

Vysoká škola ekonomická v Praze  
Fakulta informatiky a statistiky  
Katedra informačních technologií

Studijní program: Aplikovaná informatika  
Obor: Informatika

Název bakalářské práce

BAKALÁŘSKÁ PRÁCE

Student : Jméno a příjmení studenta  
Vedoucí : Jméno a příjmení s tituly  
Oponent : Jméno a příjmení s tituly

2016

## Prohlášení

Prohlašuji, že jsem bakalářskou/diplomovou práci zpracoval(a) samostatně a že jsem uvedl(a) všechny použité prameny a literaturu, ze které jsem čerpal(a).

V Praze dne den. měsíc 2016

.....  
Jméno a příjmení studenta

## Poděkování

Teoreticky tato pasáž ve vaši práci být nemusí a můžete ji smazat. Pokud byste však chtěli někomu poděkovat, zde je to správné místo. V poděkování můžete uvést vedoucího práce, případně konzultanta, nesmíte zde však uvést oponenta. Mezi ty, kterým děkujete, můžete samozřejmě zařadit i své blízké, i když to nebývá zvykem.

### **Abstrakt**

Obsahuje zaměření a hlavní cíl práce, způsob dosažení cíle, přínos práce (vlastní příspěvek k řešenému tématu) a stručně popsanou strukturu práce.

### **Klíčová slova**

Seznam nejvýznamnějších odborných výrazů charakterizujících téma závěrečné práce.

## **Abstract**

Obsahuje zaměření a hlavní cíl práce, způsob dosažení cíle, přínos práce (vlastní příspěvek k řešenému tématu) a stručně popsanou strukturu práce.

## **Key words**

Seznam nejvýznamnějších odborných výrazů charakterizujících téma závěrečné práce.

# Obsah

1	Úvod	1
1.1	vymezení tématu práce a důvod výběru tématu . . . . .	1
1.2	cíle práce . . . . .	1
1.3	způsob/metoda dosažení cíle . . . . .	1
1.4	předpoklady a omezení práce, struktura práce . . . . .	1
1.5	výstupy práce (příp. komu jsou určeny) a očekávané přínosy . . . . .	1
2	Historie	2
3	Syntaxe regulárních výrazů	3
3.1	Definice . . . . .	3
3.2	Literály a jejich řetězení . . . . .	3
3.3	Metaznaky . . . . .	4
3.3.1	Třída znaků . . . . .	5
3.3.2	Kvantifikátory . . . . .	6
3.3.3	Hladové vs. líné kvantifikátory . . . . .	7
3.4	Optimalizace regulárních výrazů . . . . .	7
3.4.1	Posesivní kvantifikátory . . . . .	7
3.4.2	Atomické seskupování . . . . .	8
3.5	UNICODE vlastnosti . . . . .	8
3.6	Look Around . . . . .	10
3.7	Zachycování (capture) . . . . .	10
4	Jazyky s vestavěnou podporou regulárních výrazů	12
4.1	Perl . . . . .	12
4.2	AWK . . . . .	12
4.2.1	Obecné informace . . . . .	12
4.2.2	Syntaxe . . . . .	12
4.3	Javascript . . . . .	13
4.3.1	Obecné informace . . . . .	13
4.3.2	Syntaxe . . . . .	13
5	Jazyky s knihovnami	16
5.1	C# . . . . .	16
5.2	Python . . . . .	16
5.3	Java . . . . .	16
5.3.1	Obecné informace . . . . .	16
5.3.2	Používané třídy a metody . . . . .	16

# 1 Úvod

1.1 vymezení tématu práce a důvod výběru tématu

1.2 cíle práce

1.3 způsob/metoda dosažení cíle

1.4 předpoklady a omezení práce, struktura práce

1.5 výstupy práce (příp. komu jsou určeny) a očekávané přínosy

## 2 Historie

V roce 1956 americký matematik a logik Stephen Cole Kleene popsal konečné deterministické automaty (DFA), které rozpoznávají formální jazyk. Jazyk rozpoznatelný konečným automatem se poté nazývá regulárním jazykem, protože se vyznačuje „pravidelností“ (regularitou) [1]

Na Kleeneho v roce 1968 navázal programátor, známý především díky operačnímu systému Unix, Kenneth Thomson. Publikoval článek „Regular Expression Search Algorithm“ [2], ve kterém popsal algoritmus pro vyhledávání pomocí regulárních výrazů v textu.

Thomson také implementoval Kleeneho notaci do textového editoru QED. Tuto funkci poté zabudoval i do unixového editoru ed, co později vedlo ke vzniku vyhledávacího počítačového programu grep. Později byly regulární výrazy zabudovány do jazyka AWT či do příkazu `expr` používaném v příkazové řádce v unixových systémech.

Komplikovanější regulární výrazy se v 80. letech minulého století začaly objevovat v jazyce Perl. Vycházely přitom z regex knihovny napsané kanadským programátorem Henry Spencerem. Začaly se zde objevovat i regulární výrazy, na které nebylo možné použít konečný automat (jsou rozpoznávány nedeterministickými automaty NFA). Spencer později regulární výrazy implementoval i do skriptovacího jazyka Tcl, známé pod názvem Advance Regular Expressions (ARE).

V roce 1992 byly regulární výrazy v původní podobě standardizovány v POSIX.2 standardu. Nicméně počítačový programátor Philip Hazel v roce 1997 vyvinul PCRE (Perl Compatible Regular Expressions), jehož syntaxe je více flexibilní než v POSIXu. Je využíván například v jazyce PHP nebo Apache HTTP Server. [3] I když úplný název PCRE zní „Perl Compatible Regular Expressions“, PCRE a Perl mají své odlišnosti [4], které v této práci nebudou podrobněji popsány.

Dnes jsou regulární výrazy podporovány v mnoha jazycích a textových editorech jako například Java, Python, JavaScript atd.



## 3 Syntaxe regulárních výrazů

### 3.1 Definice

Regulární výrazy nemají přesnou definici, nicméně dají se popsat jako textové řetězce, které slouží jako vzor pro vyhledávání ve strukturovaném nebo nestrukturovaném textu. Dalším případem užití může být nahrazování či rozdělování textu.

Jak již bylo zmíněno v předchozí kapitole, jsou využívány v mnoha aplikacích a programovacích jazycích a jejich knihovnách. [5, s.1–3] Jejich implementace není ale ve všech jazycích stejná, a proto budu v následujícím textu uvádět odlišnosti syntaxe regulárních výrazů ve vybraných jazycích (tzv. „příchuť“ regulárních výrazů).

Existují různé desktopové i webové aplikace, které umožňují rozpoznávání regulárních výrazů v různých jazycích, čímž umožňují ověření správnosti zadaného výrazu. Mezi webové aplikace patří například:

- <http://regex101.com/>
- <http://regexr.com/>

Z desktopových aplikací bych uvedla například „The Regex Couch“, který lze stáhnout zde: <http://weitz.de/regex-coach/>.

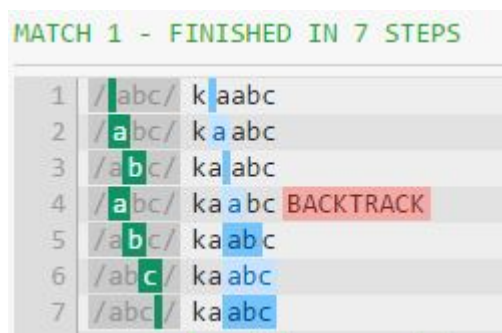
### 3.2 Literály a jejich řetězení

Řetězení umožňuje spojit jednotlivé znaky do skupin. Například regulárnímu výrazu: **abc** vyhoví jakýkoli řetězec, který obsahuje danou skupinu na libovolném místě. Regulární výraz tedy vyhoví například řetězcům: „**ijkl****abc****j**dab**d**“ nebo „**ab**cccc**d**“, ale nenajde shodu v řetězci: „**a****j****b****c**“.

Jak toto vyhledávání funguje vystvětlím na následujícím příkladu. Mějme již výše uvedený regulární výraz **abc** a řetězec: „**ka****abc**“. Při vyhledávání se začíná zleva a postuje se po jednotlivých znacích. Nejprve se vezme znak „**k**“, na kterém regulární výraz selže, protože první znak výrazu je „**a**“. Proto řetězec zkrátí o první znak a pokračuje dalším znakem, což je znak „**a**“. Zde regulární výraz uspěje. Poté ovšem selže na třetím znaku řetězce „**a**“, jelikož dalším znakem ve výrazu je znak „**b**“. Proveďte se tedy tzv. „backtrack“, což znamená, že regulární výraz nepřejde na čtvrtý znak, ale začne porovnávat regulární výraz od začátku s třetím znakem řetězce „**a**“. Zde najde shodu jelikož prvním znakem výrazu je „**a**“. Poté pokračuje v porovnávání čtvrtého znaku řetězce „**b**“ se znakem „**b**“ regulárního výrazu, zde opět najde shodu. Podobně najde shodu i při znaku „**c**“. [6]

Názorně tento proces lze vidět na obrázku 1.

Při použití regulárního výrazu v tomto tvaru musíme brát v úvahu, že lze používat pouze ASCII znaky a regulární výraz najde pouze první shodu v řetězci. Dále regulární výrazy jsou „case-sensitive“, což znamená, že výše uvedený regulární výraz nerozezná například řetězec „**Abc**“. [5, s.26]



Obrázek 1: Backtrack

Jaké části regulárních výrazů, popř. tříd je potřeba použít v daném jazyce, lze vidět v tabulce 1.

Tabulka 1: Case insensitive

Case insensitive	
Java	Pattern.CASE_INSENSITIVE nebo (?i)
Perl	\i
AWK	IGNORECASE = 0 (zajistí zapnutí módu globálně v gawk)
Javascript	\i
PCRE	\i

### 3.3 Metaznaky

Kromě alfanumerických znaků se v regulárních výrazech používají tzv. metaznaky. Metaznak má v regulárním výrazu speciální funkci a je jich celkem 12: `$()*+.[\^{}|`. Regulární výraz tyto metaznaky nerozpozná jako obyčejné alfanumerické znaky. Pro rozpoznání některého z výše uvedených znaků jako takových je třeba potlačit jejich funkci použitím zpětného lomítka `\` (tzv. escape). Znaky `]` a `}` není potřeba zapisovat se zpětným lomítkem, výjimkou jsou pouze případy, kdy následují za `[` nebo `{` bez zpětného lomítka. Jazyk AWK navíc vyžaduje i escapování dvojitých uvozovek `\"` a lomítka `\/`. Některé verze AWK navíc dovolují použít znaky `*+?` bez zpětného lomítka jako literály v případě, že se nevyskytují na místě kvantifikátoru (viz. kapitola Kvantifikátory). Při escapování jiného znaku než výše uvedené, většina jazyků bere následující znak jako literál a ne jako metaznak.

Metaznak `.` (tečka) zachytí jakýkoliv znak kromě znaku konce řádky. Toto pravidlo platí v jazycích Java, Javascript, Perl, PCRE, .NET nebo Python. Nicméně konec řádky lze tečkou zachytit použitím „single line“ módu („dot all“ módu). Tuto možnost ovšem nelze využít v jazyce Javascript. [5, s.34–36] Jazyk AWK podporuje mód multiline automaticky. Jak daný mód zapnout, lze zjistit z tabulky 2.

Dalšími metaznaky jsou `^` a `$`. `^` značí začátek řetězce a `$` konec řetězce. V angličtině se nazývají „anchors“ (kotvy). Tyto metaznaky se odlišují tím, že nezachycují žádný znak v řetězci, ale pouze pozici. Například regulární výraz: `^abc$` uspěje pouze na řetězci: „abc“,

Tabulka 2: Single line neboli Dot all mód

Dot all	
Java	Pattern.DOTALL
Perl	\s
AWK	již implicitně nastaveno
Javascript	nepodporuje
PCRE	\s

Tabulka 3: Multiline mód

Dot all	
Java	Pattern.MULTILINE
Perl	\m
AWK	dá se nastavit pomocí FS="\n"
Javascript	\m
PCRE	\m

ale neuspěje na řetězci „abcc“ nebo „aabc“. Při použití pouze jednoho z uvedených například regulární výraz `^abc` uspěje na řetězci „**abcc**“, ale ne na „aabc“. Analogicky regulární výraz `abc$` uspěje jak na řetězci „abc“, tak i na „**abc**“, ovšem selže na „abcc“. [5, s.37–40]

Důležité při používání tzv. kotev v regulárních výrazech je, že nezachycují začátek a konec řádku. Zachytí tedy celý víceřádkový řetězec. Tato vlastnost se dá změnit za pomoci „multiline“ módu.

Kulaté závorky `()` slouží pro zachycení části regulárního výrazu. Toto má celou řadu využití například za účelem alternace, nahrazení, opakování apod. Podrobněji se kulatými závorkami budu věnovat v kapitole Zachycení (capture).

Metaznak `|` značí alternaci neboli logické „nebo“. Pro příklad regulární výraz `(a|A)bc` uspěje na řetězci „abc“ nebo na řetězci „Abc“. V případě že bychom kulaté závorky vynechali, regulární výraz `a|Abc` uspěje buď na řetězci „a“ nebo „Abc“.

### 3.3.1 Třída znaků

Velmi užitečná je také třída znaků `[]`. Třída znaků slouží k rozpoznání libovolného znaku nebo několika znaků uvnitř hranatých závorek. Regulární výraz `[abc]` zachytí buď „a“ nebo „b“ nebo „c“. Uvnitř třídy znaků lze používat všechny metaznaky krom `^ \ - a ]` bez zpětného lomítka, jelikož uvnitř hranatých závorek ztrácejí svojí speciální funkci. V jazycích Java a .NET mezi metaznaky v třídě znaků patří také `[.s.31–32]`5

To proto, že třída znaků v jazyce Java také umožňuje intersekcce. Syntaxe intersekcí je následující: `[třída&&[intersekce]]`. Pomocí intersekcce je možné například vynechat některé znaky, které by jinak zahrnoval interval. Regulární výraz `[ac-fh-qs-y]` tak můžeme zapsat následovně: `[a-z&&[^bgrz]]`. [9]

NEMELO BY JIT `[A-z]`

Znak `^` uvnitř třídy znaků značí negaci, čili regulární výraz: `a[^abc]c` uspěje na řetězci „**adc**“, ale selže na „abc“. Je třeba brát v potaz, že v regulárním výrazu `[a^bc]`, kde není

`^` na prvním místě, se `^` chová jako literál. Znak `-` je mimo hranaté závorky literál, ovšem uvnitř hranatých závorek slouží k zapsání třídy znaků jako interval. Regulární výraz: `[0-9A-z]*` uspěje na řetězci „**A9g**“, ale selže na „č“ jelikož implicitně jsou rozeznány pouze znaky z ASCII tabulky. Toto platí pouze v případě, že se nenachází hned za uvozující závorkou tzn. `[-a]` nebo hned před uzavírající závorkou tzn. `[a-]`. V uvedených příkladech se pak `-` chová jako literál. [5, s.31–32]

Jako zjednodušení pro výše popsanou třídu znaků, existují zkratky tzv. „tokeny“. Znak `\w` je ekvivalentní s `[A-Za-z0-9_]` a to v jazyce Java, Javascript a PCRE. Jazyky .NET a Perl umožňují `\w` zachytit i znaky mimo ASCII tabulku (z českých znaků např. ý,á,í nebo é).

K třídě znaků `[0-9]` je ekvivalentní znak `\d`.

Dalším používaným znakem je bílý znak `\s`. Bílým znakem rozumíme znak, který označuje prázdné místo [6]. Mezi bílé znaky ve všech modifikacích regulárních výrazů patří tabulátor `\t`, nová řádka `\n`, konec stránky `\f`, vertikální tabulátor `\v`, mezeru a znak, který posune kurzor na začátek stránky `\r`. Všechny jazyky podporující Unicode vlastnosti, podporují i všechny bílé znaky v Unicode tabulce. Výjimkou jsou jazyky Java a PCRE. Horizontální tabulátor podporuje PCRE a Java8. [8]

Pro získání negace znaků `\w`, `\d` a `\s`, používáme velká písmena `\W`, `\D` a `\S`.

Pro zachycení znaků na začátku nebo konci slova lze použít znak `\b`. Například regulární výraz `\ba..g\b` uspěje na řetězci „**assg**“, ale selže při rozpoznání „**abbc**“. Znak `\B` hledá pozice, které jsou: před prvním nebo za posledním znakem slova (pokud posledním znak není znak slova), mezi dvěma znaky slova nebo jinými znaky, a prázdný řetězec. Příchutě v jazycích Javascript, Perl, .NET, PCRE a Python zachytí pozici mezi znaky rozpoznávanými `\w` a `\W`. Javascript a PCRE rozeznají pouze ASCII znaky, .NET, Java a Perl rozeznají jakékoliv znaky. Python umožňuje rozeznat i jiné než ASCII znaky, pokud použijeme Unicode flag. [5, s.42–43]

### 3.3.2 Kvantifikátory

Další skupinou metaznaků jsou kvantifikátory umožňující opakování znaku nebo skupiny znaků zachycenou v kulatých závorkách `()`.

Kvantifikátory a interval počtu opakování lze vidět v tabulce 4.

Tabulka 4: Kvantifikátory

Kvantifikátor	Počet opakování
<code>*</code>	<code>&lt;0;∞)</code>
<code>+</code>	<code>&lt;1;∞)</code>
<code>?</code>	0 nebo 1

Složené závorky `{}` též slouží jako kvantifikátor. Umožňují zadat přesný počet opakování např. `a{3}bc` uspěje na řetězci „**aaabc**“, protože obsahuje znak „a“ třikrát za sebou. Složené závorky dále umožňují nastavit minimum a maximum opakování např. `a{1,3}bc` uspěje na řetězcích „abc“, „aabc“ i „aaabc“. Další možností, kterou složené závorky nabízejí, je nastavit minimální počet opakování např. `a{2,}` uspěje na všech řetězcích, které obsahují minimálně dvě „a“ za sebou. [10]

### 3.3.3 Hladové vs. líné kvantifikátory

Všechny výše uvedené kvantifikátory jsou implicitně nastaveny jako „žravé“ (anglicky „greedy“). To znamená, že kvantifikátor pohltí, co nejvíce znaků, za kterými je umístěn. Toto chování vysvětlím na příkladě.

Mějme regulární výraz `.*abc` a textový řetězec „aabcaabc“. Kvantifikátor `*` nejprve pohltí veškeré znaky řetězce. Poté se bude snažit najít další znak „a“. Na tomto znaku, ale selže, jelikož v řetězci už nezůstává žádný další znak. Proto se provede „backtracking“, tak že postupně odebírá odzadu znaky, které pohltil kvantifikátor. To je možné díky tomu, že si zapamatovává pozice pohlcených znaků. Tím se vrátí na pozici „c“ a znovu ověří, zda znak na dané pozici je „a“. Zde selže, proto vrátí znak „b“. Opět selže a pokračuje znakem „a“. Regulární výraz zde uspěje a pokračuje znaky „b“ a „c“. Nakonec tedy regulární výraz na daném řetězci uspěje. [5, s.68]

„Žravé“ chování kvantifikátorů můžeme změnit na „líné“ (anglicky „lazy“), tak že na kvantifikátor přidáme „?“. Dostaneme tedy `*?+?/?` a `{ }?`. Po změně kvantifikátoru na lazy ve výše uvedeném regulárním výrazu, získáme `.*?abc`. Nyní se na řetězci „aabcaabc“ provádí po každém úspěšném rozeznání znaku „a“ „backtracking“, aby se ověřilo zda na následujících pozicích není „b“ a „c“. „Líný“ kvantifikátor tak opakuje znaků, co nejméně to jde. V uvedeném řetězci tedy nalezne pouze „aabcaabc“ [5, s.69]

Na obrázku 2 lze vidět srovnání postupu při rozpoznávání stejného řetězce pomocí regulárního výrazu s líným a žravým kvantifikátorem.

MATCH 1 - FINISHED IN 8 STEPS			MATCH 1 - FINISHED IN 9 STEPS		
1	/.*?abc/	aabcaabc	1	/.*abc/	aabcaabc
2	/.*?abc/	aabcaabc	2	/.*abc/	aabcaabc
3	/.*?abc/	aabcaabc	3	/.*abc/	aabcaabc
4	/.*?abc/	aabcaabc	4	/.*abc/	aabcaabc
5	/.*?abc/	aabcaabc	5	/.*abc/	aabcaabc
6	/.*?abc/	aabcaabc	6	/.*abc/	aabcaabc
7	/.*?abc/	aabcaabc	7	/.*abc/	aabcaabc
8	/.*?abc/	aabcaabc	8	/.*abc/	aabcaabc
#	Match found in 8 step(s)		9	/.*abc/	aabcaabc
			#	Match found in 9 step(s)	

Obrázek 2: Líný a žravý kvantifikátor

## 3.4 Optimalizace regulárních výrazů

Optimalizace regulárních výrazů spočítá v redukci počtu kroků, provedených při vyhodnocování regulárního výrazu. Následující metody mohou optimalizovat regulární výrazy zejména v případech, kdy má regulární výraz na daném řetězci selhat. [6, s.45]

### 3.4.1 Posesivní kvantifikátory

Jak bylo možné vidět na obrázku 2, „žravé“ i „líné“ chování kvantifikátorů vyžaduje mnohdy přebytný backtracking. Tomu lze předejít použitím posesivních kvantifikátorů.

Ty dostaneme přidáním + za kvantifikátor, tedy: `*,++,?+ a { }+`. Jsou podporovány pouze v jazycích Java (od verze 4), PCRE a Perl (od verze 5.10).

Posesivní kvantifikátor se chová podobně jako „žravý“, ovšem po pohlčení všech možných výskytů daného znaku, již neprovádí backtracking a žádný již pohlčený znak nevrací. Nezapamatovává si totiž pozice. Stejný regulární výraz jako v předchozích případech, při použití posesivního kvantifikátoru, `.*+abc` pak na řetězci „aaaabc“ neuspěje. Kvantifikátor totiž pohltní všechny znaky v řetězci a při porovnávání znaku „a“ za kvantifikátorem selže, protože se neprovede backtracking. (viz. obrázek 4). [5, s.70-71]

**MATCH 1 - FINISHED IN 13 STEPS**

1	/a*+abc/	aaaabc	
2	/a*+abc/	aaaa bc	
3	/a*+abc/	a aaabc	
4	/a*+abc/	aaaa bc	BACKTRACK
5	/a*+abc/	aa aabc	
6	/a*+abc/	aa aa bc	BACKTRACK
7	/a*+abc/	aaa abc	
8	/a*+abc/	aaa a bc	BACKTRACK
9	/a*+abc/	aaaa bc	
10	/a*+abc/	aaaa bc	BACKTRACK
11	/a*+abc/	aaaab c	
12	/a*+abc/	aaaab c	BACKTRACK
13	/a*+abc/	aaaab c	
#	Match failed in 13 step(s)		

Obrázek 3: Posesivní kvantifikátory

### 3.4.2 Atomické seskupování

Další možností, jak se předejít přebytku backtrackingu, je použití atomického seskupení (`?> výraz`). Tato syntaxe umožňuje do závorek umístit jakoukoliv část regulárního výrazu, v rámci kterého si nepřejeme provádět backtracking (tzn. výraz uvnitř závorek je „nedělitelný“). Podobně jako u posesivních kvantifikátorů se uvnitř kulatých závorek nezapamatovávají pozice pro backtracking.

Využití bychom mohli najít například při alternaci. Regulární výraz `(?>bcb)c` — uspěje pouze na „bcc“, nikoliv na „bc“. [5, s.72]

Atomické seskupování není podporováno v jazyce Javascript.

## 3.5 UNICODE vlastnosti

Unicode Standard a ISO/IEC 10646 podporují UTF-8, UTF-16 a UTF-32. [11] Což umožňuje převést libovolné znaky do strojově čitelného textu. Nejnovější verzi standardu Unicode je verze 8.0. Je podporováno mnoha jazyky včetně Javascriptu dle normy ECMA Script-262 6. edice, nicméně AWK jej nepodporuje.

Pro unicodové znaky (tzv. grafémy) mohou mít například tvar `\u2122`, což je podporovaný tvar jazyky Javascript a Java (za předpokladu zapnutí módu unicode `\u`), nebo

The image shows three sequential screenshots of a regular expression testing interface. Each screenshot displays a 'REGULAR EXPRESSION' field, a 'TEST STRING' field containing 'aaaaec', and a status bar indicating 'NO MATCH' and the number of steps taken.

- Top Screenshot:** The regular expression is `a*abc`. The status bar shows 'NO MATCH - 32 STEPS'.
- Middle Screenshot:** The regular expression is `a*+abc`. The status bar shows 'NO MATCH - 12 STEPS'.
- Bottom Screenshot:** The regular expression is `(?>a*)abc`. The status bar shows 'NO MATCH - 24 STEPS'.

Obrázek 4: Srovnání počtu kroků v jazyce PCRE (php)

`\x{2122}` podporované jazykem Perl nebo PCRE. Tvar `\u+0000` umožňuje použít právě 4 hexa číslice, zatímco `\x{00}` umožňuje použít právě tolik hexa číslic, kolik je potřeba.

V regulárních výrazech můžeme použít i unicodové vlastnosti (anglicky property). `\p{vlastnost}` pak reprezentuje jakýkoliv znak s danou vlastností ve složených závorkách. Pokud chceme její negaci použijeme `\P{vlastnost}`. Tuto funkci podporuje pouze Java, Perl, PCRE a .NET. (ne Javascript ani awk)

`\p{L}` představuje jakékoliv písmeno z kteréhokoliv jazyka. Můžeme tuto vlastnost doplnit `\p{Lu}`, což značí velké písmeno nebo `\p{Ll}`, co značí malé písmeno. `\p{Z}` zachytí mezeru nebo neviditelný separátor. Tuto vlastnost můžeme upřesnit `\p{Zs}`, což značí neviditelný prázdný znak nebo `\p{Zl}` značící oddělovač řádků U+2028 či `\p{Zp}` pro oddělovač odstavců U+2029. Pro matematické symboly, znaky měn apod. můžeme použít `\p{S}`. Pro čísla existuje vlastnost `\p{N}`, která zachytí veškerá čísla od 0 do 9 i římské číslice. Interpunkci (tzn. tečka, čárka, uvozovky, závorky apod..) je možné zachytit pomocí `\p{P}`. Pro tečku a čárku můžeme použít `\p{Pd}`, pro otevírací závorku `\p{Ps}` a uzavírací závorku `\p{Pe}`. [5, s.45-47]

Unicode bloky jsou tvořeny seskupenými unicodovými znaky. Jsou podporovány pouze v jazyce Perl, .NET a Java. Syntaxe `\p{nazev_bloku}` se ovšem v různých jazycích může lišit. Java používá vždy před názvem bloku ve složených závorkách „In“ (např. `\p{InBasicLatin}`), Perl umožňuje jak „In“ tak „Is“. Názvů bloků je celá řada, například blok pro znaky 0000..007F se nazývá Basic\_Latin, 0400..04FF pro Cyrilici nebo 0600..06FF pro arabské znaky (více viz. [12]). [5, s.44] V jazyce Javascript jde stejného výsledku dosáhnout pomocí třídy znaků např. `/[\u0400-\u04FF]+/`.

Syntaxe pro unicode písma `\p{nazev_skriptu}` je podporována v Javě (od verze 7),

PCRE i Perlu. Java vyžaduje před názvem skriptu ještě „Is“ (např. `\p{IsGreek}`) [13]. Názvy unicodových skriptů jsou například: „Latin“, „Arabic“, „Brahmi“ nebo od verze 8 také „Multani“ [14].

## 3.6 Look Around

Look around rozpoznává pozici v textu. Existují čtyři druhy look around.

„Pozitivní lookahead“ (`(?= ...)`) uspěje, pouze pokud doprava od dané pozice se nachází řetězec vyhovující regulárnímu výrazu v kulatých závorkách. Je podporován v jazycích Java, Javascript, Perl a PCRE. Například regulární výraz `aaa(=bc)` uspěje na řetězci „**aaabc**“ (pozn. zachytí se pouze zvýrazněná část).

„Negativní lookahead“ (`(?! ...)`) neuspěje, pokud se doprava od dané pozice nachází řetězec vyhovující regulárnímu výrazu v závorkách. Znamená to tedy, že regulární výraz `aaa(!bc)` neuspěje na řetězci „**aaabc**“, ale upspěje na „**aaagh**“. Je podporován v jazycích Java, Javascript, Perl i PCRE.

„Pozitivní lookbehind“ (`(?<= ...)`) uspěje, pouze pokud doleva od dané pozice se nachází řetězec vyhovující regulárnímu výrazu v kulatých závorkách. Je podporován v jazycích Java, Perl a PCRE. Regulární výraz `(?<=bc)df` uspěje na „**bcd**df****“.

„Negativní lookbehind“ (`(?<! ...)`) se chová podobně jako negativní lookahead, s tím rozdílem, že kontroluje řetězec doleva od dané pozice. Regulární výraz `(?<!bc)df` neuspěje na řetězci „**bcd**df****“, ale na „**aad**df****“ ano. Negativní lookbehind je podporován v jazycích Java, Perl a PCRE.

Negativní a pozitivní lookbehind nepodporuje kvantifikátory, které umožňují nekonečné opakování jako `* + {1,}` uvnitř kulatých závorek. [5, s.75-77]

## 3.7 Zachycování (capture)

Zachycování neboli anglicky „capture“ slouží k zachycení určité části výrazu. K tomu slouží kulaté závorky `()` (viz. podkapitola Metaznaky). Zachycení pak má celou řadu využití jako např. za účelem nahrazení, zpětné odkazy atd... Text je pak zachycen v oddělených skupinách, které se číslují od jedné do nekonečna.

V případě, že nechceme, aby regulární výraz zachytil některou část textu můžeme danou část umístit do `(?: ...)`. Tatot syntaxe se dá využít zejména při alternaci (viz. podkapitola Metaznaky). Regulární výraz `(jablka)(?:anebo)(hrušky)`— na řetězci „jablka a hrušky“ zachytí „jablka“ jako první skupinu a „hrušky“ jako druhou skupinu. Narozdíl od `(jablka)(anebo)(hrušky)`—, kde druhá skupina obsahuje „a“ a „hrušky“ jsou obsaženy až ve třetí skupině. Tato syntaxe je podporována v jazycích Java, Perl i Javascript.

Dle označení skupin je pak možné se na ně odkazovat. V jazycích Perl nebo Java je možné odkazovat na zachycený text pomocí zpětného lomítka a čísla dané skupiny (např. `\u{1}` odkazuje na první zachycenou skupinu ve výrazu tzn. „jablka“). Regulární výraz `(jablka)(a|nebo) \1`, pak uspěje na řetězci „jablka a jablka“. Jazyk AWK text zachycený v závorkách uloží do pole, tudíž se zachyceným textem může dále pracovat jako s položkami v poli (viz. kapitola AWK).

Regulární výrazy také umožňují zachycené skupiny pojmenovávat a pak na ně odkazovat pomocí daného názvu, pomocí syntaxe `(?P<nazev>...)(?P=nazev)` v jazycích Perl nebo `(\k<nazev>...)\k<nazev>`. Výše uvedený regulární výraz dospěje ke stejnému výsledku



i touto syntaxí: (?P<ovoce>jablka) a (?P=ovoce) nebo (?<ovoce>jablka) a \k<ovoce>.  
Pojmenovávání skupin není podporováno v jazyce Javascript a AWK.

## 4 Jazyky s vestavěnou podporou regulárních výrazů

### 4.1 Perl

### 4.2 AWK

#### 4.2.1 Obecné informace

AWK je jak počítačovým programem, tak i programovacím jazykem, který umožňuje daný program ovládat. Je zahrnut téměř ve všech UNIX systémech. Slouží zejména ke zpracování textových dat. K tomu hojně využívá regulární výrazy.

#### 4.2.2 Syntaxe

Jazyk AWK se vyznačuje hned několika odlišnostmi od ostatních modifikací regulárních výrazů. Regulární výraz se zapisuje mezi dvě lomítka (`\ ... \`). AWK umožňuje porovnávání řetězců pomocí operátorů `~` a `!~`. Syntaxe `pozice ~ /regVyraz/` umožňuje vyhledat všechny výrazy, které odpovídají vzoru. `pozice ! /regVyraz/` umožňuje naopak najít všechny výrazy na dané pozici, které vzoru neodpovídají. [18, s.45]

Další odlišností, která nebyla v přechozím textu zmíněna, je používání odlišný zkratek pro třídu znaků. AWK umožňuje používat třídu znaků definovanou POSIX standardem, pomocí `[:nazevTridy:]`. Seznam níže uvádí přehled názvů tříd a jejich popis. [18, s.51]

- `[:alnum:]` veškeré alfanumerické znaky
- `[:alpha:]` veškeré znaky písmen
- `[:blank:]` mezera a tabulátor
- `[:cntrl:]` control znaky
- `[:digit:]` čísla
- `[:graph:]` tištitelné a viditelné znaky např. „k“
- `[:lower:]` malá písmena
- `[:print:]` tištitelné znaky
- `[:punct:]` interpunkce, která nespádá ani do jedné z výše uvedených skupin
- `[:upper:]` kapitálky
- `[:xdigit:]` znaky v hexadecimálním tvaru

Třídu znaků je nutné zapisovat do dvojité hranaté závorky např. `[[:alpha:]]`.

AWK čte zadaný příkaz dvakrát. Poprvé přečte zadaný program a podruhé vyhledá řetězec vyhovující zadanému regulárnímu výrazu. [18, s.53]

Jak již bylo zmíněno výše, AWK má implicitně nastaven mód „single line“. Při načítání souboru rozděluje vložený text do záznamů oddělené právě znakem nové řádky „`\n`“. Pro změnu separátoru můžeme nastavit proměnnou `RS` na jakýkoli jiný znak.

Záznamy jsou pak dále děleny na pole, která jsou lépe programově zpracovatelná (program si totiž zapamatovává jejich pozice pomocí počtu záznamů, které již byly přečteny). Separátor rozdělující pole je buď literál nebo regulární výraz. Separátor můžeme nastavit pomocí proměnné „`FS`“.

---

```
echo Jimmy Weasel, 100 Any Street#Jane Weasel, 99 Same Street | awk
  'BEGIN{RS="#"; FS=","} /.+/{print $1}'

#vytiskne:
#Jimmy Weasel
#Jane Weasel
```

---

Pro nahrazování použijeme metodu `sub()`.

---

```
echo abcdefg| awk '{sub(/abc/, "cba");print}'
# vytiskne "cbadefg"
```

---

## 4.3 Javascript

### 4.3.1 Obecné informace

Javascript je objektově orientovaný jazyk, používaný jako skriptovací jazyk pro webové stránky. Nicméně je spustitelný i v nástroji `node.js`. Standardem pro Javascript je ECMAScript. Nejnovější verzí je ECMAScript 6 (neboli ECMA–262 6. edice) vydaná v roce 2015. Ovšem není podporován všemi internetovými prohlížeči ani nástrojem `node.js`, proto se v této kapitole budu držet syntaxe standardizovaném ECMAScript 5.1, která je plně podporována.

ECMAScript 6 přináší významné novinky z hlediska syntaxe<sup>1</sup>. Z hlediska regulárních výrazů byly přidány dvě vlaječky a to: `\u` pro unicodové vlastnosti a `\y` tzn. „sticky matcher“, který umožňuje procházet zadaný textový řetězec od pozice dané v proměnné `RegExp.lastIndex`.

### 4.3.2 Syntaxe

Regulární výrazy v Javascriptu se zapisují mezi dvě lomítka `/.../` a módy, které chceme použít se zapisují až nakonec, za druhé lomítko např. `/.../gim`. [16]

Pro vytvoření regulárního výrazu z textového řetězce je zapotřebí vytvořit objekt pomocí konstruktoru `RegExp`. Tento konstruktory může mít buď jeden parametr, obsahující regulární výraz bez lomítek, a nebo dva parametry, z nichž první obsahuje regulární výraz bez lomítek a druhý vlaječku. Při vytváření objektu regulárního výrazu může dojít

---

<sup>1</sup>Více informací o změnách lze nalézt zde: <http://es6-features.org/>

k výjimce *SyntaxError*, která, jak už název napovídá, je vyvolána při zadání výrazu s nesprávnou syntaxí. [16]

---

```
//Metoda zjišťuje zda byla zadána i vlaječka a poté vytvoří regulární výraz
//pomocí konstruktoru RegExp
function prevodReg(reg,f){
    var vystup;
    try{
        if(f == ""){
            //zadaný výraz se převede do regulárního výrazu bez vlaječky
            vystup = new RegExp(reg);
        }else{
            //zadaný výraz se převede do regulárního výrazu i s vlaječkou
            vystup = new RegExp(reg,f);
        }
    }catch (error){
        //odchycení výjimky SyntaxError
        console.log("SyntaxError: Nesprávně zadaný výraz");
        main();
    }
    return vystup;
};
```

---

Takto vytvořený *RegExp* objekt dědí po svém předku metody:

- *RegExp.prototype.test()*, která otestuje, zda zadaný řetězec odpovídá vzoru. Při nalezení shody vrátí *true*, v opačném případě *false*.
- *RegExp.prototype.exec()*, která vrací nalezenou část řetězce odpovídající zadanému vzoru. A to v podobě pole. V případě, že nenajde žádnou shodu, vrátí *null*. Stejného výsledku docílíme i metodou *String.prototype.match()*. Ta navíc umožňuje vložit jako parametr jak objekt *RegExp* tak i regulární výraz v typu *string*. [16]

---

```
//Metoda zjišťuje shodu zadaného výrazu (str) a vzorem (re)
function zjistitShodu(re,str){

    //zjistí se, zda daný text odpovídá vzoru
    if(re.test(str)){

        //vrací substring odpovídající vzoru
        var newstr = str.match(re);

        //použití RegExp.prototype.exec()
        //var newstr = re.exec(str);

        console.log(newstr);
    }else{
        console.log("Žádná shoda");
    }
};
```

---

Například při zadání regulárního výrazu: `[12][0-9]{3}` a řetězce „v roce 1990“, výše uvedená metoda vypíše do konzole: `['1990',index: 7,input: 'v roce 1990']`. Na první pozici se nachází část řetězce odpovídající vzoru. Index značí pozici, na které se nachází „1990“. Input pak ukazuje vložený řetězec.

Pro nahrazení části řetězce odpovídající danému vzoru, slouží metoda *String.prototype.replace()*. Metoda vrací nový řetězec tvořený původním řetězcem s nahrazenými částmi. Jako parametr může být použit, jak regulární výraz ve tvaru řetězce (string), tak i jako objekt *RegExp*. Druhým parametrem je nahrazující řetězec, který může být zadán jako string nebo jako metoda, která se zavolá při každém použití daného regulárního výrazu.. [17]

---

```
//metoda slouží k nahrazení části zadaného řetězce (str), která odpovídá
    regulárnímu výrazu (reg)
//řetězcem "repl"
function nahradit(reg,str,repl){
//zjistí, zda daný řetězec odpovídá vzoru
    if(reg.test(str)){
        //nahradí zadaný text
        var newstr = str.replace(reg,repl);
        console.log(newstr);
        main();
    }else{
        console.log("Žádná shoda");
        main();
    }
};
```

---

Metoda *replace()* neumožňuje vkládat vlaječky jako parametr. Toto omezení lze obejít využitím *RegExp* objektu. Na zachycené části textu (viz. podkapitola Zachycení), se lze odkazovat pomocí `$d`, kde *d* představuje číslo dané skupiny. [17]

## 5 Jazyky s knihovnamí

### 5.1 C#

### 5.2 Python

### 5.3 Java

#### 5.3.1 Obecné informace

Jazyk Java je objektovým programovacím jazykem. Jeho první verze vyšla v roce 1995. Ve čtvrté verzi vydané v roce 2002, byl do Javy přidán (mimo jiné) balíček podporující regulární výrazy *java.util.regex*. Java používá modifikaci regulárních výrazů velmi podobné jazyku Perl, což umožňuje v Javě používat i pokročilé regulární výrazy. Podpora dalších funkcí v regulárních výrazech přišla s šestou verzí, kdy byla přidána podpora unicodových skriptů (viz. podkapitola Unicode) a pojmenovávání zachycených skupin (viz. podkapitola Zachycování(capture)).

Významným odlišením od syntaxe regulárních výrazů v jiných jazycích (včetně jazyku Perl) je použití dvojitého zpětného lomítka při „escapování“ v datovém typu String. Tudíž je důležité například místo `\w` napsat `\\w`. To proto, aby se byly Java kompilátorem správně přeloženy. To platí i pro zapisování unicodových znaků. [13]

#### 5.3.2 Používané třídy a metody

Balíček *java.util.regex* obsahuje třídy:

- *Pattern*, která provádí kontrolu syntaxe regulárního výrazu. Nemá veřejný (public) konstruktor, k vytvoření je proto třeba *public static Pattern compile(String regex)*. Při volání této metody je možný vznik výjimky *PatternSyntaxException*. Tato výjimka se nemusí povinně odchyťovat. [13]
- *Matcher*, která zajišťuje porovnávání zadaného vzoru a textového řetězce. Podobně jako třída *Pattern*, nemá veřejný konstruktor. K jejímu vytvoření je třeba volání metody na instanci třídy *Pattern* *public Matcher matcher(CharSequence input)*. [15] Tato metoda může vyvolat výjimku *NullPointerException* v případě, že její parametr bude *null*.

---

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class RegexLogika {
```

```
private Pattern pattern;  
private Matcher m;
```

---

Třída `Matcher` pak má metody: `public boolean matches()` umožňující porovnání řetězce se zadaným vzorem. `public boolean find()` hledající počátek části zadaného řetězce, který odpovídá vzoru. Vrací `true`, v případě, že řetězec obsahuje část (tzv. „substring“) odpovídající vzoru. Vyhledaný substring, pak lze získat pomocí metody `public String group()`. [15]

---

```
/**  
 * Metoda uplatní uživatelem zadaný vzor na uživatelem zadaném řetězci  
 * @param regex - vzor  
 * @param vstup - textový řetězec  
 * @return vyhledaný substring  
 */  
public String vyhledat(String regex, String vstup) {  
    String vystup = "";  
    //regularni vyraz zadany uzivatelem  
    try {  
        pattern = Pattern.compile(regex);  
    } catch (PatternSyntaxException e) {  
        JOptionPane.showMessageDialog(null, " Chybný regulární výraz",  
            "REGEX hlášení", JOptionPane.PLAIN_MESSAGE, null);  
    }  
    //vstup zadany uzivatelem  
    try {  
        m = pattern.matcher(vstup);  
        if (m.find()) {  
            vystup = m.group();  
        } else {  
            vystup = "ŽÁDNÁ SHODA";  
        }  
    } catch (NullPointerException ex) {  
        vystup = "ŽÁDNÁ SHODA";  
    }  
    return vystup;  
}
```

---

Metodu `group()` můžeme nahradit:

---

```
//index prvního znaku nalzeného textu  
zacatek = m.start();  
//index posledního znaku nalezeného textu  
konec = m.end();  
//vytvoření substringu ze zadaného textu  
vystup = vstup.substring(zacatek, konec);
```

---

Další metodou třídy `Matcher` je `public String replaceAll(String replacement)`, která zajistí nahrazení všech výskytů nalezeného substringu v zadaném textu, textem či odkazy udanými v parametru metody.

K nahrazení pouze prvního výskytu daného substringu slouží metoda *public String replaceFirst(String replacement)*. Obě metody vrací objekt typu *String*, který obsahuje výsledný řetězec po nahrazení.

Parametr obou metod může obsahovat odkazy na zachycené skupiny (viz. kapitola Zachycení) za pomoci `${navezSkupiny}` nebo `$d` (kde *d* značí číslo skupiny). [15]

---

```
Pattern p = Pattern.compile("(\\w+)\\s(\\w+)");
Matcher m = p.matcher("John Smith");
String vysledek = m.replaceAll("$2 $1");
//proměnná vysledek je pak rovna "Smith John"
```

---



# Literatura

- [1] DOSTÁL, Hubert. *Teorie konečných automatů, regulárních gramatik, jazyků a výrazů* [online]. 2008 [cit. 2015-07-11]. Dostupné z: <http://iris.uhk.cz/tein/teorie/regJazyk.html>
- [2] THOMPSON, Ken. *Regular Expression Search Algorithm*. Communications of the ACM: Volume 11 [online]. 1968, (6): 419-422 [cit. 2015-07-11]. Dostupné z: <http://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf>
- [3] Perl Compatible Regular Expressions. *Wikipedia* [online]. 2015-07-03 [cit. 2015-07-11]. Dostupné z: <https://en.wikipedia.org/wiki/Perl-Compatible-Regular-Expressions>
- [4] Regular Expression. *Wikipedia* [online]. 2015-07-08 [cit. 2015-07-11]. Dostupné z: [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)
- [5] GOYVAERTS, Jan a Steven LEVITHAN. *Regular Expressions Cookbook*. 1. the United States of America: O'Reilly Media, Inc., 2009. ISBN 978-0-596-52068-7.
- [6] GOYVAERTS, Jan. *Regular Expressions: The Complete Tutorial* [online]. 2007, July 2007 [cit. 2015-07-19]. Dostupné z: <http://www.regular-expressions.info/print.html>
- [7] Bílý znak. *Wikipedie* [online]. 2015-07-17 [cit. 2015-07-18]. Dostupné z: [https://cs.wikipedia.org/wiki/B%C3%ADl%C3%BD\\_znak](https://cs.wikipedia.org/wiki/B%C3%ADl%C3%BD_znak)
- [8] Shorthand Character Classes. GOYVAERTS, Jan. *Regular-Expressions.info* [online]. 2014-04-24 [cit. 2015-07-18]. Dostupné z: <http://www.regular-expressions.info/shorthand.html>
- [9] Character Class Intersection. GOYVAERTS, Jan. *Regular-Expressions.info* [online]. 2014-09-26 [cit. 2015-07-19]. Dostupné z: <http://www.regular-expressions.info/charclassintersect.html>
- [10] The Java™ Tutorials. ORACLE. Oracle Java Documentation [online]. [cit. 2015-07-19]. Dostupné z: <https://docs.oracle.com/javase/tutorial/essential/regex/quant.html>
- [11] The Unicode® Standard: A Technical Introduction. UNICODE, INC. The Unicode Standard [online]. 2015-06-25 [cit. 2015-07-22]. Dostupné z: [http://www.unicode.org/standard/principles.html#Unicode\\_and\\_ISO](http://www.unicode.org/standard/principles.html#Unicode_and_ISO)
- [12] Blocks. Unicode [online]. 2014-11-10 [cit. 2015-07-23]. Dostupné z: <http://www.unicode.org/Public/UCD/latest/ucd/Blocks.txt>

- [13] Class Pattern. ORACLE. *Java™ Platform Standard Ed. 8 Documentation* [online]. 2015 [cit. 2015-07-25]. Dostupné z: <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>
- [14] Supported Scripts. UNICODE, INC. *The Unicode Standard* [online]. 2015-06-12 [cit. 2015-07-25]. Dostupné z: <http://unicode.org/standard/supported.html>
- [18] ROBBINS, Arnold D. *GAWK: Effective AWK Programming* [online]. Edition 4.1. USA: Free Software Foundation, 2015 [cit. 2015-07-28]. ISBN 1-882114-28-0. Dostupné z: <http://www.gnu.org/software/gawk/manual/gawk.pdf>
- [16] Regular Expressions. MOZILLA. Mozilla Developer Network [online]. 2015-07-21 [cit. 2015-07-26]. Dostupné z: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions)
- [17] String.prototype.replace(). MOZILLA. Mozilla Developer Network [online]. 2015-07-13 [cit. 2015-07-27]. Dostupné z: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String/replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace)
- [15] Class Matcher. ORACLE. *Java™ Platform Standard Ed. 8 Documentation* [online]. 2015 [cit. 2015-07-25]. Dostupné z: <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html>

## Seznam obrázků

1	Backtrack . . . . .	4
2	Líný a žravý kvantifikátor . . . . .	7
3	Posesivní kvantifikátory . . . . .	8
4	Srovnání počtu kroků v jazyce PCRE (php) . . . . .	9

# Seznam tabulek

1	Case insensitive (viz. [5]) . . . . .	4
2	Single line (viz. [5]) . . . . .	5
3	Multiline mód (viz. [5]) . . . . .	5
4	Kvantifikátory (viz. [5]) . . . . .	6