****

**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

**Факултет „Компютърни системи и технологии“**

**Допълнение към курсова работа по „Програмни среди“**

Изработил: Любен Димитров Кирев №121217059, група 39, КСИ, направление ИТ

Титуляр дисциплина: проф. д-р инж. Огнян **Наков**

Ръководител упражнения: ас. маг. инж. Петър **Маринов**

Дата: 12.06.2020 година

Тема: „Regular Expressions”

Регулярните изрази (на англ. Regular Expressions) или накратко Regex представляват поредица от знаци, които определят модел за търсене. Най-често такива модели се използват от алгоритми за търсене на низове за операциите за намиране или намиране и заместване, както и за валидиране на низ от входни данни. Тези операции често се използват за търсене в търсещи машини или текстови редактори, както и при подаване на входни данни от потребителския интерфейс към back end-а при редица приложения. В днешно време повечето езици имат вградена поддръжка на регулярните изрази, а където липсва, в повечето случаи има библиотеки.

Реализирането на regex се осъществява чрез метазнаци със специално значение или обикновен знак с буквално значение. Примерно в регулярния израз „**а.**“ „а“ е знаков литерал, който означава същоименната буква, а „.“ Е метазнак, който означава произволен знак, различен от нов ред.

Примери:

* ^The – намира всеки низ който започва със символите след ^(в случая думата The)
* abc+ - намира всеки низ, в който има символите ab, последвани от c
* \d – намира един символ, представляващ число
* \$\d – намира низ, който представлява единствено число, преди което има знака $(например за цена)
* [abc] – намира всеки низ,в който се съдържа a, b или c. Еквивалентно на a|b|c
* \bcat\b – намира дума, поставена между \b символите(в случая cat)
* d(?!r) – намира буквата ‘d’, само ако не е последвана от ‚r‘, но връща само ‘d’
* С вертикална черта се разделят алтернативи. Например colour|color съвпада с "colour" и "color".

Квантори:

Представляват добавени след даден синтактичен елемент или група от елементи указвайки колко срещания на елемента се допускат. Най-често се срещат ? \* +.

* ? – Означава нула или едно срещане на предходен елемент. (honou?r honor/honour)
* \* - Означава нула или повече срещания на предходния елемент. (ab\*c => “ac”, “abc”, “abbc”, „abbbc” …)
* + - Означава едно или повече срещания на предходен елемент. (ab+c => “abc”, “abbc”, “abbbc”, но не и с “ac”)
* {n} – Допускат се точно n-поредни съответсвия на предходния елемент.
* {min} - Допускат се min или повече поредни съответствия на предходния елемент.
* {min, max} – Допускат се поне min, но не повече от max поредни съответствия на предходния елемент.

Заместители:

Заместващия символ „.“ Съвпада с произволен знак. Например „a.b” съвпада с всеки низ, който съдържа „a”, последван от произволен знак и след това „b”, а „a.\*b” съответства на всеки низ, който съдържа „a”, последвано след някакъв брой значи от „b”. Тези конструкции могат да бъдат комбинирани, за да се изграждат произволно сложни изрази, точно както аритметичните изрази могат да се конструират от числа и операциите + - . : .

* a|b\* обозначава множеството {ε, "a", "b", "bb", "bbb", …}
* (a|b)\* обозначава множеството от всички думи, съставени само от символите „a” или „b”, включително празната дума {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", …}
* ab(c|ε) обозначава множеството от думите, започващи с „a”, последвано от нула или повече „b” и накрая незадължително от „c” {"a", "ac", "ab", "abc", "abb", "abbc", …}
* (0|(1(01\*0)\*1))\* обозначава множеството на двоичните числа, кратни на 3: { ε, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", … }
* Конкатенация RS обозначава множеството от думите, които могат да се получат чрез конкатениране на дума от R и дума от S. Например, ако R = {"ab", "c"} и S = {"d", "ef"}, то RS = {"abd", "abef", "cd", "cef"}.
* Алтернацията R | S обозначава обединението на множествата, описани от R и S. например, ако R описва {"ab", "c"} и S описва {"ab", "d", "ef"}, то изразът R | S описва {"ab", "c", "d", "ef"}.
* Звезда на Клийни R\* - Обозначава най-малкото надмножество на множеството, описвано от R, което съдържа ε и е затворено спрямо конкатенацията. Това е множеството на всички думи, които могат да се получат чрез конкатениране на произволен кран брой (включително нула) думи от множеството, описвано от R. Примерно {"0","1"}\* е множество на всички крайни двоични низове (включително празния) а {"ab", "c"}\* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", … }

Регулярните изрази намират много широко приложение в много сфери. Някои примери са за валидация на входни данни (подсигуряване на правилен формат и подредба), извличане на данни от уеб страници (web scraping), преобразуване на данни в друг формат, заменяне на стринг, подчертаване и уеднаквяване на текст, търсене в, преименуване и редактиране на файлове. Често използвана функция базирана на регулярни изрази е функцията find in text, която съществува под различни форми във повечето съвременни текстови редактори. Тази функционалност работи на принципа на regular expressions, като търси точно подаден низ, във цялата страница. Някои търсачки позволяват ползването освен на буквални символи, така и метасимволи.

Съществуват различни Regex Engines, като повечето от тях не са съвместими един с друг. Всеки обработва низа от буквални и метасимволи по различни начини и за това съвместимостта представлява известен проблем. Повечето езици за програмиране включват собствен, като най-добре отговаря на изискванията и формулировката на езика и средата. Има много различни, но примери за такива са .NET, Java, JS, PHP, Python и други. Въпреки, че работят на един и същ принцип, има съществени разлики като как определени метасимволи се интерпретират и каква дължина от даден текст се филтрира, как се третират празните пространства и новите редове. Основните разлики са в интерпретацията на метасимволите, тъй като буквалните символи винаги са еднакви и при тях се прави проста съпоставка със ASCII стойностите на сравнявания символ и сравняващия. RegEx работи като минава пред определен низ, който трябва да се филтрира и пробва всички възможни пермутации в зададения ред, след което преминава на следващия символ. При обработка на текст, никога не се връща назад. Това не е необходима, тъй като работата на двигателя е просто сравнение по даден шаблон и няма къде да възникне грешка. При обработване на низ, без значение дали е един символ или повече, принципа на работа е един и същ. Пробвайки даденият шаблон, с всяка възможна пермутация, връщайки резултата, било то дали съвпада или не.

Поради самият принцип на работа, на регулярните изрази, а именно множество сравнения, в зависимост от размера на зададения ред и филтрирания текст процесът може да бъде доста дълъг и да изисква сравнително много ресурс, от машината. В повечето случаи от всекидневието не се филтрират толкова големи количества текст, и за тази цел е напълно достатъчно да се ползва директно простия алгоритъм за сравнения и най-просто реализиран двигател. За случаите, когато обаче бързодействието е важно и се цели оптимизация на необходимите ресурси се използват по-развити двигатели. Поради огромният набор от метасимволи в комбинациите, които са възможни и съответно тези, които трябва да се оптимизират, са на практика безкрайни, но е възможна сериозна оптимизация на най-често използваните буквални/метасимволи. Примерна оптимизация би била претърсването на низ от тип „Певецът пее по-късно“, при използване на филтър състоящ се от „пе“. Повечето обикновени алгоритми сравняват първата буква на низа „п“ с първата от регулярния израз и тъй като тя пасва, преминават на следващата, която в случая също пасва на следващата от реда и съответно тези символи представляват съвпадения. След това работата продължава аналогично, до края на низа. В случая, когато търсачката стигне до третата дума в примера, тя вижда, че първата буква съвпада с първата буква от шаблона и проверява втората, която обаче не съвпада. Т този случай тя преминава буквата „п“ и отива на буквата „о“ въпреки, че вече е преминала през нея, за да я сравни с втория символ от шаблона. Такива повторни проверки представляват забавяния, които биха могли да се спестят. Това се прави като се използва малка по размер временна памет, която позволява да се запомнят символи, които скоро са били филтрирани.

Важни правила за използване на RegEx:

* Не трябва да използваме регулярни изрази за „Parse-ване“ на файлове/данни когато вече има готови библиотеки за това. Примерно за XML или HTML, където вече е пълно с тествани библиотеки, готови да свършат цялата работа само с извикването на 1 метод.
* Когато използването на регулярни изрази за търсене стане твърде товарещо за програмата ни. Тогава можем да се опитаме да използваме някакъв вид integer ID за всеки ресурс, по които да търсим, понеже сравнението на низове е много по-бавно от колкото сравнението между числа.
* Регулярните изрази често се ползват в Уеб апликациите, където има често търсене и заместване на текстови данни от JavaScript. Също се ползват за валидация на данни от потребител. Пример: Потребителят се регистрира на нашият сайт, попълвайки формата за регистрация. Ние взимаме паролата и с регулярен израз проверяваме дали паролата отговаря на изискванията: Да има минимум 1 главна буква, да има специални символи(#,@,!)и дали притежава забранени от системата ни символи.
* Регулярните изрази могат да бъдат имплементирани и в търсенето на данни от дадена база данни, където трябва да селектираме само специални записи от базата ни.

**Източници:**

<https://swtch.com/~rsc/regexp/regexp1.html>

<https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285>

<https://www.regular-expressions.info/engine.html>

<https://regex101.com/r/cO8lqs/25>

<https://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html>