1 Regular expressions

Regular expressions are patterns used in selecting text.

In addition to a specifying string literals, regular expressions can represent classes of strings. Strings thus represented are said to be matched by the corresponding regular expression. If it is possible for a regular expression to match several strings in a line, then the left-most longest match is the one selected.

The following symbols are used in constructing regular expressions:

c Any character *c* not listed below, including '`{`', '`}`', '`(`', '`)`', '`<`' and '`>`', matches itself.

`\`*c* Any backslash-escaped character *c*, other than '`{`', "`}`', '`(`', '`)`', '`<`', '`>`', '`b`', '`B`', '`w`', '`W`', '`+`' and '`?`', matches itself.

Note that '`\`' also has special meaning in the read syntax of Lisp strings, and must be quoted with '`\`'. For example, the regular expression that matches the '`\`' character is '`\\`'. To write a Lisp string that contains the characters '`\\`', Lisp syntax requires you to quote each '`\`' with another '`\`'. Therefore, the read syntax for a regular expression matching '`\`' is `"\\\\"`.

`.` Matches any single character.

`[`*char-class*`]`

Matches any single character in *char-class*. To include a '`]`' in *char-class*, it must be the first character. A range of characters may be specified by separating the end characters of the range with a '`-`', e.g., '`a-z`' specifies the lower case characters. The following literal expressions can also be used in *char-class* to specify sets of characters:

```
[:alnum:] [:cntrl:] [:lower:] [:space:]
[:alpha:] [:digit:] [:print:] [:upper:]
[:blank:] [:graph:] [:punct:] [:xdigit:]
```

If '`-`' appears as the first or last character of *char-class*, then it matches itself. All other characters in *char-class* match themselves.

Patterns in *char-class* of the form:

`[.`*col-elm*`.]`
`[=`*col-elm*`=]`

where *col-elm* is a *collating element* are interpreted according to `locale (5)` (not currently supported). See `regex (3)` for an explanation of these constructs.

`[^`*char-class*`]`

Matches any single character, other than newline, not in *char-class*. *char-class* is defined as above.

`^` If '`^`' is the first character of a regular expression, then it anchors the regular expression to the beginning of a line. Otherwise, it matches itself.

`$` If '`$`' is the last character of a regular expression, it anchors the regular expression to the end of a line. Otherwise, it matches itself.

`\(`*re*`\)` Defines a (possibly null) subexpression *re*. Subexpressions may be nested. A subsequent backreference of the form '`\`*n*', where *n* is a number in the range [1,9], expands to the text matched by the *n*th subexpression. For example, the regular expression '`\(a.c\)\1`' matches the string '`abcabc`', but not '`abcadc`'. Subexpressions are ordered relative to their left delimiter.

`*` Matches the single character regular expression or subexpression immediately preceding it zero or more times. If '`*`' is the first character of a regular expression or subexpression, then it matches itself. The '`*`' operator sometimes yields unexpected results. For example, the regular expression '`b*`' matches the beginning of the string '`abbb`', as opposed to the substring '`bbb`', since a null match is the only left-most match.

`\{`*n*`,`*m*`\}`
`\{`*n*`,\}`
`\{`*n*`\}` Matches the single character regular expression or subexpression immediately preceding it at least *n* and at most *m* times. If *m* is omitted, then it matches at least *n* times. If the comma is also omitted, then it matches exactly *n* times. If any of these forms occurs first in a regular expression or subexpression, then it is interpreted literally (i.e., the regular expression '`\{2\}`' matches the string '`{2}`', and so on).

`\<`
`\>` Anchors the single character regular expression or subexpression immediately following it to the beginning (in the case of '`\<`') or ending (in the case of '`\>`') of a *word*, i.e., in ASCII, a maximal string of alphanumeric characters, including the underscore (`_`).

The following extended operators are preceded by a backslash '`\`' to distinguish them from traditional `ed` syntax.

`\‘`
`\’` Unconditionally matches the beginning '`\‘`' or ending '`\’`' of a line.

`\?` Optionally matches the single character regular expression or subexpression immediately preceding it. For example, the regular expression '`a[bd]\?c`' matches the strings '`abc`', '`adc`' and '`ac`'. If '`\?`' occurs at the beginning of a regular expressions or subexpression, then it matches a literal '`?`'.

`\+` Matches the single character regular expression or subexpression immediately preceding it one or more times. So the regular expression '`a+`' is shorthand for '`aa*`'. If '`\+`' occurs at the beginning of a regular expression or subexpression, then it matches a literal '`+`'.

`\b` Matches the beginning or ending (null string) of a word. Thus the regular expression

‘`\bhello\b`’ is equivalent to ‘`\<hello\>`’. However, ‘`\b\b`’ is a valid regular expression whereas ‘`\<\>`’ is not.

`\B` Matches (a null string) inside a word.

`\w` Matches any character in a word.

`\W` Matches any character not in a word.

2 Regular Expressions

A *regular expression* (*regexp*, for short) is a pattern that denotes a (possibly infinite) set of strings. Searching for matches for a regexp is a very powerful operation. This section explains how to write regexps; the following section says how to search for them.

2.1 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`[`', '`]`' and '`\`'; no new special characters will be defined in the future. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

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

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

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something more powerful, you need to use one of the special characters. Here is a list of them:

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

`*` is not a construct by itself; it is a suffix operator that means to repeat the preceding regular expression as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches one '`f`' followed by any number of '`o`'s. The case of zero '`o`'s is allowed: '`fo*`' does match '`f`'.

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

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking

occurs, discarding some of the matches of the '`*`'-modified construct in case that makes it possible to match the rest of the pattern. For example, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

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

The usual regular expression special characters are not special inside a character set. A completely different set of special characters exists inside character sets: '`]`', '`-`' and '`^`'.

'`-`' is used for ranges of characters. To write a range, write two characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower case letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case letter or '`$`', '`%`' or a period.

The following literal expressions can also be used in *char-class* to specify sets of characters:

```
[:alnum:] [:cntrl:] [:lower:] [:space:]
[:alpha:] [:digit:] [:print:] [:upper:]
[:blank:] [:graph:] [:punct:] [:xdigit:]
```

To include a '`]`' in a character set, make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', write '`-`' as the first character in the set, or put immediately after a range. (You can replace one individual character *c* with the range '*c*-*c*' to make a place to put the '`-`'). There is no way to write a set containing just '`-`' and '`]`'.

To include '`^`' in a set, put it anywhere but at the beginning of the set.

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

'`^`' is not special in a character set unless it is the first character. The character following the '`^`' is treated as if it were first (thus, '`-`' and '`]`' are not special there).

Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.

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

When matching a string, '`^`' matches at the beginning of the string or after a newline character '`\n`'.

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

When matching a string, '`$`' matches at the end of the string or before a newline character '`\n`'.

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

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

Note that '`\`' also has special meaning in the read syntax of Lisp strings, and must be quoted with '`\`'. For example, the regular expression that matches the '`\`' character is '`\\`'. To write a Lisp string that contains the characters '`\\`', Lisp syntax requires you to quote each '`\`' with another '`\`'. Therefore, the read syntax for a regular expression matching '`\`' is `"\\\\"`.

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

`\+` is a suffix operator similar to '`*`' except that the preceding expression must match at least once. So, for example, '`ca+r`' matches the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' matches all three strings.

`\?` is a suffix operator similar to '`*`' except that the preceding expression can match either once or not at all. For example, '`ca?r`' matches '`car`' or '`cr`', but does not match anyhing else.

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

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

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

Full backtracking capability exists to handle multiple uses of '`\|`'.

`\( ... \)` is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose an expression for a suffix operator such as '`*`' to act on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number of '`na`' strings.
3. To record a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a

separate feature which happens to be assigned as a second meaning to the same '`\( ... \)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

`\`*digit* matches the same text which matched the *digit*th occurrence of a '`\( ... \)`' construct.

In other words, after the end of a '`\( ... \)`' construct. the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by *digit* to match that same text, whatever it may have been.

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding '`\( ... \)`' constructs.

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

`\w` matches any word-constituent character.

`\W` matches any character that is not a word-constituent.

These regular expression constructs match the empty string—that is, they don't use up any characters—but whether they match depends on the context.

`\‘` matches the empty string, but only at the beginning of the buffer or string being matched against.

`\’` matches the empty string, but only at the end of the buffer or string being matched against.

`\b` matches the empty string, but only at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bballs?\b`' matches '`ball`' or '`balls`' as a separate word.

`\B` matches the empty string, but *not* at the beginning or end of a word.

`\<` matches the empty string, but only at the beginning of a word.

`\>` matches the empty string, but only at the end of a word.

Not every string is a valid regular expression. For example, a string with unbalanced square brackets is invalid (with a few exceptions, such as '`[]]`', and so is a string that ends with a single '`\`'. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

3 Examples

3.1 Complex Regexp Example

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is the value of the variable `sentence-end`.

First, we show the regexp as a string in C syntax to distinguish spaces from tab characters. The string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the string, '`\\`' for a backslash as part of the string, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
```

In contrast, in Lisp, you have to type the tab as Ctrl-V Ctrl-I, producing the following:

```
sentence-end
⇒
"[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

In this output, tab and newline appear as themselves.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]` The first part of the pattern consists of three characters, a period, a question mark and an exclamation mark, within square brackets. The match must begin with one of these three characters.

`[]\"')}]*`

The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is C or Lisp syntax for a double-quote in a string. The '`*`' at the end indicates that the immediately preceding regular expression (a character set, in this case) may be repeated zero or more times.

`\\($\\| \\|\t\\|  \\)`

The third part of the pattern matches the whitespace that follows the end of a sentence: the end of a line, or a tab, or two spaces. The double backslashes mark the parentheses

and vertical bars as regular expression syntax; the parentheses mark the group and the vertical bars separate alternatives. The dollar sign is used to match the end of a line.

`[ \t\n]*` Finally, the last part of the pattern matches any additional whitespace beyond the minimum needed to end a sentence.

3.2 Common Regular Expressions Used in Editing

This section describes some common regular expressions used for certain purposes in editing:

Page delimiter: This is the regexp describing line-beginnings that separate pages. A good value is `(string #\Page)`.

Paragraph separator: This is the regular expression for recognizing the beginning of a line that separates paragraphs. A good value is (in C syntax) `"^[ \t\f]*$"`, which is a line that consists entirely of spaces, tabs, and form feeds.

Paragraph start: This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. A good value is (in C syntax) `"^[ \t\n\f]"`, which matches a line starting with a space, tab, newline, or form feed.

Sentence end: This is the regular expression describing the end of a sentence. (All paragraph boundaries also end sentences, regardless.) A good value is (in C syntax, again):

```
"[.?!][]\"')}]*\\($\\|\t\\| \\)[ \t\n]*"
```

This means a period, question mark or exclamation mark, followed by a closing brace, followed by tabs, spaces or new lines.

4 The Regular Expression Module

match *regexp string* &key *start end* Function

This function returns as first value a `match` structure containing the indices of the start and end of the first match for the regular expression *regexp* in *string*, or `nil` if there is no match. If *start* is non-`nil`, the search starts at that index in *string*. If *end* is non-`nil`, only `(subseq` *string start end*`)` is considered.

For example,

```
(match "quick" "The quick brown fox jumped quickly.")
     ⇒ #S(match :start 4 :end 9)
(match "quick" "The quick brown fox jumped quickly." :start 8)
     ⇒ #S(match :start 27 :end 32)
(match "quick" "The quick brown fox jumped quickly."
       :start 8 :end 30)
     ⇒ nil
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

The next values are `match` structures for every '`\(` ... `\)`' contruct in *regexp*, in the order that the open parentheses appear in *regexp*.

match-start *match* Function

Extracts the start index of *match*.

match-end *match* Function

Extracts the end index of *match*.

match-string *string match* Function

Extracts the substring of *string* corresponding to a given pair of start and end indices. The result is shared with *string*. If you want a freshly consed string, use `copy-string` or `(coerce (match-string ...) 'simple-string)`.

regexp-quote *string* Function

This function returns a regular expression string that matches exactly *string* and nothing else. This allows you to request an exact string match when calling a function that

wants a regular expression.

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression.