



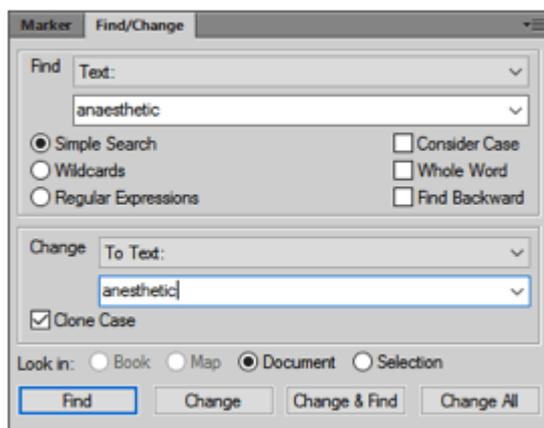
/Everybody stand back/ I know regular expressions

Do the magic: Regular Expressions in FrameMaker (2015 release)
by @wasaty

[Guest Post] "Do the magic: Regular Expressions in FrameMaker", by Marek Pawelec

□ March 10, 2016 □ Marek Pawelec □ FrameMaker, Guest Posts, Products

When working with text, whether authoring or editing, we often need to change something. Name of some feature from A to B. Spelling error. Numerical value. A phrase that does not fit the rest of text. And usually, we just delete the wrong text and type in the correct one. If a spelling error or incorrect spelling variant is repeated across the document, we may use Find/Change to correct all instances at once.



Things get a bit more difficult when we want to correct more than a single word at a time – but we can use wildcards. There are two commonly used wildcard symbols, and as you probably know, “*” stands for “any string” and “?” means “single character”. So searching for “anaestheti*” will return anaesthetic, anaesthetics, anaesthetised, anaesthetist etc., while searching for “anaestheti?” will return only “anaesthetic” (if “Whole Word” option is checked).

Wildcard-based search may be sufficient in a number of situations, where all you need is to find some text and change its attributes, for example character style. It is especially true in FrameMaker since FM wildcards support some features copied from regular expressions. However, if we need to conduct more advanced searches, conditional searches or perform some kind of operations on the text we find, we should reach for full power of regular expressions (“regex”).

In This Article

- [Meet regular expressions](#)
- [Alphabeth for Beginners](#)
- [Hang on](#)
- [Grammar Rule\[sz\]](#)
- [Benefits of Laziness](#)
- [Bean Counter](#)
- [Numbers Crunching](#)
- [What If](#)
- [Bring it on](#)
- [Reference](#)

Meet regular expressions

What are they? Regular expressions are described as “a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. “find and replace”-like operations” [[Wikipedia](#) [□]]. Somewhat like in the case of wildcards, there is a “vocabulary” – set of symbols that can be used to find a string of characters matching our expression, only the symbol set is much bigger – and a special kind of “grammar” which enables the creation of long and complex search patterns. Plus once we find something, we can usually modify it.

To give you some examples:

- Let’s say we want to find all occurrences of “FrameMaker” along with its version number in a FrameMaker Wikipedia article to change its character format. Can we do that with wildcards? Yes, but given the possible variations in numbering (2.0, 5.1.2, 2015) we would need to run the search at least three times. Why waste the time, if we can do it with one regular expression? (`FrameMaker \d+(\.\d+)*`)
- In Polish texts we often encounter single letter conjunctions – i, o, u, z, a, w – and much like widow lines for paragraphs, it is considered poor typography if a line ends with such single letter. The simplest solution is to replace standard space with non-breaking one for them. Again something that can be done with a single regular expression.

Find:

```
(\b[iouzaw]\b) (\w);
```

Change:

```
$1 $2
```

 – with a non-breaking space on the "Change" side).

- Do you need to change date notation from 12/24/2015 to 12-24-2015? Or maybe even from American to European format (24.12.215)? Piece of cake (see below).
- Remove all double spaces in one go? Easy:

Find

```
\s{2,}
```

Change:

```
\s
```

- Insert (or remove) spaces between numerical values and units? No problem (see below).
- Add a missing word to complex product name in inflected language, but only if it's really missing? Will do (see below).

So, how do we actually accomplish all these things? You need to learn basic vocabulary and grammar of regular expressions. And you can do this either by reading the introduction below, or just skip to the reference table at the end and rifle through examples. All regular expressions in the text are written with `\s{2,}` font, and quotation marks around them (if present) are not part of the expressions. And remember – while the expressions may look scary, they are, in fact, quite regular. Once you grasp the meaning of symbols and basic rules, you'll be able to use them at no time.

Alphabet for Beginners

If we want to find any particular symbol/letter/number using regular expression, we just type that symbol/letter/number. So `anaesthetic` is a regular expression which matches only that particular sequence of lower-case letters (regular expressions are case-sensitive).

If we need to find `any` character or symbol, we use period (`.`). Single dot means single symbol of any kind.

To get somewhat more specific matches we can define character `class` by using square brackets `[]`. For example `[abcd]` will match "a", "b", "c" or "d". If you are looking for a symbol within continuous range, instead of defining all elements of class, we can simply state first and last: `[a-d]`, or `[0-9]` to match any digit. The order in ranges is always the same as in ASCII character table, so to match any letter regardless of case we can use class `[A-Za-z]`.

There is also a special type of class used for negation: if the first symbol within square brackets is the circumflex (`^`), expression will match anything except the class content. For example `[^abcd]` will match any single symbol except for a, b, c or d. Class can be seen as logical alternative for set with single symbol elements.

If we need to employ alternatives for longer elements, we can use pipe character "`|`" to separate elements: `Monday|Tuesday|Wednesday|Thursday` will match any of these four weekday names.

Unfortunately there is no simple way to exclude longer strings for search – matching anything but some

string is usually bit tricky.

Do we have to define these classes every time? That depends on the class. There are some shortcuts defined in regular expression vocabulary:

- `\d` = digit = `[0-9]`
- `\D` = digit negation = `[^0-9]`
- `\w` = word character = `[A-Za-z0-9_]`
- `\W` = word negation = `[^A-Za-z0-9_]`
- `\s` = white space (includes all spaces, tabs and end of line characters)
- `\S` = white space negation
- `\u####` = Unicode character number ####, e.g. `\u2212` for n-dash (-)

Of course this is just the most basic set. Instead of shortcuts you can also use [Unicode categories](#), which are much more versatile with regard to various scripts and non-ASCII characters – see the [reference tables](#) at the end of this post.

Backslash by itself is a special **escape character**. You know that a period means any symbol, but what do we do if we want to match only period? Simply escape it with backslash: `\.`. But since backslash is special character too, if we need to match backslash, we have to escape it too: `\\`

Hang on – anchors

There are two special symbols used to define start (`^`) and end (`$`) of the line, plus shortcut defining word boundary (`\b`) they are called “anchors”. To recall our earlier example, `anaesthetic` will match that string anywhere in text, but `^anaesthetic` will match only, if the string will appear at the beginning of line (paragraph). By itself `aesthetic` will also match “an `aesthetic`” and “an`aesthetic`s”, `aesthetic\b` will match “an`aesthetic`” but not “anaesthetics” and `\baesthetic\b` will match neither. It’s a bit like searching with “Whole Word” option, but with more precise control.

Grammar rule[sz]

This was essential “vocabulary”, now we’ll see some “grammar”. Let’s start by introducing operators (quantifiers), since quite often we are interested in some longer strings, not just particular symbols. Operators works for the object immediately to their left. There are three basic operators:

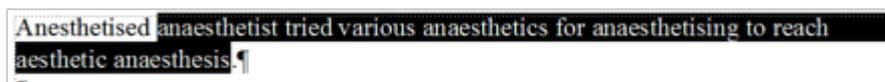
- `*` = 0 or as much as possible. “ `.*` ” matches any string. That is, it can match whole paragraph, but also a completely empty paragraph (0 occurrences). “ `\d*` ” will match any string of numbers, but also “empty” string, e.g. boundary between letters. You need to be careful with this quantifier.
- `?` = 0 or 1. `?.?` matches empty string or single symbol of any kind. “ `\d?` ” matches no digits or exactly one digit. The concept of “empty” match is a bit tricky, but I suggest to simply enter these expressions into the **Find/Change** dialog with **Regular Expressions** option checked and click **Find** repeatedly to see what will be matched.
- `+` = 1 or as much as possible. That one is much more intuitive: “ `.+` ” will match any string with at least one symbol and `\d+` will match any string of digits, but not less than one.

We can use these quantifiers for example to find both spelling versions of an[a]esthetic with a single expression. `ana?esthetic` will match both **anesthetic** and **anaesthetic**, because letter "a" can occur 0 or 1 time.

Benefits of Laziness

When using operators, we need to remember, that by default they behavior is "greedy", that is, they want to match the longest possible string within paragraph (in FM with default settings) or even whole text.

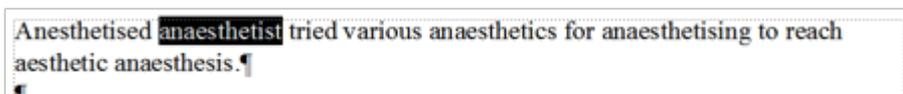
We can expand our search to cover both spelling variants and different word suffixes. `ana?esthe.*\b` will match **anaesthe** or **anesthe** followed by any letter occurring any number of times until the end of word. Unfortunately, the greedy behavior will make the expression match up to the last word boundary it can find (since period matches also a space):



Now, being eager like that can be a good thing, but as they say, laziness is one of the greatest programmer virtues. So in order to avoid such excessive matching we have to use "lazy" matching operator combinations:

- `*?` = zero or as little as possible
- `+?` = one or as little as possible

If we modify our expression accordingly: `ana?esthe.*?\b`



We can match exactly what we want. The alternative way to safely match only single word in this case would be to use `ana?esthe\w+` that way matching will end at the first non-word character (e.g. space or punctuation mark).

Bean Counter

Are these the only quantifiers you have at your disposal? No, for precise matching we can define either the exact number of occurrences we want to match or a range of occurrences using curly brackets `{}`:

- `\d{4}` = matches exactly 4 digits
- `\d{2,4}` = matches from 2 to 4 digits (2, 3, 4)
- `\d{,4}` = matches up to 4 digits
- `\d{4,}` = matches at least 4 digits

So for our (an)aesthetic example `ana?esthe\w{3}` will match **anaesthesia** and **anesthetic**, but not anesthetist (`\w{3}` will match exactly three of any "word" characters).

Now it's time to familiarize ourselves with the use of parentheses `()`. I've already written that regular expression allow us to manipulate results of our searches. This is possible, because the match is stored in memory, numbered and can be recalled. Whole match is always numbered "0". What's important, we can define parts of search expression to be separate groups, and we do that by putting them inside parentheses. Content of the groups is recalled by `$num`, e.g. `$1` for group number 1.

Numbers Crunching

Let's say we want to find a date in American format: 12/24/2015. A simple expression for matching this can look like this:

`\d\d/\d\d/\d\d\d\d` – or, a more elegant and readable version – `\d{2}/\d{2}/\d{4}`

Match two digits, slash, again two digits, slash and four digits. Now, if we want to somehow manipulate this date, we have to define groups:

Find:	Change:	Matches:	Result:
<code>(\d{2})/(\d{2})/(\d{4})</code>	<code>\$1-\$2-\$3</code>	12/24/2015	12-24-2015

Of course we can freely manipulate the results, so if we need to change date format to European, we can use this:

Find:	Change:	Matches:	Result:
<code>(\d{2})/(\d{2})/(\d{4})</code>	<code>\$2.\$1.\$3</code>	12/24/2015	24-12-2015

Quite often when writing longer expression it's convenient to create a group, that will not be numbered. In such case we use the so-called passive group: `(?:)`. An example of expression with passive groups:

Find:	Change:	Matches:	Result:
<code>(\d{2})(?:/ - :\.)(\d{2})(?:/ - :\.)(\d{4})</code>	<code>\$2.\$1.\$3</code>	12/24/2015, 12-24-2015, 12:24:2015, 12.24.2015	24.12.2015

(And if you wonder – yes, it would be simpler to write `(\d{2})[/-:\.](\d{2})[/-:\.](\d{4})`, in this case grouping is not necessary, but it's just an example.)

What if

The last "grammar" element I'm going to show you are assertions. Simply speaking, they are conditions or "ifs". Whatever is inside assertion is not considered part of the match, but must (or cannot) be present before or after particular content to match it. There are four of them, two positive and two negative.

First Example

- `(?=)` = positive lookahead
- `memo(?:=Q)` match "memo" in **memoQ**, but not in **memory**

- `(?!)` = negative lookahead
- `memo(?!Q)` will "memo" in **memory**, but not in **memoQ**

Second Example

- `(?<!)` = negative lookbehind
- `(?<!k)ot` will match "ot" for **pot**, but not in **kot**
- `(?<=)` = positive lookbehind
- `(?<=k)ot` will match "ot" for **kot**, but not in **pot**

Why do we need them? Quite often we need to match (find) some string (text) only, if it matches certain criteria, or it appears before or after some other text. Let's say we need to remove all double spaces occurring between sentences. The simplest expression for this purpose would be (without quotation marks):

Find: " `{2,}` " (space, occur at least twice)

Change: " " (single space)

Unfortunately this will convert all sequences of multiple spaces into single space. If for any reason we want to remove only double spaces between sentences, we need to be more precise:

Find: " `(\.) {2,}([A-Z])` " (period, store it as group #1, at least two spaces, capital letter, store it as group #2)

Change:

`$1 $2` (put back that period, insert single space, put back that capital letter)

Now, since the period and capital letters are only there to define matching criteria, we can make the Find rule a bit more complex, to simplify Replace:

Find: " `(?<=\.) {2,}(?=[A-Z])` " (at least two spaces, but only if there is a period before and capital letter behind them)

Change: " " " " " "

A bit more complex example from the real life scenario: let's say you have a text on "ACME super rocket engine". However, the author was somewhat inconsistent and sometimes it's referred in text as "ACME super rocket". You need to insert "engine" into text, but only where it's not already there to avoid double words. To make matter more complex, sometimes text refers to just "super rocket", without ACME, and you should not insert "engine" there either. Fortunately, we have regex:

Find: " `(?<=ACME)(super rocket)(?! engine)` " (match "super rocket" if there is "ACME" before but no "engine" after)

Change:

`$1 engine` (insert back "super rocket", but add a space and "engine" after)

(And if your text is written in an inflected language, like Polish, you can match all grammar variants in one go: " `(?<=ACME)(supe\w+ rock\w+)(?! engine)` " – in Polish "rocket" is inflected as "rakieta, rakiety, rakiemie, rakieta", and that's just singular.)

Bring it on

Below you will find several examples of regex matching

HTML and XML tags:

- Match any tag: `<.*?>` (match left angle bracket and anything up to (including) first right angle bracket)
- Match only opening tags: `<[^/]*?>` (match left angle bracket and anything up to (including) first right angle bracket, but not if the first character after left angle bracket is backslash)
- Match only closing tags: `</.*?>` (match left angle bracket, backslash and anything up to (including) first right angle bracket)
- Match **content** of tag "xxx" with attribute "translate": `(?<=<xxx.*"translate".*?>)(.*?)(?=</xxx.*?>)`

Find "Chapter" line:

- Match line starting with chapter, followed by number: `^[Cc]hapter\s+\d+\s*$` (match "Chapter" or "chapter" at the beginning of line followed by at least one space, followed by any number of digits and possibly a space(s), but nothing else before end of line. It will match "Chapter 1" but also "Chapter 11 " – because quite often text is far from perfect. In FM you can control case recognition by checking or un-checking "Consider Case" check box, but that's not usually the case with regex).

Match compound number (1.3.2): `\d+(\.\d+)+` (match string of digits followed by period and string of digits – the sequence "period and string of digits" may appear any number of times).

However, consider what will happen if you want to use the expression for find and replace `(\d+(\.\d+)+)`: for 3.5 you will get two numbered groups – `$1` = 3.5, `$2` = .5. For 3.5.9 you will get also two numbered groups, but with different result – `$1` = 3.5.9, `$2` = .9. Should your expression include more numbered groups, referencing them would become unpredictable. Therefore it is a good idea to use passive group in such case: `(\d+(?:\.\d+)+)` will return only `$1` covering whole compound number.

- Match chapter with compound number: `^[Cc]hapter\s+\d+(\.\d+)+\s*$`
- Match chapter with standard or compound number: `^[Cc]hapter\s+\d+(\.\d+)*\s*$`
- Match any combination of capital letters and numbers separated by dashes (e.g. catalog numbers): `[A-Z\d]{2,}(-[A-Z\d]+)+` (match string of at least two capital letters, numbers or capital letters and numbers combination followed by dash and any capital letter/number combination – the sequence "dash and any capital letter/number combination" may appear any number of times. The combination will match AB-123456, ABCD-21, AB56-1, GH345-123-578, etc.)

Find numbers followed by "in" designation and replace with the same number, space and "in":

Find:

`(\d)(in\b)` ("`\b`" is a safeguard against matching of longer string starting with "in" e.g. internal)

Change:

`$1 $2`

There is an important limitation you need to be aware of when changing/replacing. While we can use various symbols for matching (e.g. `\u2212` for n-dash), in the change/replace field all text is treated

literally (except for `$num`). So if we want to insert n-dash into changed text, we need to type/paste actual n-dash, not it's code. On the other hand, we can type in period without escaping it.

Reference

Should you be interested in learning more, there is plenty of resources available on the Internet: both courses (e.g. <http://www.regular-expressions.info/> [□]) and software for building and testing regular expressions – both online (e.g. <http://www.regexpal.com/> [□]) and offline (e.g. <http://www.ultrapico.com/expresso.htm> [□])

Table of basic regular expression symbols

Symbol	Meaning
<code>.</code>	Any character
<code>[]</code>	Any of the given characters (class)
<code>[-]</code>	Range of characters
<code>[^]</code>	Exclusion of matching
<code> </code>	"OR"
<code>\</code>	Modifier
<code>\d</code>	Digit
<code>\D</code>	Anything but digit
<code>\s</code>	Whitespace character
<code>\S</code>	Anything but whitespace
<code>\w</code>	Any "word" character – includes letters, numbers and underscore (<code>_</code>)
<code>\W</code>	Anything but word
<code>\r</code>	Carriage return
<code>\n</code>	Newline
	FrameMaker performs all Find operations within a single paragraph, but in other software regular expression will often match the whole file, so it is useful to know that for Windows files end of line is encoded as <code>"\r\n"</code> .
<code>\t</code>	Tab

`\xnnnn` Unicode character `nnnn`

Operators and anchors

Symbol	Meaning
<code>?</code>	Zero or one occurrence of symbol on the left
<code>*</code>	Zero or more occurrences of symbol on the left
<code>+</code>	One or more occurrences of symbol on the left
<code>*?</code>	Zero or as little as possible – lazy matching
<code>+</code>	One or as little as possible – lazy matching
<code>{ , }</code>	Range
<code>^</code>	Start of line
<code>\$</code>	End of line
<code>\b</code>	Word boundary

Grouping

Symbol	Meaning
<code>()</code>	Group
<code>(?:)</code>	Passive group (not numbered)
<code>(?=)</code>	Positive look ahead
<code>(?!)</code>	Negative look ahead
<code>(?<=)</code>	Positive look behind
<code>(?<!)</code>	Negative look behind
<code>(?#)</code>	Comment

Unicode categories

Note: Make sure to check the "Consider Case" check box.

Symbol	Meaning
<code>\p{L}</code>	Any letter from any language (equivalent to [A-Za-z] but also for non-Latin scripts)
<code>\p{Lu}</code>	Uppercase letter with lowercase variant (Letter uppercase)
<code>\p{Ll}</code>	Lowercase letter with uppercase variant (Letter lowercase)
<code>\p{Z}</code>	Any whitespace or separator
<code>\p{S}</code>	Math symbols, currency symbols, dingbats m – math symbol, c – currency symbol
<code>\p{N}</code>	Numbers
<code>\p{P}</code>	Punctuation marks d – pause and dashes, s – opening brackets, e – closing brackets, i – opening parentheses, f – closing parentheses

Share this post with your colleagues and friends:

Facebook Google+ LinkedIn Twitter VKontakte
 Weibo Email

Adobe FrameMaker Regular Expressions (RegEx)



MAREK PAWELEC

Marek Pawelec is a full-time freelance literary and technical translator since 2001. He graduated in molecular biology and worked as a researcher in faculties of Medicine and Chemistry, so he specializes in translation of medical, biochemical and chemical texts. Experienced computer translation software (CAT) user, trainer and consultant with an interest in technology and productivity tools, including regular expressions. You can follow him on Twitter: [@wasaty](#)

9 thoughts to "[Guest Post] "Do the magic: Regular Expressions in FrameMaker", by Marek Pawelec"



ADEE

February 4, 2018 at 1:41 am

Does Unicode character `\u####` really work? I tried a few codes and and it seems that no character is matched by this method. Am I missing something?

I'm specifically interested in non-breaking space which should be `\u00a0`. It is also not matched by the whitespace categories. Is there another way to match it in a regexp?

REPLY



KASHISH

May 5, 2017 at 5:16 pm

Thanks for the tutorial! Helped a lot in understanding the regular expressions.

REPLY



ROY SUBHASH

April 7, 2017 at 1:44 pm

I am creating a document in Framemaker in Arabic Language. There are some words and alphabets in English font. I want to find and replace its font. But I am unable to find that option. In Adobe InDesign that option is in "Find/Replace => GREP". Like InDesign, there is any option to find and replace only english character?

REPLY



STEFAN GENTZ

April 7, 2017 at 3:31 pm

Yes, there is. Just turn on "Regular Expressions" In the most simple case, you can search for something like `([a-z]+?)`. This finds any character in the range from a to z. You could also use `(\b[a-z]\b+?)` to find whole words.

You might want to extend it to other western characters like ä, ö, ü etc. like this: `([a-z]`