AT Computing
Arnhemsestraatweg 33
6881 ND Velp
The Netherlands

Regular Expressions

Version: 1c

Contents

1 | Introduction

Regular expressions form a specification language designed to find a specific piece within a text. The search can be performed by your computer, but it requires a precise specification of the piece you wish to retrieve. To that end, you define your query in the *regular expression language*. A notation written in this language is called a *regular expression* and we will often abbreviate this to *regex* throughout the remainder of this article.

Regular expressions belong to the most powerful mechanisms of UNIX and are used whenever a search in a text file takes place. For example, all editors have a *search-replace* operation. The *search* part of such an operation is always written as a regex.

To be able to demonstrate the use of regexes we have a text file named `words`, which contains the following:

```
earth
talking.
tree
sky
horror
whenever
              <-- this is a completely empty line
try
wheel
break.
woolen
better
world
```

Besides a text file to sift through, we require a search tool for our demonstration. UNIX comes with many such tools, from which we choose `grep`. To use `grep` a user needs to provide it with a search pattern (a regular expression) and one or more text files. The `grep` command will look inside the text files for a piece of text *matching* the given regex. Any line(s) where a match is found will than be output.

At this point it is important to distinguish between regexes as specification language, and the specific behaviour of a certain command we happen to use for demonstration purposes. *Output the line after* a match is found is behaviour specific to `grep`. Had we used an editor with a search/replace command, the matching text would have been replaced by something else. But this article is about the searching itself: how to specify what needs to be found, and how is the matching performed exactly? All commands which support regex notation, and which we could use for demonstration purposes, have this notation and matching functionality in common.

2 | Three generations regex

The UNIX regexes have been designed in the first half of the seventies. After that, new ideas led to several enhancements. Nowadays, *three* generations of regexes can be encountered:

- BREs: the *Basic* Regular Expressions. This is the basic form used in most UNIX commands.
- EREs: the *Extended* Regular Expressions. These expand upon the possibilites of the BREs, but the notation comes with a few key differences. So you'll need to know whether a given command expects either BREs or EREs. Most commands support BREs by default, and can be switched to EREs with a flag. The POSIX standard defines precise specifications for both BREs and EREs. But when the term *POSIX regular expressions* is encountered, this usually means that EREs are expected.
- The designer of the programming language Perl introduced many new ideas in the area of regex support. *Perl* Regular Expressions are the youngest of the three generations. For this generation, the same is true as for EREs: *if* a command supports Perl regexes, this usually has to be enabled by setting a special flag.

First, we will explain the Basic Regular Expressions.

3 | Basic form

The simplest regexes are literal texts: whenever a simple piece of text should be found, it is written as-is. When searching for `a`, the output of `grep` are the lines containing `a`:

```
$ grep 'a' words
earth
talking.
break.
```

Using `grep` to search for `ea` yields all lines containing `ea`:

```
$ grep 'ea' words
earth
break.
```

A regex like `talk` yields all lines containing `talk`; this can also be part of a larger word (here: `talking`). So, strictly speaking `grep` is not searching for `talk` as a word, but for `talk` as *string*.

Almost all characters used in a regex are taken *literally*. With exception of a few characters that have special meaning, which enable finding patterns, rather than literal text.

3.1 | Anchoring in regular expressions

For example, `grep` can search for lines *starting* with an `a`. To accomplish this, the regex should start with the special symbol `^`.

The regex `^w` means: search for a `w` at the start of a line. So:

```
$ grep '^w' words
whenever
wheel
woolen
world
```

Likewise, a `$` at the end of a regex means that the string we search for should match at the end of a line. Not even a whitespace character is allowed after it. So, the regex `y$` is used to find lines *ending* with a `y`.

```
$ grep 'y$' words
sky
try
```

If ee would find two consecutive es, then ^$ should find empty lines. After all, on an empty line the end immediately follows the start of it. And, indeed:

```
$ grep '^$' words
              <-- this output is the empty line from our words file.
```

Fixing a regex to the start or end of a line is called *anchoring*, and the special symbols ^ and $ are called *anchors*. It is important to note that anchors specify a *position*: the position *in front* of the first character on the line, or the position *after* the last character on the line, respectively. Therefore, anchors themselves have a width of zero characters.

^ as the first character in a regex defines that the rest of the regex should match at the beginning of a line.

$ as the last character in a regex defines that the preceding part should match at the end of a line.

Text files made for Microsoft Windows are problematic with the $ anchor. Under the hood those files are different from UNIX/Linux text files. The difference is how a line ending is encoded. A UNIX file has an ASCII linefeed code as line ending, whereas a Windows file uses an ASCII carriage return code, followed by an ASCII linefeed. If you would look for text$, the carriage return inside a Windows text file will cause the match to fail. There are many tools available to convert text files, but without conversion you will never be able to match such a $ anchor.

3.2 | Alternatives on a single position

Could grep be asked for all places containing either an a or an l? This is possible using a construction with which one can specify a choice of certain characters. By writing [al] we specify a *single* character, which is allowed to be either an a or an l. So all acceptable characters are written between square brackets. Searching for an a or an l can be done as follows:

```
$ grep '[al]' words
earth
talking.
wheel
break.
woolen
world
```

It shows that some lines contain both an a and an l, but every line found contains at least one of those letters.

With the regex [aeo][aou] we specify two *consecutive* character positions (because each set of square brackets counts for a single position), where the first position is allowed to contain an a, e, or o, and the second position an a, o, or u. Please note that in a regex each position can be specified by chaining together individual specifications, just like in our example here.

grep finds all lines ending in e or l with:

```
$ grep '[el]$' words
tree
wheel
```

3.3 | The complementary set

Let us make it even more general: how to search for lines *not* ending on an `e`? Do we need to explicitly specify all possible characters, excluding `e`? That would be very cumbersome and error-prone. Luckily, there is a notation with the meaning: *not*-`e`. It looks like this: `[^e]`. A little more specific, this means: any arbitrary character, but not the characters specified between the square brackets (here: *not* an `e`). Please note that ^ could mean two different things in a regex: if the regex begins with it, it is taken to be the anchor for a line start, but just after a square opening bracket it *negates* the character set between the brackets.

So, all lines *not* ending in `e` can be found using:

```
$ grep '[^e]$' words
earth
talking.
sky
horror
whenever
try
wheel
break.
woolen
better
world
```

Please note: the empty line will not be found, because that line does not contain any character, not even a non-`e`.

Another example: all lines starting with a non-`e` are:

```
$ grep '^[^e]' words
talking.
tree
sky
horror
whenever
try
wheel
break.
woolen
better
world
```

The first ^ is the begin-of-line anchor. The second ^ negates the set of characters between the square brackets. We still need to be careful, though, because `grep '[^e]'` is not doing the opposite of `grep 'e'`. Consider:

```
$ grep '[^e]' words
earth
talking.
tree
sky
horror
whenever
try
wheel
break.
```

(continues on next page)

(continued from previous page)

```
woolen
better
world
```

After all, every line contains a non-e, except, of course, an empty line.

3.4 A single arbitrary character

It is also possible to allow for any character for a given position. So, that is a wider definition than an explicitly specified set of characters between square brackets.

For example: search for a group of four letters, of which the first should be a w, and the last should be an l. For the two positions between those letters we accept *any* character. In regular expressions this can be written as w..l, where the dot (.) is the notation for *any character here*.

```
$ grep 'w..l' words
woolen         <-- here wool matches
world          <-- here worl matches
```

3.5 Degrading special characters

How can we find lines ending with a dot? Not by using .$, because that will show all non-empty lines (lines containing an arbitrary character just before the end of the line). For this situation a systematic solution exists: a character that has a special meaning within the regular expression notation can be neutralised into being its literal self, by prepending it with a backslash \. Please note that in doing so, the backslash itself has become a special character as well. So, searching for a literal backslash has to be done like so: \\.

Lines ending in a literal dot can be found as follows:

```
$ grep '\.$' words
talking.
break.
```

Note that using a backslash to remove any special behaviour from a particular symbol is also used in the UNIX/Linux shell language. The shell language uses quotes as an alternative notation for this same *demotion* purpose as well. The regex language does not use quotes for this.

3.6 A single character in regular expressions

In summary, there are four methods to specify a character in a regex. Ordered from most to least accurately specified, they are:

a a character that should occur literally (here: an `a`).

\. also a literal character, especially if that character means something special within a regex (in this example a literal dot).

[abc]
the desired character should be one of the specified set (here: an `a`, `b`, or `c`).

[^0123]
for this position, we are looking for a single character, as long as it is *not* one from the set between the square brackets. In this example a `0`, `1`, `2`, or `3` is not allowed.

. an arbitrary character.[1]

A word of caution here, however, because the order of characters is defined by the *ASCII character set* standard. *Lower case letters* are ordered consecutively, just like the ranges for *upper case letters* and *digits*. Never write something like `[a-Z]`, however; that will go wrong, because it is assumed then that upper case letters follow the lower case ones, while in ASCII that order is reversed. Instead, use `[a-zA-Z]` So, the hyphen simply connects two characters to denote the whole range. In case you want to find a literal hyphen, write it as the last character between the square brackets.

Accented letters are unavailable in the ASCII set. Later, more information about that will be presented.

3.7 The Kleene star

Now suppose we are looking for words where a `w` is followed by any number of `o`'s. So, we are looking for `wo`, `woo`, `wooo`, etcetera.

Such a repetition can be specified by writing a *quantifier* after a character, which defines how often that particular character may be present. The most frequently used quantifier is the *star*[2]. If we place a star after a character, we accept that character *zero* or more times in a row.

So we are looking for:

```
$ grep 'wo*' words
whenever
wheel
woolen
world
```

There are several lines containing a `w` that is *not* followed by an `o`. This is because the `w` is followed by *zero* times an `o` in those cases. If we explicitly want to prevent such *zero* cases to be found, we need to write:

[1] Ranges between square brackets can be abbreviated using a hyphen: for example, `[a-z]` means: any lower case letter from the alphabet.

[2] A character that could sometimes be confusing is the TAB (ASCII table value 9). Within a regex, the TAB counts as a single character. On your screen, however, typing a TAB causes your cursor to move to the next tabulator stop, which results in your on-screen cursor to possibly advance *multiple* spaces.

```
$ grep 'woo*' words
woolen
world
```

that is, at least one `o`, and possibly more. The first `o` is explicitly wanted, and the second one has the `*`. Put another way: we want at least one `o`, optionally followed by multiple `o`'s.

That shows the real power of the `*`: it specifies a certain component to be *optional*. Using the `*`, *optional parts* can be included in a regex.

For example, when we talked about `$` being the end-of-line anchor, we warned about *the preceding expression should match at the end of the line. Not even a whitespace character is allowed after it*. Suppose we want to be a bit more forgiving and allow for possible spaces at the end of a line. We will let our regex end with ` *$` (space star dollar), to express our wish to allow *optional* spaces just before the end of the line.

The `*` works on the preceding element; that can also be a specification in `[....]` form. So:

```
$ grep '[aeio][aeio][aeio]*' words
earth
tree
wheel
break.
woolen
```

searches for *two or more* occurrences of `a`, `e`, `i`, `o`: two occurrences of `[aeio]` are specifically requested, with optionally (zero or) more `[aeio]` after that. The star is working only on the last of the three `[aeio]` pieces, because that is the optional part. The first two `[aeio]` pieces tell what *must* be there.

Please note the correct interpretation of `[aeio][aeio][aeio]*`. It does not mean: *two or more times ``a``*, or *two or more times ``e``*, etc., but: *two or more elements, with each element matching ``[aeio]``*. So, every possible mix of `a`'s, `e`'s, `i`'s, and `o`'s is valid, as long as it is at least two characters long.

- `*` after a character (or field) means that that character may occur *zero* or more times at that position.

Now suppose, again, that we want to find the lines without any `e` in them. The requirement is now that from the first character to the last, it should not be an `e`. So:

```
$ grep '^[^e]*$' words
talking.
sky
horror

try
world
```

Because of the *anchoring*, the `^` at the start and the `$` at the end of the expression, we are preventing the problems we had earlier with `grep '[^e]'`. We should not forget to include the `*`, because without it we would be looking for a line with exactly one character in it, which, on top of that, should not be an `e`. Note that the empty line is also matched now, because the star allows for *zero* times a non-`e`.

In the same way we can look for lines containing just the letters `k` to `y`:

```
$ grep '^[k-y]*$' words
sky
              <-- the empty line is found as well
try
```

If we want to omit the empty line, it becomes:

```
$ grep '^[k-y][k-y]*$' words
sky
try
```

Let us search for lines starting with an `t` and ending in `e`. We allow for any number of arbitrary characters between the `t` and `e`. At first we try this:

```
$ grep 't.*e' words
tree
better
```

We appear to find more than we bargained for, because we have forgotten about anchoring. For example, we find `better`, because our regex does not demand the line *begin* with a `t`. Better:

```
$ grep '^t.*e$' words
tree
```

How do we find lines containing two times an `e`? As follows:

```
$ grep 'e.*e' words
tree
whenever
wheel
better
```

Is this a search for *exactly* two times an `e`? No, `whenever` contains *three* times an `e`. If we really want *exactly* two times an `e`, we will have to rule out that any other character in the line is an `e`. In order to include all characters in the line in our specification we use anchoring.

```
$ grep '^[^e]*e[^e]*e[^e]*$' words
tree
wheel
better
```

The regex above can be read as follows: from the beginning of the line, first *zero or more times not an ``e``* (that is the part `[^e]*`). After that, the first `e` followed by *zero or more times not an ``e``*, then the second `e`, followed by *zero or more times not an ``e``* until the end of the line. Or, put yet another way: because of the anchors this regex covers the whole line and buried in the regex are the two `e`'s we are looking for. In front of those, between and after them, only non-`e` characters are allowed.
Schematically drawn:

```
^     e     e     $
 [^e]* [^e]* [^e]*
```

Constructs similar to this are encountered quite often, so it is worth the trouble to study this example carefully.

3.8 More quantifiers

The Kleene star following a regex denotes that the regex is allowed *zero or more* times. There is another notation available, to express different amounts of repetitions, instead of *zero or more*. It looks like this:

```
regex{m,n}
```

The desired amount of repetition is specified after the regex with backslashes and curly braces. `m` and `n` are whole numbers (>= `0`, and specify the range of possible repetitions. These can we expressed in two ways:

```
regex\{m,\}     -> m or more repetitions (at least m)
regex\{m\}      -> exactly m repetitions
```

So, `regex*` is the same as `regex\{0,\}`

The backslashes work exactly opposite from what we have seen before. Normally, a backslash removes special behaviour from a character. In this case, it makes a normal character (a curly brace) special.

3.9 Grouping with parentheses

We have seen that quantifiers work on the preceding character. To make them work on a *group* of characters, place parentheses around them. With BREs those parentheses need to be preceded by backslashes.

```
\(at\)*
```

searches for zero or more occurrences of the string *at*. Later we will see that grouping with parentheses has a useful side effect.

Just like in the previous paragraph, the backslashes work backwards to their normal operation: they make the parentheses *special*. So this is a promotion instead of a demotion. The reason for the special notation stems from the fact that grouping was not part of regular expressions from the start. Originally, parentheses and curly braces were *normal* characters. To remain compatible, a *promotion* notation would have to be devised. In the Extended Regular Expressions (EREs), which we will discuss later, this has been corrected: parentheses and curly braces are special there, so then you will need backslashes to *demote* them.

3.10 Character classes

Earlier we saw that between square brackets a hyphen can be placed between to characters to denote a range, for example `[0-9]`, or `[a-zA-Z]`. But this is a range according to the ASCII standard, and that contains only a limited amount of characters. Letters with accents (official term: with diacritics) are lacking, just like all kinds of national extensions like the ß, the ð or the ¿.

That is why a number of *character classes* were made, with which particular sets of characters can be specified in a straightforward way, including all non-ASCII variations.

Regular expression	Meaning
`[:alpha:]`	a letter
`[:lower:]`	a lower case letter
`[:upper:]`	an upper case letter
`[:digit:]`	a decimal digit
`[:xdigit:]`	a hexadecimal digit
`[:alnum:]`	a letter or digit
`[:punct:]`	all punctuation
`[:graph:]`	a visible, printable char (no space/tab)
`[:print:]`	everything, except control characters
`[:cntrl:]`	a control character
`[:blank:]`	a space or a tab
`[:space:]`	space, tab, form feed, vertical tab, newline, carriage return

Such a character class, as written above, can only be used inside square brackets. So please take care not to get confused by all those square brackets. For example:

```
[[:alpha:]2-5]
```

specifies a position containing a letter, or a digit in the range 2 to 5.

It is important to realise that these character classes enable us to specify characters outside the ASCII range, but that we are still limited to the Latin alphabet. Expansion to other alphabets, or even to alphabet-less languages (Chinese, Japanese, Korean), is quite a different matter.

3.11 | Basic Rules

As the expressions become more complicated, it is quite possible that different interpretations arise. Here are some basic rules to aid in resolving conflicts:

- A regex only matches on a *single* line.
  A match will never extend beyond the end of a line, to include the beginning of the next line. To put it another way: for every input line, the search will *reset*.
- Within a line, only the first match is taken:
- This *first match* rule is only valid in editor *search/replace* situations. It is only influencing the replacement, not the search.
- Usually, there are possibilities within the editor's command language or user interface to circumvent the *first match* rule.
- Quantifiers are *greedy*. They always try to take the largest matching piece of text.
- For example, in a search for `o*l` in the text line `woolen blanket`, not only will `ool` match, but at the same position also `ol`, and even just the `l`. In situations like this *the longest match rule* governs.
- The *longest match rule* has lower precedence than the *first match rule*. Meaning if the first match in a line is shorter than a match further down the line, *still* the first match will be taken.
- Matching is case sensitive. Usually there is an option to specify case_in_sensitive matching. But that will be an option for the specific command, or for the programming language used.

3.12 | Back-referencing

We have said before that grouping characters between parentheses has a side effect. The side effect is that whatever is between the parentheses is not only grouped together, but also *remembered*. This remembered portion can be reused in the same regex. This is called *back-referencing*.

Let us look at an example. Earlier we were looking for lines containing `ea`. Now suppose we want all lines where a certain character occurs twice, consecutively. We could do this as follows:

```
$ grep '\(.\)\1' words
tree           <-- matches the part ee
horror         <-- matches the part rr
wheel          <-- matches the part ee
woolen         <-- matches the part oo
better         <-- matches the part tt
```

In this example we are looking for an arbitrary character, followed by the same character: `1` refers to what's between parentheses. This way, up to nine groups can be back-referenced.

So, searching for sets of four characters, with the first and last ones equal to each other:

```
$ grep '\(.\)..\1' words
horror         <-- matches the part orro
better         <-- matches the part ette
```

- `\(`*regex*`\)` defines a group of characters, of which the contents are the match of the text with *regex*. The parentheses do not influence the match directly. Put another way: a regex with or without the parentheses matches the same way. But: whatever is matched between the parentheses will be remembered for later reference. See the next point.
- `\n` (*n* is a digit between 1 and 9) matches the same characters that formed a match inside the *n* th group of parentheses.

If there are multiple groups, the groups are numbered in the order of their opening parentheses from left to right. Example with nested groups:

```
$ grep '\(.\(.\)\)\2\1' words
horror         <-- matches the part orror
```

So, all lines are found that contain a sequence of five characters, where the first two characters are the same as the last two, and the second and third characters are equal as well. Note that this is not the same as: I mean the same regex that I wrote inside the parentheses. It is not about the regex itself, but about a repetition of what that regex has found. This example also shows that nesting of groups is allowed.

In many editors it is possible to use this back-reference notation `\n` in the replace part of a search/replace operation.

3.12.1 | EREs: Extended Regular Expressions

Some years after the introduction of *normal* Basic Regular Expressions in UNIX, the Extended Regular Expression came into being, bringing improvements and extensions. They are not entirely compatible with the Basic REs. That is why existing UNIX commands were never *silently* upgraded to use these new possibilities: they either need a special flag (like the `-E` flag for `grep`, or the `-r` flag for the GNU version of `sed`), or it is a given that specific commands use only EREs (like `awk`).

Looking back at what we already said about the BREs, the most important changes are:

- Round parentheses are no longer preceded by a backslash, but they do not support back-referencing. There are implementations that do, however, but that is not official.
- With the general quantifier notation using curly braces (`{` and `}`) the need for preceding backslashes has also disappeared. Curly braces are now *always* special, making it necessary to include a backslash whenever a *literal curly brace* must be matched. The curly braces themselves kept their previous meaning of specifying repetition factors.
- The Kleene star `*` is no longer the only symbolic quantifier: `+` and `?` are added to the repertoire.
- The most important extension is that multiple regexes can be connected with a Boolean *or*.

A well-known implementation of EREs is the *regex* library by Henry Spencer. It is not fully compatible with the POSIX standard for EREs, but mostly so. This library is used by MySQL, among others. MariaDB, MySQL's *successor*, uses Perl-compatible regexes.

3.13 | Grouping in EREs

Using parentheses allows building *multi-stage* patterns:

```
([a-z]+[0-9]+ )+
```

Between the parentheses it says we are looking for one or more letters, followed by one or more digits, followed by a space. But outside the parentheses the + means we accept one or more occurrences of that whole pattern.

3.14 | + and ? quantifiers in EREs

The Kleene star `*` in its meaning of *zero or more repetitions of the previous thing* stayed available. But now, we also have the +, meaning *one* or more repetitions of the previous thing.

Earlier, we saw the following example:

```
$ grep '^[k-y][k-y]*$' words
```

and this can now be written more concisely as:

```
$ grep -E '^[k-y]+$' words       <-- flag -E switches grep to EREs
```

Moreover, we now also have the `?` quantifier, meaning *zero or one occurrance*.

A consequence of this is that from now on, the + and `?` have become special characters as well. Should you want to find them literally, they have to be preceded by a backslash.

Please note that + is equivalent to `{1,}`; and `?` is equivalent to `{0,1}`. In the Basic Regular Expressions this specification with curly braces is available as well, but there the braces must be preceded by `\`.

3.15 | Combining two regexes with Boolean OR (alternation)

The most important new feature of EREs is the possibility to chain multiple regexes together with a `|`.

```
aRegex|anotherRegex
```

tries to match either *aRegex* at this position, or *anotherRegex*. A little more complicated is:

```
firstRegex(secondRegex|thirdRegex)fourthRegex
```

This regex searches for a match for *firstRegex*, followed by either a match for *secondRegex* or *thirdRegex*, followed by a match for *fourthRegex*.

Note that in this example the two alternatives must be grouped (in parentheses) to distinguish them from the rest of the regex.

3.16 | Filename wildcards versus regular expressions

Filename wildcard notation and regular expression notation are often confused. But they are two very different notation systems!

For example, getting all filenames that start with an `a` in your current directory works in the shell like this:

```
$ ls a*
```

but if the same task would be carried out using a regex, it would be:

```
$ ls | grep '^a'
```

Filename expansion is done by the shell, after you hit *enter* to complete a command line, and before the command is actually run. In UNIX jargon this expansion is also called *globbing*. The shell does not do regular expressions, and *normal* commands do not do filename expansion (globbing), except for some exceptions, like `find`, `tar`, and `locate`[3].

Filename wildcard notation works with the following characters:

[3] This star has been named after mathematician Steve Kleene, who invented the theoretical foundation for regular expressions: https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

```
*      a range of zero or more characters
?      exactly one character
[...]  one character from the set between square brackets
[!...] not one character from the set between square brackets
```

The `[...]` notation happens to be a component present in both notational systems, even with the same meaning. But variations with it already work differently. If some characters are *unwanted* at a certain location, in regex notation `[^...]` is used, whereas `[!...]` is the correct form when using globbing. And according to the POSIX standard for filename expansion the hyphen between the brackets must behave according to the user's current *locale*. That could mean that the globbing behaviour could become case-insensitive. So, `[a-z]` would also match upper case letters in the filename in such a case.

4 | Perl Regular Expressions

At this point we will make a big jump forward: from Basic and Extended Regular Expressions and head to the third generation: Perl Regular Expressions. In doing so, it becomes more complicated. It might be wise to stop reading at this point, and return when you are more experienced in using BREs and EREs. The Perl Regular Expressions are mainly useful for programmers. But they are also available in heavy-weight tools, like the URL rewrite module for the Apache web server.

The scripting language Perl owes much of its popularity to the application of regular expressions. Perl has been available since 1987, and reached version 5 in 1995. It was this particular version that brought many enhancements to EREs. It has been the benchmark for others to follow, as far as regex support was concerned.

Most scripting and programming languages support regexes in one way or another. Quite often the Perl syntax is used for this. Sometimes the support is built into the language, but most of the time the implementation is inside a library.

This chapter brings most of Perl's extras to your attention.

In order to demonstrate the Perl regexes in a way resembling what we did with `grep`[4], we use the following command line form at a UNIX shell prompt:

```
$  perl -ne 'print if (/regex/)' text file
```

This lets `perl` show the lines from the `text file` which match the given regex. Note that in this case the regex is placed between forward slashes. In Perl, the slash is used as a separator for literal regexes, just like literal strings are placed between quotes in most languages.

4.1 | Quantifiers

A simple enhancement to the existing `{n,m}` variations is:

```
{,m}
```

This means: *zero to m times* the previous thing. Some regex implementations do not use this form, but then, of course, you could simply write `{0,m}`.

A much more important addition is the possibility to make quantifiers *lazy*. Until now, we saw quantifiers which always want to match *as much characters as possible*, and are therefore called *greedy*. So, given the following text:

[4] If *normal* commands perform globbing, it is always because they need to search somewhere else than in the current directory. For example, in a whole directory tree, inside a tar file, or in a database of names.

```
<p>An html-paragraph.</p><p>The second paragraph.</p>
```

applying this regex: `<p>(.*)</p>` yields a match for the group (the part between parentheses):

```
An html-paragraph.</p><p>The second paragraph.
```

So, not just:

```
An html-paragraph.
```

Maybe that longer match was not what we wanted. If we would have wanted to *catch* the text between the first pair of `<p>...</p>`, we should have made the `*` in our regex *lazy*. That means: it would have matched as *little* characters as possible, while still satisfying the pattern.

A quantifier can be made lazy by appending a `?` to it.

So, if we adjust our regex that way, the group in:

```
<p>(.*?)</p>
```

will most certainly only contain

```
An html-paragraph.
```

because matching stops at the first `</p>`.

There is even a third form of quantifiers, whose behaviour is like the lazy ones, but subtly different. A quantifier is made *possessive* by appending a + to it. It could then look like the following: `*+`, or `{n,m}+`, or `?+`, or even `++`

An explanation of possessive quantifiers is outside the scope of this article. More in-depth knowledge about the inner workings of regex implementations is necessary, before they can be properly explained.

4.2 Grouping

Grouping works exactly the same as with EREs, like already shown above. In Perl Regular Expressions the matched characters inside a group are always remembered, unless told otherwise.

New in Perl Regular Expressions is that it is now possible to specify explicitly that any matched characters between parentheses do not need to be remembered for later back-referencing. Remembering takes time and costs memory, after all. So if the only thing we are after is simply grouping some characters, we can save the system some work, and the search could be performed more quickly.

A group for which there is no need to remembered any matched text is called a *non-capturing group*. A group is made non-capturing by placing a question mark and a colon directly after the opening parenthesis:

```
abc(?:de)*fg
```

In this regex `de` is being grouped, so the `*` works for the whole group. The substring `de` does not need to be remembered, so we made the group non-capturing. The regex describes: `abc`, followed by zero or more times `de`, followed by `fg`.

Warning: non-capturing groups *do not count* against the numbering for capturing groups! So, only count those groups of which the opening parenthesis is *not* followed by a question mark.

In the regex:

```
ab(cd)*e(?:gh)*i(jk)lm\1\2
```

`\1` refers to `cd`, and `\2` to `jk`. The group containing `gh` does not count!

Atomic grouping is not used very often, and to be able to clearly explain what it is, more in-depth knowledge is needed about the inner workings of regex implementations. In this article we only show how they look. An atomic group is a group where directly after the opening parenthesis a question mark and a > character are placed:

```
(?>Regex)
```

Atomic groups do not count against the group numbering, just like non-capturing groups.

4.3 Backreferences

Backreferences are written in the same way as in BREs and EREs. So, `\1` up to `\9` refer to the matched characters in the corresponding group. Remember: grouping in BREs is written like `\(...\)`, whereas in EREs it looks like `(...)`.

To be able to use more than nine groups, an alternative notation exists: `\g{1}` up to `\g{99}` refers to group 1 up to group 99.

4.4 Character classes

Some shorthand notations were introduced for often-used character classes, which are easier to write than their usual `[...]` counterparts.

To specify *white space*, `\s` may be written. *White space* includes: space, tab, linefeed, carriage return, and formfeed. So, `\s` is equivalent with `[ \t\n\r\f]`. Some implementations also include the vertical tab (`\v`).

As mentioned earlier, *negating* a character class using the square bracket notation works by placing a caret `^` as the first character after the opening bracket. The shorthand notations introduced here can be negated as well, by using the upper case version of the letter. So, if you need to indicate you want anything *but* a white space character at a certain position, you write `\S`, with a capital `S`.

In the same vein, `\d` denotes a digit, equivalent with `[0-9]`. And `\D` means: anything *but* a digit, so: `[^0-9]`.

Lastly, `\w` stands for a *word character*, which is defined to be a letter, a digit, or an underscore: `[a-zA-Z0-9_]`. *Note*: if the regex implementation supports Unicode, the concept of a *letter* or *digit* can become considerably wider and could include accented letters, or even characters from other alphabets, like Greek, Cyrillic, Arabic, Hebrew, etc.

And, of course, negation works by writing `W`, meaning `[^a-zA-Z0-9_]`

Summarised:

Table 1 Shorthand notation for often-used character classes

Notation	Meaning	Equivalence
`\s`	whitespace	`[ trnf]`
`\S`	anything but whitespace	`[^ trnf]`
`\w`	word character	`[a-zA_Z0-9_]`
`\W`	anything but a word character	`[^a-zA-Z0-9_]`
`\d`	digit	`[0-9]`
`\D`	anything but a digit	`[^0-9]`

In regex implementations which support Unicode, arbitrary characters can be matched using their Unicode code point. The copyright sign (©), for example, is Unicode code point U+00A9. If you want to include it in a regex, write (in most implementations):

```
\u00a9
```

This notation does not work in Perl or in PCRE,[5] where the following should be written instead:

```
\x{a9}
```

Note that the leading zeroes can be omitted in this case. This notation cannot be confused with the quantifier `{n}`, because `\x` is not a valid independent construction. When you had written the following:

```
\x{2500}{50}
```

this would be interpreted as 50 times the Unicode code point U+2500.

Matching Unicode code points can be very tricky, however. One thing to watch out for is characters with multiple representations, like accented letters, for example. They can either be represented by a single code point for the whole character, or by two code points! One for the bare letter without accent, followed by one for the particular accent that has to be combined with it.

So, an `é` can be represented by either U+00E9 (for the complete character), or by the combination of `e` (U+0065), followed by the accent (U+0301).

With Unicode support added, a great many extra character classes have become available. There are a number of *main categories*, like letters, symbols, diacritical signs, interpunction, and such, each of which have been divided into *subcategories*, for even more fine-grained control.

Every (sub)category can be written in a longer, readable form, and in an abbreviated form. For example, a *letter* can be specified as either `\p{Letter}`, or `\p{L}`. A subcategory of `\p{Letter}` is *uppercase*: `\p{Uppercase_Letter}`, abbreviated as `\p{Lu}`

An overview of all available Unicode character classes can be found in later on in this document.

[5] The GNU version of `grep` has the `-P` flag to switch to Perl regular expressions. But to honour the inventor, we will use the `perl` command in our examples, instead of `grep`.

4.5 Anchors

New in Perl Regular Expressions, compared to EREs, is the `\b` anchor to denote a *word boundary*. This is defined as a location between two characters, where one is a word character (`\w`), and the other is not.

Please recall that anchors, like `^` and `$`, as discussed earlier, have no width! An anchor always specifies a position *between* two characters.

Specifying that there should be *no* word boundary at a particular position can be done by writing the uppercase version: `\B`

The regex `\bearth\b` matches in the sentence:

```
The earth is warming up.
```

but not in:

```
We are getting better at predicting earthquakes.
```

Now suppose you would like to find words beginning with `earth`, but not the whole word `earth`, you should better use the regex `\bearth\B` That one *would* match the first three letters of the word `earthquake`.

4.6 Assertions

Assertions do not exist in BREs or EREs, so they are a new concept. An assertion could be considered to be a *programmable anchor*. Where a hard-coded anchor like `\b` specifies that there should be a word boundary at a particular position, an assertion could be an *arbitrary regex*, functioning as a test for the position.

Just like an anchor, an assertion has *no width*, but tests whether some position *between* two characters passes some test or not, by matching a regex at its left (*lookbehind*), or at its right (*lookahead*).

The combinations of passing the test or not, and looking behind or ahead, yield four possible specific assertions:

Table 2 Lookaround assertions

Notation	Meaning	Name
`(?=regex)`	regex should match at the right	lookahead
`(?!regex)`	regex should not match at the right	negative lookahead
`(?<=regex)`	regex should match at the left	positive lookbehind
`(?<!regex)`	regex should not match at the left	negative lookbehind

The regex `[a-z](?=\d)` matches a lower case letter, followed by a digit. You could also find that with `[a-z]\d`, but then the digit would really be part of the match. That is an important fact when you use the regex for a search and replace operation:

`[a-z]\d` matches the text `c1` in `abc123`

`[a-z](?=\d)` matches the text `c` in `abc123`

It remains important to realise that an assertion never matches any characters, but simply tests whether a certain condition holds at some position in the regex.

Furthermore, assertions *do not* count against group numbering.

Example: we show all the lines from the `words` file containing an `r` which is *not* preceded by an `o`.

```
$ perl -ne 'print if (/(?<!o)r/)' words
```

This yields the following matching lines:

```
earth
tree
horror
whenever
try
break.
better
```

We also get lines with a `r` which *are* preceded by an `o`, which would seem odd, but in any case those lines also contain an `r` without an `o` in front of it.

Lastly, another example of a positive lookahead, used in a negated way in Perl. We are looking for lines which *do not* contain the same letter twice consecutively.

```
$ perl -ne 'print unless (/(.)(?=\1)/)' words
```

The regex matches every line containing an arbitrary character, immediately followed by the same character. The word `unless` is Perl language to negate the condition that follows it, so the statement only prints lines which do *not* match the regex:

```
earth
talking.
sky
whenever

try
break.
world
```

4.7 Backtracking

When using greedy quantifiers the mechanism of *backtracking* plays a role. This means that the regex *engine*, while it is scanning a text, continuously monitors the positions where matches occur. It remembers such a position, just in case it later appears to have been the longest possible match. If it encounters a longer match further along the line, it forgets any earlier position, and remembers the new one. Arriving at the end of the text the engine often discovers there is not a longer match than the one previously found, and *backtracks*.

A nice example which shows how you can shoot yourself in the foot with backtracking, is the following combination of two *consecutive* greedy quantifiers.

With this regex:

```
.*(\d+)
```

we are looking for a match in the text:

```
abc123
```

The trick question is: what is inside the group?

Answer: only the last digit, `3`!

Explanation: the `*` working on the `.` tries to match as many characters as possible, preferably up to and including the last one in the text! Arriving at the last character (the `3`), it concludes there is no match, unless it backtracks one step. The `3` is then the only available remaining character for the group, which wants to match one or more digits. What we see here is that the first greedy quantifier *wins* from the second one.

Should we want to compose a regex that collects *all* the digits into the group, we can arrange for that by making the `*` lazy:

```
$ echo 'abc123' | perl -ne 'print $1,"\n" if (/.*?(\d+)/)'
```

This would be a complete command line to test the match. The `$1` inside the `print` statement is a Perl specific back-reference notation, which indicates that we only want to print the part of the match that ended up inside the group. Every programming language has its own way to refer to a group, outside the regex itself.

4.8 Much more is possible

Perl regular expressions have even more possibilities than what we have shown here. These features are either more advanced, or not widely supported using a uniform syntax.

Regular expression cheat sheet

Basic rules

1. A regex only matches on a single line
2. The *first* match counts
3. Standard quantifiers are *greedy*
4. Matching is *case sensitive*

Metacharacters

The following characters have to be escaped using a backslash ‘\‘ if they are to be matched literally:

`^ $ ( ) [ { \ | . * + ?`

Anchors

Notation	Meaning
`^`	start of line
`$`	end of line
`\b`	word boundary
`\B`	not a word boundary

Alternation

Notation	Meaning
`oneRegex|otherRegex`	matches either `oneRegex` or `otherRegex`

Quantifiers: *Greedy*: match as *many* characters as possible

Lazy: match as *little* characters as possible

Possessive: not discussed in this document

Meaning	Greedy	Lazy	Possessive
zero or more	`*`	`*?`	`*+`
one or more	`+`	`+?`	`++`
zero or one	`?`	`??`	`?+`
exactly *n*	`{n}`	`{n}?`	`{n}+`
at least *n*	`{n,}`	`{n,}?`	`{n,}+`
at most *m*	`{,m}`	`{,m}?`	`{,m}+`
n up to *m*	`{n,m}`	`{n,m}?`	`{n,m}+`

Grouping

Notation	*Meaning*
`(regex)`	*regex* is a group; whatever matches is remembered *(capturing group)*
`(?:regex)`	*regex* is a group; match is *not* remembered *(non-capturing group)*, and not numbered
`(?>regex)`	atomic grouping (not discussed)

Character classes

Notation	**Meaning**
`\s`	whitespace
`\S`	not whitespace
`\w`	word character (letter, digit, underscore)
`\W`	not a word character
`\d`	digit
`\D`	not a digit
`[...]`	set of *desired* characters and/or character classes
`[^...]`	set of *undesired* characters and/or character classes
`[a-z]`	*range* of `a` to `z`

Unicode character class categories:

Abbrev.	**Long form**	**Meaning**
`\p{L}`	`\p{Letter}`	letter, from any alphabet
`\p{M}`	`\p{Mark}`	a character meant to be combined with another character
`\p{Z}`	`\p{Separator}`	any kind of white space or invisible divider
`\p{S}`	`\p{Symbol}`	mathematical symbols, lines, currency signs, etc.
`\p{N}`	`\p{Number}`	numerical sign
`\p{P}`	`\p{Punctuation}`	interpunction
`\p{C}`	`\p{Other}`	invisible control characters and unused code points

Unicode subcategories:

abbrev.	**long form**	**meaning**
`\p{Ll}`	`\p{Lowercase_Letter}`	lower case letter which has an upper case variant
`\p{Lu}`	`\p{Uppercase_Letter}`	upper case letter which has a lower case variant
`\p{Lt}`	`\p{Titlecase_Letter}`	upper case letter at the start of a word
`\p{L&}`	`\p{Cased_Letter}`	combination of `\p{Ll}`, `\p{Lu}` and `\p{Lt}`
`\p{Lm}`	`\p{Modifier_Letter}`	special symbol being used as a letter

continues on next page

Table 3 – continued from previous page

abbrev.	long form	meaning
\p{Lo}	\p{Other_Letter}	letter which has no lower and upper case distinction
\p{Mn}	\p{Non_Spacing_Mark}	character meant to be used in combination with another character, without claiming extra space (e.g. umlaut, accents)
\p{Mc}	\p{Spacing_Comb_Mark}	a character meant to be combined with another character, which uses extra space (vowel signs in eastern languages)
\p{Me}	\p{Enclosing_Mark}	a symbol which encloses the character with which it is combined (e.g. circle, square, keyboard key)
\p{Zs}	\p{Space_Separator}	an invisible white space symbol that does take up space
\p{Zl}	\p{Line_Separator}	line separator U+2028
\p{Zp}	\p{Paragraph_Separator}	paragraph separator U+2029
\p{Sm}	\p{Math_Symbol}	any mathematical symbol
\p{Sc}	\p{Currency_Symbol}	any currency symbol
\p{Sk}	\p{Modifier_Symbol}	an independent combining character
\p{So}	\p{Other_Symbol}	several characters or symbols not in any of the subcategories above
\p{Nd}	\p{Decimal_Digit_Number}	digit 0 till 9
\p{Nl}	\p{Letter_Number}	a digit that looks like a letter like a roman numeral
\p{No}	\p{Other_Number}	a sub- or superscripted digit, or another digit not equal to 0-9
\p{Pd}	\p{Dash_Punctuation}	any kind of horizontal line
\p{Ps}	\p{Open_Punctuation}	any kind of opening bracket
\p{Pe}	\p{Close_Punctuation}	any kind of closing bracket
\p{Pi}	\p{Initial_Punctuation}	any kind of opening quote
\p{Pf}	\p{Final_Punctuation}	any kind of closing quote
\p{Pc}	\p{Connector_Punctuation}	a connecting symbol, like underscore
\p{Po}	\p{Other_Punctuation}	any kind of punctuation not in any of the subcategories above
\p{Cc}	\p{Control}	an ASCII 0x00-0x1F or Latin-1 0x80-0x9F control character
\p{Cf}	\p{Format}	invisible layout symbol
\p{Co}	\p{Private_Use}	any code point for private use
\p{Cs}	\p{Surrogate}	half of a surrogate pair in UTF-16
\p{Cn}	\p{Unassigned}	any unassigned code point

POSIX character classes and their ASCII and Unicode equivalents

POSIX	meaning	ASCII	Unicode
[:alnum:]	alfanumeric symbol	[a-zA-Z0-9]	[\p{L}\p{Nl}\p{Nd}]
[:alpha:]	alfabetic symbol	[a-zA-Z]	[\p{L}\p{Nl}]
[:ascii:]	ASCII symbol	[\x00-\x7F]	\p{InBasicLatin}
[:blank:]	space or tab	[\t]	[\p{Zs}\t]
[:cntrl:]	control code	[\x00-\x1F\x7F]	\p{Cc}
[:digit:]	digit	[0-9]	\p{Nd}
[:graph:]	visible symbol	[\x21-\x7E]	[\p{Z}\p{C}]
[:lower:]	lowercase letter	[a-z]	\p{Ll}
[:print:]	visible symbol or space	[\x20-\x7E]	\P{C}
[:punct:]	interpunction and symbols	[!"$%&'()	\p{P}
		*+,\-./:;<=>	
		?@\[\\\]^_{|}~]	
[:space:]	all whitespace	[\t\n\r\v\f]	[\p{Z}\t\n\r\v\f]
[:upper:]	Capital letter [A-Z]	\p{Lu}	
[:word:]	word-symbol	[A-Za-z0-9_]	[\p{L}\p{Nl}
			\p{Nd}\p{Pc}]
[:xdigit:]	hexadecimal symbol	[a-fA-F0-9]	[a-fA-F0-9]

Backreferences

Numbering: capturing groups are numbered from 1, in the order of the opening parentheses ``(``

Notation	Meaning (with k = 1, 2, …, 9)
\k	matched text of the k^{th} group (only *within* regex itself)

Just like non-capturing groups, lookaround assertions are not numbered!

Notation	Meaning	Name
(?=regex)	*regex* must match at the right	positive lookahead
(?!regex)	*regex* should *not* match at the right	negative lookahead
(?<=regex)	*regex* must match at the left	positive lookbehind
(?<!regex)	*regex* should *not* match at the left	negative lookbehind

Backtracking

1. Try to match from the beginning.
2. Multiple possibilities? Remember current position and possibilities.
3. Try one.
4. No match? Backtrack to last remembered position and try again.
5. No more possibilities? Continue until either a match is found, or the whole match fails.

Flags

Most regex implementations offer possibilities to alter the standard behaviour of the regex engine. This is done using *switches* which can be *on* or *off*; they are often called *flags*. The exact use of these flags depends on the environment (command, programming language) where you are using regular expressions.

Many programming languages do not use the single letter flags given below, but, instead, make you supply your options through extra arguments to the functions doing the matching. These arguments often have the form of predefined constants, which are described in the relevant documentation.

The following flags are usually supported:

Flag	Meaning	Explanation
`i`	case insensitive	no distinction between upper and lower case letters
`g`	global search	continue after first match (e.g. during search/replace)
`s`	single-line text	`.` (arbitrary character) also matches `\n`
`m`	multi-line anchors	anchor `^` also matches after `\n` and `$` also matches before `\n`
`x`	comment mode	ignore whitespace; comments are allowed after `#`