Before you compile it, you must indicate the regular expression syntax you want it compiled according to by setting the variable `re_syntax_options` (declared in '`regex.h`' to some syntax (see Chapter 2 [Regular Expression Syntax], page 2).

To compile a regular expression use:

```
char *
re_comp (char *regex)
```

regex is the address of a null-terminated regular expression. `re_comp` uses an internal pattern buffer, so you can use only the most recently compiled pattern buffer. This means that if you want to use a given regular expression that you've already compiled—but it isn't the latest one you've compiled—you'll have to recompile it. If you call `re_comp` with the null string (*not* the empty string) as the argument, it doesn't change the contents of the pattern buffer.

If `re_comp` successfully compiles the regular expression, it returns zero. If it can't compile the regular expression, it returns an error string. `re_comp`'s error messages are identical to those of `re_compile_pattern` (see Section 7.1.2 [GNU Regular Expression Compiling], page 26).

7.3.2 BSD Searching

Searching the Berkeley UNIX way means searching in a string starting at its first character and trying successive positions within it to find a match. Once you've compiled a pattern using `re_comp` (see Section 7.3.1 [BSD Regular Expression Compiling], page 40), you can ask Regex to search for that pattern in a string using:

```
int
re_exec (char *string)
```

string is the address of the null-terminated string in which you want to search.

`re_exec` returns either 1 for success or 0 for failure. It automatically uses a GNU fastmap (see Section 7.1.6 [Searching with Fastmaps], page 30).

Appendix A GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DE-

FECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

$

$ 18

(

(16

)

) 16

*

'*' 10

-

'-' 13

.

'.' 9

:

':]' in regex 14

?

'?' 11

[

'[' 13
'[:' in regex 14
'[^' 13

]

']' 13

{

'{' 12

|

| 13

}

'}' 12

+

'+' 11

^

'^' 13
^ 18

\

\ 7
'\' 13
'\'' 21
\(16
\) 16
'\`' 21
'\{' 12
\| 13
'\}' 12
'\>' 21
'\<' 21
'\b' 20
'\B' 20
'\s' 22
'\S' 22
'\w' 21
'\W' 21

A

`allocated` initialization 26
alternation operator 13
alternation operator and '^' 18
anchoring 18
anchors 18
Awk 5

B

back references 17

backtracking 10, 13
beginning-of-line operator 18
bracket expression 13
buffer field, set by re_compile_pattern 27
buffer initialization 26

C

character classes 14

E

Egrep 5
Emacs 5
end-of-line operator 18
end in struct re_registers 32

F

fastmap initialization 26
fastmap_accurate field, set by re_compile_pattern 27
fastmaps 30

G

Grep 5
grouping 16

I

ignoring case 35
interval expression 12

M

matching list 13
matching newline 13
matching with GNU functions 27

N

newline_anchor field in pattern buffer 18
nonmatching list 13
not_bol field in pattern buffer 18
num_regs in struct re_registers 32

O

open-group operator and '^' 18
or operator 13

P

parenthesizing 16
pattern buffer initialization 26
pattern buffer, definition of 24
POSIX Awk 5

R

range argument to re_search 28
re_registers 32
RE_BACKSLASH_ESCAPE_IN_LIST 3
RE_BK_PLUS_QM 3
RE_CHAR_CLASSES 3
RE_CONTEXT_INDEP_ANCHORS 3
RE_CONTEXT_INDEP_ANCHORS (and '^') 18
RE_CONTEXT_INDEP_OPS 3
RE_CONTEXT_INVALID_OPS 3
RE_DOT_NEWLINE 3
RE_DOT_NOT_NULL 4
RE_INTERVALS 4
RE_LIMITED_OPS 4
RE_NEWLINE_ALT 4
RE_NO_BK_BRACES 4
RE_NO_BK_PARENS 4
RE_NO_BK_REFS 4
RE_NO_BK_VBAR 4
RE_NO_EMPTY_RANGES 4
re_nsub field, set by re_compile_pattern 27
re_pattern_buffer definition 24
re_syntax_options initialization 26
RE_UNMATCHED_RIGHT_PAREN_ORD 4
REG_EXTENDED 35
REG_ICASE 35
REG_NEWLINE 36
REG_NOSUB 35
regex.c 1
regex.h 1
regexp anchoring 18
regmatch_t 39
regs_allocated 32
REGS_FIXED 33
REGS_REALLOCATE 32
REGS_UNALLOCATED 32
regular expressions, syntax of 2

S

searching with GNU functions 28
start argument to re_search 28
start in struct re_registers 32
struct re_pattern_buffer definition 24
subexpressions 16
syntax bits 2
syntax field, set by re_compile_pattern 27
syntax initialization 26
syntax of regular expressions 2

T

translate initialization 26

U

used field, set by re_compile_pattern 27

W

word boundaries, matching 20

Table of Contents

1 Overview **1**

2 Regular Expression Syntax **2**

- 2.1 Syntax Bits 2
- 2.2 Predefined Syntaxes 5
- 2.3 Collating Elements vs.Characters 6
- 2.4 The Backslash Character 7

3 Common Operators **9**

- 3.1 The Match-self Operator (*ordinary character*) 9
- 3.2 The Match-any-character Operator (`.`) 9
- 3.3 The Concatenation Operator 10
- 3.4 Repetition Operators 10
  - 3.4.1 The Match-zero-or-more Operator (`*`) 10
  - 3.4.2 The Match-one-or-more Operator (`+` or `\+`) 11
  - 3.4.3 The Match-zero-or-one Operator (`?` or `\?`) 11
  - 3.4.4 Interval Operators (`{` ... `}` or `\{` ... `\}`) 12
- 3.5 The Alternation Operator (`|` or `\|`) 13
- 3.6 List Operators (`[` ... `]` and `[^` ... `]`) 13
  - 3.6.1 Character Class Operators (`[:` ... `:]`) 14
  - 3.6.2 The Range Operator (`-`) 15
- 3.7 Grouping Operators (`(` ... `)` or `\(` ... `\)`) 16
- 3.8 The Back-reference Operator (`\`*digit*) 17
- 3.9 Anchoring Operators 18
  - 3.9.1 The Match-beginning-of-line Operator (`^`) 18
  - 3.9.2 The Match-end-of-line Operator (`$`) 18

4 GNU Operators **20**

- 4.1 Word Operators 20
  - 4.1.1 Non-Emacs Syntax Tables 20
  - 4.1.2 The Match-word-boundary Operator (`\b`) 20
  - 4.1.3 The Match-within-word Operator (`\B`) 20
  - 4.1.4 The Match-beginning-of-word Operator (`\<`) 21
  - 4.1.5 The Match-end-of-word Operator (`\>`) 21
  - 4.1.6 The Match-word-constituent Operator (`\w`) 21
  - 4.1.7 The Match-non-word-constituent Operator (`\W`) 21
- 4.2 Buffer Operators 21

4.2.1 The Match-beginning-of-buffer Operator (\`) 21
4.2.2 The Match-end-of-buffer Operator (\') 21

5 GNU Emacs Operators 22
5.1 Syntactic Class Operators 22
5.1.1 Emacs Syntax Tables 22
5.1.2 The Match-syntactic-class Operator (\s*class*) 22
5.1.3 The Match-not-syntactic-class Operator (\S*class*) 22

6 What Gets Matched? 23

7 Programming with Regex 24
7.1 GNU Regex Functions.. 24
7.1.1 GNU Pattern Buffers 24
7.1.2 GNU Regular Expression Compiling..................... 26
7.1.3 GNU Matching 27
7.1.4 GNU Searching 28
7.1.5 Matching and Searching with Split Data................. 29
7.1.6 Searching with Fastmaps 30
7.1.7 GNU Translate Tables 31
7.1.8 Using Registers 32
7.1.9 Freeing GNU Pattern Buffers............................ 34
7.2 POSIX Regex Functions... 35
7.2.1 POSIX Pattern Buffers................................. 35
7.2.2 POSIX Regular Expression Compiling................... 35
7.2.3 POSIX Matching 37
7.2.4 Reporting Errors....................................... 38
7.2.5 Using Byte Offsets 39
7.2.6 Freeing POSIX Pattern Buffers.......................... 39
7.3 BSD Regex Functions 40
7.3.1 BSD Regular Expression Compiling 40
7.3.2 BSD Searching 40

Appendix A GNU GENERAL PUBLIC LICENSE
.... 42
Preamble.. 42
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND
MODIFICATION.. 43
Appendix: How to Apply These Terms to Your New Programs......... 48

Index... 50