EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/B/2019/36103158109441796

REGULÁRNE VÝRAZY

Bakalárska práca

Bratislava 2019 **Oliver Juran**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

REGULÁRNE VÝRAZY

Bakalárska práca

Študijný program: Hospodárska informatika

Študijný odbor: Hospodárska informatika

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Miroslav Kršjak, PhD.

Bratislava 2019 **Oliver Juran**

Čestné vyhlásenie

Vyhlasujem, že som svoju záverečnú prácu vypracovala samostatne na základe vlastných teoretických a praktických poznatkov s využitím uvedenej literatúry.

V Bratislave, dňa 29.7.2019 **……**

Podpis študenta

Poďakovanie

Ďakujem svojmu školiteľovi za to, že bol vždy ochotný sa mi venovať aj počas letných prázdnin a vždy odpovedal na moje otázky. Taktiež by som sa mal poďakovať svojej priateľke, ktorá ma v rámci možností psychicky zvládala aj napriek tomu, že som všetko robil na poslednú chvíľu.

ABSTRAKT

JURAN, Oliver: *Regulárne výrazy.* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Miroslav Kršjak, PhD. – Bratislava: FHI EU, 2019, 45 strán

Cieľom záverečnej práce je podrobne popísať tvorbu regulárnych výrazov na príkladoch v konkrétnom programovacom jazyku. Jej cieľom je teda aj oboznámenie čitateľa s regulárnymi výrazmi. Práca obsahuje 19 obrázkov, päť tabuliek a pozostáva z troch kapitol. V prvej z nich si v skratke prejdeme históriu a vývoj regulárnych výrazov a taktiež základné pojmy regulárny výraz a regex engine. V ďalšej časti sa charakterizuje hlavný a čiastkové ciele a popisuje sa metodika práce. Posledná kapitola sa zaoberá podrobným popisom jednotlivých znakov používaných pri tvorbe regulárnych výrazov v jazyku JavaScript a taktiež sa na niektorých príkladoch vysvetľuje fungovanie regex enginu. Za význam a prínos tejto práce považujem schopnosť čitateľa po jej preštudovaní dokázať samostatne tvoriť regulárne výrazy v jazyku JavaScript a vo veľkej časti aj v iných jazykoch.

Kľúčové slová:

Regex, regulárne výrazy, regex engine, JavaScript

ABSTRACT

JURAN, Oliver: *Regular expressions.* – University of Economics in Bratislava. Faculty of Economic Informatics; Department of applied informatics. Head of thesis: Ing. Miroslav Kršjak, PhD. – Bratislava: FHI EU, 2019, 45 pages

The main purpose of this final thesis is to describe the creation of regular expressions in a particular programming language. It´s meaning is also to familiarize the reader with the problematics of regular expressions. The thesis includes 19 pictures, 5 panels and consists of 3 chapters. The first chapter briefly introduces the history of regular expressions and also clarifies the terms “regular expressions” and “regex engine”. The next chapter defines the main and partial objects and describes the methodology. The last chapter focuses on detailed description of single characters used in creation of the regular expressions in JavaScript and also exemplarily explains the functioning of the regex engine. The purpose and benefit of this thesis stands up to the ability of the reader following the article to create regular expressions in the JavaScript language and others individually.

Key words:

Regex, regular expressions, regex engine, JavaScript

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

Úvod

Regulárne výrazy sú bežným pojmom vo svete informačných technológií. Vo veľkej miere sa používajú pri programovaní. Ich uplatnenie spočíva v rôznych vyhľadávacích algoritmoch aplikovaných na reťazce, no aj pri „find and replace“ operáciách vykonávaných na reťazcoch. Taktiež zohrávajú dôležitú úlohu pri overení vstupu používateľa programu, aplikácie či informačného systému ako celku. Je tomu tak preto, lebo nesprávny vstup by mohol neskôr spôsobovať rôzne problémy.

Obsahom prvej kapitoly tejto práce je história a vývoj regulárnych výrazov, a taktiež informujeme čitateľa o niektorých ľuďoch, ktorí sa významnou mierou pričinili o ich rozvoj. V tejto časti práce si taktiež zadefinujeme základné pojmy, teda regulárny výraz a regex engine. Zároveň si aj rozdelíme znaky používané pri tvorbe regulárnych výrazov do základných skupín. Nakoľko regex enginov je viac druhov, základné z nich si taktiež definujeme.

V strednej časti práce stanovíme a popíšeme základný cieľ tejto práce, ako aj čiastkové ciele našej práce. Ich súrhn bude mať dôležitú úlohu pri hodnotení ich naplnenia. V tejto kapitole si taktiež popíšeme metodiku práce a spôsoby využité pri tvorení praktických príkladov.

V poslednej kapitole si následne podrobne vysvetlíme fungovanie jednotlivých prvkov používaných pri tvorbe regulárnych výrazov v jazyku JavaScript. Nakoľko sú mnohé prvky z veľkej časti podobné, niektoré aj totožné s regulárnymi výrazmi v iných programovacích jazykoch, môže táto práca pomáhať pri tvorbe regulárnych výrazov aj v iných jazykoch. Do podrobna si v niektorých vybraných príkladoch vysvetlíme aj fungovanie regex enginu. V úplnom závere práce sú tabuľkovým spôsobom skonsolidované všetky znaky používané pri tvorbe regulárnych výrazov v jazyku JavaScript.

Účelom tejto práce je oboznámiť a vzdelať čitateľa v oblasti regulárnych výrazov a pomôcť mu pri práci s nimi.

1 Súčasný stav riešenej problematiky

Regulárny výraz je v podstate vzor, ktorý opisuje určitý text. Jeho názov pochádza z matematickej teórie. Pojem „regulárny výraz“ sa zvyčajne vyskytuje v skrátenej forme – „regex“, ktorá pochádza z anglického názvu „Regular Expressions“. V rámci regulárnych výrazov existujú rôzne skupiny znakov:

- Jednoduché znaky - najjednoduchší regulárny výraz pozostáva z jedného jednoduchého znaku. Ide napríklad o akýkoľvek znak abecedy.
- Špeciálne znaky - nakoľko je naším cieľom viac ako len vyhľadávanie podľa jednoduchých znakov, je treba niektoré znaky rezervovať pre špecifické účely. Tieto znaky sa môžu v rôznych programovacích jazykoch líšiť, ale v princípe ide o rôzne zátvorky, bodky, otázniky, znak plus a ďalšie. Tieto špeciálne znaky sa zvyknú nazývať metaznaky.
- Netlačiteľné znaky – je možné použiť sekvenciu špeciálnych znakov na vloženie netlačiteľného znaku do regulárneho výrazu. Napríklad „\t” nájde v texte tab, „\n“ nový riadok a ďalšie. Výhodou je, že všetky znaky sa dajú vyhľadať aj podľa ich ASCII kódu. Napríklad tab sa dá zapísať do regulárneho výrazu nasledovne: „\x09”. [10,11]

1.1 Regex engine

Regex engine je softvérový prvok, ktorý spracováva regulárne výrazy – snaží sa nájsť zhodu so vzorom v prehľadávanom texte. Zvyčajne je tento engine súčasťou väčšej aplikácie a priamo k nemu nepristupujeme. Aplikácia má vedieť rozpoznať, kedy je engine potrebný. Rozličné regex enginy spolu nie sú plne kompatibilné.

Existujú dva hlavné typy regex enginov, a to riadený textom a riadený regexom. Regex enginy riadené regexom vždy vrátia iba prvú zhodu z ľava. Text je prechádzaný zľava doprava po riadkoch, a aj napriek tomu, že niekde hlbšie v texte sa môže nachádzať možno lepšia alebo presnejšia zhoda, engine vráti prvú nájdenú zhodu. Keď je na nejaký reťazec aplikovaný regulárny výraz, regex engine začne kontrolovať prvý znak tohto reťazca. Na tomto znaku overí všetky možné permutácie daného regulárneho výrazu. Regex engine prejde na druhý znak reťazca iba v tom prípade, ak boli vyskúšané všetky možnosti a zároveň zhoda nebola nájdená. Opätovne sa overujú všetky permutácie regulárneho výrazu. Výsledkom je, že regexom riadený engine vráti iba prvú zhodu z ľava. V online testeri, ktorý budeme v tejto práci používať, sa dá

nastaviť či má engine vyhľadať iba prvú zhodu, alebo aj ďalšie. Slúži na to modifikátor „g“, teda global. Hovorí nám, že sa bude vyhľadávať globálne, teda všade. V praxi to znamená, že engine neprestane pracovať po nájdení prvej zhody. Tento online nástroj sa nachádza na adrese https://regex101.com/.

Jedným zo spôsobov ako zistiť, či je daný engine riadený textom alebo regexom je overiť, či sú k dispozícii lenivé kvantifikátory alebo spätná referencia. Ak sú k dispozícii, ide o regexom riadený engine. [10,11,12]

1.2 Vývoj a história regulárnych výrazov

Hoci sa to môže zdať zvláštne, počiatky regulárnych výrazov sú spojené s vedami skúmajúcimi neuróny. V deväťdesiatych rokoch devätnásteho storočia začali vedci, počnúc Guillaumeom-Benjaminom-Armandom Duchenneom de Boulogne, skúmať, ako fungujú neuróny. [1]

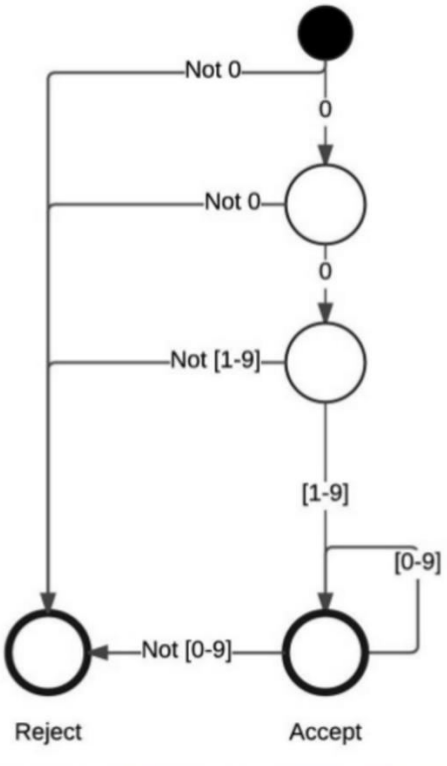
V roku 1943 neurovedec Warren S. McCulloch a logik Walter Pitts publikovali dielo „*Logický počet myšlienok imanentných v nervovej aktivite*“, v ktorom sa pokúšali pochopiť fungovanie mozgu na základe modelovania neurónov. Vytvorili modely, ktoré opisovali ako funguje ľudský nervový systém, alebo ako by mohli byť zostavené prístroje a počítače tak, aby sa správali viac ako ľudský mozog. V tom čase to netušili, no ich práca mala neskôr obrovský vplyv na počítačové vedy.

O niečo neskôr, v roku 1951, sa Stephan Kleene rozhodol dokázať, čo začali McCelloch a Pitts – že siete neurónov môžu počítať. Urobil tak v diele „*Representation of Events in Nerve Nets and Finite Automata*“. V tomto diele Kleene navrhol jednoduchú algebru, ktorá definovala Regulárny Jazyk – niečo, čo sa neustále používa v počítačovom programovaní. [1,2]

Regulárny jazyk je jazykom, ktorý môže byť vyjadrený regulárnym výrazom, alebo deterministickým, resp. nedeterministickým konečným automatom (finite automata), alebo stavovým diagramom. Jazyk je množina reťazcov, ktoré sa skladajú zo znakov zo špecifickej abecedy, alebo zo sady symbolov. Regulárne jazyky sú podmnožinou množiny všetkých reťazcov. Regulárne jazyky sa používajú pri analýze a navrhovaní programovacích jazykov a sú jedným z prvých konceptov vyučovaných na kurzoch výpočtových techník. Tie dokážu pomáhať počítačovým vedcom pri rozpoznávaní vzorov v dátach a zoskupovať určité výpočtové problémy – potom môžu použiť podobné prístupy na riešenie týchto zoskupených problémov. Regulárne jazyky sú kľúčovou témou v rámci výpočtovej teórie.

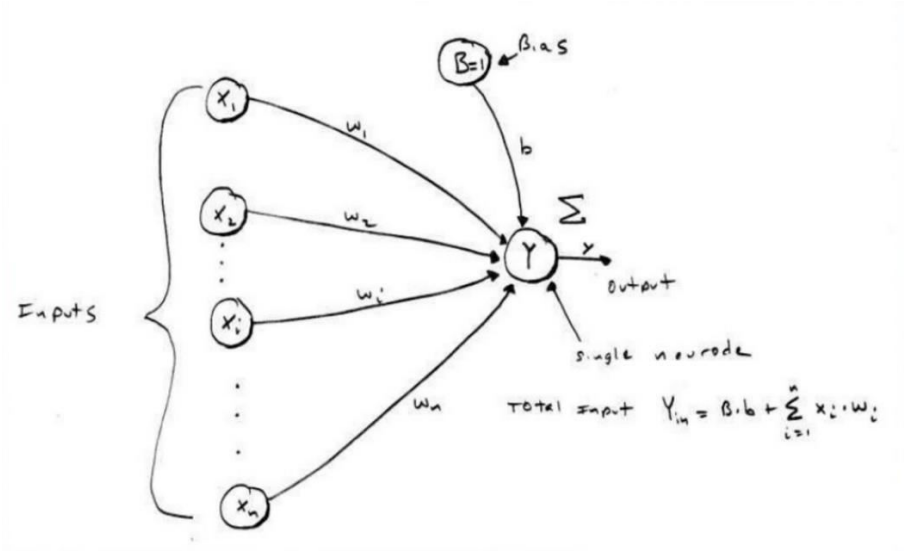
Konečný automat vytvára alebo modeluje regulárne jazyky. To znamená, že regulárny jazyk môže byť popísaný jednoduchým stavovým diagramom. Konečný stavový prístroj M opisuje konečný jazyk L. Prístroj M akceptuje reťazec w za predpokladu, že prístroj začína v počiatočnom stave, prejde zmenou alebo sériou zmien a končí v akceptovateľnom stave. Prístroj M rozpoznáva jazyk L, ak M akceptuje všetky reťazce w, ktorá patria do L. Jazyk nazývame regulárny jazyk ak existuje konečný automat, ktorý ho rozpozná. [3]

Môžeme si to vysvetliť na príklade. Majme abecedu {0,1,2,3,4,5,6,7,8,9} a slová {123 012334 23848484}. Aby tvorili jazyk, je potrebné vytvoriť nejaké pravidlá. Napríklad všetky slová v 00-jazyku musia začínať dvoma nulami. Ak je možné určiť, či je dané slovo v danom jazyku tým, že sekvenčne kontrolujeme jeho písmená ako to robí tento diagram, potom je jazyk regulárny. [1]



Obrázok č. 1: Konečný stavový prístroj – 00 jazyk

V roku 1956 Kleene opísal McCullochove a Pittsove modely algebrou, ktorú nazval regulárne sety a taktiež vytvoril zápis na ich vyjadrenie. Tento zápis nazval regulárne výrazy. Taktiež dokázal, že konečný stavový prístroj je ekvivalentný tomuto jeho náčrtu: [1,2]



Obrázok č. 2: Kleeneov náčrt

Regulárny jazyk môže byť definovaný používaním gramatiky s troma operátormi

1. Reťazenie – je to operácia, ktorá kombinuje dva symboly, reťazce či jazyky. Existujú dva spôsoby, ako zapísať reťazenie: X◦Y a XY. Oba zápisy znamenajú spojenie X a Y. Ako už bolo spomenuté, spájať sa môžu aj jazyky. Zápis L1◦L2, kde L1 a L2 sú rozličné jazyky, znamená, že najprv sa zapíšu reťazce z L1 a potom z L2. Reťazenie je definované nasledovne: $A \circ B = \{xy \mid x \in A \ \& \ y \in B\}$.
2. Zjednotenie – zjednotenie je operácia, pri ktorej konečný stavový prístroj môže vykonať jednu alebo inú možnosť. Dá sa to tiež považovať za „alebo“ operáciu. Môže napríklad existovať konečný stavový prístroj, ktorý sa môže rozhodnúť medzi reťazením X a Y alebo X a Z. „Alebo“ sa zapisuje ako vertikálna čiara – „|” a predchádzajúci výraz sa dá zapísať takto: (X◦Y|X◦Z). Formálne sa zjednotenie zapisuje nasledovne: $A \cup B = \{x \mid x \in A \vee x \in B\}$. V tomto prípade symbol „|” znamená „pri čom“.
3. Kleeneova hviezdica – Kleeneova hviezdica je operácia, ktorá sa zapisuje ako hviezdička – „*“. Táto operácia reprezentuje opakované sebareťazenie. Ak jazyk L reprezentuje všetky reťazce podľa zápisu X*, potom tento jazyk obsahuje reťazce X, XX, XXXXX, XXXXXXXX, atď. Kleeneova hviezdica môže daný symbol zopakovať koľkokoľvek krát. Regulárny jazyk X◦Y|Z* reprezentuje všetky reťazce, pre ktoré platí, že X reťazené s Y alebo Z zopakovaným koľkokoľvek krát. Po formálnej stránke je Kleeneova hviezdica definovaná nasledovne:

$A^* = \{x1,x2,\ldots,xk \mid k \geq 0 \text{ a } \forall \text{ xi } \epsilon A\}$. Kleeneova hviezdica je na rozdiel od reťazenie a zjednotenia unárny operátor. Keďže index k môže byť aj 0, x môže byť zopakované aj 0 krát. To znamená, že prázdny reťazec vždy patrí do A*. [3]

Nič, o čom sme si doteraz rozprávali, sa zatiaľ netýkalo počítačov – pohybovali sme sa iba do päťdesiatych rokov dvadsiateho storočia. Tieto myšlienky sú však veľmi dôležité pre počítačový svet. [1]

Rok 1968. Ken Thompson, skorý počítačový priekopník, publikoval článok „Regular expression search algorithm“, v ktorom opisuje metódu na lokáciu špecifického znakového reťazca vloženého v znakovom texte. Venuje sa taktiež implementácii tejto metódy vo forme kompilátora. „*Kompilátor akceptuje regulárny výraz ako zdrojový jazyk a vytvorí IBM 7094 program ako objektový jazyk. Kompilátor potom príjme text, ktorý má byť vyhľadaný, ako input a vyšle signál vždy, keď sa reťazec v texte zhoduje zo zadaným regulárnym výrazom.*“ (Thompson, 1968) Taktiež sú v tomto článku príklady, problémy a ich riešenia. [1,4]

O rok neskôr, v roku 1969 Ken Thompson, ako jeden z kľúčových vývojárov UNIXu, implementoval regulárne výrazy do skorého UNIXového textového editora nesúceho názov názov „ed“. Tento moment možno považovať za vstup regulárnych výrazov do počítačového sveta. Ak chceme v ede vyhľadať text pomocou regulárneho výrazu, používa sa zápis g/regular_expression/p, pri čom „g“ a „p“ sú modifikátory. „G“ znamená globally – vyhľadávať sa má globálne, teda všade a „p“ slúžili na output výsledku, na jeho vypísanie na obrazovku (print). Skrátene sa používa zápis grep – global regular expression print. [1,2]

Medzi rokmi 1970 a 1968 sa okrem vzniku a vývoju nových programovacích jazykov a štandardov vyvíjali aj grep a egrep. Písmeno „e“ pred grepom znamená „extended“, teda rozšírený. V regulárnych výrazoch sa teda začalo vyskytovať viac a viac nekompatibilností. [2]

V roku 1986 vnikol POSIX. Je to súbor noriem, štandardov, ktoré sú navrhnuté na zabezpečenie kompatibility medzi odlišnými operačnými systémami najmä na báze UNIXu. Samotný názov POSIX znamená Portable Operating System Interface a X je referencia na UNIX. Je to teda prenosné rozhranie pre operačný systém. Jeho cieľom bolo jednotné rozhranie, ktoré malo zabezpečiť prenosnosť programov a aplikácií medzi rôznymi hardwarovými platformami. V tom čase existovali dva druhy regulárnych výrazov - BREs a EREs. BREs znamenalo „Basic regular expressions“ a EREs „Extended regular expressions“. Teda jednoduché a rozšírené regulárne výrazy. Všetci programátori sa museli rozhodnúť, či implementujú BREs alebo EREs. Obe majú presne definovaný súbor pravidiel, kde má a kde

nemá byť nájdená zhoda, a takisto ktorý symbol čo znamená. V tom čase sa neočakávalo, že BREs a EREs by mohli byť vzájomne zameniteľné. [2,5,6]

V tomto roku taktiež Herny Spencer napísal v jazyku C regex knižnicu. Bola to prvá z troch knižníc pre regulárne výrazy, ktorú napísal. Na tom, že je to knižnica je výborné to, že môže byť zabudovaná do iných programov a teda zabezpečuje konzistentnosť, pretože ktokoľvek použije Spencerovu knižnicu, budú jeho regulárne výrazy fungovať rovnakým spôsobom. V tomto bode sa teda rôzne záležitosti ohľadom regulárnych výrazov stali konzistentnejšie a zmeny v nich sa podarilo stabilizovať. [2,7]

V roku 1987 vytvoril Larry Wall programovací jazyk Perl. Tento jazyk využíva Spencerovu knižnicu, no postupom času doňho boli pridané rôzne významné črty. Larry Wall sa snažil vytvoriť Perl tak, aby bol naozaj prospešný. Pre zvýšenie jeho užitočnosti boli doňho pridané rôzne črty. Medzi týmito črtami boli napríklad lenivé kvantifikátory, lookahead, nezachytávajúce skupiny/okrúhle zátvorky a ďalšie, ktoré sú dnes považované za samozrejmosť. Vďaka týmto črtám sa stal akýmsi zlatým štandardom pre fungovanie regexových knižníc. [2,8]

V súčasnosti existujú mnohé Perl-kompatibilné jazyky a programy. Apache, C, C++, .NET jazyky, Java, JavaScript, MySQL, PHP, Python, Ruby a ďalšie sa snažia byť Perl-kompatibilné jazyky a programy. Existuje taktiež knižnica nazývaná PCRE, teda Perl-compatible Regular Expression library. Je to sada funkcií, ktoré implementujú porovnávanie vzorov regulárnych výrazov pomocou rovnakej syntaxe a sémantiky ako Perl 5. Moderný syntax regulárnych výrazov používaných v dnešnej dobe pochádza práve z Perlu. [2,9]

2 Cieľ práce a metodika práce

V tejto časti bakalárskej práce sa zameriame na stanovenie a vymedzenie cieľa našej záverečnej práce a metodiky našej práce. Taktiež si bližšie charakterizujeme rôzne čiastkové ciele. Popíšeme si aj postupy, ktoré sme použili pri snahe o ich naplnenie.

2.1 Cieľ práce

Cieľom tejto práce je popísanie tvorby regulárnych výrazov na príkladoch v konkrétnom programovacom jazyku. V tejto práci si budeme popisovať syntax regulárnych výrazov používaný v programovacom jazyku JavaScript. Regulárne výrazy v tomto jazyku si budeme popisovať práve preto, lebo patrí k dlhodobo najpoužívanejším programovacím jazykom, a pravdepodobne tomu tak ešte nejakú dobu bude.

Za najdôležitejší čiastkový cieľ považujem skutočnosť, že čitateľ tejto práce by mohol byť po jej preštudovaní schopný samostatne používať a vytvárať regulárne výrazy. Je tomu tak najmä preto, lebo významnou súčasťou našej práce je popísanie regulárnych výrazov na konkrétnych príkladoch, ale aj podrobný popis fungovania regex enginu. Chceli by sme teda docieliť toho, aby táto záverečná práca mohla v budúcnosti slúžiť ako tutoriál pre každého, kto by sa chcel v tejto oblasti zaučiť, poprípade zdokonaľovať.

Dôležitým čiastkovým cieľom je aj zdokonalenie čitateľa v používaní regulárnych výrazov v iných programovacích jazykov, nakoľko syntax regulárnych výrazov v jazyku JavaScript je vo veľkej miere totožný s inými programovacími jazykmi.

Schopnosť vedieť používať a vytvárať komplexnejšie regulárne výrazy po podrobnom preštudovaní tejto záverečnej práce taktiež považujem za jej čiastkový cieľ. Je tomu tak najmä preto, lebo regulárne výrazy sú bežnou súčasťou života programátora a zohrávajú dôležitú úlohu pri overovaní inputu používateľa navrhnutého programu alebo aplikácie. Keďže používatelia informačných systémov bežne zadajú chybný, respektíve iný ako očakávaný input, je potrebné ho overiť. Ak teda chceme dostať správny, očakávaný input, je potrebné na tento účel vytvoriť dokonale fungujúci regulárny výraz. Ak teda používateľ danej aplikácie zadá nejaký input, regex engine sa pokúsi nájsť zhodu s našim regulárnym výrazom. Ak sa to nepodarí, používateľovi sa zobrazí chybová hláška. Keby tomu tak nebolo, nesprávny input by mohol neskôr spôsobovať problémy.

2.2 Metodika práce

Pri vypracovaní našej záverečnej práce sme využili rôzne poznatky a vedomosti, ktoré sme nadobudli počas štúdia na Fakulte hospodárskej informatiky, ale taktiež nové poznatky, ktoré sme nadobudli až počas študovania rôznych materiálov pred samotným písaním záverečnej práce. Išlo primárne o zahraničnú literatúru dostupnú na internete. Tieto materiály sme využili hlavne pri kapitole týkajúcej sa vývoja a histórie regulárnych výrazov.

Prvým krokom bolo zvolenie si programovacieho jazyka, ktorého regulárne výrazy budeme popisovať. Týmto jazykom je z už vyššie uvedeného dôvodu práve JavaScript.

V ďalšom kroku sme si podrobne preskúmali a analyzovali rôzne zahraničné zdroje zamerané na tvorbu regulárnych výrazov nie len v jazyku JavaScript. Po analýze prišla na rad syntéza týchto informácií. Postupne sme si tieto poznatky spájali do jedného rozsiahleho celku.

Následne sme v hlavnej časti bakalárskej práce popisovali jednotlivé znaky používané pri tvorbe regulárnych výrazov v jazyku JavaScript. Nakoľko bolo potrebné tieto znaky použiť v konkrétnych príkladoch, potrebovali sme na to použiť nejaký nástroj. Na tieto účely nám výborne poslúžil online nástroj nachádzajúci sa na adrese https://regex101.com/. Pri niektorých znakoch sme si taktiež podrobne popísali, ako funguje regex engine. Je to dôležitým krokom pri pochopení fungovania regulárnych výrazov.

Na konci tejto časti sme si taktiež vytvorili prehľadné tabuľky, v ktorých sa nachádzajú znaky používané pri tvorbe regulárnych výrazov v jazyku JavaScript.

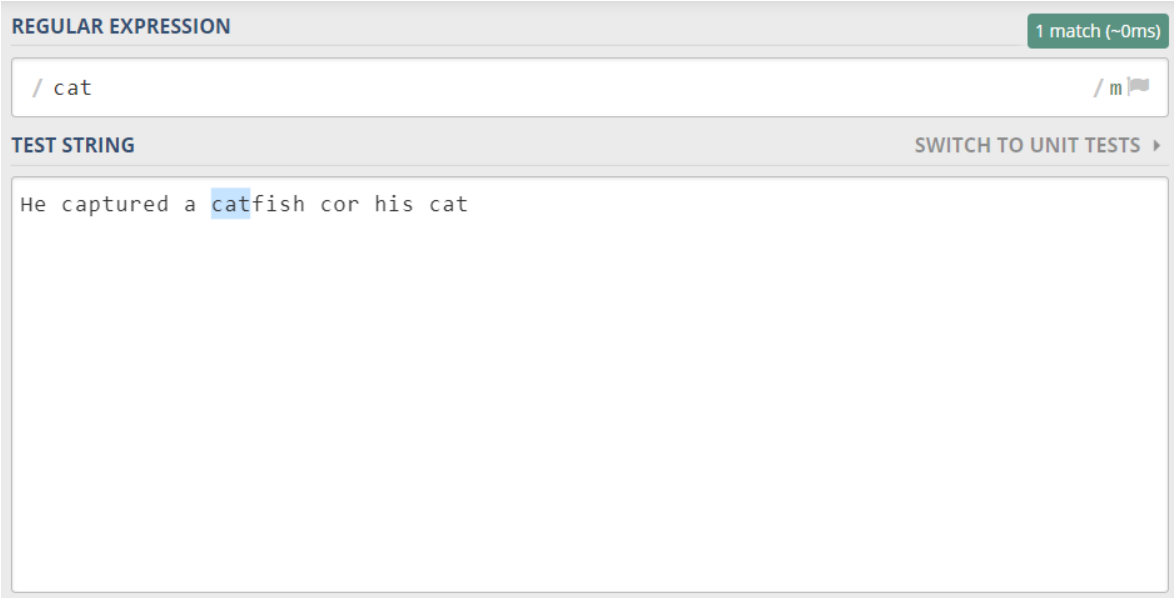
3 Výsledky práce

V tejto časti si podrobne popíšeme tvorbu regulárnych výrazov v jazyku JavaScript, ako aj fungovanie regex enginu na konkrétnych príkladoch. Čiastočne sa pri tom budeme opierať o zdroj číslo 10.

Na začiatok si popíšeme fungovanie regexom riadeného enginu na jednoduchom príklade. Regexom riadený engine budeme ďalej používať v každom jednom príklade.

3.1 Fungovanie regex enginu

Majme reťazec „He captured a catfish for his cat“, regulárny výraz „cat“ a engine nastavený na vyhľadanie iba prvej zhody. Engine sa pokusí nájsť zhodu prvého tokenu v regulárnom výraze, teda „c“ s prvým znakom v našom reťazci, teda „H“ – neúspešne. Na koľko náš regulárny výraz pozostáva iba zo sekvencie troch jednoduchých znakov, neexistujú žiadne ďalšie možnosti, engine prejde na druhý znak – „e“. Zhoda sa taktiež nenašla. Podarí sa ju nájsť až pri štvrtom znaku reťazca. Engine sa následne snaží nájsť zhodu druhého znaku nášho regulárneho výrazu, teda písmena „a“, s piatym znakom reťazca. Taktiež úspešne. Potom sa ale nenájde zhoda v treťom znaku nášho výrazu s piatym znakom v prehľadávanom reťazci. Znak „t“ sa totižto nezhoduje so znakom „p“. V tomto bode engine vie, že regulárny výraz „cat“ nebude mať zhodu začínajúcu štvrtým znakom v prehľadávanom texte. Prejde teda na piaty znak – „a“. Zhoda sa znova nenašla. Nájde sa až s pätnástym znakom v texte – „c“. Následne sa nájde zhoda aj v ďalších dvoch znakoch. Aj napriek tomu, že je tento reťazec „cat“ súčasťou dlhšieho reťazca „catfish“, nájde sa úplná zhoda. Je to preto, že nemáme nastavené žiadne slovné ohraničenia. Náš regulárny výraz sa teda zhoduje s textom počínajúcim pätnástym znakom prehľadávaného reťazca. Engine v tomto momente zahlási zhodu a prestane ďalej prehľadávať text. To aj napriek tomu, že sa v texte nachádza aj „lepšia“ zhoda. Táto je považovaná za dostatočnú.



Obrázok č. 3: Regulárny výraz „cat“

3.2 Jednoduché znaky

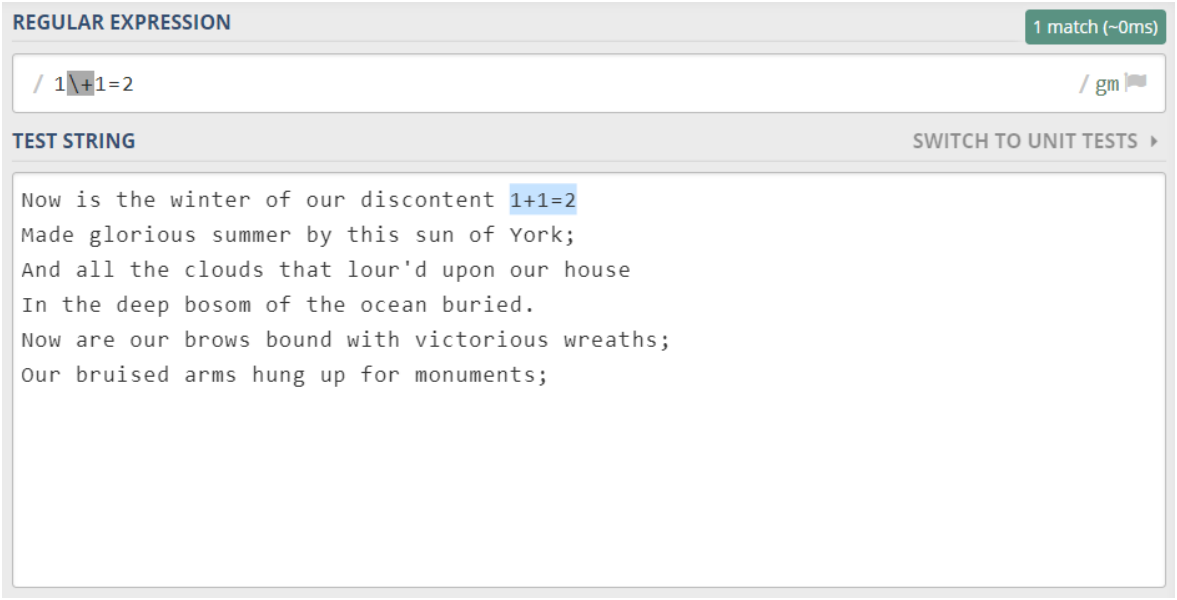
V kapitole „Súčasný stav riešenej problematiky“ sme sa už zoznámili s jednoduchými a špeciálnymi znakmi. Teraz si ich vo všeobecnosti ukážeme na príkladoch.

Pre znázornenie jednoduchého znaku si vezmeme znak „a“. V prehľadávanom texte bude teda zhoda všade, kde je písmeno „a“. Nezáleží pri tom, či je daný znak na začiatku, v strede, poprípade na konci slova. Ak by nám na tom záležalo, dajú sa použiť slovné ohraničenia. Tie si popíšeme neskôr v tejto kapitole. V texte „Moja záverečná práca“ sa teda nájdu dve zhody. Podobne, regulárny výraz „cat“ nájde reťazec „cat“ vo vete „About cats and dogs“. Tento regulárny výraz pozostáva zo série troch jednoduchých znakov. Regex engine v podstate nájde „c“, po ktorom nasleduje „a“ a „t“. Základné nastavenie regex enginu je citlivé na veľkosť písmen. Regulárny výraz „cat“ teda nebude mať zhodu s reťazcom „Cat“. Dá sa však napísať regulárny výraz tak, aby ignoroval rozdiely vo veľkých a malých písmenách. Slúži na to modifikátor „i“.

3.3 Špeciálne znaky

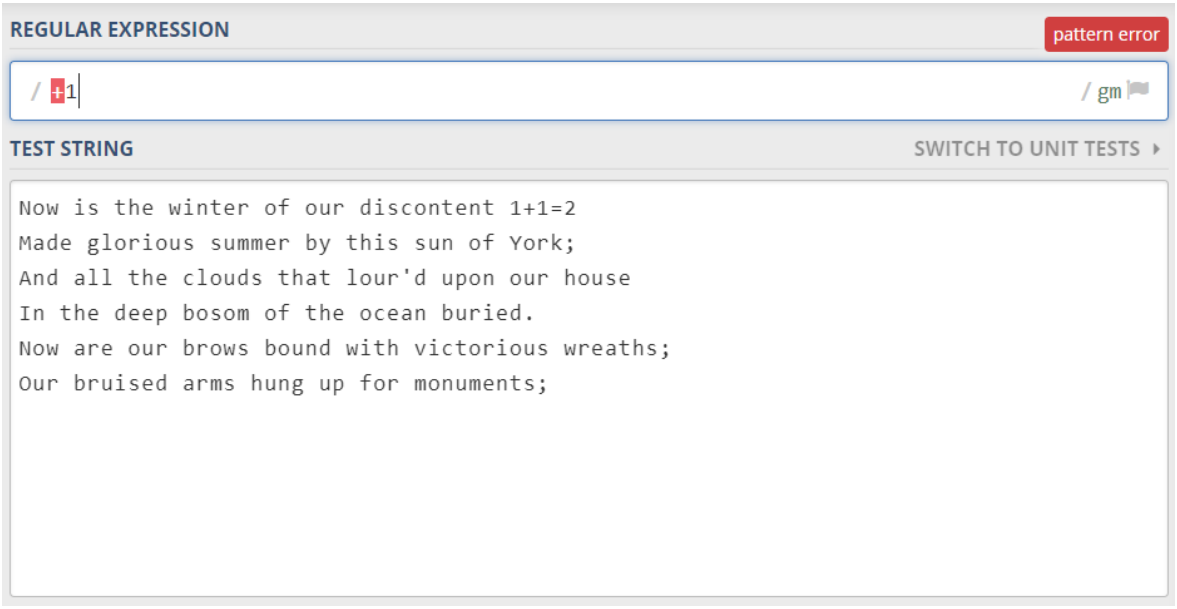
Ak chceme použiť niektorý zo špeciálnych znakov ako jednoduchý znak, je treba pred neho zadať backslash „\“. Ak chceme pomocou regulárneho výrazu vyhľadať reťazec „1+1=2“, regulárny výraz musí vyzerať takto: „1\+1=2”. Samozrejme, aj výraz „1+1=2“ je platný

regulárny výraz, ale nenašiel by zhodu v našom texte. Bol by platný pre reťazec, pre ktorý platí, že aspoň dve jednotky sú nasledované znakmi „=2“. Našiel by teda zhodu napríklad v reťazcoch „11=2“ alebo „111111111=2“. Znak „+“ znamená jeden alebo viac výskytov znaku alebo série znakov napísaných pred ním. Takýto metaznak je nazývaný kvantifikátor. Tým sa budeme taktiež venovať v ďalšej časti tejto práce. Náš vyššie uvedený príklad si môžeme overiť v našom online nástroji.



Obrázok č. 4: Regulárny výraz „1\+1=2“

Ak by sme ale špeciálny znak zabudli ohraničiť a použili tak, ako to nie je povolené, dostaneme chybové hlásenie. Je to tak napríklad v regulárnom výraze „+1“. Špeciálny znak „+“ je totiž možné použiť iba za iným znakom. Ako môžeme vidieť v pravom hornom rohu obrázka, dostávame chybové hlásenie „pattern error“.



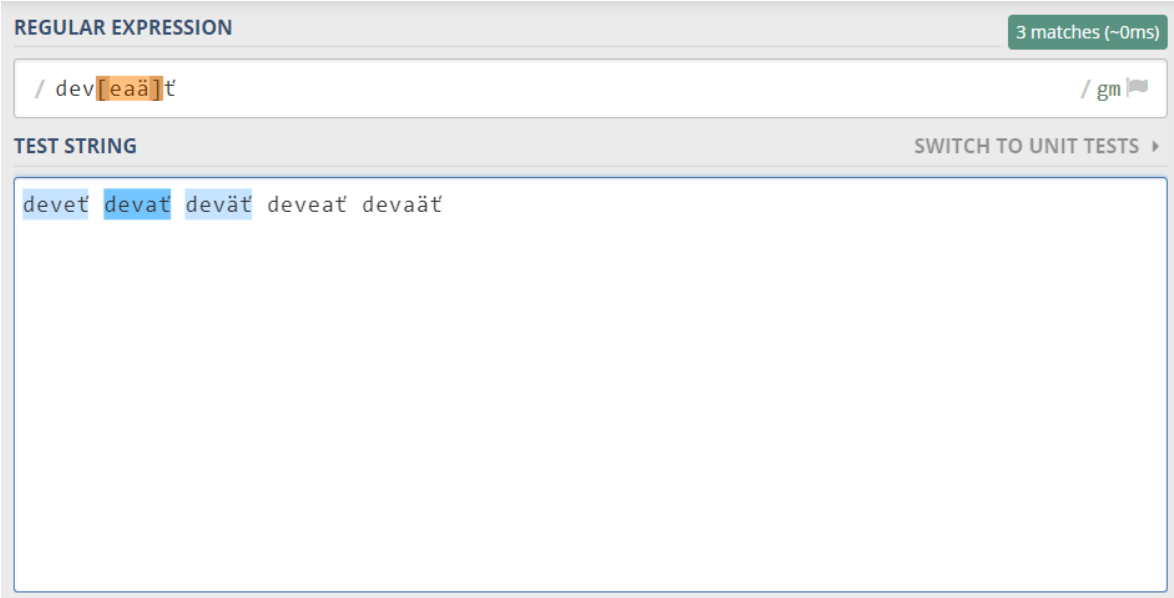
Obrázok č. 5: Regulárny výraz „+1“ – chybové hlásenie

Ostatné znaky by nemali byť používané s backslashom. Je to preto, lebo backslash je taktiež špeciálny znak. Pri použití s niektorými jednoduchými znakmi môže nechcene vytvoriť regulárny výraz so špeciálnym významom. Napríklad výraz „\d” bude mať zhodu s číslom od 0 do 9, „\D” nájde akýkoľvek nečíselný znak, „\w” zase „slovný znak“.

Neskôr v tejto časti si ešte podrobnejšie prejdeme niektoré najpoužívanejšie metaznaky a ich podrobné fungovanie.

3.4 Znakové triedy

Znakové triedy sa taktiež zvyknú nazývať znakové sady. S ich pomocou je prostredníctvom nich možné povedať regex enginu, aby vyhľadal iba jeden z viacerých znakov. Najjednoduchší spôsob je vložiť hľadané znaky do hranatých zátvoriek. Majme regulárny výraz „dev[eaä]ť”. Nájde nám slovo „deväť“ pričom nezáleží na tom, či je alebo nie je napísané správne. V slove musí byť ale napísaný iba jeden zo znakov v hranatej zátvorke. Nezáleží na tom, v akom poradí sú znaky v zátvorkách napísané. Ak by boli v danom slove napísané viaceré znaky z danej znakovej triedy za sebou, zhoda by sa nenašla.



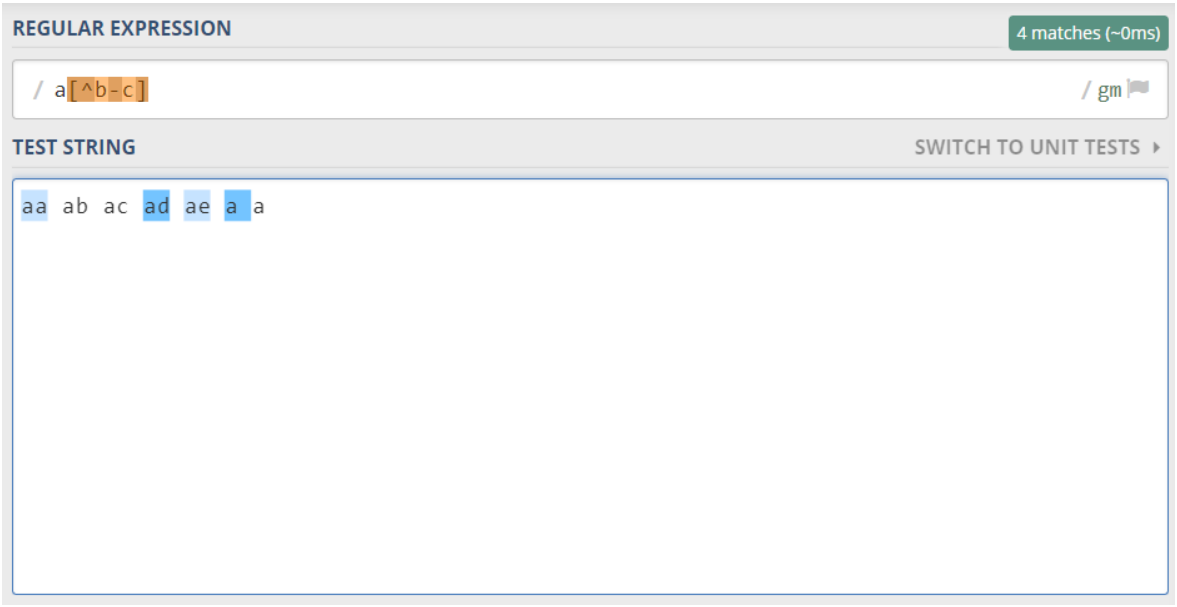
Obrázok č. 6: Regulárny výraz „dev[eaä]ť

Relatívne veľký význam má v súvislosti so znakovými triedami znak „-". Dá sa ním totižto v hranatých zátvorkách určiť interval vyhľadávaných znakov. Intervalov môže byť použitých aj viac. Regulárny výraz „[a-zA-Z]" nám vyhľadá všetky veľké aj malé znaky abecedy. Dá sa použiť aj v kombinácii s jednotlivými znakmi. Teda výraz „[a-zA-Z123]" nám okrem veľkých a malých písmen vyhľadá aj čísla jedna, dva a tri.

Pre doplnenie by bolo vhodné poznamenať, že znakové triedy sa môžu aj opakovať. Slúžia na to operátory „+", „*" a „?". Podrobnejšie si ich fungovanie vysvetlíme v ďalšej časti tejto práce. Regulárny výraz „[0-9]*" nám nájde zhodu rovnako ako v akomkoľvek jednocifernom čísle, tak aj v inom, viac-menej hocijako dlhom čísle. Tieto opakovacie znaky sa teda píšu za znakovú triedu.

3.4.1 Negované znakové *triedy*

Znakové triedy je možné aj negovať. Robí sa to pomocou takzvanej striešky „^". Zapísanie tejto striešky za úvodnú hranatú zátvorku znamená, že sa hľadajú znaky, ktoré nie sú v tejto znakovej triede. Je ale potrebné dodať, že negovaná znaková triedy sa musí taktiež zhodovať so znakom. Majme regulárny výraz „a[^b-c]". Znamená to, že znak „a" má byť nasledovaný znakmi inými ako „b" až „c". Aby bola ale nájdená zhoda, musí byť znak „a" nejakým znakom nasledovaný.



Obrázok č. 7: Regulárny výraz „a[^b-c]“

V poslednom znaku „a“ sa nenašla zhoda, preto sa za ním nenachádza žiadny iný, ani prázdny znak.

3.4.2 Metaznaky vo vnútri znakovej triedy

Jediné špeciálne znaky vo vnútri znakovej triedy sú uzatváracia hranatá zátvorka, backslash, strieška a pomlčka. Ostatné metaznaky vo vnútri hranatých zátvoriek sa považujú za jednoduché znaky, ktoré nie je potrebné osamostatniť backslashom. Ak chceme napríklad vyhľadať znaky plus a hviezdička, stačí na to takýto regulárny výraz: „[+*]“. Je to plne platný a fungujúci výraz, no vo vnútri dlhšieho regulárneho výrazu môže znížiť prehľadnosť. Ak chceme použiť backslash v znakovej sade bez jeho špeciálneho významu, treba pred neho napísať ešte jeden backslash. To isté platí aj pre ostatné z vyššie spomenutých metaznakov. Avšak v niektorých prípadoch sa to dá urobiť aj inak. Striešku napríklad stačí napísať niekde inde ako na začiatok znakovej sady. Pomlčku je možné napísať aj na začiatok a takto nebude mať špeciálny význam.

3.4.3 Skratky pre znakové triedy

Niektoré znakové triedy sú tak často používané, že sa pre ne vytvorili skratky. Ako sme si už spomínali v podkapitole „Špeciálne znaky“, napríklad výraz „\d“ má rovnaký význam ako „[0-9]“, a teda číslo od nula po deväť. Regulárny výraz „\w“ nám nájde vo väčšine syntaxov okrem „slovných znakov“ aj čísla od nula po deväť, „\s“ nám zase nájde prázdne znaky. Tieto

skrátene znakové triedy sa môžu aj negovať. V troch spomenutých prípadoch je to tak, že sa dané písmeno napíše veľké. Teda platný ekvivalent pre výraz „[^\d]“ je „\D“ a znamená nečíselný znak. Rovnako ekvivalentom pre „[^\w]“ je „\W“ a pre „[^\s]“ je to „\S“.

3.5 Operátor bodka

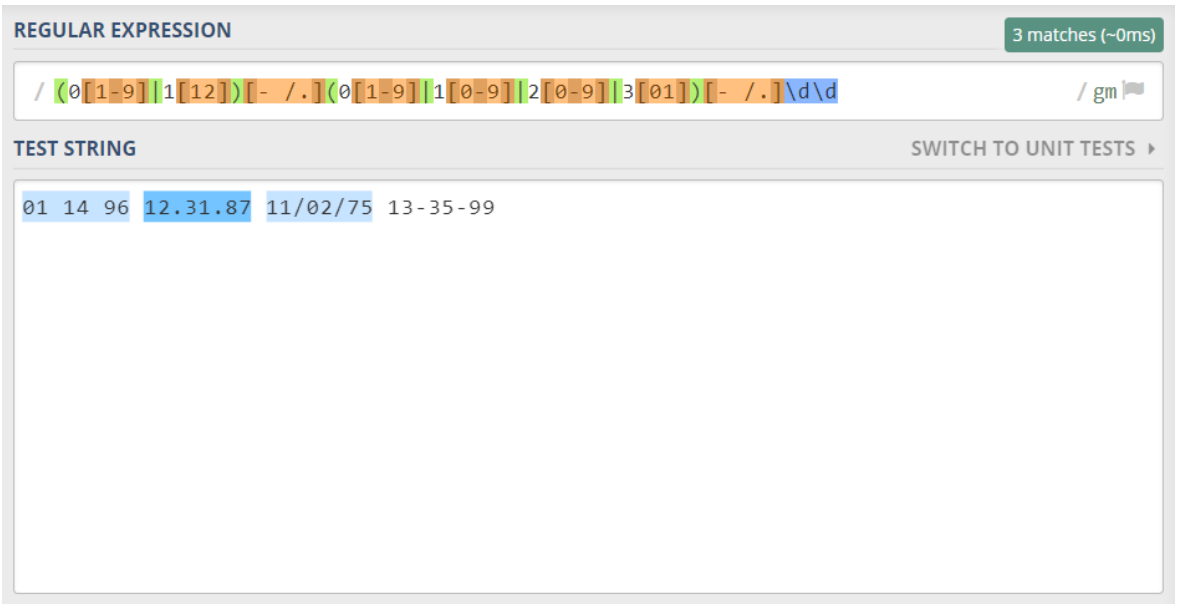
Operátor bodka „.“ v regulárnom výraze nahradzuje skoro akýkoľvek znak. Je to jeden z najpoužívanejších metaznakov. Nájde zhodu v podstate v každom znaku okrem neviditeľného znaku pre nový riadok. Takže by sme v podstate mohli povedať, že bodka je niečo ako skrátený zápis pre znakovú triedu „[^\n]”, ktorá taktiež nájde zhodu v každom znaku okrem nového riadku. Je to tak najmä z historických dôvodov. Prvé nástroje využívajúce regulárne výrazy boli totižto zostavené tak, že prechádzali súbor riadok po riadku a regulárny výraz aplikovali zvlášť na každý riadok. Výsledok bol taký, že pri používaní týchto nástrojov reťazec nemohol nikdy obsahovať neviditeľný znak pre nový riadok, a tak ich metaznak „.“ nemohol nájsť.

Moderné nástroje či jazyky môžu aplikovať regulárne výrazy na veľmi dlhé reťazce, alebo celé súbory. Väčšina z nich má možnosť nastaviť bodku tak, aby našla zhodu aj v znaku pre nový riadok. Každopádne, ak potrebujeme, na nájdenie každého existujúceho znaku sa dá použiť znaková trieda „[\s\S]“, teda každý neviditeľný a každý viditeľný znak.

Používať metaznak „.“ ale treba veľmi opatrne. Môže sa totižto stať, že nájde zhodu aj v takých znakoch, ako by sme si neželali. Pre ilustráciu si môžeme popísať nasledovný prípad. Dajme tomu, že chceme nájsť zhodu v dátume v „mm/dd/yy“ formáte, ale chceme nechať používateľa, aby si zvolil oddeľovač medzi jednotlivými číslami. Ako jednoduché riešenie sa javí regulárny výraz „\d\d.\d\d.\d\d“. Na prvý pohľad sa javí úplne v poriadku. Bez problémov nájde zhodu v dátume ako napríklad „01/14/96“ alebo „01-14-96“. Problém nastane, keď si niekto napríklad nepozrie inštrukcie alebo požadovaný formát a zadá „01141996“. Dátum je síce pre náš regulárny výraz plne platný, no prvá bodka bude znamenať znak „1“ a druhá „9“. A to teda z ďaleka nie je výsledok, aký sme očakávali. O niečo lepšou možnosťou sa môže zdať regulárny výraz „\d\d[- /.]\d\d[- /.]\d\d]“. Znak „.“ vo vnútri znakovej triedy neslúži ako metaznak, používateľ teda má možnosť použiť na oddelenie jednotlivých čísel pomlčku, medzeru, lomené a bodku. Tento výraz má však od dokonalosti taktiež ďaleko, keďže dátum ako „99/99/99“ by považoval za platný dátum.

To, aký dokonalý má regulárny výraz byť závisí od toho, na čo ho potrebujeme. Ak chceme overiť vstup od používateľa, musí byť perfektný. Skoro všetky dátumy dokáže správne

overiť tento regulárny výraz: „(0[1-9]|1[12])[- /.](0[1-9]|1[0-9]|2[0-9]|3[01])[- /.]\d\d“. Znak „|“ slúži ako „alebo“ operátor. Tento regulárny výraz najprv vyhodnocuje mesiac narodenia. Musí to byť jedna z možností 0 + 1 až 9, alebo 1 + 1 alebo 2. Potom bude považovať za platný oddeľovač vo forme pomlčky, medzery, lomené alebo bodky. V prípade platnosti dňa je to tiež viac možností. Prvou sú čísla 0 + 1 až 9, ďalšou 1 + 0 až 9, treťou 2 + 0 až 9 a poslednou 3 + 0 alebo 1. Znova sa kontrolujú oddeľovače. Ako rok budú platné akékoľvek dve čísla. Tento regulárny výraz však chybne vyhodnotí za platné dátumy ako 30. február a iné, ktoré nikdy nenastanú. Neráta s tým, že nie každý mesiac má 29 a viac dní. Na obrázku je možné vidieť, že totálne absurdný dátum je vyhodnotený za neplatný, respektíve zhoda nie je nájdená.



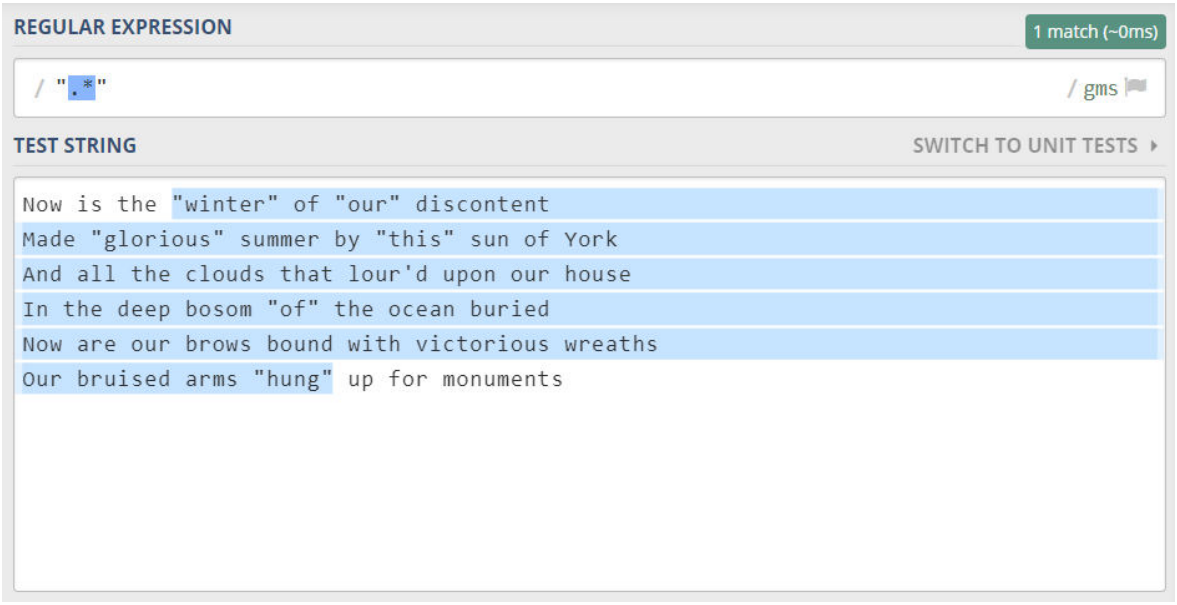
Obrázok č. 8: Regulárny výraz „(0[1-9]|1[12])[- /.](0[1-9]|1[0-9]|2[0-9]|3[01])[- /.]\d\d“

3.6 Kvantifikátory

Kvantifikátory sú v podstate špeciálne metaznaky, ktorých úlohou je znásobovať, respektíve určovať počet výskytu regulárneho výrazu alebo časti výrazu, za ktorým sa nachádzajú. Najzákladnejšími kvantifikátormi sú znaky plus, hviezdička a otáznik. Znak „+“ znamená jeden alebo viac výskytov daného výrazu, „*“ znamená nula alebo viac výskytov a „?“ slúži na nájdenie jedného alebo žiadneho výskytu.

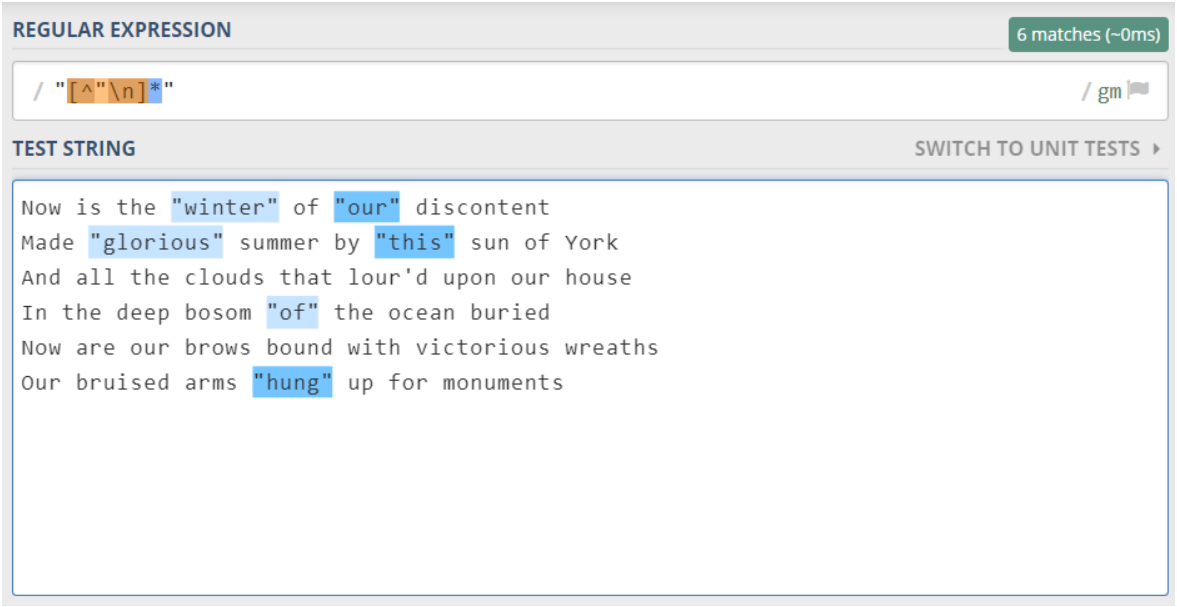
Zoberme si takú situáciu, že chceme v texte vyhľadať len reťazce v úvodzovkách. Na tomto príklade sa dá znova dobre ukázať, prečo je lepšie používať znakové triedy, respektíve negované znakové triedy, ako bodku. Prvá možnosť, čo by asi každého napadla, je vytvoriť regulárny výraz, kde v úvodzovkách bude bodka znásobená hviezdičkou, teda „“.*““. Všetko by

fungovalo správne do momentu, kedy by sme mali v texte viac výrazov v úvodzovkách. Stalo by sa totiž to, že všetky znaky od prvej do poslednej úvodzovky v každom riadku by sa zhodovali s našim regulárnym výrazom. Ak by sme mali nastavené, že „.“ nájde aj znak pre nový riadok, potom bude zhoda od prvej do poslednej úvodzovky v celom texte. Vyzerať to bude nasledovne:



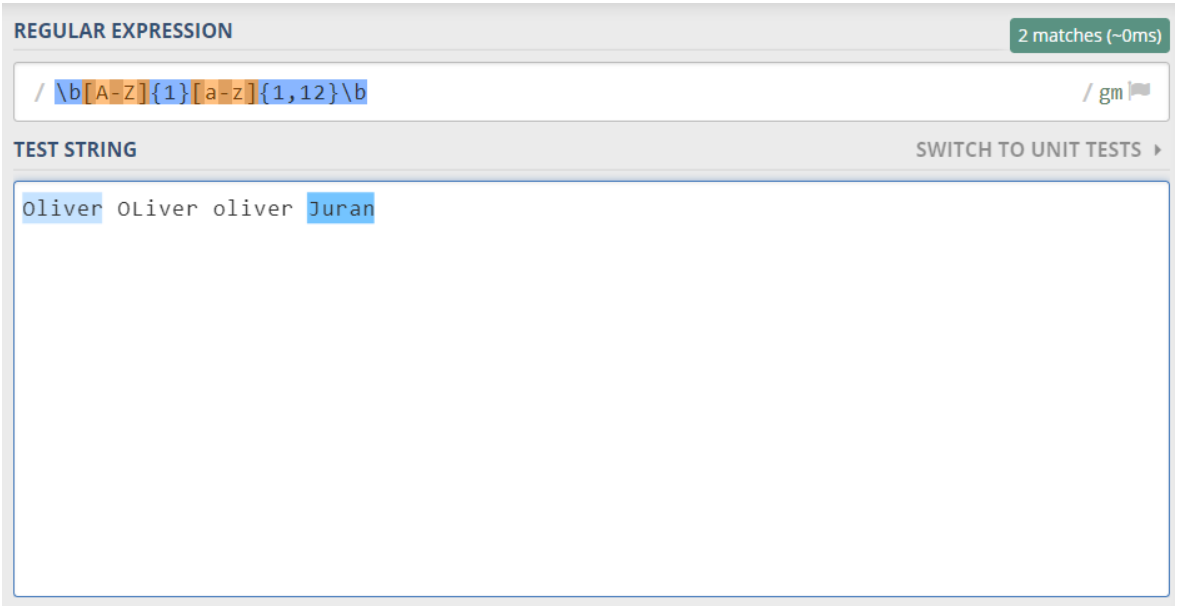
Obrázok č. 9: Regulárny výraz „“.““*

Táto situácia sa dá ošetriť negovanou znakovou triedou, kde vynecháme znak pre úvodzovku a znak pre nový riadok, a celú znakovú triedu znásobíme hviezdičkou. Všetky reťazce nachádzajúce sa v hranatých zátvorkách teda budú znamenať samostatnú zhodu. Regulárny výraz bude vyzerať nasledovne: „”[^”\n]*““. Dosiahneme teda želaný stav.



Obrázok č. 10: Regulárny výraz „ ”[^”\n]““*

Ak chcem určiť presný počet výskytu nejakého regulárneho výrazu, slúžia na to množinové zátvorky. Okrem presného počtu výskytov sa pomocou množinových zátvoriek dá určiť aj interval. Syntax je nasledovný: {minimálny počet, maximálny počet}. Ak chceme nastaviť iba spodný limit, druhé číslo v zátvorke vynecháme. Minimálny počet musí byť pri tom kladné celé číslo a maximálny počet rovnako veľké alebo väčšie kladné číslo ako minimálny počet. Pomocou množinových zátvoriek sa dajú vytvoriť ekvivalenty pre vyššie spomínané operátory „*“, „+“ a „?“. Operátor hviezdička sa dá zapísať ako „{0,}“, ekvivalent pre „+“ je výraz „{1,}“ a pre „?“ je to „{0,1}“. Tieto kvantifikátory sa dajú dobre použiť napríklad vtedy keď chceme v texte nájsť slová určitej dĺžky. V našom príklade to budú slová začínajúce veľkým písmenom nasledované minimálne jedným a maximálne dvanástimi malými písmenami. Môže ísť teda napríklad o mená, názvy miest, prvé slová vo vetách a podobne. Regulárny výraz bude vyzerať takto: „\b[A-Z]{1}[a-z]{1,12}\b”. To, aký význam má metaznak „\b”, si vysvetlíme v ďalšej časti práce.



Obrázok č. 11: Regulárny výraz „\b[A-Z]{1}[a-z]{1,12}\b”

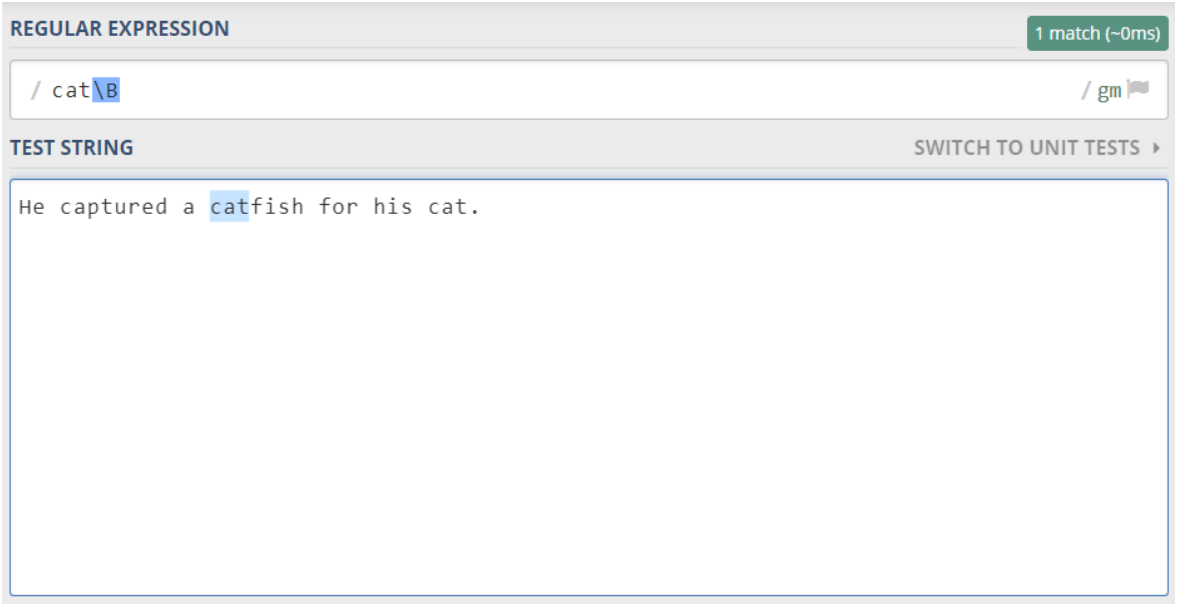
3.7 Kotvy

Kotvy sú špeciálne znaky, ktoré nám určujú, kde v texte, slove alebo riadku sa má zhoda nachádzať. Strieška, okrem toho že vo vnútri znakovej triedy znamená negáciu, má aj iný význam, a to ten, že jej prostredníctvom sa vyjadruje, že výraz sa má nachádzať iba na začiatku riadku. Naopak znak dolára – „$“, nájde zhodu iba v reťazci na konci riadku. Zaujímavé významy majú aj kotvy „\b” a jej negácia „\B”. Samotné písmeno „b“ vyplýva z anglického slova boundary, teda niečo ako ohraničenie. V tomto prípade ide o slovné ohraničenie. Tieto

kotvy sa dajú v regulárnom výraze použiť aj na začiatku, aj na konci napríklad jednoduchého znaku.

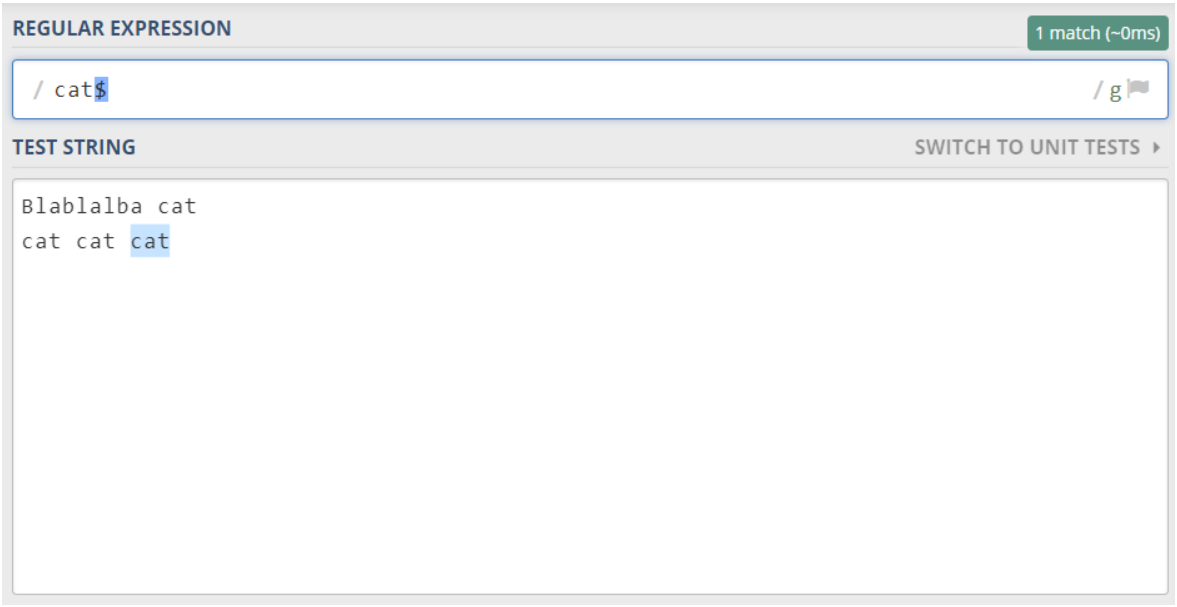
Ak použijeme kotvu „\b“ v regulárnom výraze pred jednoduchým znakom, v texte sa nájde s daným výrazom zhoda iba vtedy, ak sa znak nachádza na začiatku nejakého slova. Ak túto kotvu použijeme za jednoduchým znakom, zhoda sa nájde iba na konci slova. V prípade negovanej formy, teda výrazu „\B“, sa pri jeho použití pred jednoduchým znakom nájde zhoda tam, kde tento jednoduchý znak nie je použitý na začiatku slova. Ak sa kotva dá za jednoduchý znak, bude zhoda tam, kde sa tento znak nenachádza na konci slova.

V nižšie uvedenom príklade sa nájde zhoda s regulárnym výrazom „cat\B” iba v prvom výskyte tejto sekvencie jednoduchých znakov v texte. Je tomu tak preto, lebo v druhom výskyte je sekvencia jednoduchých znakov „cat“ použitá ako samostatné slovo, teda môžeme povedať, že sa nachádza na konci slova. Nie je za ňou žiadny iný viditeľný slovný znak. Ak by sme ale chceli nájsť slovo „cat“ iba v takom prípade, že je to samostatné slovo, môžeme tento metaznak použiť aj na začiatku aj na konci slova. Regulárny výraz by teda vyzeral takto: „\bcat\b“.



Obrázok č. 12: Regulárny výraz „cat\B”

Keď pracujeme s reťazcami dlhšími ako jeden riadok, je výhodnejšie pracovať s jednotlivými riadkami než s celým textom. Online nástroj, ktorý je v tejto práci používaný na príklady má možnosť takéto niečo nastaviť. Nazýva sa to multiline mode. Ak by sme tento mód nemali aktivovaný, znak „^“ by nám našiel zhodu iba na začiatku prvého riadku prehľadávaného textu. Podobne by to fungovalo so znakom „$“. Zhoda by sa našla iba na konci posledného riadku.

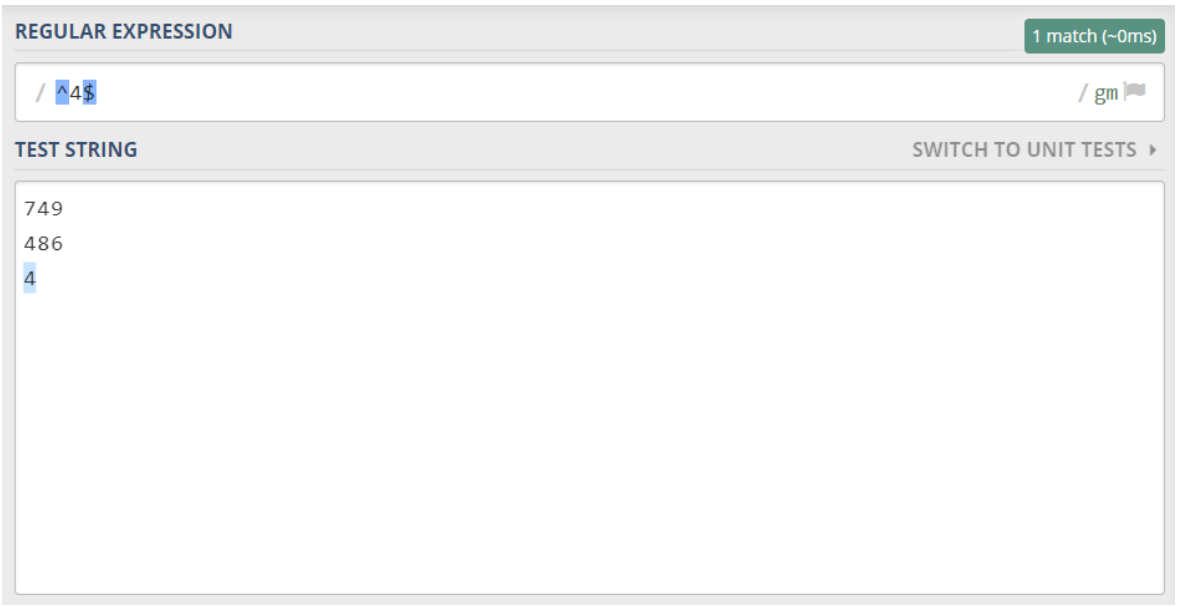


Obrázok č. 12: Regulárny výraz „$“

V pravom hornom rohu môžeme vidieť, že tam nie je vypísaný modifikátor „m“, teda multiline mode. V predchádzajúcich príkladoch bol vždy zapnutý. V tomto prípade sa nám teda vyhľadáva iba reťazec „cat“, ktorý sa nachádza na konci posledného riadku. Ak by sme mali modifikátor „m“ zapnutý, zhoda by sa našla aj na konci prvého riadku. Tieto dve kotvy as dajú použiť ako regulárny výraz aj samostatne. Moc veľký význam to ale nemá, pretože nám iba nájdu kde sú začiatky a konce riadkov.

Na príklade si teraz vysvetlíme, ako pri týchto znakoch funguje regex engine. Majme regulárny výraz „^4$“ a reťazec „749\n489\n4“, pri čom „\n“ reprezentuje nový riadok. Regex engine začína pri znaku „7“. Prvý token v regulárnom výraze je ale „^“. Nakoľko tento znak má nulovú šírku, engine sa nesnaží nájsť zhodu so znakom, ale skôr s pozíciou pred tým znakom, ku ktorému sa zatiaľ dostal. Znak „^“ teda našiel zhodu v pozícii pred znakom „7“. Engine teraz prejde na druhý token v regulárnom výraze, teda „4“. Keďže predchádzajúci symbol má nulovú šírku, engine neprechádza na ďalší znak v reťazci. Kontroluje sa teda prvý viditeľný znak – „7“. Zhoda medzi týmito znakmi sa nenašla. Ďalšie permutácie neexistujú, a tak engine znova prejde na prvý token v regulárnom výraze a začne overovať zhodu v ďalšom znaku reťazca, teda „4“. Tento krát „^“ nemôže nájsť zhodu v pozícii pred štvorkou. Je tam totiž iný jednoduchý znak, ktorý nie je znak pre nový riadok. Začne sa kontrolovať ďalší znak, znova neúspešne. Nasledujúci znak je síce „\n“, no zhoda sa taktiež nenájde, pretože na pozícii pred ním je znak „9“. Teraz sa začne kontrolovať druhá štvorka v texte. Zhoda sa nájde, pretože je pred ňou znak pre nový riadok. Rovnako sa tento znak „4“ zhoduje s druhým tokenom v našom regulárnom výraze. Teraz sa engine sa pokúša nájsť zhodu znaku „$“ s pozíciou pred znakom „8“. Zhoda sa

tu nemôže nájsť, pretože táto pozícia je nasledovaná číslom a nie znakom pre nový riadok. Engine teda musí znova pokúsiť nájsť zhodu s prvým symbolom v regexe. Predchádzajúci pokus bola úspešná zhoda v znaku „4“, takže teraz sa prejde na znak „8“. Tam sa strieška nezhoduje. Ďalšia zhoda sa potom podarí nájsť až na začiatku tretieho riadku, teda ďalšej štvorky v našom reťazci. Zhoda sa nájde aj v znaku „4“. Aktuálne sa engine snaží nájsť zhodu so znakom „$“. Kontroluje sa úplne posledný znak v reťazci, ktorým je void na konci reťazca. Znak „$“ je veľmi zvláštny. Má nulovú šírku a teda sa hľadá zhoda v pozícii pred posledným znakom. Nezáleží pri tom na fakte, že týmto „znakom“ je void. Aby sa našla zhoda, na tejto pozícii musí byť buď znak pre nový riadok, alebo void. Keďže v našom reťazci je to void, našli sme úplnú zhodu.



Obrázok č. 13: Regulárny výraz „^4$“

3.8 Nenásytné operátory

Operátory plus, bodka, hviezdička a kučeravé zátvorky sú defaultne nastavené ako „nenásytné“. Hovoria enginu, aby našiel tak veľa inštancií symbolu alebo vzoru, ktorý kvantifikujú, ako je možné. Tento jav sa nazýva nenásytnosť. Môžeme si to vysvetliť na príklade.

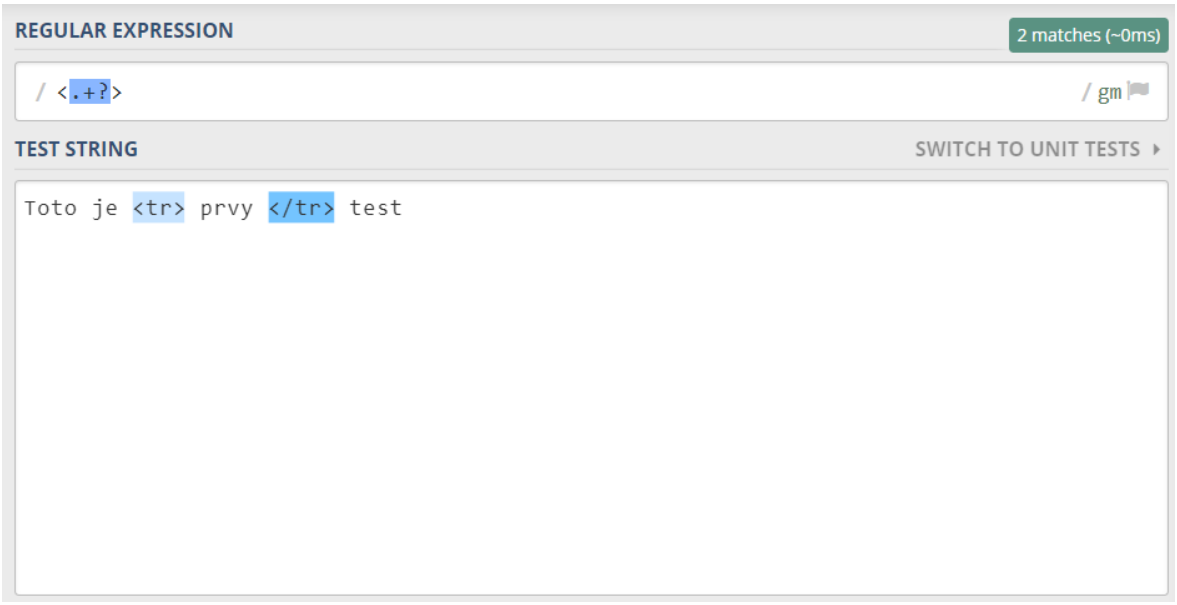
Dajme tomu, že chceme zostaviť regulárny výraz, aby našiel všetky HTML tagy. Chceme nájsť iba samotný tag, nie text, ktorý obklopuje. Vieme pri tom, že input je platný HTML dokument a regulárny výraz nemusí exkludovať žiadne neplatné výskyty lomených zátvoriek. Ak sa niečo teda nachádza medzi lomenými zátvorkami, je to HTML tag. Prvé, čo by nás mohlo napadnúť, je výraz „<.+>“. Otestujme tento výraz na reťazci „Toto je <tr> prvy </tr> test“.

Očakávame, že zhoda sa nájde v dvoch, teda „<tr>“ a „</tr>“. Nebude tomu tak. Nájde sa jedna dlhá zhoda od prvej lomenej zátvorky po poslednú. Dôvod je taký, že znak „+“ je nenásytný. To znamená, že „+“ núti regex engine zopakovať kvantifikovanú časť regulárneho výrazu toľko krát, koľko je možné. Len ak to spôsobí, že regex zlyhá, začne engine backtrackovať. Vtedy sa vráti k plusku, nechá ho vzdať sa poslednej iterácie a vykoná zbytok regexu. Teraz si podrobnejšie opíšeme, čo sa deje vo vnútri regex enginu a prečo kvôli tomuto náš regex zlyhá.

Prvý token v našom regexe je „<”. Je to jednoduchý znak a prvú zhodu nájde tam, kde je prvý rovnaký znak v našom reťazci. Ďalší token je bodka. Tá sa zhoduje s každým znakom okrem znaku pre nový riadok. Táto bodka je opakovaná znakom „+“. Následne sa nájde zhoda aj v ďalších znakoch, teda písmenách „t“ a „r“. Problém pre nás nastane vtedy, keď bodka nájde zhodu aj s ukončovacou lomenou zátvorkou a následne pokračuje v hľadaní zhody ďalej. Tu sa ukrýva príčina toho, prečo náš regex zlyhá. Bodka ďalej nájde zhodu so všetkými ostatnými znakmi v reťazci. Zlyhá až vtedy, keď engine narazí na void na konci reťazca. Až v tomto bode prejde engine na posledný token v našom regexe, ukončovaciu lomenú zátvorku. Doteraz našiel náš výraz, teda jeho časť („<.+“), zhodu s reťazcom „<tr> prvy </tr> test“ a engine prišiel na koniec reťazca. Ukončovacia lomená zátvorka tu nenašla zhodu. Engine si teraz uvedomuje, že plusko zopakovalo bodku viac krát, ako bolo požadované. Namiesto toho, aby si engine priznal chybu, začne backtrackovať. Zníži počet iterácií znaku „+“ o jeden a potom pokračuje v hľadaní zhody zvyšku regexu. Aktuálne sa teda regex „.+” zhoduje s reťazcom „tr> prvy </tr> tes“. Ďalší token v regexe je stále znak „>“, no ďalší znak v reťazci je písmeno „t“. Zhoda sa teda znova nenašla. Ako dôsledok začne engine znova backtrackovať. Zhoda je teda zredukovaná na „<tr> prvy </tr> te“. Ale znak „>“ stále nenašiel zhodu. Engine pokračuje v backtrackovaní až kým je zhoda výrazu „.+“ je zredukovaná na „tr> prvy </tr“. Až teraz nájde znak „>” zhodu s ďalším znakom v reťazci. Posledný token v reťazci bol teda nájdený. Engine teda ohlási, že sa našla zhoda s reťazcom „<tr> prvy </tr>“.

Rýchla možnosť ošetrenia tohto problému je z nenásytného pluska spraviť lenivé. Lenivé kvantifikátory sa zvyknú nazývať aj „neochotné“. Dá sa to dosiahnuť tak, že do nášho regexu sa za token „+“ pridá znak „?“. To isté sa dá urobiť aj s hviezdičkou, množinovými zátvorkami a aj s otáznikom samotným. Náš výraz po tejto zmene vyzerá takto: „<.+?>”. Pozrime sa na to, ako regex engine pracuje teraz. Ako v predošlom prípade, token „<” nájde zhodu s rovnakým znakom v reťazci. Ďalší token je bodka zopakovaná „lenivým“ pluskom. To hovorí regex enginu, aby zopakoval bodku čo najmenej krát, pričom minimum je jeden krát. Engine teda nájde zhodu v znaku „t“. Požiadavka zopakovať bodku minimálne jeden krát je teda splnená.

Engine prejde na token „>" a znak „r". To zlyhá. Engine teda aj v tomto prípade začne backtrackovať. Tento krát ale backtrackovanie donúti lenivé plusko, aby sa zopakovalo o jeden krát viac. Zhoda s výrazom „.+" sa teraz rozšírila na „tr" a engine znova pokračuje s tokenom „>". Tento krát sa úspešne našla zhoda. Posledný token v regex našiel zhodu. Engine teda ohlási, že sa našla úplná zhoda s reťazcom „<tr>". Podobne je to aj s ukončovacím tagom.



Obrázok č. 14: Regulárny výraz „<.+?>"

V tomto prípade sa ešte ako lepšia možnosť javí použiť negovanú znakovú triedu a nenásytný metaznak „+". Náš regulárny výraz potom bude vyzerať takto: „<[^>]+>". Dôvod, prečo je toto lepšia možnosť, je backtrackovanie. Keď použijeme lenivé plus, engine musí backtrackovať pre každý znak vo vnútri HTML tagu, s ktorým sa snaží nájsť zhodu. Ak použijeme negovanú znakovú triedu a reťazec obsahuje platný HTML kód, nedochádza k backtrackovaniu. Výhodné je to preto, lebo backtracking spomaľuje regex engine. Rozdiel síce nepostrehneme, ak regex len jednorazovo aplikujeme na nejaký text vo vnútri editoru, ale ušetríme množstvo cyklov procesoru, ak je takýto regex používaný opakovane vo vnútri cyklu v nejakom skripte.

Iba na pripomenutie by bolo vhodné dodať, že lenivé kvantifikátory a backtrackovanie je dostupné iba v regexom riadených enginoch.

3.9 Okrúhle zátvorky a spätná referencia

Síce sme už mali možnosť v tejto práci vidieť okrúhle zátvorky v praxi, ich činnosť sme si ešte nevysvetlili. Slúžia na zoskupovanie. Ak vložíme časť regulárneho výrazu do vnútra

okrúhlych zátvoriek, zoskupíme tým túto časť výrazu dokopy. To nám umožní aplikovať rôzne kvantifikátory na celú túto skupinu. Na tieto účely sa môžu ale používať iba okrúhle zátvorky „()“. Hranaté zátvorky nám totižto slúžia na definovanie znakovej triedy a množinové zátvorky sa používajú ako špeciálny opakovací operátor.

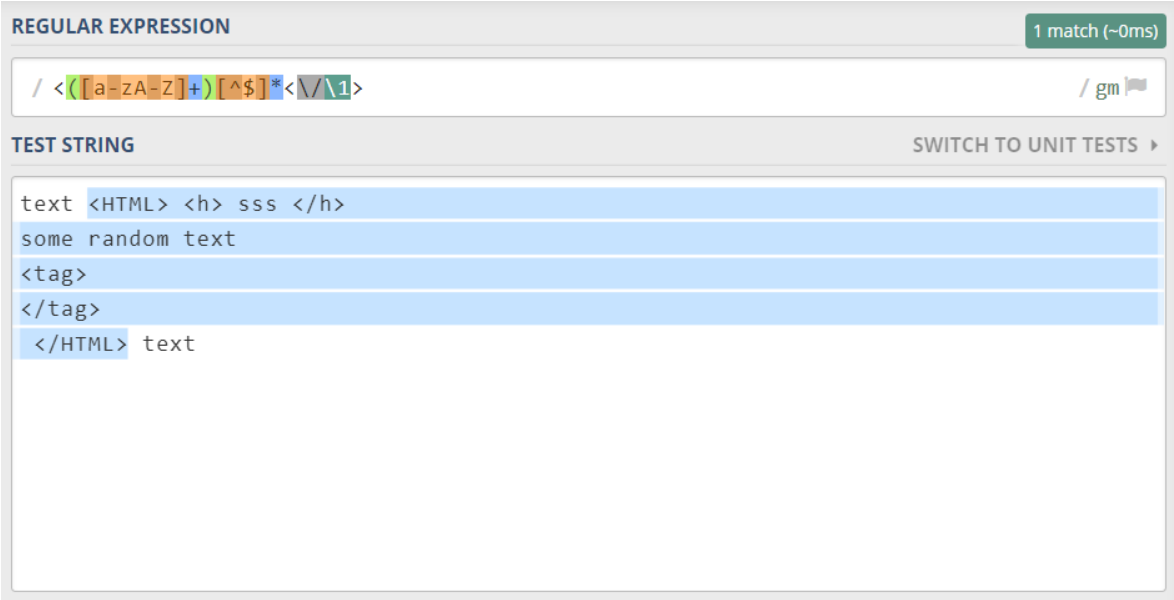
To ale nie je jediné využitie okrúhlych zátvoriek. Okrem toho, že nám zoskupujú časť regulárneho výrazu, vytvárajú nám aj spätnú referenciu. Tá ukladá časť reťazca, ktorá sa zhoduje s časťou regulárneho výrazu ukrytého v okrúhlych zátvorkách. Treba si ale uvedomiť, že používanie spätnej referencie spomaľuje regex engine. Musí totižto pracovať viac. Ak nepotrebujeme použiť spätnú referenciu, môžeme vyhľadávanie urýchliť tak, že použijeme „nezachytávajúce“ okrúhle zátvorky.

Regulárny výraz „set(Value)?“ nájde zhodu v reťazci „set“ alebo „setValue“. V prípade reťazca „set“ bude prvá spätná referencia prázdna, pretože nič nezachytila. V druhom prípade bude spätná referencia obsahovať reťazec „Value“.

Ak ale v tomto výraze spätnú referenciu nepoužijeme, lebo ju v podstate nepotrebujeme, výraz sa dá optimalizovať na „set(?:Value)?“. Sekvencia znakov „?“ a „:“ tvorí špeciálny syntax, ktorým môžeme povedať regex enginu, že táto dvojica okrúhlych zátvoriek nemá vytvárať spätnú referenciu. Je potrebné si uvedomiť, že otáznik za úvodnou okrúhlou zátvorkou nemá nič spoločné0 s otáznikom za zátvorkou uzatváracou. Zatiaľ čo otáznik na konci výrazu v zátvore robí tento výraz v texte nepovinný, prvý otáznik spolu s dvojbodkou tvoria samostatný metaznak.

Spätná referencia nám teda umožňuje znova použiť časť zhody regulárneho výrazu. Táto zhoda sa dá potom znova použiť vo vnútri regexu. Poďme si teda na príklade vysvetliť, ako sa dá spätná referencia v regulárnom výraze použiť.

Majme situáciu, že chceme v texte vyhľadať úvodný a uzatvárací HTML tag a všetok text medzi nimi. Nevieme pri tom, aký je tento úvodný tag. Zároveň máme aktívny multiline mode. Tento problém dokáže vyriešiť regulárny výraz „<([a-zA-Z]+)>[^$]*<\/\1>“ obsahujúci jeden pár okrúhlych zátvoriek, ktoré zachytia do spätnej referencie reťazec zhodujúci sa a časťou výrazu v nich. V našom prípade teda „[a-zA-Z]+“. Ide o aspoň jeden veľký alebo malý znak abecedy. Spätnú referenciu nám v tomto regexe reprezentuje zápis „\1”. Sekvencia znakov „\/“ symbolizuje ukončovací znak pred html tagom. Backslash robí zo slashu jednoduchý znak. Časť „[^$]*“ enginu hovorí, že sa majú nájsť všetky znaky vrátane znakov pre nový riadok.



Obrázok č. 15: Regulárny výraz „<([a-zA-Z]+)>[^$]<\/1>“*

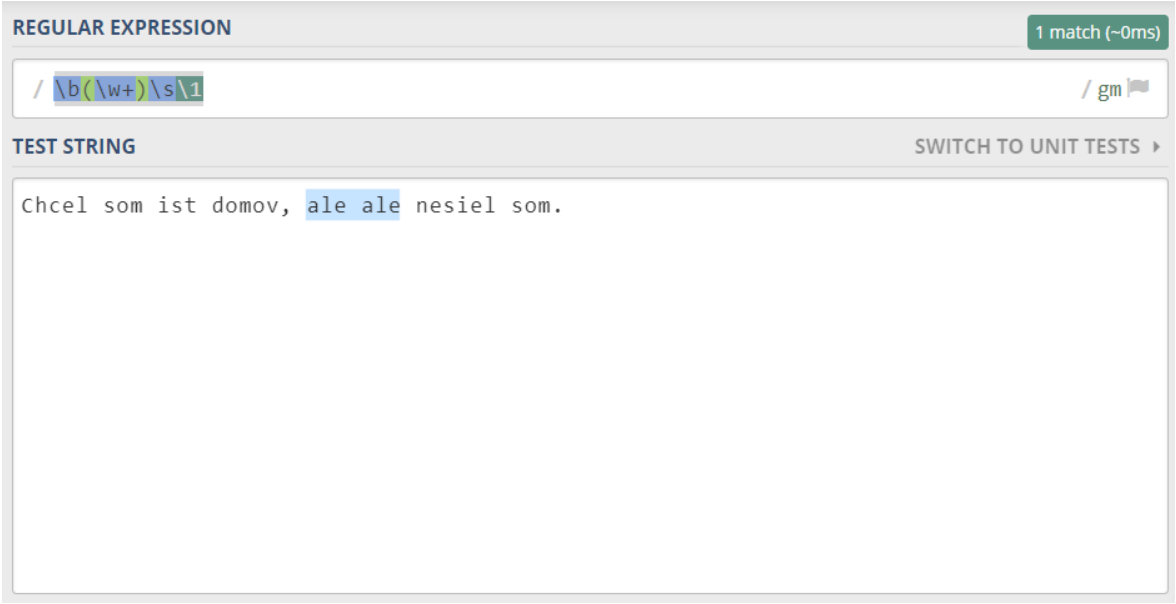
Ako môžeme vidieť na obrázku, dosiahli sme čo sme chceli. Engine našiel zhodu od prvého po posledný HTML tag v reťazci vrátane textu medzi nimi.

Spätná referencia sa môže v rámci regulárneho výrazu použiť viac ako jeden krát. Kľudne by sme teda mohli na koniec vyššie uvedeného výrazu dopísať „\1” koľkokoľvek krát. Dôležité je aj to, že v regexe sa môže vyskytovať aj viacej spätných referencií. Označujú sa potom zápismi „\2”, “\3“ atď. Nazývame ich prvá, druhá,..., n-tá referencia.

Spätná referencie sa môžu aj opakovať operátormi na to určenými. Regex engine neukladá spätné referencie permanentne. Keď bude použitá, bude sa v nej nachádzať posledná zhoda, ktorú sa podarilo nájsť. Každý jeden krát, keď sa podarí nájsť novú zhodu s výrazom v zachytávajúcich okrúhlych zátvorkách, predchádzajúca zhoda bude prepísaná.

To pre nás znamená, že existuje veľký rozdiel medzi výrazmi „([abc]+)“ a „([abc])+“. Zatiaľ čo oba tieto regulárne výrazy nájdu úplnú zhodu s reťazcom „cab“, prvý regex bude mať v spätnej referencii uložený reťazec „cab“, druhý výraz tam bude mať iba „b“. Je to tak preto, lebo v druhom výraze znak „+“ prinúti okrúhle zátvorky zopakovať sa tri krát. Prvý krát sa do spätnej referencie uložilo „c“, druhý krát „a“ a tretí krát v nej ostal uložený znak „b“. Každý jeden krát sa predchádzajúca hodnota prepísala, takže nakoniec v nej ostane „b“. Dôsledkom toho je, že kým regulárny výraz „([abc]+)=\1” sa bude zhodovať s reťazcom „cab=cab“, výraz „([abc])+=\1“ nie. Zhoda sa nenájde preto, lebo keď engine príde k tokenu „\1“, ktorý má v sebe hodnotu „b“, nenájde zhodu so znakom „c“. Preto, keď používame spätnú referenciu, mali by sme sa vždy uistiť, že v nej uchovávame to, čo naozaj chceme.

Pomocou spätnej referencie sa veľmi dobre dajú v texte odhaliť zdvojené slová. Teda napríklad keď urobíme preklep a dva krát za sebou napíšeme slovo „ale“. Na tieto účely nám perfektne poslúži regex „\b(\w+)\s\1“ .



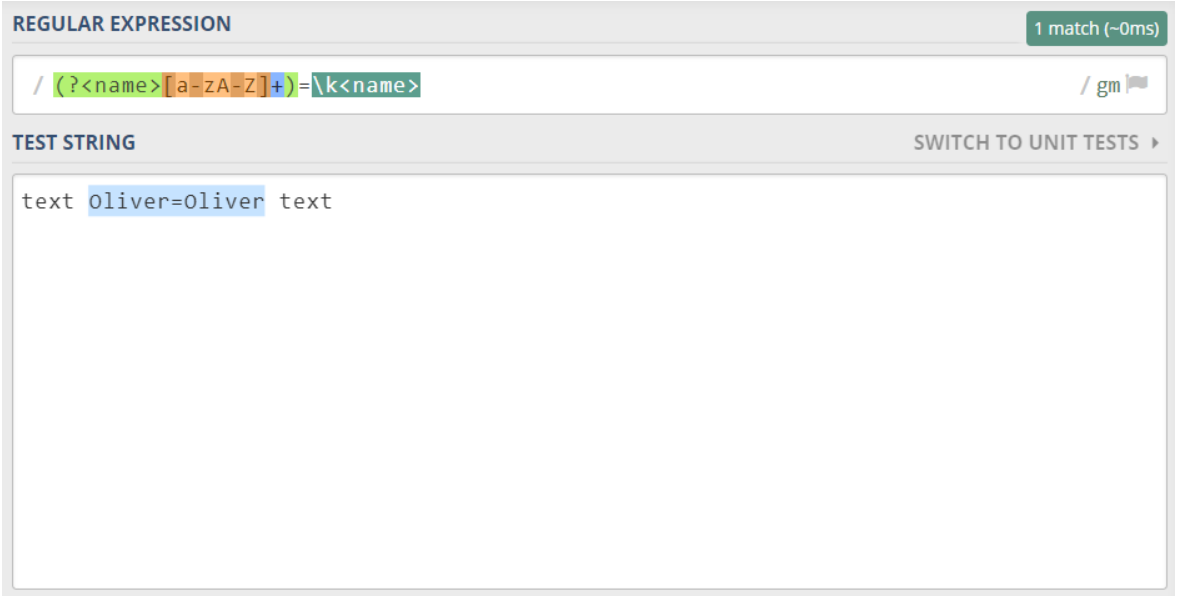
Obrázok č. 16: Regulárny výraz „\b(\w+)\s\1“

Považujeme za vhodné podotknúť, že okrúhle zátvorky a spätná referencia sa nedajú použiť vo vnútri znakovej triedy. Teda aspoň nie ako metaznaky. Keď teda okrúhle zátvorky vložíme do vnútra znakovej triedy, budú slúžiť ako jednoduchý znak. Regulárny výraz „[(a)b]” bude mať zhodu so znakmi „a“, „b“, „(“ a „)“.

V súčasnosti všetky regex enginy podporujú zachytávajúce skupiny, ktoré sú očíslované smerom zľava doprava počínajúc jednotkou. Tieto čísla môžu byť následne použité v spätnej referencii, aby znova zachytili ten istý reťazec.

V dlhších, komplexnejších regulárnych výrazom s viacerými takýmito skupinami môže byť očíslovanie mierne mätúce. Python regex module bol prvý, ktorý na tento problém navrhol riešenie. Týmto riešením je pomenovávanie zachytávajúcich skupín. V javascripte je syntax nasledovný: (?<name>group). Takýto regulárny výraz by v prehľadávanom texte našiel reťazec „group“ a uložil ho do skupiny „<name>”. Ak by sme v danom regulárnom výraze chceli tento reťazec ešte použiť, dá sa buď vykonať už popísaná spätná referencia na základe čísla, alebo sa na túto skupinu môžeme odvolať pomocou metaznaku „\k” a názvu danej skupiny. Vysvetlíme si to na príklade.

Majme regulárny výraz „(?<name>[\w]+)=\k<name>“. Tento regex sa bude zhodovať s akýmkoľvek zápisom v prehľadávanom texte, pri ktorom platí, že „n=n“, pri čom „n“ je ľubovoľný reťazec pozostávajúci z malých a veľkých písmen abecedy.



Obrázok č. 17: Regulárny výraz „(?<name>[\w]+)=\k<name>“

3.10 Lookahead a Lookbehind

Perl 5 predstavil dva veľmi silné koncepty: lookahead a lookbehind. Dokopy sa nazývajú lookaround. Zvykne sa im tiež hovoriť tvrdenia s nulovou šírkou. V tejto práci sme si už o niektorých ďalších výrazoch s nulovou šírkou hovorili. Na rozdiel od nich, lookahead a lookbehind nájdu zhodu v nejakom znaku, potom sa ale tejto zhody vzdajú a vrátia iba výsledok: či existuje alebo neexistuje zhoda. Toto je dôvod prečo sa nazývajú tvrdenia. „Nepohltia“ žiadne znaky v reťazci, ale iba nám oznámia, či je zhoda možná alebo nie. Tieto koncepty nám umožňujú vytvoriť regulárne výrazy, ktoré by bez nich nebolo možné vytvoriť, alebo by bez nich boli enormne dlhé. Lookaroundy sa dajú ešte ďalej rozčleniť na pozitívne lookheady, negatívne lookaheady, pozitívne lookbehindy a negatívne lookbehindy. Ich podrobné fungovanie si prejdeme v ďalšej časti práce.

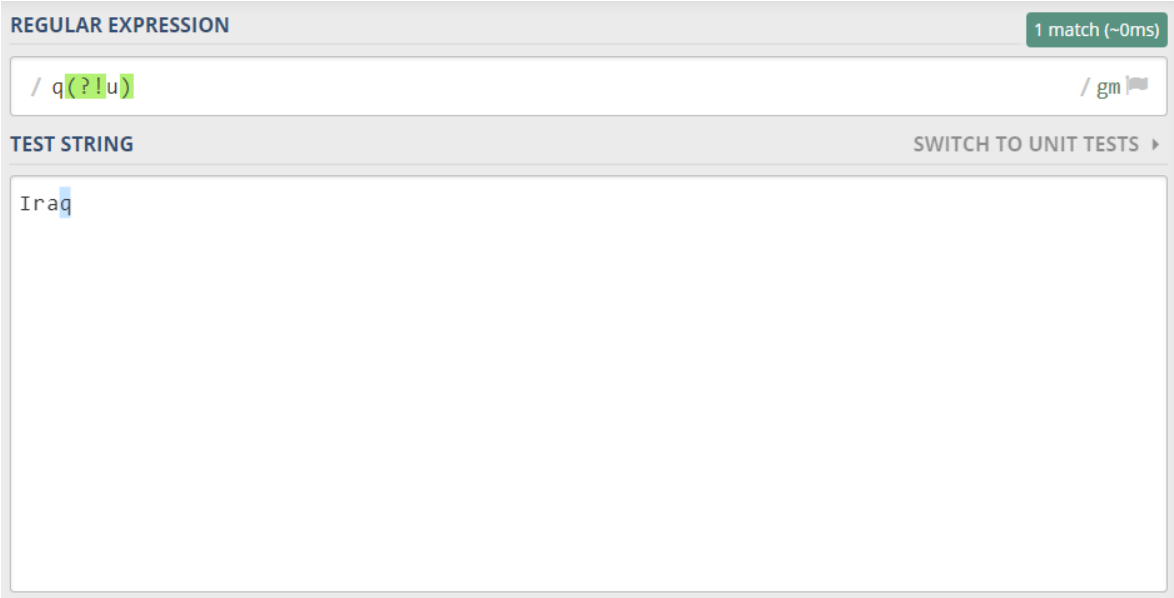
3.10.1 Negatívny a pozitívny lookahead

Negatívny lookahead je nevyhnutný, ak chceme nájsť zhodu v nejakom reťazci nenasledovanom iným reťazcom. Je to niečo, čo sa nedá docieliť ani negovanými znakovými triedami. Negatívny lookahead ponúka riešenie. Syntax je nasledovný: „(?!n)“. Ide teda o dvojicu okrúhlych zátvoriek, kde po otváracej zátvorke nasleduje sekvencia znakov „?“ a „!“.

Majme teda napríklad triviálny regulárny výraz „q(?!u)“. V texte sa teda bude hľadať znak „q“, za ktorým nenasleduje znak „u“. Ako zhoda bude nahlásený iba znak „q“.

Pozitívny lookahead funguje veľmi podobne. Rozdiel je v tom, že hľadáme reťazec nasledovaný iným reťazcom. Syntax je tiež veľmi podobný: „(?=n)“. V príklade „q(?=u)“ teda v texte hľadáme znak „q“, ktorý je nasledovaný znakom „u“. Ako zhoda bude taktiež vrátený iba znak „q“. Vo vnútri lookaheadov môžeme použiť akýkoľvek regulárny výraz. V okrúhlych zátvorkách za znakmi pre pozitívny a negatívny lookahead sa teda smie použiť akýkoľvek platný regulárny výraz. Ak napríklad obsahuje ďalšie okrúhle zátvorky, tie uložia spätnú referenciu. Lookahead samotný však spätnú referenciu nevytvára, a to aj napriek tomu, že súčasťou zápisu sú okrúhle zátvorky. Tieto zátvorky sa teda nepočítajú ani do číslovania pri používaní spätných referencií. Tu sa podľa mňa znova ukazuje výhoda pomenúvania zachytávajúcich skupín. Ak by sme teda chceli uložiť zhodu regexu vo vnútri spätnej referencie, okrúhle zátvorky musíme napísať okolo regexu vo vnútri lookaheadu. Takýto zápis pri pozitívnom lookaheade vyzerá nasledovne: „(?=(regex))“. Iný spôsob by nefungoval.

Poďme si vysvetliť, ako funguje regex engine pri aplikovaní výrazu „q(?!u)“ na reťazec „Iraq“. Prvý token v našom regexe je jednoduchý znak „q“. Zhoda sa teda, ako už vieme, nájde až pri štvrtom znaku tohto reťazca, teda znaku „q“. Ďalšia pozícia v reťazci je void na konci reťazca. Ďalší token v regexe je lookahead. Engine si tento fakt uvedomí a začne hľadať zhodu s regexom vo vnútri lookaheadu. Ďalší token je teda „u“. Ten sa nezhoduje s voidom na konci reťazcu. Zhoda s tokenom vo vnútri lookaheadu sa teda nenašla. Keďže náš lookahead je negatívny, znamená to, že lookahead úspešne našiel zhodu s aktuálnou pozíciou. Čiže v tomto momente našiel regex engine zhodu s celým regexom a vracia znak „q“ ako zhodu.



Obrázok č. 18: Regulárny výraz „q(?!u)“

Skúsme teraz ten istý regex aplikovať na reťazec „quit“. Prvý token v regexe, teda „q“, našlo zhodu v prvom znaku v prechádzanom reťazci. Ďalší token je „u“ vo vnútri lookaheadu. Ďalší znak v reťazci je tiež „u“. Tieto znaky sa zhodujú. Engine prejde na ďalší znak v reťazci. Engine už neúspešne skontroloval regex vo vnútri lookaheadu, a keďže neuspel, zhoda, ktorá sa našla na začiatku reťazca, sa zrušila. Tento fakt spôsobí, že engine sa v reťazci vráti k znaku „u“. Keďže náš lookahead je nagatívny, úspešná zhoda v jeho vnútri spôsobí zlyhanie lookaheadu. Nakoľko neexistujú ďalšie permutácie regexu, engine musí znova začať na začiatku. Znak „q“ sa ale v reťazci nenachádza nikde inde, nebude vrátená žiadna zhoda.

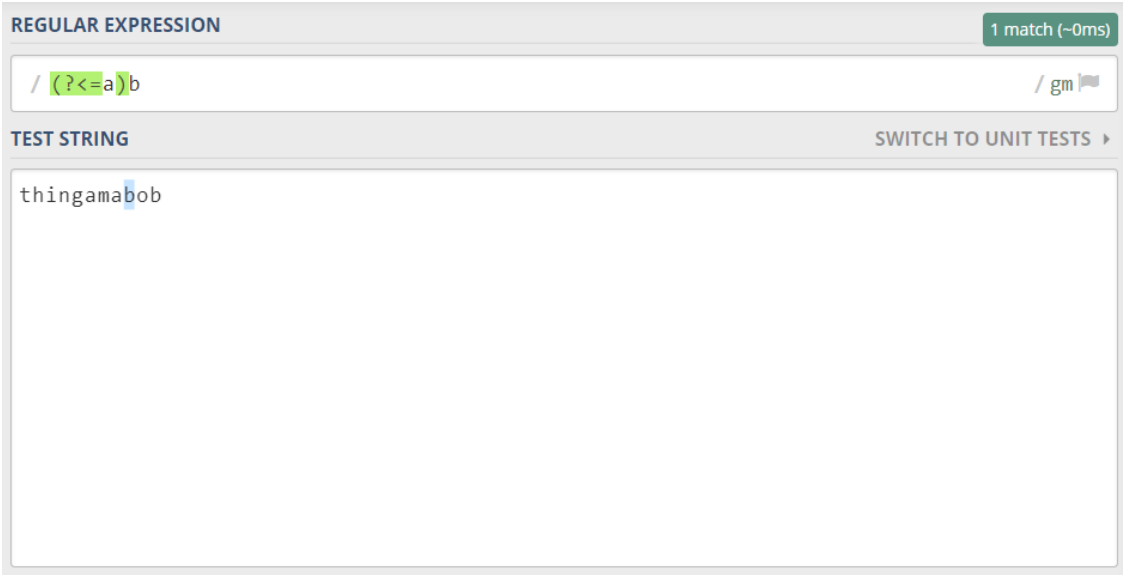
3.10.2 Negatívny a pozitívny lookbehind

Lookbehind má v podstate rovnaký efekt ako lookahead, ale pracuje obrátene. Hovorí regex enginu, aby dočasne išiel dozadu v reťazci a skontroloval, či sa tam môže nájsť zhoda s regexom vo vnútri lookbehindu. Lookbehind sa môže v regexe použiť všade, nie len na začiatku.

Negatívny lookbehind – slúži na overenie, či sa pred reťazcom nachádza alebo nenachádza iný reťazec. Syntax je nasledovný: „(?<!n)“. Majme regulárny výraz „(?<!a)b”. Tento výraz nám nájde znak „b“, pred ktorým nie je znak „a“. Rovnako ako lookahead, zhoda bude potom iba v znaku, ktorý sa zhoduje s tokenom mimo lookbehindu. Ak by sme tento regex aplikovali napríklad na reťazec „debt“, ako zhoda sa vráti iba znak „b“. Nie je pred ním totižto znak „a“.

Pozitívny lookbehind – medzi pozitívnym a negatívnym lookbehindom je taký rozdiel, že pozitívny lookbehind vráti zhodu vtedy, keď sa pred jedným reťazcom iný reťazec nachádza. Syntax je takýto: „(?<=n)". Regulárny výraz „(?<=a)b" by teda našiel zhodu napríklad v reťazci „cab". Pred písmenom „b" sa totižto nachádza „a". Ako v predchádzajúcich prípadoch, zhoda by bola len v reťazci zhodujúcim sa s regexom mimo lookbehindu, v tomto prípade s písmenom „b".

Skúsme tento regex aplikovať na reťazec „thingamabob". Engine začne s lookbehindom a prvým znakom v reťazci. V tomto prípade lookbehind povie enginu, aby sa pozrel o jeden znak v reťazci skôr, či tam je znak „a". Keďže sa engine nemôže pozrieť o jeden znak skôr, lebo je len na prvom znaku v reťazci, lookbehind zlyháva a engine sa dostane na druhý znak v reťazci, teda „h". Znova sa engine dočasne pozrie o jeden znak skôr, či je tam „a". Tento znak tam nie je, takže lookbehind znova zlyhá. Zlyháva až dovtedy, kým sa engine dostane na znak „m". Prejde o jeden znak naspäť a zistí, že sa tam nachádza „a". Pozitívny lookbehind teda našiel zhodu. Pozícia v reťazci stále ostáva na znaku „m". Ďalší token v regexe je „b", ktoré tu nenašlo zhodu. Ďalší znak je druhé „a" v reťazci. Engine znova ide o jednu pozíciu späť a zistí, že tam je „m", ktoré sa nezhoduje s „a". Ďalším znakom je „b". Engine zistí, že sa pred ním nachádza znak „a", čiže sa našla zhoda s lookbehindom a následne aj s celým regulárnym výrazom. Ak by sme nemali aktívny modifikátor „g", teda global, engine by okamžite prestal pracovať a ako zhodu by vrátil prvé „b" v reťazci. V našom prípade ale bude pokračovať ďalej, no inú zhodu už nenájde. Nájdená zhoda bude teda taká istá.



Obrázok č. 19: Regulárny výraz „(?<=a)b"

3.11 Tabuľky so znakmi v JavaScripte

Po zvládnutí lookaroundov si teraz prehľadne skonsolidujeme všetky znaky využívané v regulárnych výrazoch v javascripte.

Modifikátory – slúžia na nastavenie vyhľadávania

Tabuľka č. 1: Modifikátory

modifikátor	význam
g	vyhľadávajú sa všetky zhody, nie len prvá
i	nerozlišujú sa veľké a malé písmená
m	multiline mode
s	bodka nájde aj znak pre nový riadok

Zátvorky – slúžia na rôzne ohraničenia, zoskupenie atď.

Tabuľka č. 2: Zátvorky

výraz	popis
[abc]	nájde všetky znaky medzi zátvorkami
[^abc]	nájde všetky znaky, ktoré nie sú medzi zátvorkami
[a-z]	nájde interval medzi zátvorkami podľa ASCII tabuľky
[^0-9]	nájde všetky znaky okrem intervalu v zátvorkách
(x|y)	nájde jednu z týchto dvoch možností, vytvára spätnú referenciu
()	zoskupujú viaceré tokeny

Metaznaky – Znaky so špeciálnym významom

Tabuľka č. 3: Metaznaky

metaznak	popis
.	nájde akýkoľvek znak okrem znaku pre nový riadok
\w	nájde slovný znak
\W	nájde neslovný znak
\d	nájde číslo
\D	nájde nečíselný znak
\s	nájde neviditeľný znak

\S	nájde viditeľný znak
\b	nájde zhodu na začiatku alebo na konci slova
\B	nájde zhodu inde ako na začiatku alebo na konci slova
\0	nájde znak NUL
\n	nájde znak pre nový riadok
\f	form feed
\r	nájde znak, ktorý vráti kurzor na začiatok aktuálneho riadku
\t	nájde tab
\v	nájde vertikálnu čiaru
\xxx	nájde znak špecifikovaný oktálnym číslom xxx
/xdd	nájde znak špecifikovaný hexadecimálnym číslom dd
\udddd	nájde unicode znak špecifikovaný hexadecimálnym číslom dddd
n$	nájde n na konci reťazca
^n	nájde n na začiatku reťazca

Lookaround – Zistia, či je zhoda možná, no výraz vo vnútri lookaroundu ako zhodu nevrátia.

Tabuľka č. 4: Lookaroundy

výraz	popis
a(?=b)	nájde a, za ktorým sa nachádza b
a(?!b)	nájde a, za ktorým sa nenachádza b
(?<=b)a	nájde a, pred ktorým sa nachádza b
(?<!b)a	nájde a, pred ktorým sa nenachádza b

Kvantifikátory – Slúžia na zopakovanie tokenu, časti, alebo celého regexu.

Tabuľka č. 5: Kvantifikátory

kvantifikátor	popis
n+	nájde reťazec, ktorý pozostáva aspoň z jedného n
n*	nájde reťazec, ktorý pozostáva zo žiadneho alebo viac n
n?	nájde reťazec, ktorý pozostáva zo žiadneho alebo jedného n
n{X}	nájde reťazec, ktorý pozostáva zo sekvencie X n-iek
n{X,Y}	nájde reťazec, ktorý pozostáva zo sekvencie minimálne X,

	maximálne Y n-iek
n{X,}	nájde reťazec, ktorý pozostáva zo sekvencie minimálne X n-iek

Záver

V tejto práci sme si najprv v teoretickej rovine vysvetlili, čo tu regulárne výrazy sú. Objasnili sme si aj ďalší často využívaný pojem regex engine. Zároveň sme si aj stručne popísali históriu, vznik a vývoj regulárnych výrazov. Okrem toho sme si v tejto kapitole aj definovali základné členenie znakov používaných pri tvorbe regulárnych výrazov a rozčlenili sme si aj regex engine na dva hlavné druhy.

Následne sme si po teoretickej stránky vysvetlili jednotlivé znaky používané pri tvorbe regulárnych výrazov v jazyku JavaScript. Mnohé z nich sme si potom vysvetlili na príkladoch a aplikovali na reťazce. Pre úplné vysvetlenie a pochopenie danej tématiky sme si na niektorých vybraných príkladoch taktiež opísali akým spôsobom funguje regex engine. Z počiatku sme si po praktickej stránke prechádzali a vysvetľovali hlavne jednoduché regulárne výrazy. Neskôr sme sa dostali aj ku náročnejším, komplexnejším regulárnym výrazom. To sme robili hlavne preto, lebo rôzne jednoduché regulárne výrazy v praxi nie sú príliš aplikovateľné. Tieto príklady boli použité najmä kvôli pochopeniu základov a pre účely ich ďalšieho používania v zložitejších výrazoch.

Regulárne výrazy sú využiteľné v rôznych sférach v rámci IT oblasti. Vysvetľovali sme si teda rôzne príklady, ktoré by mohli byť v skutočnom podnikateľskom prostredí použité. Boli nimi napríklad verifikovanie vstupu od užívateľa. Týmto vstupom mal byť dátum narodenia v požadovanom formáte. Na tomto príklade sme si aj vysvetlili, prečo musí byť v prípade overenia inputu od užívateľa regulárny výraz čo najdokonalejší.

Ďalšími použiteľnými regulárnymi výrazmi boli príklady zaoberajúce sa HTML elementami. Ukázali sme si regexy, ktorými je možné v texte zachytiť rôzne elementy, text mezdi nimi alebo aj všetok text medzi úvodným a ukončovacím elementom bez toho, aby sme vedeli, aký tento element je.

Taktiež sme si popísali regulárny výraz, ktorým je možné vďaka spätnej referencii nájsť zdvojené slová. Podobný spôsob sa používa aj v rôznych textových editorov, vrátane editora Microsoft Word.

Na úplný záver sme si zostavili tabuľky, v ktorých sú v krátkosti popísané všetky elementy využívané v pri tvorbe regulárnych výrazov v jazyku JavaScript.

Zoznam použitej literatúry

[1] – FOGG, Andrew. Neural Nets: How Regular Expressions brought about Deep Learning. [online] 20.1.2016. Dostupné na internete: <https://www.import.io/post/neural-nets-how-regular-expressions-brought-about-deep-learning/>

[2] – TIKI TOKI: The Bloody History of Regex. [online] Dostupné na internete: <https://www.tiki-toki.com/timeline/entry/264740/The-Bloody-History-of-Regex/>

[3] – MOORE, Karleigh – CHUMBLEY, Alex – HO, Nein-Tai – KHIM, Jimin. Regular Languages [online] Dostupné na internete: <https://brilliant.org/wiki/regular-languages/>

[4] – THOMPSON, Ken. Programming Techniques: Regular Expressions Search Algorithm. [online] [cit. 2019-7-29] Jún 1968. Dostupné na internete: <https://www.import.io/wp-content/uploads/2016/01/p419-thompson.pdf>

[5] – TECHTARGET: POSIX (Portable Operating System Interface). [online] Dostupné na internete: <https://whatis.techtarget.com/definition/POSIX-Portable-Operating-System-Interface>

[6] – H, Zak. Posix Standard. [online] Dostupné na internete: <https://linuxhint.com/posix-standard/>

[7] – HOUSTON, Gary. Regex – Henry Spencer's regular expression libraries. [online] Dostupné na internete: <https://garyhouston.github.io/regex/>

[8] – SOFTPANORAMA: Perl Regex History. [online] Dostupné na internete: <http://www.softpanorama.org/Scripting/Perlorama/Regex/history.shtml>

[9] – BOUGIE, Patrick. PCRE – Perl Compatible Regular Expressions. [online] Dostupné na internete: <https://mac-dev-env.patrickbougie.com/pcre/>

[10] – GOYVAERTS, Jan. Regular Expressions: The Compelet Tutorial. [online] Júl 2007. Dostupné na internete: <https://www.princeton.edu/~mlovett/reference/Regular-Expressions.pdf>

[11] – REGULAR-EXPRESSIONS: Regular Expressions Turotial. [online] Dostupné na internete: < https://www.regular-expressions.info/tutorial.html>

[12] – IFMO: Mastering RegExp. [online] Dostupné na internete: <https://se.ifmo.ru/~ad/Documentation/Mastering_RegExp/mastregex2-CHP-4-SECT-3.html>