Regulární výrazy

Regulární výrazy, v angličtině regular expressions, zkráceně regex, slouží k řadě věcí v informačních technologií. Nejhlavněji však vyhledávání textu, validaci, nahrazování a obecně dalšími akcemi nad textovými řetězci. Regulární výrazy staví nad gramatikami a automaty z teoretické informatiky.

Regulární výrazy vznikli z důvodu, že vyhledávání v textu je složité. Jeden z důvodů je takový, že vyhledávání musí projít veškerý text a určování pravidel je velmi programově nepřehledné. Zkuste si schválně sami napsat nějaký program, který bude moci validovat telefonní čísla z české republiky. Program musí přijímat minimálně následující tvary:

```
+xxxXXXxxxXXX
XXXxxxXXX
xxxXXXxxxXXX
(+xxx) XXXxxxXXX
+xxx XXX xxx XXX
+xxx XXXxxxXXX
...
```

Syntaxe regulární výrazů je jednoduchá - jedná se o sekvenci (textový řetězec) znaků, které mohou mít speciální význam. Jedná se o klíčové znaky a sekvence znaků. Společně s tímto řetězcem se určují modifikátory (angl. flags), které určují speciální chování tohoto regulárního výrazu, například můžeme určit:

- `i` - ignoruje velikost písmen
- `m` - víceřádkové
- `g` - zpracovává regulární výraz vícekrát v textu

V regexu se často setkáme s literály, to je označení pro znaky, které nemají význam.

Regulární výrazy nejsou moc standardizované a různé jazyky je implementují různými způsoby a může se stát, že v různých jazycích věci fungují jinak či vůbec třeba neexistují. Je důležité vždy číst dokumentaci daného jazyka. V tomto materiálu počítáme pouze s implementací z JS/Node.js.

Použití

V JS máme k dispozici řadu metod ze standardní knihovny, které nám dovolují s regexy pracovat, jedná se o:

- `.replace`
- `.split`

- `.match` - vrací první / všechny výsledky dle /g modifikátoru
- `.exec` - stejné jako .match ale upravuje regulární výraz (spojuje vstupy)
- `.search` - stejné jako .match, ale hledá méně informací (je rychlejší)
- `.test` - vrací boolean hodnotu dle toho, zda daný regulární výraz odpovídá vstupu (volá se na reg. výrazu)
- a další

V JS můžeme regulární výrazy vytvářen dvěma způsoby. Jeden z nich je pomocí třídy RegExp a nebo přímo v kódu pomocí lomítek:

```
const regexAsObject = new RegExp('foo');
const regex = /foo/;

const input = "aaa foo bbb foo ccc";

console.log(input.split(regexAsObject));
console.log(input.split(regex));

console.log(input.replace(regexAsObject, '-'));
console.log(input.replace(/foo/g, '-'));
```

Modifikátory určujeme jako druhý argument a nebo za posledním lomítkem:

```
const regexAsObject = new RegExp('foo', "gi");
const regex = /foo/gm;

const input = "aaa foo bbb foo ccc";

console.log(input.split(regexAsObject));
console.log(input.split(regex));

console.log(input.replace(regexAsObject, '-'));
console.log(input.replace(/foo/g, '-'));
```

S globálními regulárními výrazy s jsou obecně problémy a doporučuje se pro každou kontrolu vytvářet vlastní regulární výraz. Voláním metod na regulárním výrazu se může vnitřně výsledek spojovat a obecně s tím dělat různé věci (a vracet neočekávané výsledky).

Kvantifikátory

Kvantifikátory v regexech určují, kolikrát se daný literál či skupiny (více později) může opakovat, v základu máme k dispozici:

- `*` - nepovinné, počet neomezen (0 až n)
- `+` - povinné, počet neomezen (1 až n)
- `?` - nepovinné, maximálně jednou (0 nebo 1)

Důležité je si uvědomit, že když kvantifikátor nepoužijeme, říkáme to, že chceme použít daný literál či skupinou právě jednou a to povinně.

V regulární výrazech existují i kvantifikátory pokročilé, kde můžeme definovat přesné počty. Zapisují se do složených závorek a mají právě dva argumenty, nižší a vyšší omezení. Když za literál či skupinu zapíšeme např. `{1,8}`, říkáme tím, že se může opakovat právě jednou až osmkrát. Když druhý parametr neuvedeme:

- ale zůstane `,`, znamená to, že je počet neomezený (např. `/{3,}/`)
- ale nezůstane `,`, znamená to, že počet musí být přesný (např. `/{3}/`)

Nyní jsme si definovali první znaky s významem - co kdybychom dané znaky chtěli přímo vybrat pomocí regulárního výrazu? Jednoduché řešení je zrušení významu daného znaku. To děláme pomocí zpětného lomítka, který výraz následujícího znaku zruší. Například, kdybychom chtěli vybrat hvězdičku v textu, použijeme: `/\*{3}/`.

Skupiny

Jednotlivé literály můžeme v regulárních výrazech spojovat do skupin. Těmto skupinám poté můžeme určovat další pravidla, zejména kvantifikátory. Skupina se chová jako jeden literál a určením kvantifikátoru říkáme, že všechny literály uvnitř skupiny se musí opakovat. Například pro reg. výraz `/(ab c)+/g`, říkáme, že se to, co testujeme, musí skládat alespoň z jedné skupiny (tj. `ab c`) a nebo se musí opakovat (tj. `ab cab c`, atd.).

Uvnitř skupin můžeme začít používat operátor nebo, který slouží k tomu, že se jedna z variant uvnitř skupiny použije pro vybrání. Jednotlivé možnosti rozdělujeme pomocí `|`. Např. `/(ahoj|zdar)/` vybere buď `ahoj` a nebo `zdar`. Možností v této skupině může být neomezeně. Uvnitř skupin můžeme používat další skupiny a další věci.

Jednotlivé skupiny jsou však i výsledkem volání .match a dalších. To často nechceme (např. opravdu chceme jen opakovat), a proto můžeme použít skupinu, která nevybereme daný text do výsledku těchto metod. Takový skupina se zapisuje pomocí ?: na začátku, například tedy: `/(?:ab)*/` vybere jakoukoliv sekvenci tvořenou z ab, ale nevybere ji do výsledku.

Se skupinami souvisí i podmínky. V JS jsou podmínky velmi omezené oproti ostatními implementacím, ale přesto existují. Máme několik variant:

- `/test(?= to je)/` — vybere `test`, pokud se za ním nachází `to je`
- `/(?<=test)není/` — vybere `není`, pokud se před ním nachází `test`

Rozmezí znaků

Na stránce skupin jsme si definovali operaci nebo. Co kdybychom chtěli ale vybrat hned z několika znaků a nechce se nám jednotlivé znaky vypisovat? K tomu slouží rozmezí znaků, kterými můžeme definovat různé rozmezí, které se automaticky v režimu NEBO aplikuje. To však neznamená, že jsou skupiny k ničemu. Obě dvě sekvence mají své omezení.

Můžeme např. definovat `/[a-z]/`, což nám určuje veškeré malé znak anglické abecedy. Můžeme určovat i komplikovanější věci:

- např. `/[abcde]/g` — může obsahovat a, b, c, d a e
- např. `/[a-zA-Z0-9]/g` — může obsahovat jakékoliv písmeno od a do z, od A do Z a číslice od 0 do 9

Rozmezí můžeme i negovat. Tedy, přijmeme všechny znaky, které nesplňují, že jsou uvedeny. Například `/[^ab]/` přijme vše, až na hodnoty `a` a `b`.

Je dobré i zmínit rozmezí, které jsou definované v různých implementacích regulárních výrazů. Například v PHP je k dispozici:

- `[[::ascii::]]` - znaky z tabulky ascii
- `[[::lower::]]` - malé znaky

JS však žádné takové rozmezí nemá.

S rozmezím souvisí i zástupné znaky (tokeny, meta sekvence), které mají speciální význam a vybírají různé skupiny znaků. Jedná se o:

- `.` — vybere jeden jakýkoliv znak (mimo nové řádky a podobné)
- `\s` — jakýkoliv *whitespace* znak
- `\S` — opak `\s`
- `\d` — jeden znak
- `\D` — opak `\D`
- `\w` — znak, který se může objevit ve slově

Omezení a chování

Chování funkcí pro regulární výrazů bez modifikátoru /g je takové, že daný regulární výraz se může nacházet kdekoliv uvnitř textu. To často nechceme a chceme zkusit, že celý vstup odpovídá regulárnímu výrazu. K tomu používáme klíčové sekvence:

- `^` — označuje začátek stringu
- `$` — označuje konec stringu

Můžeme je používat i dohromady, například `/^ahoj$/` znamená, že na vstupu musí být přesně ahoj, aby bylo vybráno.

Tyto znaky se chovají zajímavě i s modifikátorem /m, který dovoluje více řádků. Po zapnutí tohoto modifikátoru jsou znaky omezeny na řádek, `^` tedy znamená začátek řádku a `$` jeho konec.

Velmi používané v regulárních výrazech, jsou taktéž tzv. word boundaries, které místo znaku označují pozice znaků okolo slov (resp. okolo whitespaces). Máme k dispozici:

- `\b` — místo, kde končí či začíná slovo
- `\B` — místa, kde `\b` nevybere

Například `/\bkočička\b/g` označí kočička jako samostatné slovo.

Poslední věcí, kterou se budeme zabírat, je chování kvantifikátorů. Některé tokeny jsou totiž tzv. greedy, tedy lakomé, a chtějí vybrat co nejvíce znaků, co je možné. To se nám často nehodí, protože chceme uvnitř samotného hledání hledat další podčásti. Kvantifikátorům můžeme určit, aby byly lazy (líní) a tím jim říct, aby sebrali co nejméně, aby regulární výraz platil. Označíme ho tak, že za jeho kvantifikátor dáme otazník (funguje tedy na + a *). Například regex `/a(a)+?(a)*/` způsobí, že první skupina označí právě jedno `a`, protože zbytek může vybrat už druhá skupina.

Problém líných tokenů je takový, že hledání zpětných řešení je docela časově nákladné a tím se může výpočet regulárního výrazu zhoršit.

Další zdroje

V této lekci (a ani ve škole) jsme neprobrali další zajímavé části regulárních výrazů:

- pojmenované skupiny
- negativní podmínky
- další modifikátory - u a y
- způsoby nahrazení
- další meta sekvence

Regulární výrazy se obyčejně tvoří pomocí chytrých editorů, doporučuji je psát v nástrojích (nezapomeňte zvolit správný jazyk):

- https://regex101.com
- https://regexr.com

Procvičení

Níže máte seznam regulárních výrazů, které byste měli umět popsat, resp. vysvětlit, co označí či nahradí:

```
/foo/
/a*/
/a*/g
/bb?a*/
/b?a*/g
```

```
/(ab){3,}/g
/bb{2}/g
/baba{1,3}/g
/(lemur)*/g
/(mada(ga)?scar)/g
/(b)*o*l*e+s*í*/g
/a|b/g
/(ahoj|se)/g
/(a|j|k)/g
/((j )|(e )|(k ))/g
/[abcde]/g
/[a-z]/
/[a-z]/g
/[a-z]/gi
/[a-zA-Z0-9]/g
/[a-z]+/g
/[1-4]{2}/g
/[,\- !]+/g
/[,\- !]/g
/(+420)? ?([0-9] ?){3}/gi
/.*/g
/\d+/g
/\w+/g
/\W/g
/[^xyz]/
/[^5-8]/
/(?:ab)+/
/(?<=test)není/
/test(?= to je)/
/^ad$/
/^ad$/g
/\bkočička\b/g
/(?:\b\w+\b)+/g
/(?:\B\w+\B)+/g
/<[^<>]+>/gm
/a(a)+?(a)*/
```