8 Introduction to Perl

There is no script for this section. Have a look at the commented Perl transcript on the lecture web page and at the manual pages for Perl.

9 Regular expressions

Regular expressions (regexes or REs) are a well-known concept from theoretical computer science to describe formal languages from type 3 of the Chomsky hierarchy. In Unix systems, they occur in a wide range of tools, from text editors and mail programs to programming languages and scanner generators. Essentially they are used for two purposes: as a test (check *whether* a given regex matches a string) and as a transformation device (find those positions *where* a given regex matches a string and modify the string in some way at those positions).

Theory and practice

In formal language theory, regular expressions over some finite alphabet Σ are defined in the following way:

- the regular expression $\emptyset$ denotes the empty set $\emptyset$ of strings,
- the regular expression ε denotes the empty string,
- for every $a \in \Sigma$, the regular expression a denotes the string a,
- if r_1 and r_2 are regular expressions, then $r_1|r_2$ denotes the union of the sets denoted by r_1 and r_2,
- if r_1 and r_2 are regular expressions, then $r_1 r_2$ denotes the set of all words $w_1 w_2$ that are the concatenation of two words $w_1 \in r_1$ and $w_2 \in r_2$,
- if r is a regular expression, then r^* denotes the closure of r under concatenation, i. e., the set of all words that are the concatenation of $n \geq 0$ words in r,
- parentheses may be used to enforce precedence or increase clarity.

In practice, regular expressions as implemented in text editors and programming languages differ from the theoretical version in several ways:

- Some kind of escape machanism has to be used to distinguish characters used literally from characters used as regex operators. (We will mainly use the Perl

convention that alphanumeric characters not preceded by a backslash and non-alphanumeric characters preceded by a backslash are taken literally, whereas alphanumeric characters preceded by a backslash (e. g., `\b`, `\n`, `\1`) and non-alphanumeric characters not preceded by a backslash (e. g., `*`, `[`, `$`) may have a special semantics.)

- Usually, Unix regex engines do not test whether a regex matches a given string, but whether there exists some substring of the given string that is matched. The special operators `^` and `$` can be used to anchor a regex at the beginning or the end of a string.
- Some operators, notably the union operator "|", may be missing.
- Further operators may be added. While some of them may be considered as syntactic sugar (e. g., $r?$ is essentially an abbreviation for $(r|\varepsilon)$), others strictly increase the power of the language (the language accepted by the Perl regex `([a-z]*)\1` is *not* regular in the formal language theoretical sense).

Implementation

A finite automaton consists of a finite set of states Q, an initial state $q_I \in Q$, a set of final states $Q_F \subseteq Q$, and a set of transition rules of the form $q, a \to q'$ or $q, \varepsilon \to q'$ with $a \in \Sigma$ and $q, q' \in \Sigma$. One distinguishes between deterministic and non-deterministic finite automata: In a deterministic finite automaton (DFA), there are no transitions $q, \varepsilon \to q'$ and for every pair (q, a) there is exactly one transition $q, a \to q'$; non-deterministic finite automata (NFA) have no such restrictions. A finite automaton accepts a word w if w can be written as a concatenation $w_1 ... w_n$, where each w_i is either a letter of Σ or ε, and there is a sequence of transitions $q_i, w_i \to q_{i+1}$ with $q_1 = q_I$ and $q_{n+1} \in Q_F$.

It is easy to translate a regular expression into an NFA that accepts the same language – essentially, the states of the NFA correspond directly to subexpressions of the regex. An NFA can be converted into an equivalent DFA by taking the powerset of the state set of the NFA as state set of the DFA; the resulting DFA may be exponentially larger than the original NFA, though, even after elimination of redundant states.

Implementing a DFA is straightforward. For an NFA, there are two choices: Either one mimics the DFA powerset construction, that is one starts with the set $\{q_I\}$ and computes then for each character of the string the set of all possible successor states in parallel. This is possible in time linear with respect to the length of the string, like with a DFA, but with a larger factor. Alternatively, one can use backtracking. The latter approach makes it easy to implement backreferences, and even though it can lead to an exponential runtime, it is the approach that is found in most regex engines of typical Unix tools. The exceptions are (f)lex, most implementations of egrep and awk and some implementations of grep, which use a DFA engine.

Basic regexes: What they match

The regex implementations that one finds in typical Unix tools differ significantly, both with respect to their concrete syntax (in this section, we will use the regex syntax of Perl) and their computational power. Basic regexes, which date back to the editor `ed`, are essentially the intersection of all regex implementations. They are defined as follows:

- An alphanumeric character not escaped by a backslash or a non-alphanumeric characters escaped by a backslash matches itself.
- A character class `[...]` matches each of the characters inside the brackets. Hyphens can be used to specify intervals, e. g., `[A-Za-z]`.
- A negated character class `[^...]` matches every character that is not inside the brackets. Again, hyphens denote intervals.
- A dot `.` matches every character (in some implementation, for instance in Perl: every character except a newline).
- $r_1\, r_2$ matches every concatenation $w_1\, w_2$ of a string w_1 matched by the regex r_1 and a string w_2 matched by the regex r_2.
- If r is a character, a (negated) character class or a dot, then r`*` matches every concatenation of $n \geq 0$ strings that are matched by r. (The restriction to characters, (negated) character classes or dots holds only for basic regexes. In general regexes, the star operator may be applied to arbitrary sub-regexes.)
- The character `^` matches the beginning of the string.
- The character `$` matches the end of the string.
- Parentheses can be used to mark a sub-regex for further reference.

Basic regexes: How they match

If we use regexes not as a test but as a transformation device, it is not sufficient to know *whether* it matches a string – We have to know *how* it matches. If we have a substitution command

```
s/a.*b/c/;
```

that is, "replace the string matched by `a.*b` by `c`", and the string

```
ef a gh b ij a kl b mn
```

then there are three possible replacements: from the first `a` to the first `b`, from the first `a` to the second `b`, or from the second `a` to the second `b`. Which one is used? And similarly, if we have a substitution command

```
s/(.*),(.*)/$2 $1/;
```

that is, "replace the string matched by `(.*),(.*)` by the string matched by the second `.*`, followed by a space, followed by the string matched by the first `.*`", we have to know which substrings are matched by the first and the second `.*`.

Usual regex engines implement "greedy matching":

- Among all possible matches, they take those which start leftmost.
- Among these matches, they take those in which the first sub-regex extends as far as possible to the right.
- Among these matches, they take those in which the second sub-regex extends as far as possible to the right, and so on.

For instance, when we have the regex `[abc]*b[abc]*c` and the string

```
bacabaacaacaabaa
```

then the first `[abc]*` matches the first four characters and the second `[abc]*` matches the sixth to tenth character.

Using regexes

In Perl, the following operations make use of regular expressions:

- *expr* `=~ m/`*regex*`/`
  Searches for a substring of *expr* that is matched by *regex*. In a scalar context, it returns true if a match is found, and false otherwise. In a list context, it returns the list of substrings that are matched by parenthesized subexpressions of *regex*. If "*expr* `=~`" is omitted, the variable `$_` is used. Instead of `/`, another delimiting character can be used; if `/` is used, the `m` is optional.
- *expr* `=~ m/`*regex*`/g`
  Searches repeatedly for substrings of *expr* that are matched by *regex*. In a scalar context, each execution of " `m/`*regex*`/g`" searches for a further match; it returns true if it finds one, and false if there are no further matches. If a list context, it returns the list of substrings that are matched (repeatedly) by parenthesized subexpressions of *regex*. If "*expr* `=~`" is omitted, the variable `$_` is used.
- *variable* `=~ s/`*regex*`/`*replacement*`/`
  Searches for a substring that is matched by *regex* in *variable* and replaces it by *replacement*. If "*variable* `=~`" is omitted, the variable `$_` is used. Instead of `/`, another delimiting character can be used.
- *variable* `=~ s/`*regex*`/`*replacement*`/g`
  Searches repeatedly for substrings that are matched by *regex* in *variable* and replaces them by *replacement*. If "*variable* `=~`" is omitted, the variable `$_` is used.

- `split /`*regex*`/,`*expr*
  Splits the string *expr* into a list of strings using those substrings that match *regex* as separators; returns the list. If the second argument is omitted, the variable `$_` is used.

Examples

Matching tasks:

`if (m/a../) { print $_, "\n"; }`
Prints `$_` if it contains at least one "`a`" followed by two further characters.

`if (m/(a..)/) { print $1, "\n"; }`
The variable `$1` (or `${1}`) contains the string matched by the first parenthesized sub-regex, so this command prints the first three character substring in `$_` whose first character is in "`a`", provided that `$_` contains at least one such substring.

Some simple substitutions:

`s/abc/xyz/;`
The substitution command takes the leftmost match, hence it replaces the first "`abc`" in `$_` by "`xyz`".

`s/(.*)abc/${1}xyz/;`
The `.*` extends as far as possible to the right, so the sub-regex "`abc`" matches the last "`abc`" in `$_`. The variable `${1}` (or `$1`) contains the string matched by the first parenthesized sub-regex, that is, everything before the last "`abc`" in `$_`. Consequently, this command replaces the last "`abc`" in `$_` by "`xyz`".

`s/abc/xyz/g;`
Due to the "`g`" modifier, this command replaces every "`abc`" in `$_` by "`xyz`".

`s/a.*a/b/;`
The `.*` extends as far as possible to the right, hence this command replaces everything from the first "`a`" to the last "`a`" by "`b`".

`s/a[^a]*a/b/;`
The `[^a]*` extends as far as possible to the right, hence this command replaces everything from the first "`a`" to the second "`a`" by "`b`".

`s/(.*)a.*a/${1}b/;`
The first `.*` extends as far as possible to the right, hence this command replaces everything from the last but one "`a`" to the last "`a`" by "`b`".

`s/  */ /g;`
Replaces every non-empty sequence of spaces by exactly one space.

`s/([^ ])  */$1 /g;`
Replaces every non-empty sequence of spaces by exactly one space, except at the beginning of the line.

Note the difference global and iterated replacements or matches:

`$_ = "abababa";  s/aba/aca/g;  print $_, "\n";`
Since the search for the second substitution starts at the place where the last substitution ended, i. e., between the third and fourth character, this command yields `acabaca`. The individual substitutions do not overlap.

`$_ = "abababa";  while (s/aba/aca/) {};  print $_, "\n";`
In each iteration, the search starts at the beginning of the string, so this command yields `acacaca`.

`$_ = "abcabc";  s/a/aa/g;  print $_, "\n";`
This command yields `aabcaabc`.

`$_ = "abcabc";  while (s/a/aa/) {};  print $_, "\n";`
This command leads to an infinite loop.

`$_ = "abacadef";  while (m/(a..)/g) { print $1, "\n"; }`
Prints every three character substring in `$_` whose first character is in "`a`", except for substrings that overlap with earlier ones. So "`aba`" and "`ade`" are printed, but "`aca`" is not (after the regex has matched "`aba`", the next search starts at the following "`c`").

`$_ = "abacadef";  while (m/(a..)/) { print $1, "\n"; }`
Since the "`g`" modifier is missing, the search starts at the beginning of the string again and again, so this command prints "`aba`" infinitely often.

Caveats:

`$_ = "abcabc";  s/c*/d/;  print $_, "\n";`
A star means "0 or more iterations". The substitution command replaces the first sequence of "`c`"s by "`d`", but the first such sequence is the empty string before the first "`a`", so the result is "`dabcabc`".

`$_ = "abcabc";  s/c[^d]/e/;  print $_, "\n";`
The regex `c[^d]` means "`c` followed by a character different from `d`", not "`c` not followed by `d`". The result is "`abebc`", the last "`c`" is not replaced.

Extended regexes: What they match

Since the days of old ed, the regex operator repertoire found in various Unix tools has grown significantly. It should be noted that the availability of the regex operators discussed in this paragraph varies widely. Some are even included in current versions of

sed or grep, while others are mostly limited to Perl. For a complete list of regex operators supported by some tool, consult its manual.

- $r_1\,r_2$ matches every concatenation $w_1\,w_2$ of a string w_1 matched by the regex r_1 and a string w_2 matched by the regex r_2.
- r`*` and r`*?` match every concatenation of $n \geq 0$ strings that are matched by the regex r.
- r`+` and r`+?` match every concatenation of $n \geq 1$ strings that are matched by the regex r.
- r`?` and r`??` match every string that is matched by the regex r and the empty string.
- r`{`n`}` matches every concatenation of exactly n strings that are matched by the regex r.
- r`{`n`,}` and r`{`n`,}?` match every concatenation of at least n strings that are matched by the regex r.
- r`{`n`,`m`}` and r`{`n`,`m`}?` match every concatenation of at least n and at most m strings that are matched by the regex r.
- r_1`|`r_2 matches every string that is matched by the regexes r_1 or r_2.
- Parentheses can be used for grouping and to mark a sub-regex for further reference.

The operators `*?`, `+?`, `??`, `{`n`,}?`, and `{`n`,`m`}?` differ from `*`, `+`, `?`, `{`n`,}`, and `{`n`,`m`}` in that they are *non-greedy*, that is, given several possibilities to match, they take the shortest one. For alternations r_1`|`r_2, the left choice r_1 is the preferred one (in "traditional NFA engines", see below).

There are several abbreviations for character classes: `\w` matches any alphanumeric character and "`_`", `\d` matches any digit, and `\s` any whitespace character. `\W`, `\D`, and `\S` are the complements of `\w`, `\d`, and `\s`. These shorthands can be used both in isolation and inside of character classes, so both "`\d+`" and "`[\dA-Fa-f]`" are possible. The regex `[\d\D]` matches any character, even a newline (this is different from the default behaviour of "`.`" in Perl). The POSIX standard defines a number of abbreviations such as `[:alpha:]` (any alphanumeric character), `[:lower:]` (any lowercase letter), and `[:cntrl:]` (any control character), these are only legal inside of character classes, e. g., "`[[:lower:]]+`" for a sequence of lowercase letters.

Control characters can be written in various ways, for instance Escape (Ctrl-`[`) as `\033` (octal), `\x1B` (hexadecimal), or `\c[` (control char). For some characters, special shorthands are available: `\t` is Tab, `\n` is Newline, `\r` is Return, `\e` is Escape.

Some regexes match the empty string depending on the context: `\A` matches only at the beginning of a string, `\Z` only at end of a string, or before newline at the end, and `\z` only at end of a string. `\G` matches at the end-of-match position of a prior `m//g`. The regex `\b` matches at a word boundary, i. e., before a `\w` character that is not preceded by another

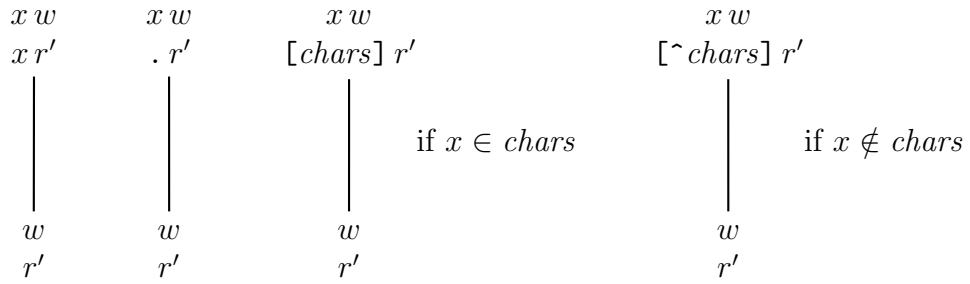
`\w` character or after a `\w` character that is not followed by another `\w` character; `\B` matches everywhere else. If r is a regex, then `(?=`r`)` matches the empty string, provided that it is followed by a string that is matched by r, and `(?!`r`)` matches the empty string, provided that it is followed by a string that is not matched by r.

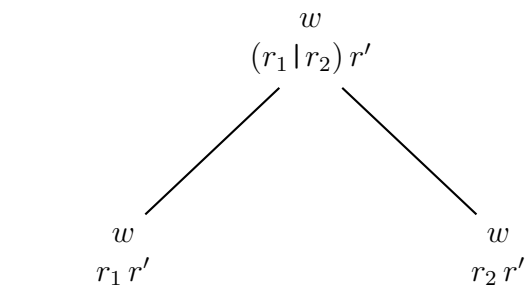
The strings matched by parenthesized sub-regexes can not only accessed in the replacement part of a substitution or in the following code, but also in the regex itself. In this case, they are written as `\1`, `\2`, ..., instead of `$1`, `$2`, ..., though. So, the regex "`\b(\w+) \1\b`" matches any repeated word. For parentheses that are only used for grouping, but not for capturing matched substrings, the notation `(?:...)` is available.

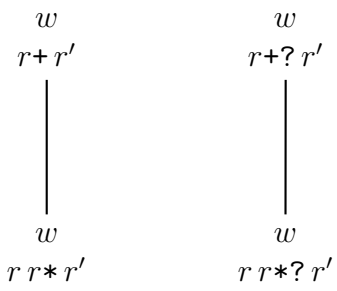
The behaviour of some regexes can be changed by adding modifiers to the match or substitution command: `m/.../i` does a case-insensitive pattern matching; `m/.../m` redefines `^` and `$` so that they match not only at the start and end of the string but at the start and end of any line anywhere within the string; `m/.../s` redefines "`.`" so that it matches every character, even a newline.

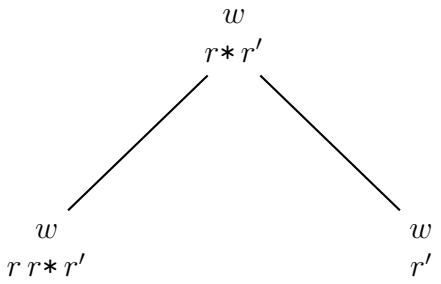
Extended regexes: How they match

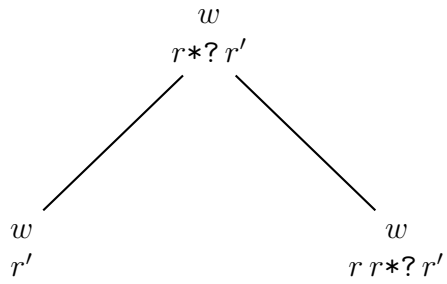
The rule "start the match at the leftmost possible position" is still valid for extended regexes. The rest of the selection process gets more complicated, though. There are regex engines (DFA and POSIX NFA) which strictly implement the rule "the longest of the leftmost matches is returned". This is an easy definition, but it is quite detrimental to the efficiency of the implementation. The behaviour of the more frequent "traditional NFA" regular expression engines can be described by a tree expansion process. Given a string w and a regex r, we start in the root node w, r and apply the following tree expansion rules:

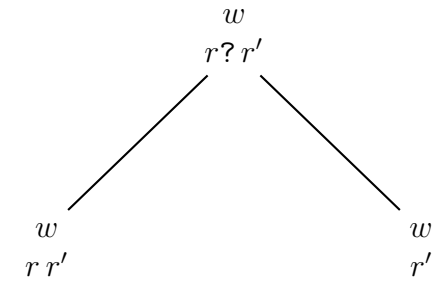


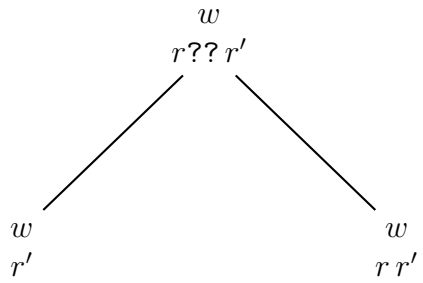

w
$(r_1 \mid r_2)\, r'$
w
$r_1\, r'$
w
$r_2\, r'$

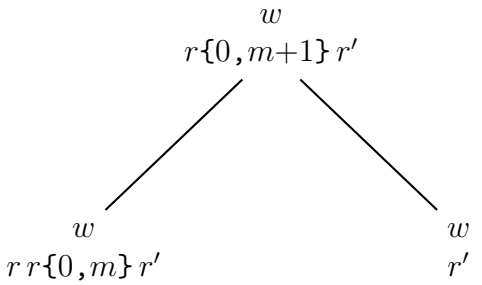

w
$r\texttt{+}\, r'$
w
$r\, r\texttt{*}\, r'$
w
$r\texttt{+?}\, r'$
w
$r\, r\texttt{*?}\, r'$

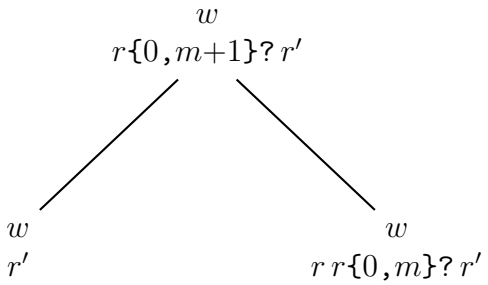

w
$r\texttt{*}\, r'$
w
$r\, r\texttt{*}\, r'$
w
r'

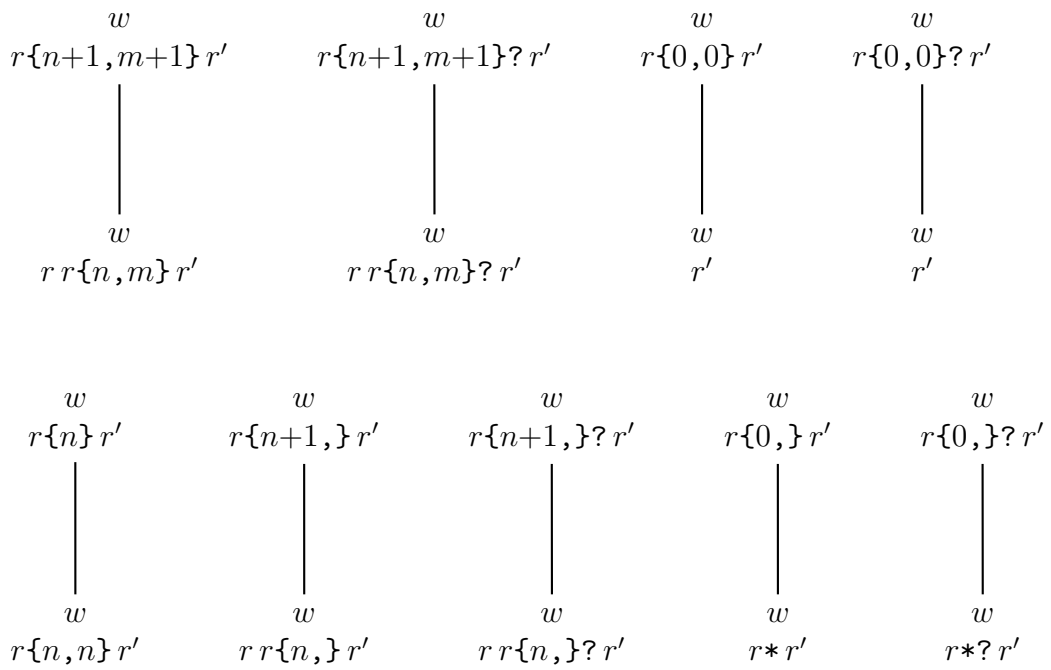

w
$r\texttt{*?}\, r'$
w
r'
w
$r\, r\texttt{*?}\, r'$


w
$r\texttt{?}\, r'$
w
$r\, r'$
w
r'


w
$r\texttt{??}\, r'$
w
r'
w
$r\, r'$


w
$r\{0,m{+}1\}\, r'$
w
$r\, r\{0,m\}\, r'$
w
r'


w
$r\{0,m{+}1\}\texttt{?}\, r'$
w
r'
w
$r\, r\{0,m\}\texttt{?}\, r'$



Starting in a root node w, r we search depth-first, left-to-right for a node whose second component is ε applying tree expansion rules whenever necessary and possible. If w'', ε is the first such node and $w = w'\,w''$, then w' is the string matched by r.

There is one additional mechanism in regex engines that is not reflected in the tree expansion rules above: If the regex r can match the empty string, then the tree expansion of $r\texttt{*}\,r'$ is infinite. To prevent this from happening, the expansion rules for `*` and `*?` are changed when a pair $w, r\texttt{*}\,r'$ or $w, r\texttt{*?}\,r'$ appears on two nodes of the same branch. In this case, the second node may only be expanded to w, r', but not to $w, r\,r\texttt{*}\,r'$ or $w, r\,r\texttt{*?}\,r'$.

Examples

Greedy vx. non-greedy operators:

```
$_ = '\xy{abc} \z{def} \xy{ghi},';
s/\\xy\{([^}]*)\}/\\XY($1)/g;
print $_, "\n";
```

Replaces `\\xy{...}` by `\\XY(...)` using greedy matching.

```
$_ = '\xy{abc} \z{def} \xy{ghi},';
s/\\xy\{(.*?)\}/\\XY($1)/g;
print $_, "\n";
```

The same using non-greedy matching.

```
$_ = '\xy{abc} \z{def} \xy{ghi},';
s/\\xy\{([^}]*)\},/\\XY($1),/g;
print $_, "\n";
```
Replaces `\\xy{...}` by `\\XY(...)` provided that it is followed by a comma (using greedy matching).

```
$_ = '\xy{abc} \z{def} \xy{ghi},';
s/\\xy\{(.*?)\},/\\XY($1),/g;
print $_, "\n";
```
This command does not yield the intended result: The `\},` following `(.*?)` matches the first *closing brace that is followed by a comma*, and therefore, the regex `(.*?)` may match some braces that are not followed by commas.

Left-to-right depth-first search:

```
$_ = "tourist";  m/(tour|to|tourist)/;  print $1, "\n";
```
The first alternative wins.

```
$_ = "baaaac";  m/b((aaa|aa)+)/;  print $1, "\n";
```
After the first alternative has been taken, only one "a" is left, and that is not enough for a second match.

```
$_ = "baaaac";  m/b((aaa|aa)+)c/;  print $1, "\n";
```
Here the trailing "c" forces backtracking.

```
$_ = "abcabc";  m/((ab[ac]*)*)/;  print $1, "\n";
```
Even without alternation, it is not necessarily the longest possible match that is taken.

Advanced techniques

Case-insensitive matching; uppercase-lowercase transformations:
The `i` option can be used for case-insensitive matching or substitution. For instance, the command

```
s/\<BR\>/\n/gi;
```

replaces every "`<BR>`", "`<Br>`", "`<bR>`", and "`<br>`" by a newline.

In the replacement part of a substitution, everything between `\U` (or `\L`) and `\E` is changed to uppercase (or lowercase). For instance,

```
s/(\<\w+)/\U$1\E/g;
```

replaces every word following a less-than sign by its uppercase version.

Regexes in variables:
Variables in the regex part of a matching or substitution command are interpolated. This is useful if some part of a regex is not fixed, if a larger regex should be used in

several matching commands, or if the regex is so large that it is better constructed step by step. Here is an example that builds a regex for balanced strings of parentheses depth 3 and uses it in a substitution: A balanced string of depth 0 is a string that contains neither parentheses nor spaces. A balanced string of depth $n+1$ is either a string that contains neither parentheses nor spaces or a sequence of spaces and balanced strings of depth n that is enclosed in parentheses:

```
$BS='[^()\s]+';
foreach (1..3) {
  $BS = '[^()\s]+|\((?:' . $BS . '|\s+)*\)';
}
while (<>) { s/($BS)/"$1"/go; print $_; }
```

The option `o` instructs Perl to that the regex will not change and therefore should be compiled only once.

Using marks:
Some replacement tasks are more easily performed in several steps using marks (auxiliary characters). For instance, it is easy to replace the last comma in a line by a semicolon, but there is no simple way to replace all commas but the last one by semicolons. However, we can first replace the last one by a mark (here: a NUL character), then replace all other commas, and finally replace the mark by a comma:

```
s/(.*),/$1\000/;
s/,/;/g;
s/\000/,/;
```

(Of course, one should pick a mark that is guaranteed not to occur in the string. If the input is processed line by line, one can take a newline character; otherwise, a NUL character is often a good choice.)

Here is another task: Put all words on the line into double quotes, except the first one. Again we use a mark – first we insert it in front of all words, then we delete the first one, and finally we put those words into double quotes where the mark is left (and delete the marks):

```
s/(\w+)/\000$1/g;
s/\000//;
s/\000(\w+)/\"$1\"/g;
```

Dealing with escape sequences:
It is easy to replace the two character sequence `\n` by an actual newline, but if the backslash itself can be escaped by another backslach, things get tricky: `\n` is an escaped `n` and should be replaced, `\\n` is an escaped backslash followed by a non-escaped `n`, which should not be replaced, `\\\n` is an escaped backslash followed by an escaped `n`, which should again be replaced, and so on. A variation of the marking technique is helpful here. We insert a space after each escaped character. After this step, the string

```
n\n\\n\\\n\\\\n,
```

is turned into

```
n\n \\ n\\ \n \\ \\ n,
```

and escaped backslashs and escaping backslashs can be easily distinguished. Now we can perform the intended substitution (taking the inserted space into account), and finally we undo the insertion of spaces:

```
s/(\\.)/$1 /g;
s/\\n /\n/g;
s/(\\.) /$1/g;
```

TeX control sequences are handled in a similar way. A TeX control sequence consists either of a backslash and a non-alphabetical character or of a backslash and a sequence of alphabetical characters. To replace the control sequence `\m` safely by `\M`, we first insert a space after each backslashed non-alphabetical character, then insert a space after each backslashed sequence of alphabetical characters. After this step, the string

```
m\m\mu\\m\\\m\\\multicolumn,
```

becomes

```
m\m \mu \\ m\\ \m \\ \multicolumn ,
```

We see that every backslashed `m` has been turned into the three character sequence "`\m `" and can now perform the intended substitution (taking the inserted spaces into account). Finally we undo the insertion of spaces:

```
s/(\\[^A-Za-z])/$1 /g;
s/(\\[A-Za-z]+)/$1 /g;
s/\\m /\\M /g;
s/(\\[A-Za-z]+) /$1/g;
s/(\\[^A-Za-z]) /$1/g;
```

Computed replacement strings:
Variables in the replacement string are interpolated automatically; this holds even for array and hash elements. For instance, the following command replaces TeX codes for German umlauts by the appropriate letters:

```
%h = ( '\"a' => 'ä', '\"o' => 'ö', '\"u' => 'ü',
       '\"A' => 'Ä', '\"O' => 'Ö', '\"U' => 'Ü' ) ;
s/(\\"[AOUaou])/$h{$1}/g;
```

Arbitrary Perl expressions are usually not evaluated in the replacement string; evaluation can be forced, though, with the "`e`" modifier. The following statement changes the format of all real numbers to three decimal digits:

```
s/(\d+\.\d+)/sprintf "%.3f", $1/ge;
```

The command

```
s,(([-(][-( ]*)?\d([\d-+*/() ]*[\d-+*/()])?),eval($1),ge;
```

evaluates everything that looks superficially like an arithmetic expression.

To replace every tab by the appropriate number of spaces (assuming tab width 8), the following command can be used:

```
while (s/\t+/" " x (length($&) * 8 - length($`) % 8)/e) {}
```

The variable `$&` contains the complete matched substring; `$`` contains the string *before* the matched substring.

Even nested substitutions are possible:

```
s/("[^"]*")/do {$m = $1; $m =~ s#,#;#g; $m;}/ge;
```

This command replaces every comma within double quotes by a semicolon. The `do {...}` construct turns a sequence of statements into an expression. The first regex matches a string that is delimited by double quotes; this string is first assigned to `$m`, then commas are replaced by semicolons in `$m`, and finally the new value of `$m` becomes the replacement string of the outer substitution.