**Регулярные Выражения**

**Введение**

**Регуля́рные выраже́ния** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *regular expressions*) — формальный язык поиска и осуществления манипуляций с [подстроками](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0) в тексте, основанный на использовании метасимволов ([символов-джокеров](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%8B-%D0%B4%D0%B6%D0%BE%D0%BA%D0%B5%D1%80%D1%8B), [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *wildcard characters*). По сути это строка-образец ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *pattern*, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

Регулярные выражения – мощное, гибкое и эффективное средство обработки текстов. Универсальные шаблоны регулярных выражений сами по себе напоминают миниатюрный язык программирования, предназначенный для описания и разбора текста. При дополнительной поддержке со стороны конкретной утилиты или языка программирования регулярные выражения способны вставлять, удалять, выделять и выполнять самые невероятные операции с текстовыми данными любого вида. Они бывают очень простыми, вроде команды поиска в текстовом редакторе, или очень сложными, как специализированные языки обработки текстов. Во многих популярных современных языках программа поиска повторяющихся слов занимает всего несколько строк. Всего одна команда поиска/замены находит и выделяет повторяющиеся слова во всем документе. Другая команда удаляет из отчета все строки, не содержащие повторяющихся слов (и оставляет только те строки, которые включаются в отчет).

## Реализации

## Отдельные системы используют различную реализацию регулярных выражений, из конечных автоматов.

## NFA ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *nondeterministic finite-state automata* — [недетерминированные конечные автоматы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82#.D0.94.D0.B5.D1.82.D0.B5.D1.80.D0.BC.D0.B8.D0.BD.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.BD.D0.BE.D1.81.D1.82.D1.8C)) используют [жадный алгоритм](https://ru.wikipedia.org/wiki/%D0%96%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) отката, проверяя все возможные расширения регулярного выражения в определённом порядке и выбирая первое подходящее значение. NFA может обрабатывать подвыражения и обратные ссылки. Но из-за алгоритма отката традиционный NFA может проверять одно и то же место несколько раз, что отрицательно сказывается на скорости работы. Поскольку традиционный NFA принимает первое найденное соответствие, он может и не найти самое длинное из вхождений Именно такой механизм регулярных выражений используется, например, в [Perl](https://ru.wikipedia.org/wiki/Perl), [Tcl](https://ru.wikipedia.org/wiki/Tcl) и [.NET](https://ru.wikipedia.org/wiki/.NET_Framework).

## DFA ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *deterministic finite-state automata* — [детерминированные конечные автоматы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82#.D0.94.D0.B5.D1.82.D0.B5.D1.80.D0.BC.D0.B8.D0.BD.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.BD.D0.BE.D1.81.D1.82.D1.8C)) работают линейно по времени, поскольку не используют откаты и никогда не проверяют какую-либо часть текста дважды. Они могут гарантированно найти самую длинную строку из возможных. DFA содержит только конечное состояние, следовательно, не обрабатывает обратных ссылок, а также не поддерживает конструкций с явным расширением, то есть не способен обработать и подвыражения. DFA используется, например, в [egrep](https://ru.wikipedia.org/wiki/Egrep).

**Различные диалекты регулярных выражений. Соответствие стандарту POSIX**

Существуют два основных диалекта (или типа) регулярных выражений: простые и расширенные. При этом граница между ними является условной и со временем становится всё менее чёткой.

Программы vi(m), sed, grep, less понимают только простые регулярные выражения, а утилиты (g)awk, egrep, а также интерпретаторы языков Perl, Tcl, Python – расширенные регулярные выражения. В то же время в каждой из программ существуют собственные усовершенствования, т.е. создаются поддиалекты регулярных выражений. Рассмотрим сходства и различия этих диалектов.

**Общая схема регулярного выражения**

Как правило, регулярное выражение состоит из трёх основных частей:

1. Якорь – определяет позицию шаблона в строке текста:
   * ^ – якорь, определяющий начало строки;
   * $ – якорь, определяющий конец строки.
2. Набор (последовательность) символов – для поиска соответствий в заданных позициях строки текста:
   * символ "точка" (.) соответствует любому произвольному символу;
   * алфавитно-цифровые символы и пробел представляют сами себя;
   * прочие символы – интерпретация зависит от диалекта.
3. Модификатор – задаёт количество повторов предыдущего символа или набора символов (в зависимости от диалекта):
   * \* – любое количество повторов символа/набора, в том числе и нулевое;
   * ? – соответствует нулю или одному экземпляру символа/набора;
   * + – соответствует одному или большему количеству экземпляров символа/набора.

Пример: необходимо найти все директивы определения макроконстант в исходном коде на языке С.

grep '^ \*#define.\*' \*.c \*.h

Здесь учтено, что в начале строки макроопределения может быть вставлено любое количество пробелов или же пробелы отсутствуют. Часть шаблона #define является литеральной, т.е. каждый символ интерпретируется "как есть". Заключительная часть шаблона означает "любые символы в любых количествах".

Отметим, что символ ^ интерпретируется как якорь, обозначающий начало строки, только в том случае, если он является самым первым символом шаблона. Точно так же символ $ обозначает конец строки при условии, что является самым последним символом шаблона. Во всех прочих случаях эти символы становятся литералами, т.е. представляют сами себя.

**Определение диапазонов символов в регулярных выражениях**

Если возникает необходимость задать символ из определённой группы, например, только цифровой символ, или только гласную букву нижнего регистра, или только символы пунктуации, то используются квадратные скобки, внутри которых определяются требуемые символы. Таким образом:

* [012345789] – соответствует одному цифровому символу из заданного набора;
* [аеёиоуыэюя] – соответствует одной из перечисленных гласных букв;
* [,.:;] – соответствует одному из символов пунктуации.

Обратите внимание на то, что в последнем случае точка в квадратных скобках утрачивает свой особый статус и обозначает не "любой символ", а собственно символ "точка".

Непрерывные диапазоны символов можно записывать в сокращённой форме с использованием дефиса: первый пример удобнее записать в виде [0–9]. Кроме того, допускаются любые сочетания диапазонов и конкретных символов.

Имеется также возможность исключать заданные наборы символов из поиска, которая осуществляется следующим образом:

* [^0-9] – соответствует любому символу, кроме цифрового;
* [^аеёиоуыэюя] – соответствует любой НЕ гласной букве.

Сейчас рассмотрим модификаторы на примере шаблона поиска цифрового IP-адреса.

**Модификаторы количества повторений символов**

Здесь сложность состоит в том, что модификатор \* для поиска IP-адреса не годится – попытка использовать шаблон [0-9]\*\.[0-9]\*\.[0-9]\*\.[0-9] приведёт к выводу строк, содержащих элементы типа 2344.5657.11.00000, не являющихся IP-адресами. Для уточнения количества повторений наборов символов применяется модификатор \{min,max\}. Зная, что в каждой части IP-адреса может содержаться от одной до трёх цифр, запишем модификатор в виде \{1,3\}. Символы "обратный слэш" перед точками необходимы для того, чтобы отменить их специальный статус универсального метасимвола. Также следует учесть, что значение 0 не используется в качестве первого байта обычных IP-адресов. В итоге получим следующий шаблон поиска:

grep '[1-9][0-9]\{0,2\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}' \*.txt

Модификатор \{min,max\} работает только в простых регулярных выражениях. В расширенных регулярных выражениях нельзя использовать конструкции \{ \}, но можно применять модификатор ? в качестве эквивалента выражения \{0,1\}, а модификатор + как эквивалент выражения \{1,\}. Во втором случае после запятой не указано числовое значение – это означает, что максимальное количество совпадений не ограничено.

#### Жадная и ленивая квантификация

Здесь применяются ещё два элемента регулярных выражений: \< для обозначения начальной границы слова и \> для обозначения конечной границы слова. Таким образом, мы запоминаем только отдельные слова, а не любые последовательности символов. Выражение ..\* соответствует любому слову, состоящему по крайней мере из одного символа. В результате мы сможем найти такие опечатки-повторения, как "и и", "не не", "для для" и т.п.

В некоторых реализациях квантификаторам в регулярных выражениях соответствует максимально длинная строка из возможных (квантификаторы являются *жадными*, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *greedy*). Это может оказаться значительной проблемой. Например, часто ожидают, что выражение (<.\*>) найдёт в тексте [теги](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%B3_%28%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8%29) [HTML](https://ru.wikipedia.org/wiki/HTML). Однако, если в тексте есть более одного HTML-тега, то этому выражению соответствует целиком строка, содержащая множество тегов.

<p><b>Lorem</b> — Ipsum Dollor, set <i>ammet</i> asterious igniti.</p>

Эту проблему можно решить двумя способами.

1. Учитывать символы, *не* соответствующие желаемому образцу (<[^>]\*>) для вышеописанного случая).
2. Определить квантификатор как *нежадный* (*ленивый*, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *lazy*) — большинство реализаций позволяют это сделать, добавив после него знак вопроса.

Использование ленивых квантификаторов может повлечь за собой обратную проблему, когда выражению соответствует слишком короткая, в частности, пустая строка.

|  |  |
| --- | --- |
| Жадный | Ленивый |
| \* | \*? |
| + | +? |
| {*n*,} | {*n*,}? |

Также общей проблемой как жадных, так и ленивых выражений являются точки возврата для перебора вариантов выражения. Точки ставятся после каждой итерации квантификатора. Если интерпретатор не нашёл соответствия после квантификатора, то он начинает возвращаться по всем установленным точкам, пересчитывая оттуда выражение по-другому.

**Запоминание и повторное использование элемента шаблона**

Этот механизм также работает только в простых регулярных выражениях. (Впрочем, в языках программирования Perl, Python и т.п. данный механизм поддерживается – граница между диалектами становится всё менее различимой)

В простых регулярных выражениях части шаблона, заключённые внутри конструкции \( \), запоминаются и нумеруются, после чего их можно использовать повторно. Всего можно запомнить до девяти пронумерованных шаблонов. Наиболее показательным примером использования механизма запоминания является поиск палиндромов (слов, которые одинаково читаются как слева направо, так и справа налево):

* \([a-z]\)\([a-z]\)[a-z]\2\1 – для пятибуквенных палиндромов (например, level, rotor, madam и т.д.)
* \([a-z]\)\([a-z]\)\([a-z]\)\3\2\1 – для шестибуквенных палиндромов (например, redder, succus, terret и т.д.)

**Соответствие стандарту POSIX**

Стандарт POSIX также делит регулярные выражения на две категории: BRE (Basic Regular Expressions) и ERE (Extended Regular Expressions). В обеих категориях поддерживаются метасимволы . и \*, якоря ^ и $, группирование символов в скобках (для BRE скобки экранируются обратным слэшем), применение квантификаторов \{min,max\} к группам в скобках. Запоминание и повторное использование \1...\9 поддерживает только категория BRE, а квантификаторы + и ? и конструкцию выбора – только категория ERE.

В стандарте POSIX используется понятие локального контекста (locale) – совокупности параметров, описывающих языковые и культурные правила: формат даты и времени, интерпретация символов активной кодировки и т.д. Это не относится напрямую к регулярным выражениям, но влияет на их функционирование. При работе в локальном контексте с кодировкой UTF-8, принятой почти во всех современных дистрибутивах, корректно обрабатываются символы русского алфавита и их диапазоны, т.е. можно указывать диапазоны [а-я] и [А-Я] в шаблонах поиска.

**Примеры составления полезных регулярных выражений**

Для создания правильно работающих регулярных выражений одной теории мало. Необходимо научиться не только конструировать и записывать шаблон, но и в полной мере учитывать контекст, в котором будет производиться его сравнение. Написание и усовершенствование шаблона является итерационным процессом, в ходе которого решаются две главные задачи: с одной стороны, получить все требуемые строки, не пропуская те, которые по замыслу должны были совпасть, но почему-либо не совпали; с другой стороны, исключить все ненужные строки, в том числе и те, которые по замыслу должны быть отброшены, но почему-либо совпали.

**Пример шаблона для поиска денежной суммы, записываемой в формате "10000 руб. 00 коп."**

[0-9]\{1,\} руб\. [0-9]\{2\} коп\.

Необходимое пояснение: если в модификаторе типа \{min,max\} отсутствует и запятая, и максимальное значение, то такая конструкция задаёт точное количество ожидаемых повторов элемента шаблона. В нашем примере определяются ровно два цифровых символа для обозначения копеек.

**Пример шаблона для поиска URL-строки, соответствующей Web-ресурсу в Интернете:**

http://[a-zA-Z0-9]\{1,\}\.[-a-zA-Z0-9\_]\{1,\}/\*

Необходимое пояснение: дефис теряет своё специальное значение, если он указан в самой первой позиции сразу после открывающей квадратной скобки в диапазоне. По данному шаблону могут быть найдены и такие "экзотические" URL-строки, как, например, http://my.home-server/

В формате расширенных регулярных выражений этот шаблон можно было бы записать более компактно:

http://[a-zA-Z0-9]+\.[-a-zA-Z0-9\_]+/\*

**Шаблон для поиска любого HTML-тэга выглядит на удивление просто:**

<[^>]+>

Совпадает с любой последовательностью символов за исключением > в количестве от одного и более, заключённой в угловые скобки. Иными словами, будет найден и односимвольный тэг <p>, и более "многословные" тэги, подобные <h1 class="empty">.

**Вариант шаблона для поиска дат**

Расширенные регулярные выражения позволяют написать несколько громоздкий, но тем не менее корректно работающий шаблон для поиска дат, имеющих вид "13 ноября 2009 г.":

[12]?[0-9] (янв|фев|мар|апр|мая|июн|июл|авг|сен|окт|ноя|дек).\* [0-9][0-9][0-9][0-9] г\.

Недостаток этого шаблона заключается в том, что с его помощью невозможно найти даты из древней истории, например, "13 ноября 245 г." или 1 января 88 г.", но для работы с современными документами он вполне годится (учитываем контекст поиска!).

**Практическое применение нумерованных частей шаблона**

В предыдущем разделе я уже приводил пример шаблона для поиска палиндромов. Его функциональность также можно немного улучшить, если переписать выражение следующим образом:

\(.\)\(.\)\(.\)\3\2\1

С помощью такого шаблона можно находить шестисимвольные палиндромы не только на английском, но и на русском и на любых других языках, а также последовательности символов, не относящихся к алфавитным, например /\*!!\*/

Более практичным способом использования запомненных и пронумерованных частей шаблона является поиск стоящих рядом повторяющихся слов, что позволяет обнаружить такие часто встречающиеся в текстах ошибки (опечатки), как "для для". Шаблон можно записать так:

\<\(..\*\)\> \<\1\>

**Ограничение размера совпадающей части шаблона**

Ещё одна особенность "характера" регулярных выражений – они являются неимоверно "жадными" (greedy), т.е. стремятся обеспечить совпадение с как можно более длинной строкой. Из-за этой "жадности" могут возникать неожиданные проблемы. Например, имеется шаблон для поиска любого количества символов, заключённых в кавычки:

".\*"

Строки, в которых производится поиск, имеют следующий вид:

"Петров" "охранник"

"Иванов" "отдел снабжения" "экспедитор"

"Сидоров" "администрация" "директор"

Если была поставлена задача извлечения из данных строк только первого аргумента (фамилия сотрудника), то предложенный выше шаблон выполнит её некорректно, поскольку вторая кавычка шаблона соответствует последней кавычке строки (из-за стремления получить максимальное совпадение). Изменение шаблона:

".\*" ".\*"

решает проблему только для первой строки, а во второй и третьей к фамилии подцепляется ещё и место работы – опять не то, что нам нужно!

Данная задача корректно решается с помощью регулярного выражения, соответствующего самому короткому из всех возможных фрагментов строки, расположенному между двумя кавычками:

"[^"]\*"

Здесь после открывающей кавычки должно следовать любое количество символов, не являющихся кавычками, до тех пор, пока не встретится завершающая эту последовательность кавычка.

##### Группы с захватом содержимого и именованные группы

на примере Python

Регулярные выражения могут использовать множество групп, как для захвата необходимой подстроки, так и для группировки и структурирования самих РВ. В сложных регулярных выражениях становится трудно отслеживать номера групп. Есть две особенности, которые помогают справиться с этой проблемой. Обе из них используют общий синтаксис для расширения регулярных выражений, который мы поэтому и рассмотрим сначала.  
  
В Perl 5 было добавлено несколько дополнительных функций для стандартных регулярных выражений, и модуль re поддерживает большинство из них. Было бы сложно выбрать новые односимвольные метасимволы или новые последовательности с бэкслешем, для того чтобы представить новые особенности так, чтобы регулярные выражения Perl без путаницы отличались от стандартных регулярных выражений. Если выбрать в качестве нового метасимвола, например, &, то старые регулярные выражения принимали бы его как обычный символ и нельзя было бы экранировать его \& или [&].  
  
Решение, выбранное разработчиками Perl состояло в том, чтобы использовать в качестве расширения синтаксиса (?...). Знак вопроса после скобки в случае обычного РВ это синтаксическая ошибка, поскольку ? нечего повторять, так что это не приводит к каким-либо проблемам в совместимости. Символы сразу после ? показывают какое расширение используется, так (?=foo) это одно (положительное утверждение о предпросмотре), а (?:foo) это что-то другое (группа без захвата содержимого, включающая подвыражение foo).  
  
К расширенному синтаксису Perl в Python добавляется собственное расширение. Если первый символ после знака вопроса P, то это означает что используется расширение, специфичное для Python. В настоящее время существуют два таких расширения: (?P<some\_name>... )определяет именованную группу, а (?P=some\_name) служит для нее обратной ссылкой. Если в будущих версиях Perl 5 добавятся аналогичные возможности, использующие другой синтаксис, модуль re будет изменен для поддержки нового синтаксиса, сохраняя при этом для совместимости Python-синтаксис.  
  
Иногда вам нужно использовать группу для сбора части регулярного выражения, но вы не заинтересованы в извлечении содержимого группы. Вы можете сделать это, используя группу без захвата содержимого: (?:...), где вы можете заменить ... любым другим регулярным выражением:

>>> m = re.match("([abc])+", "abc")  
>>> m.groups()  
('c',)  
>>> m = re.match("(?:[abc])+", "abc")  
>>> m.groups()  
()

За исключением того, что вы не получаете содержимого того, с чем совпала группа, эти группы ведут себя также, как и обычные; вы можете в них поместить что угодно, повторить с помощью соответствующего символа, такого как \*, и вставить их в другие группы (собирающие данные или нет).  
  
Более важной особенностью являются именованные группы: вместо ссылки на них по номерам, на эти группы можно ссылаться по имени.  
  
Синтаксис именованных групп это одно из специфичных Python-расширений: (?P<some\_name>...). Именованные группы ведут себя в точности как обычные, но вдобавок к этому ассоциируются с каким-то именем. Методы MatchObject, которые использовались для обычных групп принимают как числа, ссылающиеся на номер группы, так и строки, содержащие имя необходимой группы. То есть именованные группы все также принимают и числа, так что вы можете получить информацию о группе двумя способами:

>>> p = re.compile(r'(?P<word>\b\w+\b)')  
>>> m = p.search( '(((( Lots of punctuation )))' )  
>>> m.group('word')  
'Lots'  
>>> m.group(1)  
'Lots'

Именованные группы удобны тем, что они позволяют использовать вместо цифр легко запоминающиеся имена. Вот пример регулярного выражения из модуля imaplib:

InternalDate = re.compile(r'INTERNALDATE "'  
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'  
        r'(?P<year>[0-9][0-9][0-9][0-9])'  
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'  
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'  
        r'"')

Синтаксис обратных ссылок в регулярном выражение типа (...)\1 ссылается на номер группы. Более естественно было бы использовать вместо номеров имена групп. Другое Python расширение: (?P=name) показывает, что содержимое названной группы снова должно быть сопоставлено в текущей позиции. Наше прежнее регулярное выражение для поиска дублирующихся слов, (\b\w+)\s+\1 может быть также записано как (?P<doble\_word>\b\w+)\s+(?P=doble\_word):

>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')  
>>> p.search('Paris in the the spring').group()  
'the the'