Rx

Tom Lord

except the chapter "Posix Entry Points"
from *The GNU C Library* reference manual
by Sandra Loosemore
with
Richard M. Stallman, Roland McGrath, and Andrew Oram

Short Contents

Table of Contents

Rx

This document describes Rx.

1 Copying

2 Overview

Nothing to say here, yet.

3 Posix Basic Regular Expressions

The Posix Basic Regular Expression language is a notation for describing textual patterns. Regexps are typically used by comparing them to a string to see if that string matches the pattern, or by searching within a string for a substring that matches.

This chapter introduces the Posix regexp notation. This is not a formal or precise definition of Posix regexps – it is an intuitive and hopefully expository description of them.

3.1 An Introduction to Regexps

In the simplest cases, a regexp is just a literal string that must match exactly. For example, the pattern:

```
regexp
```

matches the string "regexp" and no others.

Some characters have a special meaning when they occur in a regexp. They aren't matched literally as in the previous example, but instead denote a more general pattern. For example, the character `*` is used to indicate that the preceeding element of a regexp may be repeated 0, 1, or more times. In the pattern:

```
smooo*th
```

the `*` indicates that the preceeding `o` can be repeated 0 or more times. So the pattern matches:

```
smooth
smoooth
smooooth
smoooooth
...
```

Suppose you want to write a pattern that literally matches a special character like `*` – in other words, you don't want to `*` to indicate a permissible repetition, but to match `*` literally. This is accomplished by quoting the special character with a backslash. The pattern:

```
smoo\*th
```

matches the string:

```
smoo*th
```

and no other strings.

In seven cases, the pattern is reversed – a backslash makes the character special instead of making a special character normal. The characters `+`, `?`, `|`, `(`, and `)` are normal but the sequences `\+`, `\?`, `\|`, `\(`, `\)`, `\{`, and `\}` are special (their meaning is described later).

The remaining sections of this chapter introduce and explain the various special characters that can occur in regexps.

3.2 Literal Regexps

A literal regexp is a string which contains no special characters. A literal regexp matches an identical string, but no other characters. For example:

```
literally
```

matches

```
literally
```

and nothing else.

Generally, whitespace characters, numbers, and letters are not special. Some punctuation characters are special and some are not (the syntax summary at the end of this chapter makes a convenient reference for which characters are special and which aren't).

3.3 Character Sets

This section introduces the special characters `.` and `[`.

`.` matches any character except the NULL character. For example:

```
p.ck
```

matches

```
pick
pack
puck
pbck
pcck
p.ck

...
```

`[` begins a *character set*. A character set is similar to `.` in that it matches not a single, literal character, but any of a set of characters. `[` is different from `.` in that with `[`, you define the set of characters explicitly.

There are three basic forms a character set can take.

In the first form, the character set is spelled out:

```
[<cset-spec>] -- every character in <cset-spec> is in the set.
```

In the second form, the character set indicated is the negation of a character set is explicitly spelled out:

```
[^<cset-spec>] -- every character *not* in <cset-spec> is in the set.
```

A `<cset-spec>` is more or less an explicit enumeration of a set of characters. It can be written as a string of individual characters:

```
[aeiou]
```

or as a range of characters:

```
[0-9]
```

These two forms can be mixed:

```
[A-za-z0-9_$]
```

Note that special regexp characters (such as `*`) are *not* special within a character set. `-`, as illustrated above, *is* special, except, as illustrated below, when it is the first character mentioned.

This is a four-character set:

```
[-+*/]
```

The third form of a character set makes use of a pre-defined "character class":

```
[[:class-name:]] -- every character described by class-name is in the set.
```

The supported character classes are:

```
alnum - the set of alpha-numeric characters
alpha - the set of alphabetic characters
blank - tab and space
cntrl - the control characters
digit - decimal digits
graph - all printable characters except space
lower - lower case letters
print - the "printable" characters
punct - punctuation
space - whitespace characters
```

```
upper - upper case letters
xdigit - hexidecimal digits
```

Finally, character class sets can also be inverted:

```
[^[:space:]] - all non-whitespace characters
```

Character sets can be used in a regular expression anywhere a literal character can.

3.4 Subexpressions

A subexpression is a regular expression enclosed in `\(` and `\)`. A subexpression can be used anywhere a single character or character set can be used.

Subexpressions are useful for grouping regexp constructs. For example, the repeat operator, `*`, usually applies to just the preceeding character. Recall that:

```
smooo*th
```

matches

```
smooth
smoooth
...
```

Using a subexpression, we can apply `*` to a longer string:

```
banan\(an\)*a
```

matches

```
banana
bananana
banananana
...
```

Subexpressions also have a special meaning with regard to backreferences and substitutions (see See Section 3.9 [Backreferences], page 10).

3.5 Repeated Subexpressions

`*` is the repeat operator. It applies to the preceeding character, character set, subexpression or backreference. It indicates that the preceeding element can be matched 0 or more times:

```
bana\(na\)*
```

matches

```
bana
banana
bananana
banananana
...
```

`\+` is similar to `*` except that `\+` requires the preceeding element to be matched at least once. So while:

```
bana\(na\)*
```

matches

```
bana
```

```
bana(na\)\+
```

does not. Both match

```
banana
bananana
banananana
```

```
...
```

Thus, `bana\(na\)+` is short-hand for `banana\(na\)*`.

3.6 Optional Subexpressions

`\?` indicates that the preceeding character, character set, or subexpression is optional. It is permitted to match, or to be skipped:

```
CSNY\?
```

matches both

```
CSN
```

and

```
CSNY
```

3.7 Counted Subexpressions

An interval expression, `\{m,n\}` where `m` and `n` are non-negative integers with `n >= m`, applies to the preceeding character, character set, subexpression or backreference. It indicates that the preceeding element must match at least `m` times and may match as many as `n` times.

For example:

```
c\([ad]\)\{1,4\}
```

matches

```
car
cdr
caar
cdar
...
caaar
cdaar
...
cadddr
cddddr
```

3.8 Alternative Subexpressions

An alternative is written:

```
regexp-1\|regexp-2\|regexp-3\|...
```

It matches anything matched by some `regexp-n`. For example:

```
Crosby, Stills, \(and Nash\|Nash, and Young\)
```

matches

```
Crosby, Stills, and Nash
```

and

```
Crosby, Stills, Nash, and Young
```

3.9 Backreferences, Extractions and Substitutions

A backreference is written `\n` where `n` is some single digit other than 0. To be a valid backreference, there must be at least `n` parenthesized subexpressions in the pattern prior to the backreference.

A backreference matches a literal copy of whatever was matched by the corresponding subexpression. For example,

```
\(.*\)-\1
```

matches:

```
go-go
ha-ha
wakka-wakka
...
```

In some applications, subexpressions are used to extract substrings. For example, Emacs has the functions `match-beginnning` and `match-end` which report the positions of strings matched by subexpressions. These functions use the same numbering scheme for subexpressions as backreferences, with the additional rule that subexpression 0 is defined to be the whole regexp.

In some applications, subexpressions are used in string substitution. This again uses the backreference numbering scheme. For example, this sed command:

```
s/From:.*<\(.*\)>/To: \1/
```

first matches the line:

```
From: Joe Schmoe <schmoe@uspringfield.edu>
```

when it does, subexpression 1 matches "schmoe@uspringfield.edu". The command replaces the matched line with "To: \1" after doing subexpression substitution on it to get:

```
To: schmoe@uspringfield.edu
```

3.10 A Summary of Regexp Syntax

In summary, regexps can be:

`abcd` – matching a string literally

`.` – matching everything except NULL

`[a-z_?]`, `^[a-z_?]`, `[[:alpha:]]` and `[^[:alpha:]]` – matching character sets

`\(subexp\)` – grouping an expression into a subexpression.

`\n` – match a copy of whatever was matched by the nth subexpression.

The following special characters and sequences can be applied to a character, character set, subexpression, or backreference:

`*` – repeat the preceeding element 0 or more times.

`\+` – repeat the preceeding element 1 or more times.

`\?` – match the preceeding element 0 or 1 time.

`{m,n}` – match the preceeding element at least `m`, and as many as `n` times.

`regexp-1\|regexp-2\|..` – match any regexp-n.

A special character, like `.` or `*` can be made into a literal character by prefixing it with `\`.

A special sequence, like `\+` or `\?` can be made into a literal character by dropping the `\`.

3.11 Ambiguous Patterns

Sometimes a regular expression appears to be ambiguous. For example, suppose we compare the pattern:

```
begin\|beginning
```

to the string

```
beginning
```

either just the first 5 characters will match, or the whole string will match.

In every case like this, the longer match is preferred. The whole string will match.

Sometimes there is ambiguity not about how many characters to match, but where the subexpressions occur within the match. This can effect extraction functions like Emacs' `match-beginning` or rewrite functions like sed's `s` command. For example, consider matching the pattern:

```
b\(\[^q]*\)\(ing\)?
```

against the string

```
beginning
```

One possibility is that the first subexpression matches "eginning" and the second is skipped. Another possibility is that the first subexpression matches "eginn" and the second matches "ing".

The rule is that consistant with matching as many characters as possible, the length of lower numbered subexpressions is maximized in preference to maximizing the length of later subexpressions.

In the case of the above example, the two possible matches are equal in overall length. Therefore, it comes down to maximizing the lower-numbered subexpression, \1. The correct answer is that \1 matches "eginning" and \2 is skipped.

4 Posix Entry Points

This section is excerpted from *The GNU C Library* reference manual by Sandra Loosemore with Richard M. Stallman, Roland McGrath, and Andrew Oram.

The GNU C library supports the standard POSIX.2 interface. Programs using this interface should include the header file '`rxposix.h`'.

4.1 POSIX Regular Expression Compilation

Before you can actually match a regular expression, you must *compile* it. This is not true compilation—it produces a special data structure, not machine instructions. But it is like ordinary compilation in that its purpose is to enable you to "execute" the pattern fast. (See Section 4.3 [Matching POSIX Regexps], page 15, for how to use the compiled regular expression for matching.)

There is a special data type for compiled regular expressions:

`regex_t` [Data Type]

This type of object holds a compiled regular expression. It is actually a structure. It has just one field that your programs should look at:

`re_nsub` This field holds the number of parenthetical subexpressions in the regular expression that was compiled.

There are several other fields, but we don't describe them here, because only the functions in the library should use them.

After you create a `regex_t` object, you can compile a regular expression into it by calling `regcomp`.

`int regcomp` (*regex_t* *`*compiled`*, *const char* *`*pattern`*, *int* *`cflags`*) [Function]

`int regncomp` (*regex_t* *`*compiled`*, *const char* *`*pattern`*, *int* *`len`*, *int* *`cflags`*) [Function]

The function `regcomp` "compiles" a regular expression into a data structure that you can use with `regexec` to match against a string. The compiled regular expression format is designed for efficient matching. `regcomp` stores it into `*compiled`.

The parameter *pattern* points to the regular expression to be compiled. When using `regcomp`, *pattern* must be 0-terminated. When using `regncomp`, *pattern* must be *len* characters long.

`regncomp` is not a standard function; strictly POSIX programs should avoid using it.

It's up to you to allocate an object of type `regex_t` and pass its address to `regcomp`.

Before freeing the object of type `regex_t` You *must* pass it to `regfree`. Not doing so may cause subsequent calls to Rx functions to behave strangely.

The argument *cflags* lets you specify various options that control the syntax and semantics of regular expressions. See Section 4.2 [Flags for POSIX Regexps], page 14.

If you use the flag `REG_NOSUB`, then `regcomp` omits from the compiled regular expression the information necessary to record how subexpressions actually match. In this case, you might as well pass `0` for the *matchptr* and *nmatch* arguments when you call `regexec`.

If you don't use `REG_NOSUB`, then the compiled regular expression does have the capacity to record how subexpressions match. Also, `regcomp` tells you how many subexpressions *pattern* has, by storing the number in `compiled->re_nsub`. You can use that value to decide how long an array to allocate to hold information about subexpression matches.

`regcomp` returns `0` if it succeeds in compiling the regular expression; otherwise, it returns a nonzero error code (see the table below). You can use `regerror` to produce an error message string describing the reason for a nonzero value; see Section 4.6 [Regexp Cleanup], page 16.

Here are the possible nonzero values that `regcomp` can return:

`REG_BADBR`
: There was an invalid '`\{...\}`' construct in the regular expression. A valid '`\{...\}`' construct must contain either a single number, or two numbers in increasing order separated by a comma.

`REG_BADPAT`
: There was a syntax error in the regular expression.

`REG_BADRPT`
: A repetition operator such as '`?`' or '`*`' appeared in a bad position (with no preceding subexpression to act on).

`REG_ECOLLATE`
: The regular expression referred to an invalid collating element (one not defined in the current locale for string collation).

`REG_ECTYPE`
: The regular expression referred to an invalid character class name.

`REG_EESCAPE`
: The regular expression ended with '`\`'.

`REG_ESUBREG`
: There was an invalid number in the '`\digit`' construct.

`REG_EBRACK`
: There were unbalanced square brackets in the regular expression.

`REG_EPAREN`
: An extended regular expression had unbalanced parentheses, or a basic regular expression had unbalanced '`\(`' and '`\)`'.

`REG_EBRACE`
: The regular expression had unbalanced '`\{`' and '`\}`'.

`REG_ERANGE`
: One of the endpoints in a range expression was invalid.

`REG_ESPACE`
: `regcomp` ran out of memory.

4.2 Flags for POSIX Regular Expressions

These are the bit flags that you can use in the *cflags* operand when compiling a regular expression with `regcomp`.

`REG_EXTENDED`
: Treat the pattern as an extended regular expression, rather than as a basic regular expression.

`REG_ICASE`
: Ignore case when matching letters.

`REG_NOSUB`
: Don't bother storing the contents of the *matches-ptr* array.

`REG_NEWLINE`
: Treat a newline in *string* as dividing *string* into multiple lines, so that '`$`' can match before the newline and '`^`' can match after. Also, don't permit '`.`' to match a newline, and don't permit '`[^...]`' to match a newline.

: Otherwise, newline acts like any other ordinary character.

4.3 Matching a Compiled POSIX Regular Expression

Once you have compiled a regular expression, as described in Section 4.1 [POSIX Regexp Compilation], page 13, you can match it against strings using `regexec`. A match anywhere inside the string counts as success, unless the regular expression contains anchor characters ('`^`' or '`$`').

`int regexec` (*regex_t* ***`compiled`**, *char* ***`string`**, *size_t* **`nmatch`**, *regmatch_t* **`matchptr`** [], *int* **`eflags`**) [Function]

`int regnexec` (*regex_t* ***`compiled`**, *char* ***`string`**, *int* **`len`**, *size_t* **`nmatch`**, *regmatch_t* **`matchptr`** [], *int* **`eflags`**) [Function]

This function tries to match the compiled regular expression *`*compiled`* against *string*.

`regexec` returns `0` if the regular expression matches; otherwise, it returns a nonzero value. See the table below for what nonzero values mean. You can use `regerror` to produce an error message string describing the reason for a nonzero value; see Section 4.6 [Regexp Cleanup], page 16.

The parameter *string* points to the text to search. When using `regexec`, *string* must be 0-terminated. When using `regnexec`, *string* must be *len* characters long.

`regnexec` is not a standard function; strictly POSIX programs should avoid using it.

The argument *eflags* is a word of bit flags that enable various options.

If you want to get information about what part of *string* actually matched the regular expression or its subexpressions, use the arguments *matchptr* and *nmatch*. Otherwise, pass `0` for *nmatch*, and `NULL` for *matchptr*. See Section 4.4 [Regexp Subexpressions], page 15.

You must match the regular expression with the same set of current locales that were in effect when you compiled the regular expression.

The function `regexec` accepts the following flags in the *eflags* argument:

`REG_NOTBOL`
: Do not regard the beginning of the specified string as the beginning of a line; more generally, don't make any assumptions about what text might precede it.

`REG_NOTEOL`
: Do not regard the end of the specified string as the end of a line; more generally, don't make any assumptions about what text might follow it.

Here are the possible nonzero values that `regexec` can return:

`REG_NOMATCH`
: The pattern didn't match the string. This isn't really an error.

`REG_ESPACE`
: `regexec` ran out of memory.

4.4 Match Results with Subexpressions

When `regexec` matches parenthetical subexpressions of *pattern*, it records which parts of *string* they match. It returns that information by storing the offsets into an array whose elements are structures of type `regmatch_t`. The first element of the array (index `0`) records the part of the string that matched the entire regular expression. Each other element of the array records the beginning and end of the part that matched a single parenthetical subexpression.

`regmatch_t` [Data Type]

This is the data type of the *matcharray* array that you pass to `regexec`. It containes two structure fields, as follows:

`rm_so` The offset in *string* of the beginning of a substring. Add this value to *string* to get the address of that part.

`rm_eo` The offset in *string* of the end of the substring.

`regoff_t` [Data Type]

`regoff_t` is an alias for another signed integer type. The fields of `regmatch_t` have type `regoff_t`.

The `regmatch_t` elements correspond to subexpressions positionally; the first element (index `1`) records where the first subexpression matched, the second element records the second subexpression, and so on. The order of the subexpressions is the order in which they begin.

When you call `regexec`, you specify how long the *matchptr* array is, with the *nmatch* argument. This tells `regexec` how many elements to store. If the actual regular expression has more than *nmatch* subexpressions, then you won't get offset information about the rest of them. But this doesn't alter whether the pattern matches a particular string or not.

If you don't want `regexec` to return any information about where the subexpressions matched, you can either supply `0` for *nmatch*, or use the flag `REG_NOSUB` when you compile the pattern with `regcomp`.

4.5 Complications in Subexpression Matching

Sometimes a subexpression matches a substring of no characters. This happens when '`f\(o*\)`' matches the string '`fum`'. (It really matches just the '`f`'.) In this case, both of the offsets identify the point in the string where the null substring was found. In this example, the offsets are both `1`.

Sometimes the entire regular expression can match without using some of its subexpressions at all—for example, when '`ba\(na\)*`' matches the string '`ba`', the parenthetical subexpression is not used. When this happens, `regexec` stores `-1` in both fields of the element for that subexpression.

Sometimes matching the entire regular expression can match a particular subexpression more than once—for example, when '`ba\(na\)*`' matches the string '`bananana`', the parenthetical subexpression matches three times. When this happens, `regexec` usually stores the offsets of the last part of the string that matched the subexpression. In the case of '`bananana`', these offsets are `6` and `8`.

But the last match is not always the one that is chosen. It's more accurate to say that the last *opportunity* to match is the one that takes precedence. What this means is that when one subexpression appears within another, then the results reported for the inner subexpression reflect whatever happened on the last match of the outer subexpression. For an example, consider '`\(ba\(na\)*s \)*`' matching the string '`bananas bas `'. The last time the inner expression actually matches is near the end of the first word. But it is *considered* again in the second word, and fails to match there. `regexec` reports nonuse of the "na" subexpression.

Another place where this rule applies is when the regular expression '`\(ba\(na\)*s \|nefer\(ti\)* \)*`' matches '`bananas nefertiti`'. The "na" subexpression does match in the first word, but it doesn't match in the second word because the other alternative is used there. Once again, the second repetition of the outer subexpression overrides the first, and within that second repetition, the "na" subexpression is not used. So `regexec` reports nonuse of the "na" subexpression.

4.6 POSIX Regexp Matching Cleanup

When you are finished using a compiled regular expression, you must free the storage it uses by calling `regfree`.

`void regfree` (*regex_t* `*compiled`) [Function]

Calling `regfree` frees all the storage that `*compiled` points to. This includes various internal fields of the `regex_t` structure that aren't documented in this manual.

`regfree` does not free the object `*compiled` itself.

You should always free the space in a `regex_t` structure with `regfree` before using the structure to compile another regular expression.

When `regcomp` or `regexec` reports an error, you can use the function `regerror` to turn it into an error message string.

`size_t regerror` (*int* `errcode`, *regex_t* `*compiled`, *char* `*buffer`, *size_t* `length`) [Function]

This function produces an error message string for the error code *errcode*, and stores the string in *length* bytes of memory starting at *buffer*. For the *compiled* argument, supply the same compiled regular expression structure that `regcomp` or `regexec` was working with when it got the error. Alternatively, you can supply `NULL` for *compiled*; you will still get a meaningful error message, but it might not be as detailed.

If the error message can't fit in *length* bytes (including a terminating null character), then `regerror` truncates it. The string that `regerror` stores is always null-terminated even if it has been truncated.

The return value of `regerror` is the minimum length needed to store the entire error message. If this is less than *length*, then the error message was not truncated, and you can use it. Otherwise, you should call `regerror` again with a larger buffer.

Here is a function which uses `regerror`, but always dynamically allocates a buffer for the error message:

```
char *get_regerror (int errcode, regex_t *compiled)
{
  size_t length = regerror (errcode, compiled, NULL, 0);
  char *buffer = xmalloc (length);
  (void) regerror (errcode, compiled, buffer, length);
  return buffer;
}
```

5 Beyond POSIX

This section is not finished documentation, but rather a collection of pointers towards some of the interesting, non-standard features of Rx.

5.1 New Regexp Operators

Rx supports some unusual regexp syntax.

`[[:cut N:]]` sets `pmatch[0].final_tag` to N and causes the matching to stop instantly. If N is 0, the overall match fails, otherwise it succeeds.

`[[:(:]] ... [[:):]]` is just like `\( ... \)` except that in the first case, no pmatch entries are changed, and the subexpression is not counted in the numbering of parenthesized subexpressions.

`[[:(:]] ... [[:):]]` can be used when you do not need to know where a subexpression matched but are only using parentheses to effect the parsing of the regexp.

There are two reasons to use `[[:(:]] ... [[:):]]`:

1. regexec will run faster.
2. Currently, only 8 backreferencable subexpressions are supported: `\1 .. \9`. Using `[[:(:]] ... [[:):]]` is a way to conserve backreferencable subexpression names in an expression with many parentheses.

5.2 New POSIX Functions

`regncomp` and `regnexec` are non-standard generalizations of `regcomp` and `regexec`.

5.3 Tuning POSIX performance

Two mysterious parmaters can be used to trade-off performance and memory use.

At compile-time they are `RX_DEFAULT_DFA_CACHE_SIZE` and `RX_DEFAULT_NFA_DELAY`.

If you want to mess with these (I generally don't advise it), I suggest experimenting for your particular application/memory situation; frob these by powers of two and try out the results on what you expect will be typical regexp workloads.

You can also set those parameters at run-time (before calling any regexp functions) by tweaking the corresponding variables:

`rx_default_cache->bytes_allowed`

and

`rx_basic_unfaniverse_delay`

5.4 POSIX stream-style interface

`rx_make_solutions`, `rx_next_solution`, and `rx_free_solutions` are a lower level alternative to the posix functions. Using those functions, you can compare a compiled regexp to a string that is not contiguous in memory or even a string that is not entirely in memory at any one time.

The code in rxposix.c points out how those functions are used.

5.5 DFAs Directly

If you are only interested in pure regular expressions (no pmatch data, no backreferences, and no counted subexpressions), you can parse a regexp using `rx_parse`, convert it to an nfa using `rx_unfa`, and run the dfa using `rx_init_system`, `rx_advance_to_final`, and `rx_terminate_system`. The dfa Scheme primitives in '`rgx.c`' may provide some guide.

6 Rx Theory

There are two match algorithms. One is for truly regular regexps (those that can be reduced to a dfa). The other is for non-regular regexps.

The dfa algorithm implements the idea suggested in *Compilers* by Aho, Sethi and Ullman:

> [One] approach [to pattern matching regular expressions] is to use a DFA, but avoid constructing all of the transition table by using a technique called "lazy transition evaluation". Here, transitions are computed at run time [when] actually needed. [T]ransitions are stored in a cache. [....] If the cache becomes full, we can erase some previously computed transition to make room for the new transition.

The implementation in Rx is generalized from that, but the above description covers what is used for Posix patterns.

The non-dfa algorithm implements a "recursive decomposition" technique described in email by Henry Spencer. For a given pattern, this algorithm first checks to see if a simpler, superset language, DFA-pattern matches. If it does, then this algorithm does the detail-work to see if the non-DFA pattern matches.

The detail work usually involves recursing on subpatterns. For example, a concatentation of two subexpressions matches a string if the string can be divided into two parts, each matching one subexpression, in the right order. More than one solution is often possible for a given pattern. This ambiguity is the subject of the "leftmost longest" rules in the spec, and the back-tracking oriented stream-of-solution functions `rx_make_solutions`, `rx_next_solution` and `rx_free_solutions`.

```
rxspencer.[ch]   -- The non-DFA algorithm
rxanal.[ch] rxsuper.[ch] rxnfa.[ch] -- The DFA algorithm
```