# Regulární výrazy a gramatiky

Petr Kolář

Bakalářská práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Petr Kolář** |
| Osobní číslo: | **A16081** |
| Studijní program: | **B3902 Inženýrská informatika** |
| Studijní obor: | **Softwarové inženýrství** |
| Forma studia: | **prezenční** |
| Téma práce: | **Regulární výrazy a gramatiky** |
| Téma anglicky: | **Regular Expressions and Grammars** |

Zásady pro vypracování:

1. **Popište regulární gramatiky a jejich vztah ke konečným automatům, regulárním výrazům.**
2. **Popište bezkontextové jazyky, gramatiky a jejich vztah k zásobníkovým automatům.**
3. **Vysvětlete postup zpracování výrazů.**
4. **Uveďte základní syntaxi pro psaní regulárních výrazů a gramatik.**
5. **Tuto syntaxi demonstrujte na ilustrativních příkladech.**
6. **Vyřešte komplexní problém pomocí regulárních výrazů a gramatik.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:


1. **ŠESTÁKOVÁ, Eliška. Automaty a gramatiky: sbírka řešených příkladů. V Praze: České vysoké učení technické, 2017. ISBN 978-80-01-06306-4.**
2. **HABIBALLA, Hashim. Regulární a bezkontextové jazyky I. Ostrava: Ostravská univerzita, 2003. Systém celoživotního vzdělávání Moravskoslezska. ISBN 80-7042-852-X.**
3. **STUBBLEBINE, Tony. Regular expression: pocket reference. 2nd ed. Farnham: O'Reilly, 2007, vii, 117 s. ISBN 978-0-596-51427-3.**
4. **FRIEDL, Jeffrey E. F. Mastering regular expressions. 3rd ed. Farnham: O'Reilly, 2006, xxiv, 515 s. ISBN 978-0-596-52812-6.**
5. **GOYVAERTS, Jan; LEVITHAN, Steven. Regulární výrazy: Kuchařka programátora. Computer Press, 2010.**
6. **MORITZ, Lenz. Parsing with Perl 6 Regexes and Grammars. Apress, 2017.**
7. **NAGY, Zsolt. Regex Quick Syntax Reference: Understanding and Using Regular Expressions. Apress, 2018.**


Vedoucí bakalářské práce: **Mgr. Jan Krňávek, PhD.**
Ústav matematiky

Datum zadání bakalářské práce: **3. prosince 2018**

Termín odevzdání bakalářské práce: **15. května 2019**

Ve Zlíně dne 7. prosince 2018


doc. Mgr. Milan Adámek, Ph.D.
*děkan*

prof. Mgr. Roman Jašek, Ph.D.
*garant oboru*

**Prohlašuji, že**

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 17. 5. 2019

Petr Kolář, v. r.
podpis diplomanta

## ABSTRAKT

Tato bakalářská práce se zabývá použitím regulárních výrazů a gramatik v jazyce Perl 6 k parsování datových struktur. Teoretická část obsahuje popis a klasifikaci formálních jazyků. Následně je uveden jejich vztah ke gramatikám, regulárním výrazům a konečným automatům. Praktická část popisuje základní syntaxi pro psaní regulárních výrazů v jazyce Perl 6 a vytvořenou knihovnu pro parsování formátu vCard.



Klíčová slova: regulární výraz, gramatika, regex, vCard, jCard, parsování, Perl 6



## ABSTRACT

This Bachelor's thesis deals with the usage of regular expressions and grammars in Perl 6 for parsing data structures. The theoretical part contains description and classification of formal languages. Their relations to grammars, regular expressions and finite automata is presented subsequently. The practical part describes the basic Perl 6 syntax for writing regexes and the created vCard library for parsing.



Keywords: regular expression, grammar, regex, vCard, jCard, parsing, Perl 6

Tímto bych chtěl poděkovat vedoucímu práce, panu Mgr. Janu Krňávkovi, Ph.D. za odborné rady, vstřícný přístup, a především všechen čas, který mi věnoval v průběhu zpracování bakalářské práce.

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

# ÚVOD

Teorie formálních jazyků je pro informatiku velmi důležitá. Vychází z ní například regulární výrazy, které se v současnosti nejčastěji využívají k vyhledání v textu nebo validaci uživatelských vstupů. Pro tyto účely existuje již mnoho knihoven, využívajících právě regulární výrazy. Oproti jiným přístupům umožňují vyhledávat komplexnější shody. Jsou rychlé a spolehlivé. Umožňují jednodušším způsobem, oproti manuální definici podmínek, specifikovat potřebnou množinu povolených vstupních hodnot.

Většina programovacích jazyků však regulární výrazy nepodporuje přímo v jejich kódu. Využívají k tomu starší knihovny, které jsou založeny na syntaxi jazyka Perl. Nová verze tohoto jazyka (Perl 6) spoustu věcí změnila, a to včetně zápisu regulárních výrazů. Perl 6 má tudíž značné množství chybějících knihoven, které stále čekají na implementaci.

Formát vCard se používá pro sdílení kontaktních informací. V jazyce Perl 6 v současné době zatím neexistuje žádná funkční implementace tohoto formátu.

Cílem této práce je v teoretické části popsat formální jazyky, gramatiky a jejich vztah. V praktické části zase popsat základní syntaxi Perlu 6 a vytvořit knihovnu, která by přijímala na vstupu formát vCard a za pomocí gramatik a regulárních výrazů Perlu 6 jej zpracovala syntaktickou analýzou. Výstupem bude objekt, který bude možno převést na jiný formát.

# I. TEORETICKÁ ČÁST

# 1 FORMÁLNÍ JAZYKY

Před klasifikací a popisem jazyků musíme zavést následující pojmy.

**Definice 1.1.** *Abeceda* Σ je libovolná konečná neprázdná množina prvků, které nazýváme *symboly* abecedy. [1]

**Definice 1.2.** *Řetězec (slovo)* nad abecedou Σ je každá konečná posloupnost symbolů abecedy. Prázdný řetězec ε je taková posloupnost, která neobsahuje žádný symbol. [2]

**Definice 1.3.** *Množinu všech řetězců nad abecedou Σ značíme $\Sigma^*$ a nazýváme uzávěrem množiny Σ. Jestliže z $\Sigma^*$ odebereme ε, dostaneme pozitivní uzávěr množiny, který značíme $\Sigma^+$.* [1, s. 5]

**Definice 1.4.** Nechť $\alpha$ a $\beta$ jsou řetězce nad abecedou Σ. *Zřetězením* těchto dvou řetězců (značíme $\alpha \cdot \beta$) vznikne řetězec $\alpha\beta$. [2]

**Příklad 1.1.**

Nad abecedou $\Sigma = \{a, b, 0\}$ můžeme například sestavit řetězce:

$$\alpha = aaa$$

$$\beta = b$$

$$\gamma = 0ab$$

Zřetězením $\alpha$ a $\beta$ vzniká řetězec $\alpha\beta = aaab$.

**Definice 1.5.** *Formálním jazykem* L nad abecedou Σ nazýváme libovolnou podmnožinu množiny všech možných řetězců nad abecedou Σ. [3]

*Formální jazyk L je:* [1, s. 5]

- *prázdný, jestliže $L = \emptyset$,*
- *konečný, jestliže obsahuje konečně mnoho slov,*
- *nekonečný, jestliže obsahuje nekonečně mnoho slov.*

Platí operace nad jazyky: [3]

$L_1 \cdot L_2 = \{ \alpha\beta : \alpha \in L_1; \beta \in L_2 \}$

$L^* = \{ \alpha_1\alpha_2 \cdots \alpha_n : \alpha_1, \alpha_2, \ldots, \alpha_n \in L_1; n \in N \} \cup \{ \varepsilon \}$

## 1.1 Klasifikace formálních jazyků

Formální jazyky členíme na: [3]

- jazyky nerozpoznatelné Turingovým strojem,
- rekurzivně spočetné jazyky,
- kontextové jazyky,
- bezkontextové jazyky,
- regulární jazyky – jsou nejjednodušší množinou formálních jazyků.

Regulární jazyk je vždy zároveň jazykem bezkontextovým, kontextovým a rekurzivně spočetným.

Bezkontextový jazyk je vždy zároveň kontextový a rekurzivně spočetný.

## 1.2 Gramatiky

*Gramatika je prostředek pro popis jazyků.* [3, s. 10]

**Definice 1.6.** *Gramatikou* nazýváme uspořádanou čtveřici $G = (N, \Sigma, P, S)$, kde

- $N$ je abeceda se symboly zvanými neterminály,
- $\Sigma$ je abeceda se symboly zvanými terminály, splňující podmínku $N \cap \Sigma = \emptyset$ (jedná se o abecedu jazyka generovaného gramatikou),
- $P$ je konečná množina přepisovacích pravidel ve tvaru
$$\alpha A \beta \rightarrow \gamma \qquad (\alpha, \beta, \gamma \in (N \cup \Sigma)^*, A \in N)$$
- $S \in N$ je počáteční neterminální symbol. [1; 3]

Skupinu pravidel můžeme místo

$$A \rightarrow t_1$$
$$\vdots$$
$$A \rightarrow t_n$$

zapisovat zkráceně jako

$$A \rightarrow t_1 \mid t_2 \mid \ldots \mid t_n.$$

### 1.2.1 Chomského klasifikace gramatik

Vymezuje čtyři typy gramatik podle tvaru přepisovacích pravidel: [3; 4]

*Neomezená gramatika* (typ 0) má pravidla ve tvaru:

$$\alpha A\beta \rightarrow \gamma \qquad (\alpha, \beta, \gamma \in (N \cup \Sigma)^*,\ A \in N)$$

*Kontextová gramatika* (typ 1) má pravidla ve tvaru:

$$\gamma A\beta \rightarrow \gamma\alpha\beta \qquad (\beta, \gamma \in (N \cup \Sigma)^*,\ A \in N,\ \alpha \in (N \cup \Sigma)^+)$$

*Bezkontextová gramatika* (typ 2) má pravidla tvaru:

$$A \rightarrow \alpha \qquad (\alpha \in (N \cup \Sigma)^*,\ A \in N)$$

Bezkontextová gramatika je vždy zároveň neomezená. Může být také kontextová nebo regulární.

*Regulární gramatika* (typ 3) má pravidla tvaru:

$$A \rightarrow a \text{ nebo } A \rightarrow aB \qquad (a \in \Sigma^*,\ A, B \in N)$$

Regulární gramatika je vždy zároveň bezkontextová, kontextová a neomezená.

V gramatikách typu 1, 2 a 3 se prázdný řetězec generuje pravidlem $S \rightarrow \varepsilon$. V takovém případě počáteční symbol $S$ nesmí být na pravé straně žádného pravidla dané gramatiky.

**Příklad 1.2.**

Nechť je dán regulární jazyk L.

L = $\{\alpha : \alpha \in \{a, b\}^* \wedge \alpha$ obsahuje podřetězec *abba*$\}$

Jazyk je generován regulární gramatikou $G = (\{S, A, B, C, D\}, \{a, b\}, P, S)$, kde

$P = \{S \rightarrow aS \mid bS \mid aA,$

$A \rightarrow bB,$

$B \rightarrow bC,$

$C \rightarrow aD,$

$D \rightarrow aD \mid bD \mid a \mid b\}.$

**Příklad 1.3.**

Nechť je dán regulární (tj. i bezkontextový) jazyk L z příkladu 1.2.

Jazyk je také generován bezkontextovou gramatikou (která není regulární) $G = (\{S, A\}, \{a, b\}, P, S)$, kde

$P = \{S \rightarrow AabbaA$

$A \rightarrow aA \mid bA \mid \varepsilon\}$.

**Příklad 1.4.**

Nechť je dán bezkontextový jazyk L, který není regulární.

$L = \{a^n b^n : n \in N\}$

Jazyk je generován bezkontextovou gramatikou (která není regulární) $G = (\{S\}, \{a, b\}, P, S)$, kde $P = \{S \rightarrow aSb\}$.

Platí: [2]

Jazyk je bezkontextový právě tehdy, když je generován bezkontextovou gramatikou.

Jazyk je regulární právě tehdy, když je generován regulární gramatikou.

## 1.3 Konečné automaty

**Definice 1.7.** *Konečný automat* je uspořádaná pětice $M = (Q, \Sigma, \delta, q0, F)$, kde

- $Q$ je konečná množina stavů,
- $\Sigma$ je konečná vstupní abeceda,
- $\delta$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů. [3]

Je to jednoduchý výpočetní model. Můžeme ho chápat jako stroj, který nemá paměť. Nachází se vždy v jednom z konečné množiny stavů a postupně čte symboly vstupního řetězce. V závislosti na přečteném symbolu a momentálním stavu automat buď setrvá ve stejném stavu, nebo přejde do stavu nového. [4]

Rozlišujeme dva základní typy konečných automatů.

### 1.3.1 Deterministický konečný automat

Přechodová funkce deterministického automatu (dále DKA) je definována následovně: [3]

*$\delta$ je (částečné) zobrazení z množiny $Q \times \Sigma$ do množiny stavů $Q$.*

Přechodová funkce nemusí být definována pro všechny dvojice stavů $q \in Q$ a vstupních symbolů $a \in \Sigma$.

DKA přijímá vstupní řetězec právě tehdy, když je celý přečten a automat skončí v některém z koncových stavů.

DKA nepřijímá vstupní řetězec právě tehdy, když

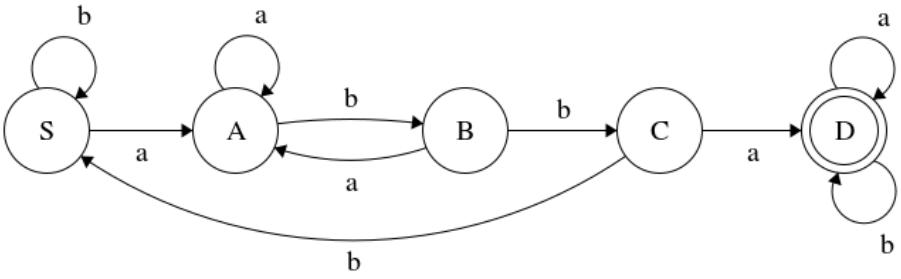
- je celý přečten a automat skončí v nekoncovém stavu,
- při čtení není pro danou dvojici stav-symbol definována přechodová funkce.

**Definice 1.8.** *Jazyk přijímaný konečným automatem* je množina všech řetězců, které automat přijímá. [3]

Libovolný jazyk je regulární, právě tehdy když jej lze přijmout konečným automatem. [4, s. 66]

**Příklad 1.5.**

Sestrojme DKA k jazyku L z příkladu 1.2.




*Obr. 1. DKA jazyka s podřetězcem abba.*

### 1.3.2 Nedeterministický konečný automat

Přechodová funkce nedeterministického konečného automatu (dále NKA) je definována následovně: [3]

*$\delta$ je zobrazení z množiny $Q \times \Sigma$ do množiny všech podmnožin množiny $Q$.*

Takto definovaná přechodová funkce umožňuje automatu pro danou dvojici stav-symbol přejít do libovolného z odpovídajících stavů.

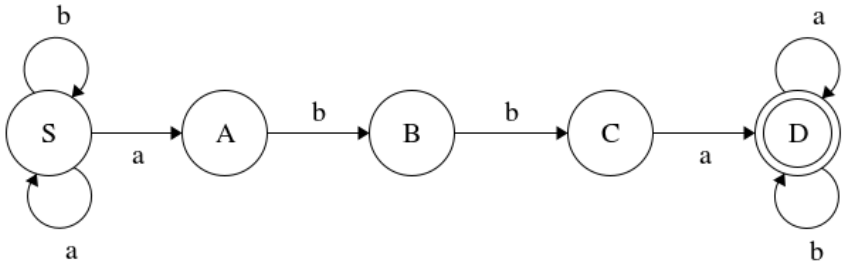
NKA přijímá vstupní řetězec právě tehdy, když existuje alespoň jedna posloupnost přechodů, která po přečtení celého vstupního řetězce skončí v koncovém stavu.

NKA nepřijímá vstupní řetězec právě tehdy, když

- je řetězec přečten a automat skončí v nekoncovém stavu,
- je při čtení výsledek přechodové funkce pro danou dvojici stav-symbol prázdná množina možných stavů.

**Příklad 1.6.**

Sestrojme NKA k jazyku L z příkladu 1.2.




*Obr. 2. NKA jazyka s podřetězcem abba.*

DKA a NKA jsou výpočetně ekvivalentní a jdou mezi sebou převádět. Důkazy tvrzení a příklady jednotlivých převodů jsou uvedeny v [1, s. 21; 4, s. 28].

## 1.4 Regulární výrazy

*Regulární výrazy slouží k přehlednějšímu zápisu regulárních jazyků*. [4, s. 63]

**Definice 1.9.** Nechť Σ abeceda neobsahující symboly $\varepsilon$, $\emptyset$, $+$, $\cdot$, $*$, $($, $)$. Řekněme, že $R$ je *regulární výraz* nad Σ, jestliže je $R$ rovno: [1]

- $\varepsilon$,
- $\emptyset$,
- $a$, kde $a \in \Sigma$,

- $(R_1 + R_2)$, kde $R_1$, $R_2$ jsou regulární výrazy,
- $(R_1 \cdot R_2)$, kde $R_1$, $R_2$ jsou regulární výrazy,
- $(R_1)^*$, kde $R_1$ je regulární výraz.

Jestliže regulární výraz $R_1$ reprezentuje regulární jazyk $L_1$ a $R_2$ reprezentuje $L_2$, pak

- $R_1 + R_2$ označuje jazyk $L_1 \cup L_2$,
- $R_1 \cdot R_2$, označuje jazyk $L_1 \cdot L_2$,
- $R_1^*$ označuje jazyk $L_1^*$. [4]

*Třída regulárních jazyků je rovna třídě jazyků popsatelných regulárními výrazy.* [1, s. 31]

**Příklad 1.7.**

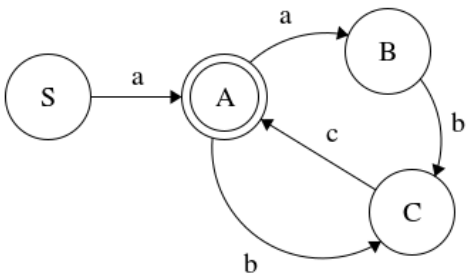
Nechť $\Sigma = \{a, b, c\}$.

Nad danou abecedou můžeme sestrojit například regulární výrazy:

$R_1 = a \cdot (bc + abc)^*$,

$R_2 = (a + b)^* \cdot abba \cdot (a + b)^*$.

Pro $R_1$ můžeme navrhnout konečný automat:




*Obr. 3. DKA výrazu $a(bc + abc)^*$.*

a převést na gramatiku:

$G = (\{S, A, B, C, D, E\}, \{a, b, c\}, P, S)$, kde

$P = \{S \rightarrow aA,$

$A \rightarrow \varepsilon \mid aB \mid bC,$

$B \rightarrow bC,$

$C \rightarrow cA\}$.

Výraz $R_2$ lze převést na DKA z příkladu 1.5. a gramatiku z příkladu 1.2.

Existuje několik metod převodu výraz/automat/gramatika. Podrobně se jim i s příklady věnuje [3, s. 69].

## 1.5 Zásobníkové automaty

Zásobníkový automat (ZKA) můžeme chápat jako NKA rozšířený o paměť typu zásobník. [1]

**Definice 1.10.** *Zásobníkový automat* je uspořádaná sedmice $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$, kde

- $Q$ je konečná neprázdná množina stavů,
- $\Sigma$ je konečná vstupní abeceda,
- $G$ je konečná neprázdná abeceda zásobníku,
- $\delta$ je přechodová funkce, definována jako zobrazení $Q \times (\Sigma \cup \{\varepsilon\}) \times G^*$ do množiny konečných podmnožin množiny $Q \times G^*$,
- $q_0 \in Q$ je počáteční stav,
- $Z_0 \in G$ je počáteční symbol na zásobníku,
- $F \subseteq Q$ je množina koncových stavů. [3, s. 153]

Automat se na základě aktuálního stavu, vstupního symbolu (nemusí se číst) a symbolu na vrcholu zásobníku může rozhodnout, jak změní svůj stav a jaký symbol zapíše na vrchol zásobníku.

Jsou dva typy ZKA. Rozdělují se podle přijímání jazyka. [3]

### 1.5.1 ZKA přijímající prázdným zásobníkem

ZKA přijímá vstupní řetězec právě tehdy, když existuje alespoň jedna posloupnost přechodů, která po přečtení celého vstupního řetězce skončí v libovolném stavu s prázdným zásobníkem.

ZKA nepřijímá vstupní řetězec právě tehdy, když

- je řetězec přečten a má neprázdný zásobník,
- je při čtení výsledek přechodové funkce pro danou trojici stav-symbol-zásobník prázdná množina možných stavů.

### 1.5.2 ZKA přijímající koncovým stavem

ZKA přijímá vstupní řetězec právě tehdy, když existuje alespoň jedna posloupnost přechodů, která po přečtení celého vstupního řetězce skončí v koncovém stavu.

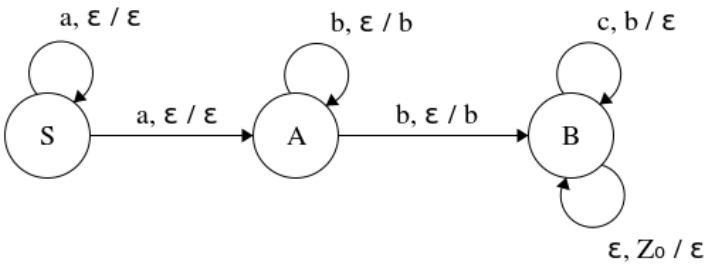
ZKA nepřijímá vstupní řetězec právě tehdy, když

- je řetězec přečten a automat skončí v nekoncovém stavu,
- je při čtení výsledek přechodové funkce pro danou trojici stav-symbol-zásobník prázdná množina možných stavů.

*Formální jazyk je bezkontextový právě tehdy, když je přijatelný zásobníkovým automatem.* [2, s. 51]

**Příklad 1.8.**

Nechť je dán bezkontextový jazyk L = $\{a^n b^m c^m : m, n \in N\}$. ZKA na obrázku (Obr. 4.), přijímá jazyk L prázdným zásobníkem.




*Obr. 4. Zásobníkový automat.*

Čárka odděluje znak na vstupu a zásobník (čte/zapisuje). Když se ve stavu B objeví na zásobníku $Z_0$, tak automat přijme slovo.

Bezkontextovou gramatiku můžeme převádět na zásobníkový automat. Podrobně je převod uveden v knize [3].

# II. PRAKTICKÁ ČÁST

# 2 ZPRACOVÁNÍ A SYNTAXE REGEXŮ V PERLU 6

Pojem regulární výraz se v Perlu 6 v mnohém liší od regulárního výrazu z výše uvedené teorie. V praxi se pro něj často používá pojem *regex*. Touto zkratkou budeme rozumět právě regulární výraz Perlu 6.

Zpracování výrazů v Perlu 6 úzce souvisí s jejich syntaxí. Postup jejich zpracování je tedy uváděn průběžně v aktuální kapitole spolu se stručným přehledem základní syntaxe Perlu 6 pro zápis regexů. Podrobný popis syntaxe, nejen regulárních výrazů, lze najít v [5; 6].

V Perlu 6 se regexy dají nejjednodušeji napsat pomocí uvození lomítky.

```
/regex/
```

Všechny znaky uvnitř tedy tvoří regex. Alfanumerické znaky a podtržítko jsou doslovné. To znamená, že jediné mají ve výrazu jejich původní význam. Všechny ostatní znaky mají význam jiný. Pokud je chceme použít pro doslovné vyhledávání, musíme je napsat do uvozovek, nebo před ně dát zpětné lomítko. [5]

```
/Ah0j_světe ‘,‘/
```

Tenhle výraz nalezne shodu, když se v řetězci vyskytuje přesně:

```
Ah0j_světe,
```

Shoda se hledá vždy zleva doprava ve vstupním textu, na který je regex zavolán. [7]

Znak mezery se pro lepší přehlednost regexu ignoruje. Význam mezery jde ale zapnout.

## 2.1 Znakové třídy

Znaková třída je skupina znaků, jejichž prvky spolu nějak souvisí. Můžeme je dělit na předdefinované a vlastní. Používají se pro zjednodušení psaní regexů.

Většina předdefinovaných znakových tříd je reprezentována zástupným symbolem. V následující tabulce je stručný přehled často používaných znakových tříd. [8]

*Tab. 1. Základní znakové třídy.*

| Značení | Popis | Příklad výrazu | Příklad shody |
|---|---|---|---|
| . | jakýkoliv znak | a.c | aac, abc, a@c |
| \d | číslice | \d\d | 00, 12 |
| \w | alfanumerický znak a podtržítko | a\w | aa, a0, a_ |

| | | | |
|---|---|---|---|
| `\n` | znak nového řádku | `ahoj\nsvete` | `ahoj`<br>`svete` |
| `\t` | znak tabulátoru | `a\tb` | `a  b` |
| `\s` | bílý znak | `a\sb` | `a b` |

Perl 6 umožnuje také používání vlastních znakových tříd. Zapisují se výčtem jednotlivých znaků do `<[]>` závorek.

Například následující třída odpovídá číslici a písmenům `c` a `d`.

```
<[cd\d]>
```

Negaci třídy je možné zapsat pomocí znaménka minus před vnitřní závorkou. [9]

```
<-[\s]>
```

Předcházející třída odpovídá všem znakům, které nejsou ve třídě `\s`. Jedná se tedy o všechny nebílé znaky. Takovou třídu ale nemá smysl používat, protože již existuje jako předdefinovaná.

Každá ze tříd, která začíná zpětným lomítkem, má svůj negovaný ekvivalent, který se liší velikostí písmena za lomítkem. Negace třídy bílých znaků se pak zapisuje jako `\S`. [5; 7]

Třídy znaků jdou dále sjednocovat a rozdělovat pomocí operátorů `+` a `−`. Podporují také rozsah, pro zkrácení zápisu, značící se dvěma tečkami mezi požadovanými krajovými znaky. [7]

```
<[\d] - [0123] + [a..f]>
```

Tato třída obsahuje všechny číslice, vyjma `0`, `1`, `2`, `3` a dále šest písmen `a`, `b`, `c`, `d`, `e`, `f`.

Perl 6, na rozdíl od jiných programovacích jazyků, standardně využívá kódování UTF-8. To se promítlo i do znakových tříd, které tak mají širší okruh znaků než ASCII. [5; 8]

**Příklad 2.1.**

Následující kód prohledává řetězec složený ze dvou arabských a indických číslic, na výskyt čtyř číslic.

```
5 say so '13৭৯' ~~ /\d\d\d\d/;
```

Protože Perl 6 podporuje Unicode znaky ve znakových třídách, bude na výstupu:

```
2 True
```

**Příklad 2.2.**

Mějme řetězec obsahující písmeno:

- latinky, azbuky, alfabety, arménštiny

A druhý řetězec, který obsahuje navíc znak `$`.

Oba testujeme na přítomnost výhradně alfanumerických znaků.

```
1 say so 'D3βψ' ~~ /\w\w\w\w/;
2 say so 'D3βψ$' ~~ /\w\w\w\w\w/;
```

Výstupy jsou následující:

```
1 True
2 False
```

Druhý řetězec `D3βψ$` není přijat, protože obsahuje nealfanumerický znak `$`.

## 2.2 Kvantifikátory

Kvantifikátory se vztahují k bezprostředně předcházejícímu znaku (může být oddělen mezerou). Vyjadřují, kolikrát se může předcházející znak vyskytnout. [7; 8]

*Tab. 2. Přehled kvantifikátorů.*

| Značení | Popis | Příklad výrazu | Příklad shody |
|---|---|---|---|
| `?` | znak se může vyskytnout pouze jednou, nebo vůbec | `a?b` | `b, ab` |
| `*` | žádný výskyt, nebo libovolněkrát | `a*b` | `b, ab, aab` |
| `+` | znak se musí vyskytnout alespoň jedenkrát | `a+b` | `ab, aab` |
| `** N` | znak se opakuje přesně N-krát | `a**2` | `aa` |

Výše uvedené `N` vyjadřuje přesný počet opakování, ovšem nemusí se uvádět pouze jako celé číslo. Místo něj lze definovat celočíselný rozsah opakování znaku pomocí dvou proměnných, které jsou odděleny dvěma tečkami. [8]

*Tab. 3. Celočíselný rozsah.*

| Značení | Popis | Příklad výrazu | Příklad shody |
|---|---|---|---|
| `N .. M` | znak se objeví minimálně N-krát a maximálně M-krát | `a**3..6` | `aaa` |
| `N^ .. M` | znak se může vyskytnout v intervalu (N, M> | `a**3^..6` | `aaaa` |
| `N .. *` | znak nemá definovaný maximální limit výskytů | `a**3..*` | `aaaaa` |

V Perlu 6 můžeme také tyto kvantifikátory použít, pokud chceme vyhledávat stejné vzory oddělené libovolným znakem. Používá se při tom znak `%`. [8] Např. výraz

```
[\d ** 2..4]+ %  ‘,‘
```

nalezne shodu s následujícími dvěma řetězci.

```
12, 345, 5678, 11

12
```

Z předcházející ukázky vyplívá, že každé dvoumístné až čtyřmístné číslo musí být odděleno čárkou, která se na konci nezachytává.

Když budeme chtít zachytit i čárku na konci, použijeme znaky `%%`.

## 2.3 Seskupení

Seskupení se značí hranatými závorkami. [5] Bylo již využito v minulém výrazu.

```
[\d ** 2..4]+ %  ‘,‘
```

Jeho cílem je spojit danou část výrazu v jeden logický celek. V tomto případě se jednalo o seskupení dvou až čtyř číslic v jeden celek, kterým bylo číslo. Takovýto celek se musel minimálně jednou vyskytnout ve zdrojovém textu a v případě vícero výskytů být oddělen čárkami.

Bez seskupení by nebylo jasné, k čemu náleží kvantifikátor +, jelikož bezprostředně před ním je definován rozsah.

Seskupení můžeme také realizovat pomocí kulatých závorek. Jejich používání má ale ještě další význam. Provádějí totiž zachytávání. Uzávorkovaný celek je uložen v Perlu 6 do proměnné a můžeme s ním později pracovat. [8; 9]

Pokud zachycení neplánujeme využít, je lepší použít seskupení bez zachycení. Smysl výrazu je tak jasnější a vyhledávání rychlejší.

**Příklad 2.3.**

Zapišme regex pro reálné číslo s pravidly:

- Před první číslicí a za exponentem může být znaménko.
- Alespoň jedna číslice před desetinnou tečkou je povinná.
- Po desetinné tečce musí vždy přijít číslice.
- Za znakem exponentu musí být minimálně jedna číslice.

Kód v Perlu 6:

```
1 for '0', '1.1', '+54.','3e+1', '-7.8E-6' {
2   say $_ ~~ /
3     [\+|\-]? \d+
4     [\.\d+]?
5     [
6       [e|E] [\+|\-]? \d+
7     ]? /
8 }
```

Výstup:

```
1 「0」
2 「1.1」
3 「+54」
4 「3e+1」
```

```
5 「-7.8E-6」
```

Věnujme pozornost třetímu výstupu. Výraz našel shodu v první části výrazu, ale ve druhé ne. Po tečce totiž očekával číslici, avšak řetězec skončil. Přesto jsme obdrželi alespoň první část s dosazením neúplné shody. Tuto problematiku řeší tzv. kotvy.

## 2.4 Kotvy

Perl 6 vyhledává shody s výrazem v celém vstupním textu. [7] Regex

```
/bys/
```

proto nalezne shodu ve všech následujících řetězcích:

```
byl bys
```

```
bystrý
```

```
Přibyslav
```

Pokud bychom požadovali přesnější výsledky, např. shodu pouze na okraji slova, můžeme využít tzv. kotvy. V tabulce je stručný přehled nejpoužívanějších kotev. [8]

*Tab. 4. Přehled základních kotev.*

| Symbol | Popis |
|---|---|
| ^ | značí začátek řetězce |
| ^^ | značí začátek řádku |
| $ | značí konec řetězce |
| $$ | značí konec řádku |
| << | levý okraj slova |
| >> | pravý okraj slova |

**Příklad 2.4.**

Mějme řetězec

```
Vlk nese domovníkovi psaní.
```

a použijme na něj následující regexy obsahující kotvu.

```
/^Vlk/, /aní.$/, /^ne/, /ovi >>/
```

V Perlu 6 je můžeme zapsat například takto:

```
1 for / ^Vlk /, /ané.$/, /^ne/, /ovi>>/ {
2     say 'Vlk nese domovníkovi psaní.' ~~ $_
3 }
```

Výstup:

```
1 ｢Vlk｣
2 ｢ané.｣
3 Nil
4 ｢ovi｣
```

V prvním případě výraz očekává `Vlk` na začátku celého řetězce. Ve druhém očekává, že řetězec po písmenech `ané.` skončí. Oba tyto výrazy shodu naleznou.

Třetí výraz hledá na začátku řetězce `ne`, ale nalezne písmeno `V`. Vyhledávání proto skončí neúspěchem.

Kdybychom nepoužili kotvy, všechny výrazy by uspěly, protože dané sekvence znaků v kontextu celého řetězce existují.

Poslední výraz hledá část slova, která končí na `ovi`, a to tím způsobem, že napravo od výrazu musí být znak ze třídy `\W` a nalevo znak od něj znak z `\w`.

## 2.5 Alternace

Alternace se použijí, když budeme chtít najít pouze jednu z několika možností, resp. daný regex bude mít více podob, a každá z nich bude mít vlastní pravidla.

Za alternativní část se považuje celý výraz od znaku alternace do ohraničení výrazu, popřípadě dalšího znaku alternace. [8]

Jsou dvě varianty alternací. První se značí pomocí jedné svislé čáry

```
/děd | děda | dědovo/
```

a druhá pomocí dvou.

```
/děd || děda || dědovo/
```

U první varianty se hledá shoda pro každou alternativu paralelně a je vybrána nejdelší část výrazu, která neselže. U druhé se hledá shoda postupně zleva a vezme se první varianta, která neselže.

**Příklad 2.5.**

Na řetězec `Vlci vyli naMěsíc` aplikujme regex s oběma variantami

```
1 for / Měsíc | na /, / Měsíc || na / {
2     say 'Vlci vyli naMěsíc.' ~~ $_ ;
3 }
```

Výstup:

```
1 ｢na｣
2 ｢na｣
```

První výraz hledá paralelně nejdelší možnou variantu. Výraz je ale vždy zpracováván zleva podle vstupního řetězce. Jako první je nalezena alternativní část obsahující `na`. K druhé části (obsahující `Měsíc`) se vyhodnocení nedostalo. Obě varianty proto vrací shodu: `na`.

Pořadí částí výrazu zde nemá vliv. Stejného výsledku dosáhneme i s výrazy:

```
/ na | Měsíc /, / na || Měsíc /
```

Upravením výrazu následujícím způsobem

```
1 for / na || naMěsíc /, / na | naMěsíc / {
2     say 'Vlci vyli naMěsíc.' ~~ $_ ;
3 }
```

vznikají dvě alternativy, přičemž jedna z nich rozšiřuje tu druhou.

Výstup:

```
1 ｢na｣
2 ｢naMěsíc｣
```

Zde je již pozorovatelný rozdíl. První výraz je zpracován sekvenčně, a zvítězí tedy první část. Druhý výraz napřed nalezne shodu u obou a vrátí druhou (nejdelší) alternativu.

## 2.6 Gramatiky

Struktury se složitějšími pravidly není vhodné popisovat pouze jedním regexem. U komplexnějších problémů často potřebujeme zavést určitou modularitu. Rozdělit problém na menší logické celky, které pojmenujeme a budeme zachytávat, abychom s nimi později mohli pracovat. K tomu můžeme v Perlu 6 použít gramatiky. [10; 11]

Gramatika seskupuje dílčí regexy v jeden celek a pomáhá tak udržovat kód čitelným a snadněji modifikovatelným. Využívá se převážně pro parsování textu a s tím související kontrolu syntaxe. [12]

Vytváří se pomocí klíčového slova `grammar`.

```
1 grammar My-grammar {
2 }
```

V jazyce Perl 6 jsou gramatiky interpretovány jako speciální třídy, které obsahují tři základní typy metod: [11]

- `regex` – metoda, která má aktivní backtracking, proto nalezne shodu tam, kde by ostatní metody selhaly. Je ale pomalejší než zbylé dva typy.
- `token` – velmi rychlá metoda s nastaveným přehlížením bílých znaků.
- `rule` – podobná metodě token, avšak bílé znaky mají speciální význam.

V gramatikách můžeme tyto metody kombinovat a vzájemně volat. Na pořadí metod v gramatice nezáleží:

```
3 grammar Contact {
4     token name    { \w ** 3..15 }
5     rule  number  {'+'[ \d]**6..14 \d}
6     token contact { <name> ':' <number> }
7 }
```

Pokud použijeme `rule`, pak bílé znaky zastupují volání metody s názvem `ws`, která bílým znakům přiděluje význam. Výstup z metody se poté nezachytává. Metodu `ws` můžeme redefinovat podle potřeby. [11]

Zachytávání se obecně vypíná tečkou před názvem volané metody. Metoda

```
8 rule line {<name> ':' <number>}
```

je ekvivalentní s

```
9 token line { <name> <.ws> ':' <.ws> <number> }
```

Gramatika dále obsahuje metodu `TOP`, která je standardně volána vždy při parsování jako první. Zastřešuje všechny ostatní metody a musí vždy nalézt úplnou shodu ve vstupním textu. Pokud by kterákoliv část této metody selhala, gramatika vrátí prázdný objekt `nil`. Při nálezu je vrácen objekt se shodou. Parsování se spouští metodou `parse`, kterou třída `grammar` obsahuje. [10]

**Příklad 2.6.**

Vytvořme gramatiku pro parsování jednoho řetězce s kontaktu ve formě:

- Jméno – bude obsahovat 3 až 15 písmen.
- Separátor – `\s` nebo podtržítko.
- Znak +
- Telefonní číslo – bude obsahovat 7 až 15 číslic, které můžou být odděleny Separátorem

Příklad gramatiky v Perlu 6:

```
1 grammar Contact {
2     token name { \w ** 3..15 }
3     token number { \d ** 7..15 % <.ws> }
4     token ws { <[\s_]>? }
5     rule TOP {<name> '+'<number>}
6 }
7 say Contact.parse('John +1_23 4 56_7');
```

K oddělení jednotlivých číslic se používá metoda `ws`. Na řádku 4 vidíme předefinování metody `ws`. Protože je `TOP` typu `rule`, má mezera před znakem + význam `<.ws>`.

Výsledek parsování je následující:

```
1 ｢John +1_23 4 56_7｣
2  name => ｢John｣
3  number => ｢1_23 4 56_7｣
```

Parsování proběhlo úspěšně, a proto jsme obdrželi výstup z metody `TOP` (řádek 1), který obsahuje dva objekty (`name` a `number`).

**Příklad 2.7.**

Gramatika na parsování zdrojového kódu z hlediska komentářů:

- Za komentář bude považováno všechno za znakem #, za kód všechno na řádku před komentářem.
- Ve zdrojovém kódu může být libovolný počet prázdných řádků.
- Řádek může obsahovat jenom kód, jenom komentář, nebo obojí.

Řešení gramatiky:

```
1 grammar Comment {
2     token code { <[\N] - [#]>+ }
3     token comment { \N* }
4     token line { <code>? ['#' <comment> ]? }
5     token TOP { <line>+ %% \n }
6 }
7 my $text = "\nAhoj#koment1\nDruhy radek \nTreti radek #koment2\n#";
8 say Comment.parse($text)
```

Výstup:

```
1 「
2 Ahoj#koment1
3 Druhy radek
4 Treti radek #koment2
5 #」
6 line => 「Ahoj#koment1」
7   code => 「Ahoj」
8   comment => 「koment1」
9  line => 「Druhy radek 」
10   code => 「Druhy radek 」
11  line => 「Treti radek #koment2」
12   code => 「Treti radek 」
13   comment => 「koment2」
14  line => 「#」
15   comment => 「」
```

Můžeme si povšimnout, že na výstupu byla dodržena struktura gramatiky. `TOP` vrátil jednotlivé `line` oddělené novým řádkem. Například na řádku 11 je `line`, jejíž hodnota je dále parsována. Vznikne tak část `code` a `comment`, které jsou rozděleny znakem `#`. Parsování dále pokračuje řádkem 14 s novou `line`.

## 2.7 Backtracking

Základní pravidlo zpracovávání výrazů říká, že kvantifikátory jsou hladové. Určují minimum a maximum výskytů znaků pro úspěšný nález a vždycky se snaží dosáhnout největšího počtu znaků. Dokud jsou úspěšné, pokračují ve hledání. [7; 10]

**Příklad 2.8.**

Porovnejme následující kód:

```
1 say 'Vysoko' ~~ / \w+ /;
2 say 'Vysoko' ~~ / \w+ o /;
3 say 'Vysoko' ~~ / \w+: o /;
```

Výstup:

```
1 ｢Vysoko｣
2 ｢Vysoko｣
3 Nil
```

Na prvním řádku, díky kvantifikátoru, musí řetězec obsahovat alespoň jeden alfanumerický znak. Maximální počet však není omezen. Kvantifikátor tedy nalezne všech šest znaků.

Na druhém řádku ale dochází k problému. Je vyžadován neomezený počet znaků. Na konci řetězce, ale musí být přesně znak `o`. Takový zápis výrazu je chybný, protože kvantifikátor se vztahuje k třídě `\w`, která `o` již obsahuje. Řetězec je pak celý nalezen pod kvantifikátorem a část s `o` nikdy nenalezne shodu.

Avšak na výstupu je shoda nalezena. To proto, že je standardně v každém regexu zapnut backtracking. Ten jde na jednotlivé kvantifikátory vypnout pomocí znaku `:`, jako tomu je ve třetím případě. Můžeme tak vidět, že bez backtrackingu došlo k selhání.

V následujícím příkladu je názorně ukázáno, jak backtracking postupuje při hledání shody. Využíváme při tom gramatiku pro snadnější sledování hodnot.

**Příklad 2.9.**

Mějme řetězec `abcabca` a parsujme jej následující gramatikou:

```
1 grammar Back {
2     regex pre  { .* {say 'pre'=> $/}};
3     regex post { .* {say 'post' => $/}};
4     regex TOP  { <pre> bc <post> bc .* };
5 }
6 say Back.parse('abcabca');
```

Výstup:

```
7 pre => ｢abcabca｣
```

```
 8 pre => ⌜abcabc⌟
 9 pre => ⌜abcab⌟
10 pre => ⌜abca⌟
11 post => ⌜a⌟
12 post => ⌜⌟
13 pre => ⌜abc⌟
14 pre => ⌜ab⌟
15 pre => ⌜a⌟
16 post => ⌜abca⌟
17 post => ⌜abc⌟
18 post => ⌜ab⌟
19 post => ⌜a⌟
20 ⌜abcabca⌟
21  pre => ⌜a⌟
22  post => ⌜a⌟
```

Na příkladu vídíme, že token <pre> nejprve pojmul celý vstup. Nastalo selhání a backtracking začal od konce ubírat znaky tak, aby další část metody TOP našla shodu (bc). Nyní (řádek 11) začala ze vstupu brát metoda <post>. Na řádku 12 můžeme vidět, že backtracking odebral i poslední symbol, ale shoda stejně nenastala. Vyhodnocení se tedy vrátilo zpět k metodě <pre> a odebralo další symbol (řádek 17). Znova se hledala část s bc. Po jejím nalezení zůstal metodě <pre> s hladovým kvantifikátorem pouze jeden symbol. Zbytek výrazu poté prošel a od řádku 20 vidíme, že celé slovo bylo úspěšně rozparsováno.

Kvantifikátory a znakové třídy sice hodně usnadňují zápis regexů, avšak v některých případech mohou způsobovat neočekávané problémy. Backtracking je může vyřešit a díky tomu můžeme snadněji zapsat složitý výraz, avšak za cenu výkonu. [10]

Backtrackingem se velmi podrobně zabývá Friedl. [7, s. 157]

## 2.8 Akce

Jelikož přistupování na jednotlivé objekty až po parsování, a jejich následné zpracovávání, je neefektivní. Můžeme tento proces pomocí akcí zautomatizovat a spustit při probíhajícím parsování. [10]

Akce se předávají jako parametr metody `parse`, protože jsou definovány v jiné třídě. [11]

```
Grammar.parse($text, actions => Grammar-actions.new);
```

Pro každou metodu, která bude použita při parsování gramatikou, můžeme použít metodu se stejným názvem v akční třídě. Každá příslušná metoda v akci je volána automaticky vždy, když dojde k použití metody v gramatice. To nám umožní oddělit logiku parsování (v gramatice) a sestavení (v akcích). [12]

**Příklad 2.10.**

Upravme gramatiku z příkladu 2.6. následovně:

```
1 grammar Contact {
2     token name { \w ** 3..15 }
3     token number { \d ** 7..15 % <.ws> }
4     token ws { <[\s_]>? }
5     rule contact {<name> '+'<number>}
6     token TOP { <contact>+ % [';'<.ws>] }
7 }
8 say Contact.parse('John+1234567; Jan +420_676_445');
```

Jediným rozdílem je nová metoda `TOP`, která může obsahovat více kontaktů oddělených alespoň středníky.

Výstupem parsování je:

```
 9 「John+1234567; Jan +420_676_445」
10  contact => 「John+1234567」
11   name => 「John」
12   number => 「1234567」
13  contact => 「Jan +420_676_445」
14   name => 「Jan」
15   number => 「420_676_445」
```

Je patrné, že metoda `TOP` vrátila objekt, který obsahuje dvě instance kontaktu. Pokud bychom chtěli získat např. jména ze všech kontaktů, museli bychom objekt procházet postupně v cyklu. Využitím akčních metod tomu předcházíme:

```
16 class Contact-action {
17     method TOP ($/) {
```

```
18          make $<contact>».made;
19      }
20      method contact ($/) {
21          make $<name>.Str;
22      }
23  }
24  say Contact.parse('John+1234567; Jan +420_676_445',
      actions => Contact-action).made;
```

Ukázková akční třída implementuje metody `TOP` a `contact`. Výstup parsování gramatiky (řádek) je argumentem akční metody `TOP`. Můžeme si všimnout, že akce používají metody `make` a `made`.

Na řádku 18 `make` umístí všechny objekty `contact` do zvláštní paměti. Při parsování kontaktu gramatikou je potom zavolána akční metoda `contact`, která vrací řetězec se jménem. V separátní paměti jsou tak po skončení parsování uložena všechna jména. Výstup akce (řádek 24) získáme zavoláním metody `made` na výstup parsování.

Výstup akce:

```
25  [John Jan]
```

Na gramatiky, akce a parsování s praktickými ukázkami se zaměřuje především Lenz. [10]

## 2.9 Proto-regexy

Proto-regexy zlepšují čitelnost a také modifikovatelnost zápisu. Předcházejí vzniku metod, které by obsahovaly velké množství alternací.

Klíčové slovo `sym` slouží k vytvoření jednotlivých variant daného proto regexu.

V těle metody můžeme použít `<sym>`, a získat tak název varianty.

**Příklad 2.11.**

Vytvořte gramatiku pro parsování řetězce s podporou escapování určených znaků. Příklad je založen na požadavcích jazyka TOML. [13]

Řešení s použitím proto-regexů:

```
1  grammar String {
```

```
2     token TOP { <string> }
3     token string { \" ~ \" [ <char> | \\ <escaped> ]+ }
4     token char { <-[ \x00..\x1F \x7F \\ \' \" ]> }
5     proto token escaped {*}
6      token escaped:sym<b>            { <.sym> }
7      token escaped:sym<t>            { <.sym> }
8      token escaped:sym<n>            { <.sym> }
9      token escaped:sym<f>            { <.sym> }
10     token escaped:sym<r>            { <.sym> }
11     token escaped:sym<quote>        { \" }
12     token escaped:sym<backslash>    { \\ }
13     token escaped:sym<u>            { u <[0..9A..F]>**4 }
14     token escaped:sym<U>            { U <[0..9A..F]>**8 }
15 }
16 my $text = '"Ahoj \"svět\ne!"';
17 say String.parse($text);
```

Metoda `string` musí přijmout alespoň jeden znak z metody `char` nebo `escaped`, který je ohraničen dvojitými uvozovkami. Na řádku 6 se nachází varianta metody `escaped` s názvem `b`. V jejím těle se očekává název varianty – symbol `b`. Na řádku 11 se nachází varianta metody `escaped` s názvem `quote`. V jejím těle se očekává dvojitá uvozovka.

Výstup:

```
1 「"Hoj\"svět\ne!"」
2  string => 「"Hoj\"svět\ne!"」
3   char => 「H」
4   char => 「o」
5   char => 「j」
6   escaped => 「"」
7   char => 「s」
8   char => 「v」
9   char => 「ě」
10  char => 「t」
11  escaped => 「n」
12  char => 「e」
13  char => 「!」
```

Další praktické příklady jsou v knihách [14; 15].

# 3 VCARD

Formát vCard (také známý pod zkratkou VCF) je standardní souborový formát elektronických vizitek, který se hojně využívá v business prostředí. Používá se ke sdílení informací formou kontaktů v emailové komunikaci, IM nebo pomocí QR kódů. Tyto informace mohou být textového charakteru, jako jsou jména, emaily, data, nebo to mohou být fotografie, loga, odkazy apod. [16]

Formát využívá kódování UTF-8 a jeho soubor má příponu `.vcard` nebo `.vcf`. Takový soubor musí obsahovat minimálně jedenu instanci objektu typu vCard. [16]

Příklad souboru:

```
1  BEGIN:VCARD
2  VERSION:4.0
3  KIND:group
4  FN:Jankovi
5  MEMBER:urn:uuid:01a7e90f-e1b8-b8a5-0458-etdd8d4a8b27
6  END:VCARD
7  BEGIN:VCARD
8  VERSION:4.0
9  FN:Martin Janek
10 UID:urn:uuid:01a7e90f-e1b8-b8a5-0458-etdd8d4a8b27
11 END:VCARD
```

Tento formát budeme parsovat pomocí gramatiky a regulárních výrazů v Perlu 6.

## 3.1 Struktura

Objekt typu vCard musí mít následující strukturu: [16]

```
"BEGIN:VCARD"
"VERSION:4.0"
ContentLine+
"END:VCARD"
```

Text v uvozovkách musí být doslovný, ale nezáleží na velikosti písmen. Každá nová položka musí být na novém řádku. Všechny tyto položky jsou povinné a musí být v uvedeném pořadí. Položka *ContentLine* musí být přítomna alespoň jednou.

❖ Objekt vCard je zpracován následovně:

```
1 token vcard {<begin> \n <version> \n <content-line>+ %% \n <endI>};
```

Token `<vcard>` odpovídá požadované struktuře objektu. Token `<content-line>` musí být oddělen znakem nového řádku, a proto je použita dvojice znaků `%%`.

Předchozí příklad souboru se parsuje následovně:

```
 2 vcard  BEGIN:VCARD
 3        VERSION:4.0
 4        KIND:group
 5        FN:Jankovi
 6        MEMBER:urn:uuid:01a7e90f-e1b8-b8a5-0458-etdd8d4a8b27
 7        END:VCARD
 8 ~
 9 vcard  BEGIN:VCARD
10        VERSION:4.0
11        FN:Martin Janek
12        UID:urn:uuid:01a7e90f-e1b8-b8a5-0458-etdd8d4a8b27
13        END:VCARD
```

❖

### 3.1.1 ContentLine

*ContentLine* může mít mnoho podob. Skládá se z těchto částí: [16]

- skupina – je nepovinná,
- název položky,
- parametr – jeden nebo více nepovinných parametrů oddělených středníkem (i od názvu položky),
- hodnota položky – jedna nebo více hodnot oddělených středníkem.

Například ContentLine

`G.N;ALTID=1:Martin`

obsahuje skupinu `G`, název položky `N`, parametr `ALTID=1` a hodnotu položky `Martin`.

Můžeme si povšimnout, že se parametr dále skládá z názvu a hodnoty parametru. Prozatím to ale zanedbejme.

Skupina je tvořena alespoň jedním alfanumerickým znakem nebo pomlčkou. Je nezávislá na velikosti písmen. Název skupiny se používá jako informace k seskupení položek, které spolu nějak souvisí. Aplikace je pak může příslušně zobrazit. [16]

❖ Každá *ContentLine* je parsována následovně:

```
14 token content-line {
       [<group> '.']?
       <property-name>
       [';' <parameter>]*
       ':' <property-value>+ % <[;]>
   };
```

Token <group> představující skupinu není povinný, ale vždy musí být následován tečkou. Parametry jsou nepovinné a může jich být libovolně mnoho. Alespoň jedna hodnota (<property-value>) je povinná. Odděluje se od zbytku řádku znakem dvojtečky.

Předchozí ukázka je zpracována takhle:

```
15 group              G
16 ~.
17 property-name      N
18 ~;
19 parameter          ALTID=1
20 ~:
21 property-value     Martin
```

❖

Pokud je na novém řádku na prvním místě bílý znak, je celý řádek považován za pokračování řádku předchozího. Tento znak i odřádkování se musí odstranit ze vstupu ještě před samotným parsováním. Vznikne tak jeden spojený řádek, který však musí splňovat pravidla *Contentline*. [16]

Řádek

```
G.N;ALTID=1:Martin
```

tedy můžeme zapsat i jako

```
G.N;ALT
ID=1:Martin
```

❖ Před zpracováním je toto odřádkování odstraněno následovně:

```
22 sub line-folding (Str $_) {
23     return $_.subst(/ \n [ ' ' | \t ] /, Q{}, :g);
24 };
```

Celý parsovaný text je argumentem funkce `line-folding()`. Tato funkce na něj spustí funkci `subst()`, která se v Perlu 6 využívá na náhradu vybraných částí textu. Tato funkce podporuje zápis nahrazované části regexem. Je proto vyhledán bílý znak (podle specifikace mezera nebo tabulátor), který bezprostředně následuje za znakem nového řádku. Každý nález je poté nahrazen prázdným řetězcem.

❖

#### *3.1.1.1 Název položky*

Hodnota položky závisí na jejím názvu. Ten je jednoznačně dán ve specifikaci. Jiná nebo vlastní jména položek nejsou podporována. Název položky dále určuje počet hodnot, typ povolených parametrů a hodnotu parametrů. [16]

❖ Názvy položek jsou pevně dány, ale nezáleží na velikosti písmen. Je proto vhodné je implementovat pomocí proto-regexů:

```
25 proto token property-name {*}
26 token property-name:sym<source>          { :i source }
27 token property-name:sym<kind>            { :i kind }
28 token property-name:sym<fn>              { :i fn }
29 token property-name:sym<n>               { :i n }
30 token property-name:sym<nickname>        { :i nickname }
                            ...
31 token property-name:sym<interest>        { :i interest }
32 token property-name:sym<org-directory>   { :i org'-'directory }
33 token property-name:sym<x-name>  { <[xX]> '-' <.alpha-num-dash>+ }
```

❖

Speciální položku tvoří ty, jejichž název začíná na `x-` (řádek 33). Jsou to položky vyhrazené pro soukromé experimentální účely. Nemusí se parsovat, protože mají rozdílnou implementaci. Standard je pouze povoluje používat. Jejich obsah závisí na uživateli.

#### *3.1.1.2 Hodnota položky*

Typ hodnoty je například text, datum, pole čísel nebo URI. Tyto hodnoty se řídí vlastní specifikací a mají tak pevně určený formát.

Hodnota položky může být i pole. Jednotlivé prvky pole mohou být i prázdné. Všechny se musí oddělovat středníkem. [16]

```
N;ALTID=1;LANGUAGE=cs:Janek;Martin;;
```

Prvkem pole může být další pole, jehož prvky se oddělují čárkami. [16]

```
N:Jiří,Jakub;Toman;;
```

Z těchto důvodu se musí před všechny čárky a středníky v hodnotách psát zpětné lomítko, stejně jako před znak nového řádku. [16]

```
ADR:;;31\,Plt St.;Bay\;twn;LA;30;Uni\\nted States of America
```

❖ Implementace hodnoty položky vypadá následovně:

```
34 token property-value  { <property-simple-value>+ % ',' };
35     token property-simple-value  { [ \\ . | <-[\n;,]> ]* };
```

Jedna `<property-value>` odpovídá jedné hodnotě položky, v které může být pole oddělené čárkami. Pokud v ní pole není (pouze jedna hodnota), nesmí být na konci hodnoty čárka. Proto používáme pro oddělení pole pouze jeden znak `%`.

Prvek pole poté může být prázdný, nebo může obsahovat sekvenci povolených znaků. Část `\\ .` zase umožní psát zmíněné znaky pouze za zpětné lomítko.

❖

### 3.1.2 Parametr

Parametr je závislý na položce. Představuje dodatečnou informaci k její hodnotě, jako je jazyk, preference a třeba řazení. [16]

Skládá se z názvu a hodnoty. Jsou odděleny znakem =.

Názvy parametrů jsou, podobně jako názvy položek, pevně dány.

#### *3.1.2.1 Hodnota parametru*

Parametry mohou mít více hodnot. Oddělují se čárkou.

```
…;TYPE=work,voice;VALUE=uri:…
```

Hodnoty se mohou se zapisovat napřímo, nebo do dvojitých uvozovek, pokud obsahují znaky jako středník, čárka, nový řádek apod.

```
…;LABEL="42 Plantation St.\nBaytown\,":…
```

❖ Řešení logiky parametru:

```
36 token parameter  { <parameter-name> '=' <parameter-value>+ % ',' };
```

```
37  token parameter-value  { <q-safe-char> | <safe-char> };
38      token q-safe-char      { [<["]> <[] ~ []> <["]>] <-["]>+ };
39      token safe-char        { <-:C-[:;,"]>+ };
```

Na řádku 36 vidíme, že parametr se skládá z jednoho jména a alespoň jedné hodnoty parametru. Řádek 38 umožňuje hodnotě parametru být ve dvou formách. Token na řádku 39 odpovídá hodnotě v uvozovkách. Může obsahovat cokoli, kromě znaku uvozovky. Uvozovky nejsou součástí `<parameter-value>`. Token `<safe-char>` nepovoluje kontrolní znaky a znaky `:;,"`.

Ukázka parsování parametru na *ContentLine*:

```
TEL;TYPE=work,voice;VALUE=uri:tel:+1-111-555-1212
```

```
40 parameter-name     TYPE
41 ~=
42 parameter-value    work
43 ~,
44 parameter-value    voice
45 parameter-name     VALUE
46 ~=
47 parameter-value    uri
```

❖

## 3.2 jCard

Aby byl formát vCard podporován v různých zařízeních a webových aplikacích, vznikly jeho reprezentace v běžných formátech jako XML nebo JSON.

Jedním z nich je jCard. Jedná se o standardní formát, který reprezentuje data z formátu vCard ve formátu JSON.

JSON používá jinou specifikaci pro formátování hodnot např. času a dat. Pro správný převod by proto měly být tyto hodnoty přeformátovány. [17]

Jedna instance jCard objektu může vypadat následovně: [17]

```
1 ["vcard",
2      [
3        ["version", {}, "text", "4.0"],
4        ["fn", {}, "text", "Martin Janek"],
5        ["lang", {"pref": "1"}, "language-tag", "cs"],
```

```
6       ]
7 ]
```

Stejná instance ve formě vCard: [16]

```
8 BEGIN:VCARD
9 VERSION:4.0
10 FN:Martin Janek
11 LANG;PREF=1:cs
12 END:VCARD
```

Taková struktura musí být vždy dodržena. Verze a řetězec *vcard* jsou povinné. [17]

Můžeme vidět, že pro ohraničení objektů se již nepoužívají *BEGIN* a *END*. Místo nich se do souboru přidává další pole, v němž jsou jednotlivé jCard objekty.

Po parsování objektu vCard můžeme získanou strukturu upravit a zformátovat. Získáme tak odlišný formát, obsahující stejné informace. Budeme takto vytvářet objekt typu jCard.

❖ Nabízí se akce, které umožňují sestavit jCard od nejmenších částí z parsování [12]:

```
13 method TOP ($/) { make $.made-value($<vcard>) }
14 method vcard ($/) {
15     make ["vcard",
16       [
17         ["version", %(), "text", "4.0"],
18         |$<content-line>».made;
19       ]
20     ]
21 };
```

Metoda *TOP* zastřešuje soubor, ve kterém může být více instancí vCard. Tuto problematiku řeší speciální metoda `made-value()`, která bude probrána později. Na řádku 15 začíná metoda `vcard`. Obsahuje stejnou strukturu jako výše zmíněná instance objektu. Řádek 17 obsahuje pole, jehož druhý prvek má být asociativní pole. Pomocí `%()` se na daném místě vytvoří prázdné asociativní pole.

❖

### 3.2.1 ContentLine

Každá položka je ve zvláštním čtyřmístném poli. Pozice přestavují: [17]

- Název položky – musí být převeden na malá písmena.

- Parametr – objekt, který obsahuje všechny parametry.
- Typ hodnoty položky
- Hodnota položky

❖ Implementace ContentLine:

```
22 method content-line ($/) {
                ...
23 make [
24          $<property-name>.lc,
25          %parameter,
26          $parameter-of-type-value,
27          $.made-value($<property-value>)
28      ]
29 };
```

`Make` vytvoří čtyřmístné pole. První je název položky, který se pouze převádí na malá písmena. Na řádku 25 je zmiňovaný objekt s parametry.

❖

Když je hodnota pouze jedna, je zapsána jako řetězec. [17]

```
["fn", {}, "text", "Martin Janek"]

FN:Martin Janek
```

Pokud je hodnot víc, je na čtvrtou pozici vloženo pole s danými hodnotami. [17]

```
["adr", {}, "text", ["Město", "Ulice 5"]]

ADR:Město;Ulice 5
```

Stejně jako u formátu vCard může každá hodnota mít několik dílčích prvků. Všechny zapisujeme do dalšího pole. [17]

```
["adr", {}, "text", ["Město", ["Ulice 5", "Ulice 4"]]]

ADR:Město;Ulice 5, Ulice 4
```

❖ Pomocí metody `made-value()` předcházíme vzniku jednoprvkového vnitřního pole:

```
30 method made-value ($_) {
31          when $_.elems > 1 {$_».made}
32          default {$_[0].made}
33 };
```

❖

Z hodnoty položky musí být odstraněny všechny znaky zpětného lomítka, které sloužily k zápisu nepovolených znaků. [17]

Typ hodnoty položky má výchozí nastavení podle názvu položky.

❖ Pro snadný přístup si vytvoříme asociativní pole s výchozími hodnotami:

```
34 my %default-type-of-value = %(
35     source => 'uri',
                  ...
36     deathdate => 'date-and-or-time',
37     lang => 'language-tag',
38     rev => 'timestamp',
39 );
```

❖

Tento typ můžeme změnit pomocí parametru *value* v původním vCard objektu. Tento parametr ale nesmí být přidán do objektu s parametry, protože patří na třetí pozici pole položky. Pokud neexistuje, zvolíme výchozí typ hodnoty. Pokud parametr nemá výchozí hodnotu, bude typ *unknown*. Specifikace dále uvádí, že všechny experimentální položky mají být typu *unknown*. [17]

❖ Toho docílíme vytvořením proměnné, jejíž obsah se vloží na třetí pozici *ContentLine*:

```
40 my $parameter-of-type-value = %parameter<value>[0] //
       %default-type-of-value{$<property-name>.lc} //
       'unknown';
41 %parameter<value>:delete;
```

❖

### 3.2.2 Parametr

Parametry jsou reprezentovány jako asociativní pole.

❖ Vytvoříme proto proměnnou:

```
42 my %parameter = %($<parameter>».made);
```

❖

Pokud má jeden parametr více hodnot, musí se hodnoty oddělené čárkami připsat do společného pole následujícím způsobem.

```
…, {"type": ["work", "voice"]},…
```

❖ Tento problém opět dokáže vyřešit metoda `made-value()`:

```
43 method parameter ($/) {
44     make $<parameter-name>.lc => $.made-value($<parameter-value>);
45 };
46 method parameter-value ($/) {
47     with $<q-safe-char> {make $_.made}
48     orwith $<safe-char> {make $_.Str}
49 };
50 method q-safe-char ($/) {
51  make $/.subst(/ \\ )> <[,\;]> /, Q{}, :g ).subst(/ \\n /, "\n", :g)
52 };
```

Metoda `parameter` vrací dvojici *name* => *value*. Název parametru musí být převeden na malá písmena.

Gramatika obsahuje dvě varianty zápisu hodnot parametrů. Stejně musíme mít i akčních metod: Pro metodu na řádku 48 stačí uložit řetězec tak, jak je. Z parametrů ve dvojitých uvozovkách musíme odstranit nepovolené znaky, před kterými bylo zpětné lomítko.

❖

Ve formátu jCard se musí skupina uvádět pouze pomocí *group* parametru uvnitř objektu s parametry.

### 3.2.3 Sestavení

❖ Po dokončení parsování formátu vCard, provádíme sestavení jCard formátu pomocí akcí. Výsledek celého procesu je dostupný pomocí funkce `from-vCard()`:

```
53 sub from-vCard ($_) is export {
54     my $preprocessed-vcard = line-folding($_);
55     sub parser (|c) {vCard::Parser::Grammar.parse(|c)};
56     my $vcard = parser($preprocessed-vcard, actions =>
   vCard::Parser::Actions.new);
57     with $vcard {
58         return $_.made
59     };
60 };
```

Mějme následující vCard:

```
61 my $vcf = 'BEGIN:VCARD
62 VERSION:4.0
63 N:Gump;Forrest;;Mr.;
64 x-qq:21588891
65 END:VCARD';
```

Zavoláme funkci `from-vCard();`

```
66 say from-vCard($vcf);
67 ["vcard",
68      [
69        ["version", {}, "text", "4.0"],
70        ["n", {}, "text", ["Gump", "Forrest", "", "Mr.", ""]],
71        ["x-qq", {}, "unknown", "21588891"],
72      ]
73 ]
```

❖

Předchozí funkce vrací strukturální reprezentaci formátu jCard. Nejedná se tedy ještě o formát jCard, ale o objekt Perlu 6, který má shodnou strukturu. Objekt jCard lze ale snadno vytvořit pomocí externích knihoven pro převod datové struktury na JSON.

Celý kód je hostován na GitHubu: https://github.com/petrkol72/vCard-Parser ve formě knihovny. Je zařazena mezi moduly Perlu 6: https://modules.perl6.org a lze ji nainstalovat standardním způsobem pro Perl 6, a to příkazem:

```
zef install vCard::Parser
```

# ZÁVĚR

V teoretické části bakalářské práce byly popsány formální jazyky a gramatiky včetně jejich klasifikace. Práce se dále zaměřovala na bezkontextové jazyky a především regulární jazyky. Byly definovány konečné a zásobníkové automaty a regulární výrazy. Je rovněž uvedeno, že regulární výrazy, regulární gramatiky a konečné automaty jdou mezi sebou převádět a popisují, generují nebo přijímají stejný regulární jazyk.

V praktické části je popisována základní syntaxe, která je nutná pro realizaci modulu vCard, a postup zpracování výrazu, který se standardně řídí pravidly hladových kvantifikátorů a nejlevější shody.

Regulární výrazy v Perlu 6 se v mnohém liší od těch teoretických. V praxi se využívají především na vyhledávání řetězců v textu. Obsahují užitečná rozšíření, jako jsou kvantifikátory, zástupné znakové třídy, zachytávání proměnných, kotvy aj. Dále pak implementují mechanizmus zvaný backtracking. Ten narušuje teoretický základ regulárních výrazů, ale umožňuje programátorům zapsat jednodušší regulární výraz. Backtrackingu je vhodné se vyhýbat, pokud to jde, protože je časově náročnější.

Gramatiky ve formě tříd se používají pro zastřešení dílčích regexů a jsou vhodé pro parsování. Pro následnou interpretaci získaných dat využíváme akce.

Výsledkem této bakalářské práce je také knihovna s názvem *vCard::Parser*, který je zařazen mezi moduly Perlu 6. Tento modul je první funkční parser formátu vCard napsaný v jazyce Perl 6. Umožňuje také z výsledku parsování vytvořit nový objekt, který má shodnou strukturu s formátem jCard. Daný objekt se poté dá přeformátovat pomocí knihoven pro vytváření JSON objektů. K modulu jsou také připojeny vlastní automatické testy.

Modul z části nedodržuje specifikaci v místech, kde by byla implementace příliš složitá, či omezující. Jelikož se nejedná o standard, tak podobné úpravy nejsou výjimkou ani u profesionálních aplikací. Největším nedostatkem modulu je absence přeformátování dat a času při vytváření objektu. Bylo by nutné nastudovat několik dalších specifikací k formátování jednotlivých částí objektu JSON.

# SEZNAM POUŽITÉ LITERATURY

[1] MARTÍNEK, Pavel. *Základy teoretické informatiky* [online]. Olomouc: Univerzita Palackého v Olomouci, 2006 [cit. 2019-05-14]. Dostupné z: https://phoenix.inf.upol.cz/esf/ucebni/zti.pdf

[2] ČEŠKA, Milan. *Teoretická informatika* [online]. Brno: Vysoké učení technické v Brně, 2002 [cit. 2019-05-14]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/ti.pdf

[3] ŠESTÁKOVÁ, Eliška. *Automaty a gramatiky: sbírka řešených příkladů*. Praha: České vysoké učení technické v Praze, 2017. ISBN 978-800-1063-064.

[4] HABIBALLA, Hashim. *Regulární a bezkontextové jazyky I*. [online]. Ostrava: Ostravská univerzita, 2003 [cit. 2019-05-14]. Dostupné z: http://www1.osu.cz/home/habibal/dizertace/texty/xrab1.pdf

[5] ROSENFELD, Laurent a Allen DOWNEY. *Think Perl 6* [online]. Needham (Massachusetts): Green Tea Press, 2017 [cit. 2019-05-14]. Dostupné z: http://greenteapress.com/thinkperl6/thinkperl6.pdf

[6] Perl 6 Language Documentation. Perl 6 Documentation [online]. [cit. 2019-05-14]. Dostupné z: https://docs.perl6.org/language.html

[7] FRIEDL, Jeffrey E. F. *Mastering regular expressions*. 3rd ed. Farnham: O'Reilly, c2006. ISBN 978-0-596-52812-6.

[8] Regexes. *Perl 6 Documentation* [online]. [cit. 2019-05-14]. Dostupné z: https://docs.perl6.org/language/regexes

[9] NAGY, Zsolt. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. Berlin: Apress, c2018. ISBN 978-1-4842-3876-9.

[10] LENZ, Moritz. *Parsing with Perl 6 Regexes and Grammars*. Fürth: Apress, c2017. ISBN 978-1-4842-3228-6.

[11] Grammars. *Perl 6 Documentation* [online]. [cit. 2019-05-14]. Dostupné z: https://docs.perl6.org/language/grammars

[12] Grammar tutorial. *Perl 6 Documentation* [online]. [cit. 2019-05-14]. Dostupné z: https://docs.perl6.org/language/grammar_tutorial

[13] PRESTON-WERNER, Tom. TOML. *GitHub* [online]. San Francisco, 2019 [cit. 2019-05-15]. Dostupné z: https://github.com/toml-lang/toml

[14] GOYVAERTS, Jan a Steven LEVITHAN. *Regulární výrazy: Kuchařka programátora*. Brno: Computer Press, c2010. ISBN 978-80-251-1935-8.

[15] STUBBLEBINE, Tony. *Regular expression pocket reference*. 2nd ed. Sebastopol: O'Reilly, c2007. ISBN 978-0-596-51427-3.

[16] PERREAULT, Simon. *vCard Format Specification*. IETF, 2011. Dostupné z: https://tools.ietf.org/html/rfc6350

[17] KEWISCH, Philipp. *JCard: The JSON Format for vCard*. IETF, 2014. Dostupné z: https://tools.ietf.org/html/rfc7095

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ASCII American Standard Code for Information Interchange.

DKA Deterministický konečný automat.

IM Instant Messaging.

JSON JavaScript Object Notation

NKA Nedeterministický konečný automat.

QR Quick Response.

TOML Tom’s Obvious, Minimal Language.

URL Uniform Resource Identifier.

UTF Unicode Transformation Format.

VCF Virtual Contact File.

XML Extensible Markup Language

ZKA Zásobníkový automat.

# SEZNAM OBRÁZKŮ

# SEZNAM TABULEK

# SEZNAM PŘÍLOH

# PŘÍLOHA P I: NÁZEV PŘÍLOHY

Obsah CD:

- Bakalářská práce v elektronické podobě
- Zdrojové kódy modulu vCard::Parser