35.3.3.1 Constructs in `rx` regexps

The various forms in `rx` regexps are described below. The shorthand `rx` represents any `rx` form. `rx...` means zero or more `rx` forms and, unless stated otherwise, matches these forms in sequence as if wrapped in a `(seq ...)` subform.
These are all valid arguments to the `rx` macro. All forms are defined by their described semantics; the corresponding string regexps are provided for ease of understanding only. `A`, `B`, ... denote (suitably bracketed) string regexp subexpressions therein.

Literals

`"some-string"` Match the string `some-string` literally. There are no characters with special meaning, unlike in string regexps.

`?C` Match the character `C` literally.

Sequence and alternative

`(seq x...)` `(sequence x...)` `(: x...)` `(and x...)` Match the `rx`s in sequence. Without arguments, the expression matches the empty string. Corresponding string regexp: `AB...` (subexpressions in sequence).

`(or x...)` `(| x...)` Match exactly one of the `rx`s. If all arguments are

= strings, characters, or `or` forms so constrained, the longest possible match will always be used. Otherwise, either the longest match or the first (in left-to-right order) will be used. Without arguments, the expression will not match anything at all. Corresponding string regexp: `A\|B\|...`.

`unmatchable` Refuse any match. Equivalent to `(or)`. See regexp-unmatchable

Repetition

Normally, repetition forms are greedy, in that they attempt to match as many times as possible. Some forms are non-greedy; they try to match as few times as possible

rx Macros	rx	Regexp	Matches	Greedy?
`(zero-or-more x...)`	`(0+ x...)`		the `rx`s zero or more times.	Default
`(one-or-more x...)`	`(1+ x...)`		1+ times	Default
`(zero-or-one x...)` `(optional x...)`	`(opt x...)`		once or an empty string	Default
	`(* x...)`	`A*`	0+ times	Gr
	`(+ x...)`	`A+`	1+ times	Gr
	`(? x...)`	`A?`	once or an empty string	Gr
	`(*? x...)`	`A*?`	0+ times	Non-Gr
	`(+? x...)`	`A+?`	1+ times	Non-Gr
	`(?? x...)`	`A??`	once or an empty string	Non-Gr
`(repeat n x)`	`(= n x...)`	`A\{n\}`	exactly `n` times	Gr
	`(>= n x...)`	`A\{n,\}`	`n` or more times	Gr
`(repeat n x)` `(repeat n m x...)`	`(** n m x...)`	`A\{n,m\}`	`n` to `m` times	Gr

The greediness of some repetition forms can be controlled using the following constructs. However, it is usually better to use the explicit non-greedy forms above when such matching is required.

`(minimal-match x)` Match `rx`, with `zero-or-more`, `0+`, `one-or-more`, `1+`, `zero-or-one`, `opt` and `optional` using non-greedy matching.

`(maximal-match x)` Match `rx`, with `zero-or-more`, `0+`, `one-or-more`, `1+`, `zero-or-one`, `opt` and `optional` using greedy matching. This is the default.

Matching single characters

`(any t...)` `(char t...)` `(in t...)` Match a single character from one of the `set=s`. Each `=set` is a character, a string representing the set of its characters, a range or a character class (see below). A range is either a hyphen-separated string like `"A-Z"`, or a cons of characters like `(?A . ?Z)`.

Note that hyphen (`-`) is special in strings in this construct, since it acts as a range separator. To include a hyphen, add it as a separate character or single-character string. String regexp: `[...]`

`(not c)` Match a character not included in `charspec`. `charspec` can be a character, a single-character string, an `any`, `not`, `or`, `intersection`, `syntax` or `category` form, or a character class. If `charspec` is an `or` form, its arguments have the same restrictions as those of `intersection`; see below. String regexp: `[^...]`, `\Scode`, `\Ccode`

`(intersection t...)` Match a character included in all of the `charset=s`. Each `=charset` can be a character, a single-character string, an `any` form without character classes, or an `intersection`, `or` or `not` form whose arguments are also `charset`'s.

`not-newline, nonl` Any character except a newline. String regexp: `.`

`anychar, anything` Any character. String regexp: `.\|\n`

Character Class

Match a character from a named character class:

Regexp	rx Macros	matches
[0-9]	alpha, alphabetic, letter	digits
[a-zA-Z]	digit, numeric, num	general-category
[0-9a-zA-Z]	alnum, alphanumeric	
[0-9a-fA-F]	xdigit, hex-digit, hex	hex
	cntrl, control	bytes 0 to 31
	blank	horizontal whitespace (general-category spacing seps)
	space, whitespace, white	any character with whitespace syntax
	lower, lower-case	lower-case (using case-table) unless case-fold-search
	upper, upper-case	upper-case (using case-table) unless case-fold-search
	graph, graphic	any character except whitespace and ... **
	print, printing	whitespace or a character matched by graph
	punct, punctuation	Match any punctuation character. ***
	word, wordchar	Match any character that has word syntax
	ascii	Match any ASCII character (bytes 0 to 127).
=[[:s:]]=	nonascii	Match any non-ASCII character (but not raw bytes)

- ** ... whitespace and ASCII and non-ASCII control characters, surrogates, and codepoints unassigned by Unicode, as indicated by the Unicode general-category property.
- *** At present, for multibyte characters, anything that has non-word syntax.

Syntax Classes

(syntax x) Match a character with syntax syntax, being one of the following names:

Char	Syntax	Char	Syntax
-	whitespace	(	open-parenthesis
.	punctuation	)	close-parenthesis
w	word	'	expression-prefix
_	symbol	"	string-quote
<	comment-start	$	paired-delimiter
>	comment-end	\	escape
|	string-delimiter	/	character-quote
!	comment-delimiter		

(syntax punctuation) is *not* equivalent to the character class punctuation Corresponding string regexp: =\s==char= where char is the syntax character.

Categories

(category y) Match a character in category category, which is either one of the names below or its category character.

	Category		Category		Category		Category
space	space-for-indent	<	not-at-end-of-line	a	ascii	t	thai
.	base	>	not-at-beginning-of-line	b	arabic	v	vietnamese
0	consonant	A	alpha-numeric-two-byte	c	chinese	w	hebrew
1	base-vowel	C	chinese-two-byte	e	ethiopic	y	cyrillic
2	upper-diacritical-mark	G	greek-two-byte	g	greek		
3	lower-diacritical-mark	H	japanese-hiragana-two-byte	h	korean		
4	tone-mark	I	indian-two-byte	i	indian		
5	symbol	K	japanese-katakana-two-byte	j	japanese		
6	digit	L	strong-left-to-right	k	japanese-katakana		
7	vowel-modifying-diacritical-mark	N	korean-hangul-two-byte	l	latin		
8	vowel-sign	R	strong-right-to-left	o	lao		
9	semivowel-lower	Y	cyrillic-two-byte	q	tibetan		
|	can-break	^	combining-diacritic	r	japanese-roman		

For more information about currently defined categories, run the command M-x describe-categories RET. For how to define new categories, see Categories. Corresponding string regexp: =\c==char= where char is the category character.

Zero-width assertions

These all match the empty string, but only in specific places.

rx Macros	Regexp	Matches
line-start, bol	^	the beginning of a line.
line-end, eol	$	the end of a line.
string-start, bos, buffer-start, bot	\`	start of string/buffer
string-end, eos, buffer-end, eot	\'	end of string/buffer
point	\=	point
word-start, bow	\<	the beginning of a word.
word-end, eow	\>	the end of a word.
word-boundary	\b	beginning/end of a word
not-word-boundary	\B	anywhere except beginning/end of a word
symbol-start	_<	the beginning of a symbol.
symbol-end	_>	the end of a symbol.

Capture groups

(group x...) (submatch x...) Match the rx=s, making the matched text and position accessible in the match data. The first group in a regexp is numbered 1; subsequent groups will be numbered one above the previously hig group in the pattern so far. Corresponding string regexp: ~...=~

(group-n n x...) (submatch-n n x...) Like group, but explicitly assign the group number n. n must be positive. Corresponding string regexp: =\(?n:...\)=

(backref n) Match the text previously matched by group number n. n must be in the range 1–9. Corresponding string regexp: =\==n=

Dynamic inclusion

(literal r) Match the literal string that is the result from evaluating the Lisp expression expr. The evaluation takes place at call time, in the current lexical environment.

(regexp r) (regex r) Match the string regexp that is the result from evaluating the Lisp expression expr. The evaluation takes place at call time, in the current lexical environment.

(eval r) Match the rx form that is the result from evaluating the Lisp expression expr. The evaluation takes place at macro-expansion time for rx, at call time for rx-to-string, in the current global environment.