Регулярные выражения

Материал из Википедии — свободной энциклопедии

Регуля́рные выраже́ния (англ. *regular expressions*) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. *wildcard characters*). Для поиска используется строка-образец (англ. *pattern*, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.

Содержание

1 Возможности
2 История
3 В теории формальных языков
4 Синтаксис
4.1 Представление символов
4.1.1 Обычные символы (литералы) и специальные символы (метасимволы)
4.1.2 Любой символ
4.1.3 Символьные классы (наборы символов)
4.2 Позиция внутри строки
4.3 Обозначение группы
4.4 Перечисление
4.5 Квантификация (поиск последовательностей)
4.5.1 Жадная и ленивая квантификация
4.5.2 Ревнивая квантификация (Сверхжадная)
4.6 Группировка
4.6.1 Обратная связь
4.6.2 Группировка без обратной связи
4.6.3 Атомарная группировка
4.6.4 Модификаторы
4.6.5 Комментарии

Заглавная страница
Рубрикация
Указатель А—Я
Избранные статьи
Случайная статья
Текущие события

Участие
Сообщить об ошибке
Портал сообщества
Форум
Свежие правки
Новые страницы
Справка
Пожертвовать

Инструменты
Ссылки сюда
Связанные правки
Спецстраницы
Постоянная ссылка
Сведения о странице
Элемент Викиданных
Цитировать страницу

Печать/экспорт
Создать книгу
Скачать как PDF
Версия для печати

Возможности [править вики-текст]

Регулярные выражения произвели прорыв в электронной обработке текстов в конце XX века.[*источник не указан 278 дней*] Набор утилит (включая редактор sed и фильтр grep), поставляемых в дистрибутивах UNIX, одним из первых способствовал популяризации регулярных выражений для обработки текстов. Многие современные языки программирования имеют встроенную поддержку регулярных выражений. Среди них ActionScript, Perl, Java[1],PHP, JavaScript, языки платформы .NET Framework[2], Python, Tcl, Ruby, Lua, Gambas, C++ (стандарт 2011 года), Delphi, D, Haxe и другие.

Регулярные выражения используются некоторыми текстовыми редакторами и утилитами для поиска и подстановки текста. Например, при помощи регулярных выражений можно задать шаблоны, позволяющие:

- найти все последовательности символов *«кот»* в любом контексте, как то: *«кот»*, *«котлета»*, *«терракотовый»*;
- найти отдельно стоящее слово *«кот»* и заменить его на *«кошка»*;
- найти слово *«кот»*, которому предшествует слово *«персидский»* или *«чеширский»*;
- убрать из текста все предложения, в которых упоминается слово *кот* или *кошка*.

Регулярные выражения позволяют задавать и гораздо более сложные шаблоны поиска или замены.

Результатом работы с регулярным выражением может быть:

- проверка наличия искомого образца в заданном тексте;
- определение подстроки текста, которая сопоставляется образцу;
- определение групп символов, соответствующих отдельным частям образца.

Română
Simple English
Slovenčina
Shqip
Српски / srpski
Svenska
தமிழ்
ไทย
Türkçe
Українська
اردو
Tiếng Việt
Walon
中文
Править ссылки

Если регулярное выражение используется для замены текста, то результатом работы будет новая текстовая строка, представляющая из себя исходный текст, из которого удалены найденные подстроки (сопоставленные образцу), а вместо них подставлены строки замены (возможно, модифицированные запомненными при разборе группами символов из исходного текста). Частным случаем модификации текста является удаление всех вхождений найденного образца — для чего строка замены указывается пустой.

История [править вики-текст]

Истоки регулярных выражений лежат в теории автоматов, теории формальных языков и классификации формальных грамматик по Хомскому[3].

Эти области изучают вычислительные модели (автоматы) и способы описания и классификации формальных языков. В 1940-х гг. Уоррен Маккалок и Уолтер Питтс описали нейронную систему, используя простой автомат в качестве модели нейрона.

Математик Стивен Клини позже описал эти модели, используя свою систему математических обозначений, названную «регулярные множества».

Кен Томпсон встроил их в редактор QED, а затем в редактор ed под UNIX. С этого времени регулярные выражения стали широко использоваться в UNIX и UNIX-подобных утилитах, например в expr, awk, Emacs, vi, lex и Perl.

Регулярные выражения в Perl и Tcl происходят от реализации, написанной Генри Спенсером. Филип Хейзел разработал библиотеку PCRE (англ. *Perl-compatible regular expressions* — Perl-совместимые регулярные выражения), которая используется во многих современных инструментах, таких как PHP и Apache.

В теории формальных языков [править вики-текст]

Регулярные выражения состоят из констант и операторов, которые определяют множества строк и множества операций на них соответственно. На данном конечном алфавите Σ определены следующие константы:

- (*пустое множество*) ∅.
- (*пустая строка*) ε обозначает строку, не содержащую ни одного символа. Эквивалентно «».
- (*символьный литерал*) «*a*», где *a* — символ алфавита Σ.
- (*множество*) из символов, либо из других множеств.

и следующие операции:

- (*сцепление*, *конкатенация*) *RS* обозначает множество {αβ | α ∈ *R* & β ∈ *S*}. Например, {"boy", "girl"}{"friend", "cott"} = {"boyfriend", "girlfriend", "boycott", "girlcott"}.
- (*дизъюнкция*, *чередование*) *R*|*S* обозначает объединение *R* и *S*. Например, {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}.[4]

- (*замыкание Клини*, *звезда Клини*) *R** обозначает минимальное надмножество множества *R*, которое содержит ε и замкнуто относительно конкатенации. Это есть множество всех строк, полученных конкатенацией нуля или более строк из *R*. Например, {"Run", «Forrest»}* = {ε, «Run», «Forrest», «RunRun», «RunForrest», «ForrestRun», «ForrestForrest», «RunRunRun», «RunRunForrest», «RunForrestRun», …}.

Синтаксис [править вики-текст]

Представление символов [править вики-текст]

Основная статья: ***Представление символов в регулярных выражениях***

Обычные символы (литералы) и специальные символы (метасимволы) [править вики-текст]

Большинство символов в регулярном выражении представляют сами себя за исключением специальных символов `[` `]` `\` `/` `^` `$` `.` `|` `?` `*` `+` `(` `)` `{` `}` (в разных типах регулярных выражений этот набор различается, см. Разновидности регулярных выражений), которые могут быть экранированы символом `\` (обратная косая черта) для представления самих себя в качестве символов текста. Можно экранировать целую последовательность символов, заключив её между `\Q` и `\E`.

Пример	Соответствие
`a\.?`	`a.` или `a`
`a\\\\b`	`a\\b`
`a\[F\]`	`a[F]`
`\Q+-*/\E`	`+-*/`

Аналогично могут быть представлены другие специальные символы (набор символов, требующих экранирования, может отличаться в зависимости от конкретной реализации). Часть символов, которые в той или иной реализации не требуют экранирования (например, угловые скобки `<` `>`), могут быть экранированы из соображений удобочитаемости.

Любой символ [править вики-текст]

Метасимвол `.` (точка) означает один любой символ, но в некоторых реализациях исключая символ новой строки.

Вместо символа `.` можно использовать `[\s\S]` (все пробельные и непробельные символы, включая символ новой строки).

Символьные классы (наборы символов) [править вики-текст]

Набор символов в квадратных скобках `[` `]` именуется символьным классом и позволяет указать интерпретатору регулярных

выражений, что на данном месте в строке может стоять один из перечисленных символов. В частности, `[абв]` задаёт возможность появления в тексте одного из трёх указанных символов, а `[1234567890]` задаёт соответствие одной из цифр. Возможно указание диапазонов символов: например, `[А-Яа-я]` соответствует всем буквам русского алфавита, за исключением букв «Ё» и «ё».[5]

Если требуется указать символы, которые не входят в указанный набор, то используют символ `^` внутри квадратных скобок, например `[^0-9]` означает любой символ, кроме цифр.

Добавление в набор специальных символов путём экранирования — самый бесхитростный способ. Однако в современных регулярных выражениях унаследован также и традиционный подход — см. Традиционные регулярные выражения.

Некоторые символьные классы можно заменить специальными метасимволами:

Символ	Эквивалент	Соответствие
`\d`	`[0-9]`	Цифровой символ
`\D`	`[^0-9]`	Нецифровой символ
`\s`	`[ \f\n\r\t\v]`	Пробельный символ
`\S`	`[^ \f\n\r\t\v]`	Непробельный символ
`\w`	`[[:word:]]`	Буквенный или цифровой символ или знак подчёркивания
`\W`	`[^[:word:]]`	Любой символ, кроме буквенного или цифрового символа или знака подчёркивания

Позиция внутри строки [править вики-текст]

Следующие символы позволяют спозиционировать регулярное выражение относительно элементов текста: начала и конца строки, границ слова.

Представление	Позиция	Пример	Соответствие
`^`	Начало текста (или строки при модификаторе ?m)	`^a`	`a`aa aaa
`$`	Конец текста (или строки при модификаторе ?m)	`a$`	aaa aa`a`
`\b`	Граница слова	`a\b`	aa`a` aa`a`
		`\ba`	`a`aa `a`aa

`\B`	Не граница слова	`\Ba\B`	aaa aaa
`\G`	Предыдущий успешный поиск	`\Ga`	aaa aaa (поиск остановился на 4-й позиции — там, где не нашлось `a`)

Обозначение группы [править вики-текст]

Круглые скобки используются для определения области действия и приоритета операций. Шаблон внутри группы обрабатывается как единое целое и может быть квантифицирован. Например, выражение `(тр[ау]м-?)*` найдёт последовательность вида `трам-трам-трумтрам-трум-трамтрум`.

Перечисление [править вики-текст]

Вертикальная черта разделяет допустимые варианты. Например, `gray|grey` соответствует `gray` или `grey`. Следует помнить, что перебор вариантов выполняется слева направо, как они указаны.

Если требуется указать перечень вариантов внутри более сложного регулярного выражения, то его нужно заключить в группу. Например, `gray|grey` или `gr(a|e)y` описывают строку `gray` или `grey`. В случае с односимвольными альтернативами предпочтителен вариант `gr[ae]y`, так как сравнение с символьным классом выполняется проще, чем обработка группы с проверкой на все её возможные модификаторы и генерацией обратной связи.

Квантификация (поиск последовательностей) [править вики-текст]

Квантификатор после символа, символьного класса или группы определяет, сколько раз предшествующее выражение может встречаться. Следует учитывать, что квантификатор может относиться более чем к одному символу в регулярном выражении, только если это символьный класс или группа.

Представление	Число повторений	Пример	Соответствие
`{n}`	Ровно *n* раз	`colou{3}r`	`colouuur`
`{m,n}`	От *m* до *n* включительно	`colou{2,4}r`	`colouur`, `colouuur`, `colouuuur`
`{m,}`	Не менее *m*	`colou{2,}r`	`colouur`, `colouuur`, `colouuuur` и т. д.
`{,n}`	Не более *n*	`colou{,3}r`	`color`, `colour`, `colouur`, `colouuur`

Представление	Число повторений	Эквивалент	Пример	Соответствие
`?`	Ноль или одно	`{0,1}`	`colou?r`	`color`, `colour`

`*`	Ноль или более	`{0,}`	`colou*r`	`color`, `colour`, `colouur` и т. д.
`+`	Одно или более	`{1,}`	`colou+r`	`colour`, `colouur` и т. д. (но не `color`)

Часто используется последовательность `.*` для обозначения любого количества любых символов между двумя частями регулярного выражения.

Символьные классы в сочетании с квантификаторами позволяют устанавливать соответствия с реальными текстами. Например, столбцами цифр, телефонами, почтовыми адресами, элементами HTML-разметки и др.

Если символы `{` `}` не образуют квантификатор, их специальное значение игнорируется.

Жадная и ленивая квантификация [править вики-текст]

В некоторых реализациях квантификаторам в регулярных выражениях соответствует максимально длинная строка из возможных (квантификаторы являются *жадными*, англ. *greedy*). Это может оказаться значительной проблемой. Например, часто ожидают, что выражение `(<.*>)` найдёт в тексте теги HTML. Однако если в тексте есть более одного HTML-тега, то этому выражению соответствует целиком строка, содержащая множество тегов.

```
<p><b>Википедия</b> — свободная энциклопедия, в которой <i>каждый</i> может изменить или дополнить любую статью.</p>
```

Эту проблему можно решить двумя способами.

1. Учитывать символы, *не* соответствующие желаемому образцу (`<[^>]*>` для вышеописанного случая).
2. Определить квантификатор как *нежадный* (*ленивый*, англ. *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
`*`	`*?`
`+`	`+?`

Пример использования жадных и ленивых выражений

Выражение `(<.*>)` соответствует строке, содержащей несколько тегов HTML-разметки, целиком.

```
<p><b>Википедия</b> — свободная энциклопедия, в которой <i>каждый</i> может изменить или дополнить любую статью.</p>
```

Чтобы выделить отдельные теги, можно применить ленивую версию этого выражения: `(<.*?>)` Ей соответствует не вся показанная выше строка, а отдельные теги (выделены цветом):

```
<p><b>Википедия</b> — свободная энциклопедия, в которой <i>каждый</i> может изменить или дополнить любую статью.</p>
```

`{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.

Ревнивая квантификация (Сверхжадная) [править вики-текст]

В отличие от обычной (жадной) квантификации, ревнивая (possessive) квантификация не только старается найти максимально длинный вариант, но ещё и не позволяет алгоритму возвращаться к предыдущим шагам поиска для того, чтобы найти возможные соответствия для оставшейся части регулярного выражения.

Использование квантификаторов увеличивает скорость поиска, особенно в тех случаях, когда строка не соответствует регулярному выражению. Кроме того, ревнивые квантификаторы могут быть использованы для исключения нежелательных совпадений.

При поиске выражения `(a+a+)+a` в строке `aaaaa` интерпретатор пойдёт приблизительно по следующему пути:

1. `aaaaa`
2. `aaaa`
3. `aaaaa`
4. `aaa`
5. `aaaaa`
6. `aaaa`
7. `aaaaa` — и только тут, проверив все точки возврата, остановится.

При использовании ревнивого квантификатора будет выполнен только первый шаг алгоритма.

Жадный	Ревнивый
`*`	`*+`
`?`	`?+`
`+`	`++`
`{n,}`	`{n,}+`

Пример	Соответствие
`ab(xa)*+a`	`abxaabxaa`; но не `abxaabxaa`, так как буква **a** уже занята

Это аналогично атомарной группировке.

Группировка [править вики-текст]

Обратная связь [править вики-текст]

Одно из применений группировки — повторное использование ранее найденных групп символов (*подстрок*, *блоков*, *отмеченных подвыражений*). При обработке выражения подстро́ки, найденные по шаблону внутри группы, сохраняются в отдельной области памяти и получают номер начиная с единицы. Каждой подстроке соответствует пара скобок в регулярном выражении. Квантификация группы не влияет на сохранённый результат, то есть сохраняется лишь первое вхождение. Обычно поддерживается до 9 нумерованных подстрок с номерами от 1 до 9, но некоторые интерпретаторы позволяют работать с бо́льшим количеством. Впоследствии в пределах данного регулярного выражения можно использовать обозначения от `\1` до `\9` для проверки на совпадение с ранее найденной подстрокой.

Например, регулярное выражение `(та|ту)-\1` найдёт строку `та-та` или `ту-ту`, но пропустит строку `та-ту`.

Также ранее найденные подстро́ки можно использовать при замене по регулярному выражению. В таком случае в замещающий текст вставляются те же обозначения, что и в пределах самого выражения.

Группировка без обратной связи [править вики-текст]

Если группа используется только для группировки и её результат в дальнейшем не потребуется, то можно использовать группировку вида `(?:`*`шаблон`*`)`. Под результат такой группировки не выделяется отдельная область памяти и, соответственно, ей не назначается номер. Это положительно влияет на скорость выполнения выражения, но понижает удобочитаемость.

Атомарная группировка [править вики-текст]

Атомарная группировка вида `(?>`*`шаблон`*`)`, также как и группировка без обратной связи, не создаёт обратных связей. В отличие от неё, такая группировка запрещает возвращаться назад по строке, если часть шаблона уже найдена.

Пример	Соответствие	Создаваемые группы
`a(bc|b|x)cc`	`abcc`axcc abcc`axcc`	a`bc`caxcc abcca`x`cc
`a(?:bc|b|x)cc`	`abcc`axcc, abcc`axcc`	нет
`a(?>bc|b|x)cc`	abcc`axcc` но не `abcc`axcc: вариант `x` найден, остальные проигнорированы	нет
`a(?>x*)xa`	не найдётся `axxxa`: все `x` заняты, и нет возврата внутрь группы	

Атомарная группировка выполняется ещё быстрее, чем группировка без обратной связи, и сохраняет процессорное время при

выполнении остального выражения, так как запрещает проверку любых других вариантов внутри группы, когда один вариант уже найден. Это очень полезно при оптимизации групп со множеством различных вариантов.

Это аналогично ревнивой квантификации.

Модификаторы [править вики-текст]

Модификаторы действуют с момента вхождения и до конца регулярного выражения или противоположного модификатора. Некоторые интерпретаторы могут применить модификатор ко всему выражению, а не с момента его вхождения.

Синтаксис	Описание	
`(?i)`	Включает	нечувствительность выражения к регистру символов (англ. *case insensitivity*)
`(?-i)`	Выключает	
`(?s)`	Включает	режим соответствия точки символам переноса строки и возврата каретки
`(?-s)`	Выключает	
`(?m)`	Символы `^` и `$` вызывают соответствие только	после и до символов новой строки
`(?-m)`		с началом и концом текста
`(?x)`	Включает	режим без учёта пробелов между частями регулярного выражения и позволяет использовать `#` для комментариев
`(?-x)`	Выключает	

Группы-модификаторы можно объединять в одну группу: `(?i-sm)`. Такая группа включает режим `i` и выключает режим `s`, `m`. Если использование модификаторов требуется только в пределах группы, то нужный шаблон указывается внутри группы после модификаторов но перед двоеточием. Например, `(?-i)(?i:tv)set` найдёт `TVset`, но не `TVSET`.

Комментарии [править вики-текст]

Для добавления комментариев в регулярное выражение можно использовать группы-комментарии вида `(?#комментарий)`. Такая группа интерпретатором полностью игнорируется и не проверяется на вхождение в текст. Например, выражение `А(?#тут комментарий)Б` соответствует строке `АБ`.

Просмотр вперёд и назад [править вики-текст]

В большинстве реализаций регулярных выражений есть способ производить поиск фрагмента текста, «просматривая» (но не включая в найденное) окружающий текст, который расположен до или после искомого фрагмента текста. Просмотр с отрицанием используется реже и «следит» за тем, чтобы указанные соответствия, напротив, не встречались до или после искомого текстового

фрагмента.

Представление	Вид просмотра	Пример	Соответствие
`(?=шаблон)`	Позитивный просмотр вперёд	`Людовик(?=XVI)`	`ЛюдовикXV, ЛюдовикXVI, ЛюдовикXVIII, ЛюдовикLXVII, ЛюдовикXXL`
`(?!шаблон)`	Негативный просмотр вперёд (с отрицанием)	`Людовик(?!XVI)`	`ЛюдовикXV, ЛюдовикXVI, ЛюдовикXVIII, ЛюдовикLXVII, ЛюдовикXXL`
`(?<=шаблон)`	Позитивный просмотр назад	`(?<=Сергей )Иванов`	`Сергей Иванов, Игорь Иванов`
`(?<!шаблон)`	Негативный просмотр назад (с отрицанием)	`(?<!Сергей )Иванов`	`Сергей Иванов, Игорь Иванов`

Поиск по условию [править вики-текст]

Во многих реализациях регулярных выражений существует возможность выбирать, по какому пути пойдёт проверка в том или ином месте регулярного выражения на основании уже найденных значений.

Представление	Пояснение	Пример	Соответствие
`(?(?=если)то|иначе)`	Если операция просмотра успешна, то далее выполняется часть *то*, иначе выполняется часть *иначе*. В выражении может использоваться любая из четырёх операций просмотра. Следует учитывать, что операция просмотра нулевой ширины, поэтому части *то* в случае позитивного или *иначе* в случае негативного просмотра должны включать в себя описание шаблона из операции просмотра.	`(?(?<=а)м|п)`	`мам,пап`
`(?(n)то|иначе)`	Если *n*-я группа вернула значение, то поиск по условию	`(а)?(?(1)м|п)`	`мам,пап`

	выполняется по шаблону `то`, иначе по шаблону `иначе`.		

Флаги [править вики-текст]

В некоторых языках (например, в JavaScript) реализованы т. н. «флаги», которые расширяют функции регэкспа. Флаги указываются после регулярного выражения (порядок флагов значения не имеет). Типичные флаги:

- **g** — глобальный поиск (обрабатываются все совпадения с шаблоном поиска).
- **i** — регистр букв не имеет значения;
- **m** — многострочный поиск.
- **s** — текст трактуется как одна строка, в этом случае метасимволу . (точка) соответствует любой одиночный символ, включая символ новой строки;

Флаг указывается после паттерна, например, вот так: `/[0-9]$/m`

Разновидности регулярных выражений [править вики-текст]

Базовые регулярные выражения POSIX [править вики-текст]

(англ. *basic regular expressions* (BRE)). Традиционные регулярные выражения UNIX. Синтаксис базовых регулярных выражений на данный момент определён POSIX как устаревший, но он до сих пор широко распространён из соображений обратной совместимости. Многие UNIX-утилиты используют такие регулярные выражения по умолчанию.

В данную версию включены метасимволы:

- `.`
- `[` `]`
- `[^` `]`
- `^` (действует только в начале выражения)
- `$` (действует только в конце выражения)
- `*`
- `\{` `\}` — первоначальный вариант для `{` `}`
- `\(` `\)` — первоначальный вариант для `(` `)`
- `\n`, где *n* — номер от 1 до 9

Особенности:

- Звёздочка должна следовать после выражения, соответствующего единичному символу. Пример: `[xyz]*`.
- Выражение `\(блок\)*` следует считать неправильным. В некоторых случаях оно соответствует нулю или более повторений строки `блок`. В других оно соответствует строке `блок*`.
- Внутри символьного класса специальные значения символов, в основном, игнорируются. Особые случаи:
  - Чтобы добавить символ `^` в набор, его следует поместить туда не первым.
  - Чтобы добавить символ `-` в набор, его следует поместить туда первым или последним. Например:
    - шаблон DNS-имени, куда могут входить буквы, цифры, минус и точка-разделитель: `[-0-9a-zA-Z.]`;
    - любой символ, кроме минуса и цифры: `[^-0-9]`.
  - Чтобы добавить символ `[` или `]` в набор, его следует поместить туда первым. Например:
    - `[][ab]` соответствует `]`, `[`, `a` или `b`.

Расширенные регулярные выражения POSIX [править вики-текст]

(англ. *extended regular expressions* (ERE)). Синтаксис в основном аналогичен традиционному.

- Отменено использование обратной косой черты для метасимволов `{ }` и `( )`.
- Обратная косая черта перед метасимволом отменяет его специальное значение (см. Представление специальных символов).
- Отвергнута теоретически **нерегулярная** конструкция `\n`.
- Добавлены метасимволы `+`, `?`, `|`.

См. также: Символьные классы POSIX

Регулярные выражения, совместимые с Perl [править вики-текст]

*Основная статья: **PCRE***

Perl-совместимые регулярные выражения (англ. *Perl-compatible regular expressions* (PCRE)) имеют более богатый и в то же время предсказуемый[*источник не указан 2676 дней*] синтаксис, чем даже POSIX ERE. По этой причине очень многие приложения используют именно Perl-совместимый синтаксис регулярных выражений.

Нечёткие регулярные выражения [править вики-текст]

В некоторых случаях регулярные выражения удобно применить для анализа текстовых фрагментов на естественном языке, то есть написанных людьми, и возможно содержащих опечатки, либо нестандартные варианты употреблений слов. Например, если проводить опрос (допустим, на веб-сайте) «какой станцией метро вы пользуетесь», может оказаться что «Невский проспект» посетители могут указать как:

- Невский
- Невск. просп.
- Нев. проспект
- наб. Канала Грибоедова («Канал Грибоедова» это название второго выхода ст.м. Невский Проспект)

Здесь обычные регулярные выражения неприменимы, в первую очередь из-за того что входящие в образцы слова могут совпадать не очень точно (нечётко), но тем не менее было бы удобно описывать регулярными выражениями структурные зависимости между элементами образца, например, в нашем случае, указать, что совпадение может быть с образцом «Невский проспект» ИЛИ «Канал Грибоедова», притом «проспект» может быть сокращено до «пр» или отсутствовать, а перед «Канал» может находиться сокращение «наб.»

Эта задача сродни полнотекстовому поиску, отличаясь в том, что здесь короткий фрагмент должен сравниваться с набором образцов, а при полнотекстовом поиске, наоборот, образец обычно один, в то время как фрагмент текста очень большой, или задаче разрешения лексической многозначности, которая, однако, не позволяет задать структурирующие отношения между элементами образца.

Существует небольшое количество библиотек, реализующих механизм регулярных выражений с возможностью нечёткого сравнения:

- TRE — бесплатная библиотека на C, использующая синтаксис регулярных выражений, похожий на POSIX (стабильный проект);
- FREJ — open-source библиотека на Java, использующая Lisp-образный синтаксис и лишённая многих возможностей обычных регулярных выражений, но сосредоточенная на различного рода автоматических заменах фрагментов текста (бета-версия).

Реализации [править вики-текст]

- NFA (англ. *nondeterministic finite-state automata* — недетерминированные конечные автоматы) используют жадный алгоритм 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 (этого требует стандарт POSIX, и существуют модификации NFA выполняющие это требование — GNU sed). Именно такой механизм регулярных выражений используется, например, в Perl, Tcl и .NET.
- DFA (англ. *deterministic finite-state automata* — детерминированные конечные автоматы) 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 lex и egrep.

См. также [править вики-текст]

- Шаблон поиска

Регулярные выражения в Викиучебнике?

Регулярные выражения на Викискладе?

Примечания [править вики-текст]

1. ↑ docs.oracle.com
2. ↑ MSDN
3. ↑ *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ. — Мир. — М., 1978. — Т. 2.
4. ↑ Во многих книгах используются символы ∪, + или ∨ вместо |.
5. ↑ Для использования последовательностей букв необходимо установить правильную кодовую страницу, в которой эти последовательности будут идти в порядке от и до указанных символов. Для русского языка это Windows-1251, ISO 8859-5 и Юникод, так как в DOS-855, DOS-866 и KOI8-R русские буквы не идут одной целой группой или не упорядочены по алфавиту. Отдельное внимание следует уделять буквам с диакритическими знаками, наподобие русских Ё/ё, которые, как правило, разбросаны вне основных диапазонов символов.

Литература [править вики-текст]

- *Фридл, Дж.* Регулярные выражения. — СПб.: «Питер», 2001. — 352 с. — (Библиотека программиста). — ISBN 5-318-00056-8.
- *Смит, Билл.* Методы и алгоритмы вычислений на строках (regexp) = Computing Patterns in Strings. — М.: «Вильямс», 2006. — 496 с. — ISBN 0-201-39839-7.
- *Форта, Бен.* Освой самостоятельно регулярные выражения. 10 минут на урок = Sams Teach Yourself Regular Expressions in 10 Minutes. — М.: «Вильямс», 2005. — 184 с. — ISBN 5-8459-0713-6.
- *Ян Гойвертс, Стивен Левитан.* Регулярные выражения. Сборник рецептов. — СПб.: «Символ-Плюс», 2010. — 608 с. — ISBN 978-5-93286-181-3.
- *Мельников С. В.* Perl для профессиональных программистов. Регулярные выражения. — М.: «Бином», 2007. — 190 с. — (Основы информационных технологий). — ISBN 978-5-94774-797-3.

Ссылки [править вики-текст]

- Справочник и ресурсы по регулярным выражениям (англ.) — учебник и детальное описание синтаксиса с примерами, сравнение различных интерпретаторов и прочее
- MSDN — Знакомство с регулярными выражениями (рус.)
- Реализация механизма обработки регулярных выражений на языке C++
- Теория и методика построения регулярных выражений. Проблема самообразования (рус.)
- Онлайн-генератор регулярных выражений на PHP для новичков (рус.)

- Конструктор регулярных выражений с подсветкой синтаксиса (англ.)
- TRE — Бесплатная и переносимая библиотека для нечёткого сравнения с помощью регулярных выражений
- FREJ (Fuzzy Regular Expressions for Java) — Нечёткие регулярные выражения для Java на SourceForge.net

Категории: Программирование | Формальные языки | Интерфейс командной строки | Сопоставление с образцом