

UNIVERZITA KOMENSKÉHO, BRATISLAVA

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SILA A ZLOŽITOSŤ MODERNÝCH REGULÁRNYCH  
VÝRAZOV

Diplomová práca

2015

Tatiana Tóthová

UNIVERZITA KOMENSKÉHO, BRATISLAVA

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

# SILA A ZLOŽITOSŤ MODERNÝCH REGULÁRNYCH VÝRAZOV

Diplomová práca

Študijný program: Informatika  
Študijný odbor: .... Informatika  
Školiace pracovisko: Katedra Informatiky  
Školiteľ: RNDr. Michal Forišek, PhD.

Bratislava, 2015

Tatiana Tóthová



Univerzita Komenského v Bratislave  
Fakulta matematiky, fyziky a informatiky

---

## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Tatiana Tóthová  
**Študijný program:** informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)  
**Študijný odbor:** 9.2.1. informatika  
**Typ záverečnej práce:** bakalárska  
**Jazyk záverečnej práce:** slovenský

**Názov:** Moderné regulárne výrazy

**Cieľ:** Spraviť prehľad nových konštrukcií používaných v moderných knižniciach s regulárnymi výrazmi (ako napr. look-ahead a look-behind assertions). Analyzovať tieto rozšírenia z hľadiska formálnych jazykov a prípadne tiež z hľadiska algoritmickej výpočtovej zložitosti.

**Vedúci:** RNDr. Michal Forišek, PhD.  
**Katedra:** FMFI.KI - Katedra informatiky  
**Vedúci katedry:** doc. RNDr. Daniel Olejár, PhD.  
**Dátum zadania:** 23.10.2012

**Dátum schválenia:** 24.10.2012

doc. RNDr. Daniel Olejár, PhD.  
garant študijného programu

*.....tu bude podakovanie.....*

*Tatiana Tóthová*

## Abstrakt

Tu bude abstrakt po slovensky.

**Kľúčové slová:** nejaké, kľúčové, slová

## Abstract

Here will be abstract in english...

**Key words:** some, key, words

# Obsah

<b>Úvod</b>	<b>1</b>
<b>Použité pojmy a skratky</b>	<b>2</b>
<b>1 Súčasný stav problematiky</b>	<b>3</b>
1.1 Základné definície . . . . .	3
1.1.1 Nové konštrukcie . . . . .	4
1.2 Vlastnosti a sila . . . . .	8
1.3 Popisná zložitosť . . . . .	10
<b>2 Naše výsledky</b>	<b>14</b>
2.1 Sila negatívneho lookaroundu . . . . .	14
2.2 Vlastnosti triedy $\mathcal{L}_{LERE}$ . . . . .	19
2.3 Chomského hierarchia . . . . .	20
2.4 Priestorová zložitosť . . . . .	22
2.5 Popisná zložitosť . . . . .	39
<b>Záver</b>	<b>40</b>
<b>Literatúra</b>	<b>41</b>

# Úvod

Pár slov na úvod...



# Použité pojmy a skratky

$\varepsilon$  – prázdne slovo

$L_1L_2$  – zretazenie jazykov  $L_1$  a  $L_2$

$L^*$  – Kleeneho iterácia ( $L^* = \cup_{i=0}^{\infty} L^i$ , kde  $L^0 = \{\varepsilon\}$ ,  $L^1 = L$  a  $L^{i+1} = L^iL$ )

$\mathcal{R}$  – trieda regulárnych jazykov

$\mathcal{L}_{CF}$  – trieda bezkontextových jazykov

$\mathcal{L}_{CS}$  – trieda kontextových jazykov

**DKA/NKA** – deterministický/nedeterministický konečný automat

**LBA** – lineárne ohraničený Turingov stroj

**TS** – Turingov stroj

**matchovať** – keď regex  $\alpha$  matchuje slovo  $w$  (vyhlási zhodu), znamená to, že  $w \in L(\alpha)$

**lookaround** – spoločný názov pre lookahead a lookbehind

*Regex* – regexy nad množinou operácií, ktorými vieme popísať iba regulárne jazyky  
(základná definícia)

*Eregex* – *Regex* so spätnými referenciami

*LRegex* – *Eregex* s operáciami lookahead a lookbehind

*nLRegex* – *LRegex* s operáciami negatívny lookahead a negatívny lookbehind

$\mathcal{L}_{RE}$  – trieda jazykov nad *Regex*, ekvivalentná  $\mathcal{R}$

$\mathcal{L}_{ERE}$  – trieda jazykov nad *Eregex*

$\mathcal{L}_{LERE}$  – trieda jazykov nad *LRegex*

$\mathcal{L}_{nLERE}$  – trieda jazykov nad *nLRegex*

# 1 Úvod do súčasného stavu problematiky

Myšlienka regulárnych výrazov bola prvýkrát spomenutá v teórii formálnych jazykov a automatov ako iný spôsob popisu regulárnych jazykov. Vtedy pozostávali z operácií zjednotenia, zretazenia a Kleeného uzáveru. Z takéhoto zápisu bolo ľahšie vidieť, o aký jazyk ide, než z konečných automatov alebo regulárnych gramatík. Vďaka jednoduchosti boli implementované ako nástroj na vyhľadávanie slov zo špecifikovaného jazyka. Postupom času a s inšpiráciou zo strany užívateľov k nim pribúdali ďalšie operácie. Niektoré boli len skratkou k tomu, čo sa už dalo zapísať – umožnili zapísať to isté, ale menej znakmi – ostatné otvárali dvere k popisu dovtedy nedosiahnuteľných jazykov.

Zmes týchto operácií nazývame moderné regulárne výrazy alebo regexy. Tento model opäť vraciame do teórie jazykov a skúmame jeho vlastnosti, zaradenie v Chomského hierarchii a zložitosť.

## 1.1 Základné definície

Každý regulárny výraz sa skladá zo znakov a metaznakov. Znak je regulárny výraz reprezentujúci sám seba, teda  $L(a) = \{a\}$ . Metaznaky popisujú operáciu nad regulárnymi výrazmi. Ak potrebujeme použiť metaznak ako znak, stačí pred neho dať  $\backslash$ , teda  $L(\backslash x) = \{x\}$ . Ak metaznak vyžaduje vstup, je ním posledný znak/metaznak/uzávierkový výraz naľavo od neho.

Nech  $\alpha, \beta$  sú regexy. Základné operácie sú:

- zretazenie  $(\alpha)(\beta)$
- alternácia  $(\alpha)|(\beta)$

- Kleeneho uzáver  $(\alpha)^*$

Platí  $L((\alpha)(\beta)) = L(\alpha)L(\beta)$ ,  $L((\alpha)|(\beta)) = L(\alpha) \cup L(\beta)$  a  $L((\alpha)^*) = L(\alpha)^*$ .

Aby sme špecifikovali pole pôsobnosti operácií, používame v regulárnych výrazoch okrúhle zátvorky  $()$ . Je možné ich vynechať, potom sa operácie budú vyhodnocovať podľa priorít – tie uvedieme po definovaní všetkých operácií.

Regulárny výraz je vykonávaný zľava doprava. Keď sa dostaneme na koniec výrazu aj slova (za posledný znak), hovoríme, že výraz *matchuje*<sup>1</sup> (z angl. *match* – zhoda) slovo, výraz vyhovuje slovu, alebo slovo vyhovuje výrazu.

Jazyk vyhovujúci regexu  $\alpha$  je množina slov matchovaných týmto regexom. Hovoríme aj, že regex generuje tento jazyk.

### 1.1.1 Nové konštrukcie

Najprv si uvedieme konštrukcie, ktoré sú len „kozmetickou úpravou základných regulárnych výrazov. Všetky sa dajú zapísať pomocou základných operácií, ale nový zápis je stručnejší a prehľadnejší. Pre formálnejšiu definíciu odporúčame siahnuť po [Tó13].

- $+$  – opakuj 1 alebo viackrát:  $(\alpha)^+ = \alpha(\alpha)^*$
- $\{ \}$  – zložené zátvorky sú používané ako  $\{n, m\}$  (opakuj aspoň  $n$  a najviac  $m$ -krát) a  $\{n\} = \{n, n\}$  (opakuj  $n$ -krát)
- $[ ]$  – predstavuje ľubovoľný znak, z tých, ktoré má vnútri zapísané. Vieme použiť aj intervaly, napr.  $a-z$ ,  $A-Z$ ,  $0-9$ ,  $\dots$  a kombinovať ich. Všetky metaznaky vnútri  $[ ]$  sa považujú za normálne znaky.
- $[^ ]$  – predstavuje ľubovoľný znak, ktorý nepatrí medzi zapísané. Ostatné ako  $[ ]$ .
- $?$  – ak samostatne: opakuj 0 alebo 1-krát  
ak za operáciou: namiesto greedy implementácie použi minimalistickú, t.j. zober čo najmenej znakov (platí pre  $*$ ,  $+$ ,  $?$ ,  $\{n, m\}$ )<sup>2</sup>
- $.$  – predstavuje ľubovoľný znak
- $^$  – metaznak označujúci začiatok slova

<sup>1</sup>Preklad „regex sa zhoduje so slovom“ nevyjadruje presne to, čo je vyjadrené v angličtine. Znie to, akoby regex bol to isté, čo slovo. Preto sme sa rozhodli zostať pri slove *match* a slovese *matchovať*.

<sup>2</sup>Všetky spomenuté operácie „žerú“ znaky a na ich implementáciu bol použitý greedy algoritmus.

- \$ – metaznak označujúci koniec slova

Okrem operácií označených metaznakmi vznikli aj zložitejšie operácie, na popis ktorých treba dlhšie konštrukcie. V prvom rade sa zaviedlo **číslovanie okrúhlych zátvoriek**. Čísľuje sa zľava doprava podľa poradia ľavej (otváracej) zátvorky, ale konštrukcie tvaru  $(?..)$  sa vynechávajú. Toto číslovanie sa deje automaticky pri každom matchovaní, teda už pri písaní výrazu ho môžeme využívať. Popíšme si teraz zložitejšie operácie:

- **komentár** ozn.  $(? \# \text{text komentáru})$  – klasický komentár, určený čitateľom kódu; nemá vplyv na výraz, pri matchovaní sa ignoruje
- **spätné referencie** ozn.  $\backslash k$ ,  $k \in \mathbb{N}$  – môže sa nachádzať na ľubovoľnom mieste vo výraze ZA  $k$ -tou pravou (zatváracou) zátvorkou. Odkazuje sa na  $k$ -te zátvorky a presný význam sa určuje až pri hľadaní zhody na konkrétnom vstupe. Algoritmus si zapamätá aké podslovo zo vstupu matchoval výraz vnútri  $k$ -tych zátvoriek a presne toto podslovo matchuje konštrukcia  $\backslash k$ , keď ďalej vo výraze narazí na  $\backslash k$ . Počas matchovania môžu  $k$ -te zátvorky matchovať viackrát. V momente, keď ideme vykonávať  $\backslash k$  si vyberáme doteraz posledné z nich.

Ukážeme si to na príklade, ako regex  $\alpha(\beta)\gamma\backslash k\delta$  matchuje slovo  $w$ :

$$w = \underbrace{x_1 \dots x_{i-1}}_{\alpha} \underbrace{\overbrace{x_i \dots x_{j-1}}^{w_k}}_{\substack{(\beta) \\ k \quad k}} \underbrace{x_j \dots x_{l-1}}_{\gamma} \underbrace{\overbrace{x_l \dots x_{m-1}}^{w_k}}_{\backslash k} \underbrace{x_m \dots x_n}_{\delta}$$

musí platiť  $w_k = x_i \dots x_{j-1} = x_l \dots x_{m-1}$  a zároveň  $w_k$  je posledné matchované podslovo  $k$ -tych zátvoriek v momente, keď sme narazili na konštrukciu  $\backslash k$ .

Predpokladáme, že číslo  $k$  je zapísané znakmi z inej abecedy ako zvyšok regexu – t.j. je jednoznačne určiteľné, kde končí zápis  $k$ .<sup>3</sup>

- **lookahead** – nazeranie dopredu.

<sup>3</sup>V implementovaných regexoch vieme deterministicky určiť, kde končí číslo  $k$  – sú povolené maximálne trojciferné čísla. My chceme všeobecnejší model, bez obmedzení na veľkosť  $k$ , a práve preto je tento predpoklad oprávnený. Zároveň si to môžeme dovoliť, lebo ak máme  $k$  zapísané ako číslo v desiatkovej sústave, v teórii je jednoduché zasubstituovať hľadaný jazyk tak, aby neobsahoval znaky  $0, \dots, 9$ . Viac informácií o obmedzeniach regexov v praxi nájdete v [Tó13, kapitola Prax].

- **pozitívny** ozn.  $(?= \alpha)$ , kde  $\alpha$  je nejaký moderný regulárny výraz

V momente, keď na lookahead narazíme, zapamätáme si aktuálnu pozíciu v slove. Od tejto pozície začneme hľadať zhodu s výrazom  $\alpha$ . Akonáhle vyhlási zhodu (nemusí ani dočítať slovo do konca), vrátime sa naspäť na zapamätané miesto a odtiaľ pokračujeme v slove vykonávaním regexu za lookaheadom – ako keby tam lookahead nikdy nebol. Inak povedané lookahead nevyžiera písmenká a robí akýsi prienik. Ak nenájde zhodu na žiadnom prefixe od zapamätaného miesta, znamená to nezhodu pre celý regex – ak sa dá, musíme sa vo výpočte vrátiť naspäť a vyskúšať inú možnosť predošlých operácií.<sup>4</sup> Pokiaľ chceme, aby lookahead slovo dočítal do konca, použijeme metaznak \$.

Ukážme si priebeh matchovania regexom  $\alpha(=?\beta)\gamma$  na slove  $w$ . Vidíme, že  $\beta$  a  $\gamma$  začínajú na rovnakom mieste v slove.

$$w = \underbrace{x_1 \dots x_{i-1}}_{\alpha} \overbrace{x_i \dots x_j}^{\beta} \underbrace{x_{j+1} \dots x_n}_{\gamma}$$

- **negatívny** ozn.  $(?! \alpha)$ , kde  $\alpha$  je nejaký moderný regulárny výraz

Nesmie nájsť slovo z jazyka  $L(\alpha)$ . Postup je ako pri lookaheade, ale končí úspešne len ak by lookahead zamietol.

- **lookbehind** – nazeranie dozadu.

- **pozitívny** ozn.  $(?<= \alpha)$ , kde  $\alpha$  je nejaký moderný regulárny výraz

Hľadá slovo z  $L(\alpha)$  naľavo od aktuálneho miesta v slove (musí končiť na susediacom políčku vľavo od aktuálnej pozície v slove). Opäť nie je určená hranica slova, môže začínať kdekoľvek, ak nie je vynútený začiatok slova znakom ^.

Ak by sme chceli deterministický algoritmus, vyzeral by nasledovne: od teraz do konca vykonávania lookbehindu bude na pozícii, na ktorej v slove sme, špeciálny znak pre koniec slova – endmarker (nie \$). Najprv vyskúšame 1 susedný symbol naľavo, či patrí do jazyka. Ak nie skúsime čítať o jeden znak viac (2 symboly naľavo), keď neuspejeme, opäť posunieme pomyselný

<sup>4</sup>V praxi je lookahead atomická operácia, ale my opäť chceme model v plnej sile.

začiatok slova doľava. Ak akceptujeme, môže to byť len na endmarkeri. Ak sme neakceptovali a začiatok slova nejde viac posunúť, zamietneme.

Opäť to bude dobre vidieť na príklade, výpočet regexu  $\alpha(?<=\beta)\gamma$  na slove  $w$ . Podslová  $\alpha, \beta$  končia na tej istej pozícii.

$$w = \underbrace{x_1 \dots x_{i-1}}_{\alpha} \overbrace{x_i \dots x_j}^{\beta} \underbrace{x_{j+1} \dots x_n}_{\gamma}$$

– **negatívny** ozn.  $(?! \alpha)$ , kde  $\alpha$  je nejaký moderný regulárny výraz

Hľadá ako pozitívny lookbehind, ale akceptuje len ak neexistuje slovo z jazyka  $L(\alpha)$  vyskytujúce sa naľavo od aktuálnej pozície.

Pre posledné 2 operácie zostaneme pri ich anglických názvoch, pretože sú stručnejšie a zvučnejšie. Existuje pre ne aj spoločný názov lookaround, takisto s prívlastkom pozitívny/negatívny, ak myslíme iba ich pozitívne/negatívne verzie. Bude našim zvykom ponechávať pozitívny lookahead/lookahead/lookbehind bez prívlastku. Pokiaľ budeme myslieť negatívnu verziu alebo to nebude z kontextu jasné, explicitne to napíšeme.

Formálnejšia definícia zložitejších konštrukcií sa nachádza v [Tó13].

Moderné regulárne výrazy sa skladajú z **konečného počtu** znakov, metaznakov a zložitejších operácií.

## Priortity operácií

Ako sme už spomínali, pre korektné matchovanie regexom potrebujeme vedieť priority operácií. Zároveň nám to umožní niekedy vynechať zátvorky a regex tak bude prehľadnejší.

Nové konštrukcie lookahead a lookbehind neberú zo slova žiadne písmenká. Môžeme si ich predstaviť ako overenie nejakej podmienky, je to taká odbočka od výpočtu. Preto nemá zmysel na ne aplikovať operácie ako  $*$ ,  $+$ ,  $?$ ,  $\{ \}$ ,  $|$  a teda taký regex nebude validný. Je možné ich uzavrieť do ďalších lookaroundov, je možné sa odkazovať na zátvorky, ktoré majú vnútri (iba ak sa jedná o pozitívny lookahead, odkazovať sa dovnútra negatívneho lookaroundsu nemá zmysel). Pokiaľ však aplikujeme operáciu na nejaký

regex obsahujúci lookahead, prirodzene je lookahead vykonaný zakaždým, keď naň príde rad.

Niektoré operácie sa správajú ako znak  $- [ ]$ ,  $[^ ]$ ,  $.$ ,  $^$ ,  $\$$ ,  $\backslash k$  – teda je s nimi možné narábať ako s obyčajným znakom (okrem  $^$  a  $\$$ , tieto nemá zmysel iterovať (napr.  $\$*$ ) – existuje vždy práve 1 začiatok a 1 koniec slova). Ostatné kombinujú znaky s nasledovnými prioritami:

priorita	najvyššia	<	<	najnižšia
operácia	()	* + ? {}	zretazenie	

Máme veľkú množinu operácií a budeme chcieť rozlišovať regexy používajúce rôzne operácie. Preto si zavedieme triedy regexov nad konkrétnymi množinami operácií. Regex z takejto triedy bude obsahovať iba operácie z povolenej množiny, v ľubovoľnom počte (stále však platí, že regex je konečnej dĺžky).

Zároveň chceme pomenovať triedy jazykov nad jednotlivými triedami regexov.

*Regex* – trieda regexov nad množinou operácií, pomocou ktorých vieme popísať iba regulárne jazyky. Množina operácií obsahuje všetky znaky a metaznaky bez zložitejších operácií.

*Eregex* – *Regex* so spätnými referenciami

*LEregex* – *Eregex* s pozitívnym lookaheadom

*nLEregex* – *LEregex* s negatívnym lookaheadom

$\mathcal{L}_{RE}$  – trieda jazykov nad *Regex* ( $= \mathcal{R}$ )

$\mathcal{L}_{ERE}$  – trieda jazykov nad *Eregex*

$\mathcal{L}_{LERE}$  – trieda jazykov nad *LEregex*

$\mathcal{L}_{nLERE}$  – trieda jazykov nad *nLEregex*

## 1.2 Vlastnosti a sila moderných regulárnych výrazov

Keď sme hľadali články o moderných regulárnych výrazoch, našli sme výsledky prevažne o regulárnych výrazoch schopných generovať iba regulárne jazyky a preštudovaná bola trieda *Eregex* [CSY03], [CN09] (v článkoch pomenovaná EREG). Tému

moderných regulárnych výrazov sme rozvinuli v bakalárskej práci [Tó13], v tejto práci naviažeme na tieto výsledky. Popíšme si teraz poznatky nadobudnuté v oblasti nášho záujmu.

Keďže operácie lookahead a lookbehind sú v regexoch nové, na počiatku sme skúmali ich vlastnosti. Uzáverové vlastnosti tried Chomského hierarchie dopadli nasledovne:

**Veta 1.2.1. [Tó13, Veta 2.2.5.]**

*Trieda  $\mathcal{R}$  je uzavretá na pozitívny lookaround.*

Trieda  $\mathcal{L}_{CS}$  je tiež uzavretá a trieda  $\mathcal{L}_{CF}$  nie je.

Základnou vlastnosťou pozitívneho lookaheadu je, že do jeho vnútra sa oplatí dávať iba prefixové jazyky. Vyplýva to z toho, že lookahead matchuje tak skoro, ako len môže – akonáhle nájde prvé slovo patriace do jeho jazyka, zvyšok vstupu už neskúma. A tým pádom môžeme z daného jazyka vyhodit všetky slová, ktorých prefix doň tiež patrí. Analogicky to platí pre pozitívny lookbehind a sufixové jazyky. Dôsledkom je, že pokiaľ jazyk v pozitívnom lookarounde obsahuje prázdne slovo ( $\varepsilon$ ), potom je možné celý lookaround vynechať. Dôvodom je to, že lookaround „nevyužíra písmenká”, preto si môže zakaždým zvolit matchovať  $\varepsilon$  a automaticky vždy akceptovať.

Prejdime teraz k regexom. Pridaním pozitívneho lookarouнду k triede *Regex* získame nový model. Avšak silnejší ako *Regex* nie je:

**Veta 1.2.2. [Tó13, Veta 2.2.10.]**

*Trieda jazykov nad *Regex* s pozitívnym lookarouandom je  $\mathcal{R}$ .*

Skúmaním modelu so spätnými referenciami aj pozitívnym lookarouandom vznikla hierarchia:

**Veta 1.2.3. [Tó13, Vety 2.2.13 a 2.2.14.]**

$$\mathcal{L}_{ERE} \subsetneq \mathcal{L}_{LERE} \subseteq \mathcal{L}_{CS}$$

Z viet 1.2.2 a 1.2.3 vyplýva dôležité pozorovanie, že síce lookaround samostatne regexom silu nepridáva, ale ak ho pridáme k modelu so spätnými referenciami, pridá do triedy nové jazyky. Z tohto hľadiska nabera na význame a vidíme dôvod, prečo sa ním zaoberať.

Je to spôsobené tým, že trieda  $\mathcal{L}_{ERE}$  nie je uzavretá na prienik [CN09] a pozitívny lookaround je nástroj na zapísanie prieniku 2 jazykov. Navyše pridaním negatívneho



lookaroundu je zaručená aj uzavretosť na komplement. Avšak to, či je naň uzavretá aj  $\mathcal{L}_{LRE}$  je otvorený problém.

Zaujímavá vlastnosť triedy  $\mathcal{L}_{LRE}$  je, že obsahuje jednoduché jazyky a to v tom zmysle, že pre ňu existuje pumpovacia lema ([CSY03] aj [CN09]). Pridaním lookaroundu táto vlastnosť mizne – do triedy  $\mathcal{L}_{LRE}$  patrí jazyk všetkých platných výpočtov Turingovho stroja [Tó13, Veta 2.2.16.]. Pre unárne jazyky z  $\mathcal{L}_{LRE}$  bola dokázaná kvázi-pumpovacia lema. Kvázi preto, že platí len pre množinu regexov, ktoré vnútri Kleeneho  $*$  neobsahujú žiadnu konštrukciu lookaheadu obsahujúceho metaznak  $\$$  alebo lookbehindu obsahujúceho metaznak  $\wedge$  (t.j. lookaround spúšťaný niekoľkokrát v iterácii, vždy potenciálne z inej pozície je problém).

## 1.3 Popisná zložitosť regulárnych výrazov

V cieľoch tejto práce je skúmať aj zložitosť moderných regulárnych výrazov. Opäť sa odvolávame na to, že k našej téme sme nenašli články. Je zrejmé, že skúmať NSPACE a DSPACE základných regulárnych výrazov nemá zmysel – sú konštantné. Preto zostala popisná zložitosť.

Z oblasti popisnej zložitosti sú známe len výsledky pre regulárne výrazy s operáciami zjednotenia, zretazenia a Kleeneho  $*$  (RE), prípadne ešte prieniku a komplementu (GRE). Navyše mali ešte znak pre prázdny jazyk  $\emptyset$  a prázdne slovo  $\varepsilon$ .

Spomeňme si najprv ako možno zadefinovať popisnú zložitosť [EKSW04] [EZ76]:

Nech  $E$  je regulárny výraz, potom

- $|E|$  je jeho celková dĺžka
- $rpn(E)$  je jeho celková dĺžka v reverznej poľskej notácii. Táto notácia oproti klasickej neobsahuje okrúhle zátvorky a používa explicitný metaznak pre zretazenie  $\bullet$ . Napríklad regulárny výraz  $(a|ab) * (c|d)$  má 12 znakov a v reverznej poľskej notácii  $aab \bullet | * cd| \bullet$  má 10 znakov.

Uvedomme si, že dĺžka reverznej poľskej notácie je rovnaká ako počet vrcholov v syntaktickom strome pre daný výraz. Preto možno vernejšie popisuje skutočnú zložitosť regulárneho výrazu (žiadne pomocné symboly, len znaky a operácie).

Nakoľko však nie je veľmi prehľadná a ťažko sa v nej hľadajú chyby, nie je tak často používaná.

- $|alph(E)| = N(E)$  je počet alfabetických symbolov v  $E$
- $H(E)$  je hĺbka vzhľadom na  $*$ , počet vnorení  $*$
- $L(E)$  je dĺžka najdlhšej neopakujúcej sa cesty cez výraz
- $W(E)$  je šírka, počet zjednotených symbolov. Za touto mierou zložitosti nie je žiadna intuitívna predstava. Ako vidieť neskôr v definícii, dôvodom jej vzniku bola duálnosť k  $L$ .

V nasledujúcej tabuľke sú uvedené indukčné definície jednotlivých mier zložitosti. Zložitost regulárneho jazyka vzhľadom na ľubovoľnú z týchto mier zložitosti je minimálna miera cez všetky regulárne výrazy pre daný regulárny jazyk.

	Alphabetical Symbol	$E \cup F$	$E \cdot F$	$E^*$
N	1	$N(E) + N(F)$	$N(E) + N(F)$	$N(E)$
H	0	$\max(H(E), H(F))$	$\max(H(E), H(F))$	$H(E) + 1$
L	1	$\max(L(E), L(F))$	$L(E) + L(F)$	$L(E)$
W	1	$W(E) + W(F)$	$\max(W(E), W(F))$	$W(E)$

Ako pri mnohých iných modeloch, aj do regulárnych výrazov vieme zakomponovať časti, ktoré nič nerobia (okrem toho, že zaberajú miesto). V rámci skúmania najjednoduchších výrazov sa prišlo k nasledujúcim definíciám [EKSW04]:

**Definícia 1.3.1.** *Nech  $E$  je regulárny výraz nad abecedou  $\Sigma$  a nech  $L(E)$  je jazyk špecifikovaný výrazom  $E$ . Hovoríme, že  $E$  je **zmenšiteľný**, ak platí nejaká z nasledujúcich podmienok:*

1.  $E$  obsahuje  $\emptyset$  a  $|E| > 1$
2.  $E$  obsahuje podvýraz tvaru  $FG$  alebo  $GF$ , kde  $L(F) = \varepsilon$
3.  $E$  obsahuje podvýraz tvaru  $F|G$  alebo  $G|F$ , kde  $L(F) = \{\varepsilon\}$  a  $\varepsilon \in L(G)$

Inak, ak žiadna z nich neplatí,  $E$  nazývame **nezmenšiteľným**.

V originále sa používajú výrazy *collapsible* a *uncollapsible*. Táto definícia neodhalí všetky zbytočne zopakované časti, napríklad  $a|a$  je nezmenšiteľný, aj keď by sa dal zapísať jednoduchým  $a$ . Problém je, že identity výrazov nie sú konečne axiomatizovateľné (ani nad unárnou abecedou)[EKSW04]. Teda nie je reálne určiť také pravidlá, aby sme dosiahli konečné zjednodušenie.

**Definícia 1.3.2.** Ak  $E$  je nezmenšiteľný regulárny výraz taký, že

1.  $E$  nemá nadbytočné  $()$ ; a zároveň
2.  $E$  neobsahuje podvýraz tvaru  $F **$

potom vravíme, že  $E$  je **neredukovateľný** (*irreducible*).

Môžeme vyvodiť, že minimálny regulárny výraz pre daný jazyk bude nezmenšiteľný a neredukovateľný, naopak to však nemusí platiť. S takýmto základom možno dokázať napríklad tvrdenie: Ak  $E$  je neredukovateľný a  $|\text{alph}(E)| \geq 1$ , potom  $|E| \leq 11|\text{alph}(E)| - 4$ .

Iné spôsoby na ohraničenie regulárnych výrazov poskytujú nasledujúce pozorovanie a veta.

**Veta 1.3.3** (Proposition 6 [EKSW04]). Nech  $L$  je neprázdny regulárny jazyk.

- (a) Ak dĺžka najkratšieho slova v  $L$  je  $n$ , potom  $|\text{alph}(E)| \geq n$  pre ľubovoľný regulárny výraz  $E$ , kde  $L(E) = L$ .
- (b) Ak navyše  $L$  je konečný a dĺžka najdlhšieho slova v  $L$  je  $n$ , potom  $|\text{alph}(E)| \geq n$  pre ľubovoľný regulárny výraz, kde  $L(E) = L$ .

Definícia non-returning NKA hovorí, že sa nevracia do počiatočného stavu, t.j. žiadne prechody nevedú smerom do  $q_0$ .

**Veta 1.3.4** (Theorem 10). Nech  $E$  je regulárny jazyk s  $|\text{alph}(E)| = n$ . Potom existuje non-returning NKA akceptujúci  $L(E)$  s  $\leq n + 1$  stavmi a DKA akceptujúci  $L(E)$  s  $\leq 2n + 1$  stavmi.

Rozbehnutých je viacero oblastí skúmania. Regulárne výrazy sú zaujímavé hlavne preto, že tvoria stručnejší popis jazyka a sú ekvivalentné automatom. S tým súvisí prvá oblasť. Skúma sa stavová zložitosť automatu ekvivalentného konkrétnemu výrazu

a naopak tiež popisná zložitosť výrazu ekvivalentného konkrétnemu automatu. Dá sa to zhrnúť ako problémy konverzií medzi modelmi. Niektorí autori siahajú po väčšej abstrakcii a namiesto automatov uvažujú iba orientované grafy s hranami označenými symbolmi. Určia si parametre grafu a snažia sa napríklad čo najlepšie popísať cesty medzi vrcholmi.

Ďalšia oblasť zahŕňa operácie nad regulárnymi výrazmi. Jeden z problémov je napríklad vzťah popisnej zložitosti výrazu pre jazyk  $L$  a  $L^c$ . Pre automaty existuje konštrukcia pre vyrobenie komplementu jazyka. Avšak pre komplementárny regulárny výraz to nie je také jednoznačné.

Ako poslednú by sme spomenuli problém najkratšieho slova nešpecifikovaného regulárnym výrazom. Predpokladajme regulárny výraz  $E$ , kde  $|alph(E)| = n$  nad konečnou abecedou  $\Sigma$ , pričom  $L(E) \neq \Sigma^*$ . Aké dlhé môže byť najkratšie slovo NEšpecifikované výrazom  $E$ ? Najzaujímavejší výsledok je pre výraz s  $|alph(E)| = 75n + 361$ , kde najkratšie nešpecifikované slovo je dĺžky  $3(2n - 1)(n + 1) + 3$ .

## 2 Naše výsledky

**TODO!!!** Sem by bolo fajn niečo napísať.

### 2.1 Sila negatívneho lookaroundu

V bakalárskej práci bol skúmaný hlavne pozitívny lookaround. Teraz si pozrieme na vlastnosti negatívneho lookaroundu. Budú to vety s podobným znením ako v prípade pozitívnej verzie.

Pripomenieme, že negatívny lookahead skúša všetky prefixy a pokiaľ žiadny z nich nepatrí do jeho jazyka, akceptuje a hľadanie zhody môže pokračovať v regexe ďalej. Čo je jednoduché pri pozitívnej verzii – stačí si nedeterministicky tipnúť ten správny prefix patriaci do jazyka – to je komplikované pri tej negatívnej. Aby sme mohli s istotou povedať, že akceptuje, potrebujeme deterministický algoritmus, ktorý vyskúša všetky možnosti. Preto veľakrát budeme budovať negatívny lookaround tak, že vezmeme nejaký deterministický model akceptujúci jeho jazyk a upravíme akceptáciu.

**Lema 2.1.1.** *Trieda  $\mathcal{R}$  je uzavretá na negatívny lookahead.*

*Dôkaz.* Nech  $L_1, L_2, L_3 \in \mathcal{R}$ . Ukážeme, že  $L = L_1(?! L_2)L_3 \in \mathcal{R}$ .

Jazyky  $L_1, L_2, L_3$  sú regulárne, teda existujú DKA  $A_i = (K_i, \Sigma_i, \delta_i, q_{0i}, F_i)$  také, že  $L(A_i) = L_i$ ,  $i \in \{1, 2, 3\}$ . Zostrojíme NKA  $A$  pre  $L$ .

Konstrukcia bude veľmi podobná ako v obdobnej vete pre pozitívny lookahead v [Tó13], keď si správne predpripravíme  $A_2$ . Negatívny lookahead sa snaží za každú cenu nájsť slovo z  $L_2$  (t.j. uspieť s  $A_2$ ) a ak sa mu to nepodarí, akceptuje. Preto musíme zaručiť, že sa  $A_2$  buď zasekne alebo prejde až do konca slova<sup>1</sup>. Akonáhle  $A_2$  dosiahne akceptačný stav, negatívny lookahead musí zamietnuť.

---

<sup>1</sup>Ak sa zasekne, slovo z  $L_2$  tam určite nebude, lebo  $A_2$  je deterministický a teda neexistuje akceptačný výpočet. Ak sa nezasekne, nevieme povedať o tom slove nič, pokiaľ ho neprejdeme celé.

Ďalšie pozorovanie nám hovorí, že negatívny lookahead akceptuje vždy nejaké slovo z komplementu  $L_2$ . Ale komplement vzhľadom na akú abecedu? V skutočnosti nekonečnú – ľubovoľný znak, ktorý nepatrí do  $\Sigma_2$ , znamená zaseknutie  $A_2$ , t.j. akceptáciu. Ak sa na to pozrieme z väčšej diaľky, uvidíme  $L_3$ , s ktorým robíme prenik.  $A_3$  musí dočítať slovo do konca a na úplne neznámom znaku sa zasekne, takže z pohľadu výslednej akceptácie slova nevadí, ak by kvôli tomu znaku zamietol už negatívny lookahead. Preto stačí urobiť komplement vzhľadom na  $\Sigma_2 \cup \Sigma_3$ .

Podme upravovať  $A_2$ , začneme vytváraním komplementu. Ak  $\Sigma_3 \setminus \Sigma_2 \neq \emptyset^2$ , potom vytvoríme  $A'_2 = (K'_2, \Sigma'_2, \delta'_2, q_0, F'_2)$  tak, že pridáme nové znaky  $\Sigma'_2 = \Sigma_2 \cup \Sigma_3$ , jeden nový stav<sup>3</sup>  $K'_2 = K_2 \cup \{q_{ZLE}\}$  a prechody na nové znaky z každého stavu:

$$\begin{aligned} \forall q \in K_2, \forall a \in \Sigma_2 & : \delta'_2(q, a) = \delta_2(q, a) \\ \forall q \in K_2, \forall a \in \Sigma_3 \setminus \Sigma_2 & : \delta'_2(q, a) = q_{ZLE} \\ \forall a \in \Sigma'_2 & : \delta'_2(q_{ZLE}, a) = q_{ZLE} \end{aligned}$$

$F'_2 = F_2$ . Do všetkých stavov sme pridali prechody na nové znaky a  $q_{ZLE}$  má 1 prechod na každý znak, takže  $A'_2$  je stále deterministický. Zároveň nové znaky vedú do stavu, z ktorého sa nedá dostať do žiadneho akceptačného, z čoho vyplýva  $L(A'_2) = L_2$ .

Teraz  $A'_2$  zmeníme na  $A''_2$  tak, aby akceptoval práve vtedy, keď (!  $L_2$ ). Najprv si skonštruujeme množinu stavov, z ktorých sa vieme dostať do akceptačného stavu:  $H = \{q \in K'_2 \mid \exists w \in \Sigma_2^*, \exists q_A \in F'_2 : (q, w) \vdash_{A'_2} (q_A, \varepsilon)\}$ , nasledovným spôsobom:

$$H_0 = F'_2$$

$$H_{i+1} = H_i \cup \{q \in K'_2 \mid \exists p \in H_i \exists a \in \Sigma'_2 : \delta(q, a) = p\}$$

Zrejme  $\exists j \in \mathbb{N} : H_{j+1} = H_j = H$ . Nás zaujíma množina  $K'_2 \setminus H$ , v ktorej sa nachádzajú všetky stavy, z ktorých sa na žiadnu postupnosť znakov nedá dostať do žiadneho akceptačného stavu.

$$A''_2 = (K''_2, \Sigma''_2, \delta''_2, q_0, F''_2):$$

$$K''_2 = K'_2 \cup \{q_A, q_Z\}, \Sigma''_2 = \Sigma'_2, F''_2 = (K'_2 \setminus H) \cup \{q_A\}$$

<sup>2</sup>Inak  $A'_2 = A_2$ .

<sup>3</sup>Pre každý nový stav predpokladáme, že je jedinečný – t.j. žiaden taký v množine stavov ešte nie je.

## 2.1. SILA NEGATÍVNEHO LOOKAROUNDU KAPITOLA 2. NAŠE VÝSLEDKY

$\delta''$  je definovaná nasledovne:

$$\begin{aligned} \forall q \in K'_2 \setminus F'_2 \quad \forall a \in \Sigma'_2 & : \delta''_2(q, a) = \delta'_2(q, a) \\ \forall q \in K'_2 \setminus F'_2 \quad \forall a \in \Sigma'_2 & : \delta''_2(q, \varepsilon) = q_A \\ \forall q \in F'_2 \quad \forall a \in \Sigma'_2 & : \delta''_2(q, a) = q^4 \\ \forall a \in \Sigma'_2 & : \delta''_2(q_A, a) = q_Z \end{aligned}$$

Je dobré poznamenať, že  $A''_2$  je takmer deterministický. Chýbajú mu prechody z  $q_Z$  – môžeme ho nechať cykliť v tomto stave, ale nevadí nám ani keď sa zasekne. Nedeterministické rozhodnutie vykonáva iba jedno – pri prechode do stavu  $q_A$ . Vtedy háda, že už je na konci slova. Ak nie je,  $\delta$ -funkcia ho pošle do stavu  $q_Z$ . Podľa toho je zrejmé, že ak existuje akceptačný výpočet a dočítal slovo do konca, je práve jeden<sup>5 6</sup>.

Tvrdíme  $L_1(=L(A''_2))L_3 = L_1(?! L_2)L_3 = L$ . Keďže  $A_1$  a  $A_3$  sa pri oboch výpočtoch budú chovať rovnako (sú nezávislé od lookaheadov), nebudeme sa nimi pri dôkaze zaoberať.

$\subseteq$ : Máme akceptačný výpočet pre  $A''_2$  na nejakom vstupe. Akceptačný stav mohol byť buď z množiny  $K'_2 \setminus H$ , to znamená, že pôvodný automat  $A'_2$  sa dostal do stavu, z ktorého už nemohol nijak akceptovať<sup>7</sup>. V takom prípade  $(?! L_2)$  akceptuje. V druhom prípade mohol  $A''_2$  akceptovať pomocou  $q_A$ , čo znamená, že dočítal slovo do konca (pretože z  $q_A$  sa na ľubovoľný znak dostaneme do  $q_Z$ , v ktorom sa  $A''_2$  zasekne a nebude akceptovať) a ani raz sa nedostal do akceptačného stavu automatu  $A'_2$ . Teda aj  $(?! L_2)$  akceptuje.

$\supseteq$ : Nech  $(?! L_2)$  akceptoval, t.j. na  $A_2$  bol vykonaný nejaký neakceptujúci výpočet ( $A_2$  je deterministický, takže existuje práve jeden pre každé slovo). Rovnakú postupnosť stavov bude mať aj výpočet na  $A''_2$  (automat obsahuje všetky stavy aj  $\delta$ -funkciu z  $A_2$ , počiatočný stav je ten istý). Ak sa  $A_2$  zasekol na neznámom znaku, v  $A'_2$  na ten znak pridáme prechod do stavu  $q_{ZLE}$  a v ňom zostaneme až do konca slova. Keďže  $q_{ZLE}$  nie je v  $A'_2$  akceptačný a nedá sa z neho na žiadne slovo dostať do ľubovoľného akceptačného

<sup>4</sup>Tento riadok v podstate netreba,  $A''_2$  sa môže rovno zaseknúť, lebo  $A'_2$  práve akceptoval.

<sup>5</sup>Okrem prípadu, kedy dočíta slovo a je v stave z  $(K'_2 \setminus H)$  – vieme ho predĺžiť o krok na  $\varepsilon$  a skončiť v  $q_A$ . Jadro výpočtu ale zostáva rovnaké – jednoznačné.

<sup>6</sup>Zaujímavé je, že aj zamietací výpočet je pre každé slovo jednoznačný.

<sup>7</sup>Sem spadá aj prípad, keď  $A_2$  prečítal neznámy znak a zasekol sa –  $A'_2$  zostáva až do konca slova v stave  $q_{ZLE}$ .

stavu,  $q_{ZLE} \in K'_2 \setminus H$  a teda  $q_{ZLE} \in F''_2$ . Ak sa  $A_2$  nezasekol, tak dočítal slovo do konca a v postupnosti stavov jeho výpočtu sa nenachádza žiaden z množiny  $F_2$ . V tom prípade  $A''_2$  z posledného stavu prejde na  $\varepsilon$  do  $q_A$  (2. riadok definície  $\delta''_2$ ) a akceptuje.

Z práve dokázaného tvrdenia a vety 1.2.1 už vyplýva aj platnosť našej lemy.  $\square$

**Lema 2.1.2.** *Trieda  $\mathcal{R}$  je uzavretá na negatívny lookbehind.*

*Dôkaz.* Nech  $L_1, L_2, L_3 \in \mathcal{R}$ , existujú DKA  $A_i = (K_i, \Sigma_i, \delta_i, q_{0i}, F_i)$  také, že  $L(A_i) = L_i$ ,  $i \in \{1, 2, 3\}$ . Ukážeme, že  $L = L_1(?< L_2)L_3 \in \mathcal{R}$ .

Nemôžeme skonštruovať  $A''_2$  také, že bude akceptovať práve vtedy, keď  $(?< L_2)$ , pretože nevieme čítať doľava – teda nevieme zaručiť, že miesto, kde začína výpočet  $A''_2$  je skutočný začiatok slova. Mechanizmus lookbehindu musí byť riadený zvonku, automatom pre  $L$ . Skonštruujeme ho.  $C = (K, \Sigma, \delta, q_0, F)$  :

$$K = K_3 \cup \{(q, A) \mid q \in K_1 \wedge A \subseteq K_2\}, \quad \Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3, \quad q_0 = (q_{01}, \{q_{02}\}), \quad F = F_3$$

$\delta$  :

$$\begin{aligned} \forall q \in K_1 \quad \forall A \subseteq K_2 \quad \forall a \in \Sigma_1 \cap \Sigma_2 & : \delta((q, A), a) \ni (\delta_1(q, a), B \cup \{q_{02}\}) \\ & B = \{r \mid \exists s \in A : \delta_2(s, a) = r\} \\ \forall q \in K_1 \quad \forall A \subseteq K_2 \quad \forall a \in \Sigma_1 \setminus \Sigma_2 & : \delta((q, A), a) \ni (\delta_1(q, a), \{q_{02}\}) \\ \forall q \in F_1 \quad \forall A \subseteq K_2 \setminus F_2 & : \delta((q, A), \varepsilon) \ni q_{03} \\ \forall q \in K_3 \quad \forall a \in \Sigma_3 & : \delta(q, a) \ni \delta_3(q, a) \end{aligned}$$

Prvý riadok  $\delta$ -funkcie hovorí, že  $A_1$  a všetky rozbehnuté výpočty  $A_2$  prejdú do ďalšieho stavu a zároveň sa rozbehne nový výpočet  $A_2$ . Ak by sme hľadali jeden akceptačný výpočet  $A_2$  stačilo by tipnúť si začiatok a odtiaľ ho simulovať. Lenže simulujeme negatívny lookbehind, teda ak neexistuje akceptačný výpočet, tak my akceptujeme (prejdeme na simulovanie  $A_3$ , 4. riadok). A to vieme povedať len v prípade, že sme videli všetky možné výpočty. Keďže  $A_2$  je deterministický, pre každé slovo existuje práve jeden výpočet a zároveň sa nikdy nezasekne (na svojej abecede), teda nám stačí skontrolovať množinu stavov  $A$  vtedy, keď sa  $C$  rozhodne, že  $A_1$  končí výpočet. Ak v  $A$  je nejaký akceptačný stav, potom  $C$  nevie prejsť do stavu  $q_3$ . Tým pádom sa nedostane k simulovaniu  $A_3$  a ani k svojim akceptačným stavom.

$$L(C) = L.$$



## 2.1. SILA NEGATÍVNEHO LOOKAROUNDU KAPITOLA 2. NAŠE VÝSLEDKY

$\subseteq$ : Majme akceptačný výpočet  $C$  na  $w$ . Z definície  $F$  vyplýva, že  $A_3$  akceptoval, teda  $\exists u, v$  také, že  $w = uv$  a  $v \in L_3$ . Do  $q_{03}$  sa dá dostať len z dvojice  $(q, A)$  takej, že  $q \in F_1$ , teda  $u \in L_1$ , a  $A \cap F_2 = \emptyset$ , teda  $\nexists x, y$  také, že  $u = xy$  a  $y \in L_2$ . To vyplýva z toho, že v každom znaku  $u$  začal jeden výpočet na  $A_2$  a žiaden z nich neakceptoval. Teda negatívny lookbehind akceptoval a  $w \in L$ .

$\supseteq$ : Nech  $w \in L$ , teda  $\exists u, v$  také, že  $w = uv$ ,  $u \in L_1, v \in L_3$  a  $\forall x, y : u = xy$  platí  $y \notin L_2$ . Pre  $u, v$  teda existujú akceptačné výpočty na  $A_1, A_3$  a pre všetky  $y$  neakceptačné na  $A_2$ . Z toho už vieme vyskladať akceptačný výpočet na  $C$ .  $\square$

**Veta 2.1.3.** *Trieda  $\mathcal{R}$  je uzavretá na negatívny lookaround.*

Zjavne konečné automaty si s negatívnym lookaroundom poradia. Pozrime sa teraz na model *Regex* s pridaným negatívnym lookaroundom. Najprv musíme skontrolovať kombinácie operácií ako v [Tó13, Lema. 2.2.8. a 2.2.9]. Ukazuje sa, že aj tu dostávame rovnaké výsledky ako pri pozitívnej verzii.

**Lema 2.1.4.** *Nech  $L_1, L_2, L_3, L_4 \in \mathcal{R}$ ,  $\alpha = (L_1 (?! L_2) L_3) * L_4$ . Potom  $L(\alpha) \in \mathcal{R}$ .*

*Dôkaz.* Podobne ako v dôkaze vety 2.1.1 pretransformujeme  $(?! L_2)$  na akýsi  $(?=L(A_2''))$ , kde  $A_2''$  bude akceptovať práve vtedy, keď  $(?! L_2)$ . Potom  $\beta = (L_1(?=L(A_2''))L_3) * L_4$ ,  $L(\beta) = L(\alpha) \in \mathcal{R}$  podľa lemy k vete 1.2.1.  $\square$

**Lema 2.1.5.** *Nech  $L_1, L_2, L_3, L_4 \in \mathcal{R}$ ,  $\alpha = L_4 (L_1 (? <! L_2) L_3) *$ . Potom  $L(\alpha) \in \mathcal{R}$ .*

*Dôkaz.* Použijeme konštrukciu ako v dôkaze vety 2.1.2, kde pretransformujeme  $(? <! L_2)$  na  $(? <= L(A_2''))$ , kde  $A_2''$  bude akceptovať práve vtedy, keď  $(? <! L_2)$ . Z toho vznikne  $\beta = L_4 (L_1 (? <= L(A_2'')) L_3) *$ ,  $L(\beta) = L(\alpha) \in \mathcal{R}$  podľa lemy k vete 1.2.1.  $\square$

**Veta 2.1.6.** *Trieda nad regexami s negatívnym lookaroundom je  $\mathcal{R}$ .*

*Dôkaz.* Podobne ako v dôkaze vety 1.2.2 je netriviálna iba kombinácia operácií negatívny lookaround a  $*$ . Podľa lemy 2.1.4 a 2.1.5 máme stále regulárne jazyky.  $\square$

**Dôsledok 2.1.7.** *Trieda nad regexami s pozitívnym a negatívnym lookaroundom je  $\mathcal{R}$ .*

*Dôkaz.* Lookaround nevyžiera písmenká, takže neovplyvňuje zvyšok regexu, teda ani iné lookaroundy. Musíme vyriešiť iba prípad, kedy je do seba vnorených niekoľko pozitívnych a negatívnych lookaroundov. V takejto situácii vieme postupovať z najvnútornejšieho regexu s lookaroundom s hĺbkou vnorenia 1. Jeho jazyk je regulárny a teda

ho vieme prepísať na regex z *Regex*. Takýmto spôsobom vieme znútra von prepisovať regex, až skončíme s regexom z *Regex*, teda matchujúcim regulárny jazyk.  $\square$

## 2.2 Vlastnosti triedy $\mathcal{L}_{LRE}$

Pridanie nových operácií medzi regulárne výrazy síce pridalo na silu modelu, ale mohlo pokaziť jeho uzáverové vlastnosti. Vieme, že regulárne jazyky sú uzavreté na všetky možné operácie, ktoré poznáme. Posilnenie modelu spätnými referenciami a následne lookaroundom však ohrozilo uzavretosť na základnú operáciu regulárnych výrazov – zretazenie. Pokiaľ zretazíme 2 regexy obsahujúce lookahead alebo lookbehind, tieto operácie začnú zasahovať do slova z vedľajšieho jazyka. Ak by išlo o zretazenie so zarážkou (napr.  $\alpha\#\beta$ ), na koniec každého lookaheadu v  $\alpha$  by stačilo pripojiť  $. * \#$ , prípadne znak  $\$$  nahradiť znakom  $\#$ . Podobne by sme v regexe  $\beta$  pridali na začiatok každého lookbehindu  $\#.*$  prípadne by sme vymenili znak  $\wedge$  znakom  $\#$ . Potom by tieto operácie zostali „skrotené” na území slova, ktoré danému regexu prislúcha.

Otázka nastáva, ako to spraviť, keď zarážku k dispozícii nemáme. Odpoveďou je nasledujúca veta.

**Veta 2.2.1.** *Trieda  $\mathcal{L}_{LRE}$  je uzavretá na zretazenie.*

*Dôkaz.* Nech  $\alpha, \beta \in LRE_{regex}$ . Chceme ukázať, že jazyk  $L(\alpha)L(\beta) \in \mathcal{L}_{LRE}$ . Intuitívne nás to vedie k riešeniu  $\alpha\beta$ , čo ale nemusí byť vždy správne. Problémom sú operácie lookaround, presnejšie každý lookahead v  $\alpha$  môže zasahovať do slova z  $L(\beta)$  a takisto každý lookbehind z  $\beta$  môže zasahovať do slova z  $L(\alpha)$ . Navyše, ak lookahead obsahuje  $\$$  a lookbehind  $\wedge$ , potom zasahujú do slova z iného jazyka určite. V takom prípade môže regex  $\alpha\beta$  vynechať niektoré slová z  $L_1L_2$  a tiež pridať nejaké nevhodné slová navyše. Preto treba nájsť spôsob, ako operácie lookaroundu vhodne „skrotiť”. Predpokladajme, že  $\alpha$  má  $k$  označených zátvoriek, potom regex pre jazyk  $L(\alpha)L(\beta)$  vyzerá nasledovne:

$$(?=(\alpha) (\beta) \$)\alpha' \backslash k+2 (?<=\wedge 1\beta') \quad (2.1)$$

$\begin{matrix} 1 & 1 & k+2 & k+2 \end{matrix}$

V  $\alpha, \beta$  treba vhodne prepísať spätné referencie podľa nového prečíslovania zátvoriek.  $\alpha'$  je  $\alpha$  prepísaný tak, že pre každý lookahead:

- bez  $\$$  – na koniec pridáme  $. * \backslash k+2 \$$

- s \$ – pred \$ pridáme  $\backslash k+2$

$\beta'$  je  $\beta$  prepísaný tak, že pre každý lookbehind:

- bez  $\wedge$  – na začiatok pridáme  $\wedge \backslash 1.*$
- s  $\wedge$  – za  $\wedge$  pridáme  $\backslash 1$

Čo presne robí regex 2.1? Nech  $w$  je vstupné slovo, ktoré chceme matchovať. Lookahead na začiatku ho nejako rozdelí na  $w_1$  a  $w_2$ , pričom  $w = w_1 w_2$ , tak, že  $w_1$  bude patriť do  $L(\alpha)$  a podslovo  $w_2$  do  $L(\beta)$ . Ešte musíme overiť, či  $\alpha$  matchuje  $w_1$  samostatne.

Preto sme v  $\alpha$  prepísali všetky lookaheady. V  $\alpha'$  každý z nich musí na konci slova  $w$  matchovať  $w_2$  (toto zabezpečí spätná referencia  $\backslash k+2$  a \$) a teda matchovanie regexu  $\alpha$  zostane výlučne na podslove  $w_1$ . Analogicky to platí pre upravené lookbehindy v  $\beta'$ .

□

## 2.3 Zaradenie do Chomského hierarchie

**Veta 2.3.1.**  $\mathcal{L}_{LRE}$  je neporovnateľná s  $\mathcal{L}_{CF}$ .

*Dôkaz.* Majme jazyk  $L = \{ww \mid w \in \{a,b\}^*\} \in \mathcal{L}_{CF}$ , nech  $\alpha = ([ab]^*)\backslash 1$ . Zrejme  $L(\alpha) = L$ , teda  $L \in \mathcal{L}_{LRE}$ .

Ukážeme, že jazyk  $L = \{a^n b^n \mid n \in \mathbb{N}\} \notin \mathcal{L}_{LRE}$ . Sporom, nech  $L \in \mathcal{L}_{LRE}$ .

Vieme, že  $L \notin \mathcal{L}_{RE}$ , preto musí obsahovať nejaký lookaround. Zároveň z  $L \notin \mathcal{R}$  a vety 1.2.2 vyplýva, že musí obsahovať aj spätné referencie.

Kam sa môžu spätné referencie odkazovať a kam ich potom môžeme umiestniť? Nech výraz, na ktorý ukazujú, vyrobí nejaké:

- $a^i$ , potom  $\backslash k$  musí byť v prvej polovičke slova (medzi  $a$ , inak by pokazil štruktúru slova), takže nevplýva na časť s  $b$  a teda sa zaobídeme bez nich.
- $a^i b^j$ , potom  $\backslash k$  by mohol byť len medzi  $b$ , ale tam by pokazil štruktúru slova  $a^n b^n$
- $b^j$ , potom  $\backslash k$  môže byť len medzi  $b$  a je tam zbytočný z rovnakých dôvodov, aké má prípad  $a^i$

Vidíme, že so spätnými referenciami dosiahneme rovnaký výsledok ako bez nich, čo je spor s tým, že sa vo výraze musia nachádzať (bez nich vieme urobiť len regulárny jazyk).  $\square$

**Dôsledok 2.3.2.**  $\mathcal{L}_{LERE} \subsetneq \mathcal{L}_{CS}$

**Veta 2.3.3.**  $\mathcal{L}_{nLERE} \subseteq \mathcal{L}_{CS}$

*Dôkaz.* Vieme, že  $\mathcal{L}_{LERE} \in \mathcal{L}_{CS}$  (veta 1.2.3), teda ľubovoľný regex z  $LEregex$  vieme simulovať pomocou LBA. Ukážeme, že ak pridáme operáciu negatívny lookahead/lookbehind, vieme to simulovať tiež.

Nech  $\alpha \in nLEregex$ . Potom nech  $A$  je LBA pre  $\alpha$ , ktorý ignoruje negatívny lookaround (t.j. vyrábame LBA pre regex z  $LEregex$ ). Teraz vytvoríme LBA  $B$  pre regex vnútri negatívneho lookarounds. Z nich vytvoríme LBA  $C$  pre úplný regex  $\alpha$  tak, že  $C$  bude simulovať  $A$  a keď príde na rad negatívny lookaround zaznačí si, v akom stave je  $A$  a na ktorom políčku skončil. Skopíruje slovo na ďalšiu stopu a na nej simuluje od/do toho miesta  $B$  (podľa toho, či je to lookahead alebo lookbehind).

Teraz je to s akceptáciou náročnejšie ako pri pozitívnom lookarounds. Pokiaľ  $B$  akceptoval,  $C$  sa zasekne. Ak sa  $B$  zasekol,  $C$  sa vráti naspäť k zastavenému výpočtu  $A$  a pokračuje v ňom. Keďže slovo je konečné a  $B$  na ňom testuje regex, ktorý postupne vyjedá písmenká, určite raz príde na koniec slova. Môže sa stať, že bude skúšať viaceré možnosti – napr. skúsi pre  $*$  zobrať menej znakov, teda príde na koniec slova viackrát. Ako určíme, že skončil výpočet? Bez újmy na všeobecnosti môžeme predpokladať, že sa  $B$  nezacyklí, ale zamietne slovo na konci výpočtu. To preto, že všetkých možností na rozdelenie znakov medzi operácie  $*$ ,  $+$ ,  $?$ ,  $\{n, m\}$  je konečne veľa a keď ich systematicky skúša<sup>8</sup>, raz sa mu musia minúť. To znamená, že ak nemá v  $\delta$ -funkcii umelo vsunuté zacyklenie, nezacyklí sa. Teda ak slovo neakceptuje, tak ho určite zamietne a vtedy  $C$  môže prejsť na zastavený výpočet  $A$  a pokračovať v ňom.

Podobne ako pri lookarounds aj jeho negatívna verzia môže obsahovať vnorený negatívny lookaround. Tých však môže byť iba konečne veľa, keďže každý regex musí mať

<sup>8</sup>Algoritmus pre  $*$  je v praxi greedy. Ak slovo nesedí, tak sa vráti, odoberie posledný znak zožratý  $*$  a opäť skúša zvyšok výrazu, či slovo sedí. Ak stále nevyhovuje, algoritmus odoberá hviezdičky znaky dovtedy, dokým nenájde zhodu alebo odoberie všetky znaky – vtedy sa mu minuli všetky možnosti a môže slovo zamietnuť.

konečný zápis. Čo znamená, že aj stôp bude konečne veľa a naznačeným postupom si vieme postupne vybudovať LBA, ktorý bude simulovať  $\alpha$ .  $\square$

**TODO!!!**check & rewrite

**Veta 2.3.4.**  $\mathcal{L}_{nLRE}$  je neporovnateľná s  $\mathcal{L}_{CF}$ .

*Dôkaz.* Opäť použijeme jazyk  $L = \{a^n b^n \mid n \in \mathbb{N}\}$  a budeme dokazovať sporom. Nech  $L \in \mathcal{L}_{LRE}$ .

Z vety 2.3.1 vidíme, že výraz musí obsahovať negatívny lookaround. Z jej dôkazu vidíme, že spätné referencie nepomôžu, nech sa ich pokúsime hocijako použiť. Z toho vyplýva, že ich nepoužijeme a z vety 2.1.6 zistíme, že nám zostal model schopný vyrobiť iba regulárne jazyky. Spor.  $\square$

**Dôsledok 2.3.5.**  $\mathcal{L}_{nLRE} \subsetneq \mathcal{L}_{CS}$

**TODO!!!**obkec o tom, že doplniť substitúciu nestačí **TODO!!!**?? a čo keby sme pridali aj reverz ?? (pozn. obe operácie len upravujú spätné referencie)

## 2.4 Priestorová zložitosť

**Veta 2.4.1.**  $\mathcal{L}_{LRE} \subseteq NSPACE(\log n)$ , kde  $n > 1$  je veľkosť vstupu.

*Dôkaz.* Ukážeme, že ľubovoľný  $\alpha \in LERegex$  vieme simulovať nedeterministickým Turingovým strojom s jednou vstupnou read-only páskou a jednou pracovnou páskou, na ktorej použijeme maximálne logaritmický počet políčok.

Celý regex si budeme uchovávať v stavoch<sup>9</sup> a pridáme doňho špeciálny znak  $\blacktriangleright$ , ktorým si budeme ukazovať, kam sme sa v regexe dopracovali – bude to akýsi smerník na znak, ktorý práve spracovávame. Teda stav bude vyzeráť takto:  $q_{\beta\blacktriangleright\xi}$  (pre lepšiu čitateľnosť budeme uvádzať namiesto stavu iba jeho dolný index:  $\beta\blacktriangleright\xi$ ), kde  $\beta\xi = \alpha$ , čiast  $\beta$  sme už namatchovali na vstup a práve sa chystáme pokračovať čiastou  $\xi$ . Ak  $\blacktriangleright$  ukazuje na znak, porovnáme ho s aktuálnym znakom na vstupnej páske. Pokiaľ sa nezhodujú, výpočet sa zasekne. Pri zhode pokračujeme až kým sa nám neminie vstup aj regex. Ak  $\blacktriangleright$  ukazuje na metaznak, potom zistíme o akú operáciu ide (ak má viac znakov, treba na to niekoľko stavov – napr. lookahead) a začneme ju vykonávať.

<sup>9</sup>Každý regex je má z definície konečnú dĺžku, to znamená aj konečný počet stavov.

Pracovná páska bude slúžiť ako úložisko smerníkov na rôzne miesta vstupnej pásky. Bude mať formát  $A_1\#A_2\#\dots\#A_m$ . Adresu nejakého políčka na vstupnej páske vieme zapísať v priestore  $\log n$ . Ak ukážeme, že  $m$  je konštanta, potom  $m \cdot \log n \in O(\log n)$ . Adresa  $A_1$  bude rezervovaná pre prvý adresný slot na aktuálnu pozíciu hlavy na vstupe, nech si ju máme odkiaľ okopírovať, keď treba. Adresy budeme využívať pri operáciách spätná referencia, lookahead a lookbehind.

Keďže celý regex vidíme pri konštrukcii Turingovho stroja, vieme si do stavov zakódovať význam a poradie adresných slotov. A celú pracovnú pásku si predpripraviť (napísať potrebný počet  $\#$ ).

Teraz skonštruujeme Turingov stroj  $M = (K, \Sigma, \delta, q_0, F)$  k regexu  $\alpha$ .

$$K = \{\beta \blacktriangleright \xi \mid \beta, \xi \text{ sú podslová } \alpha \text{ také, že } \beta\xi = \alpha\}$$

$$\Sigma = \Sigma(\alpha)^{10}, \quad q_0 = \blacktriangleright \alpha, \quad F = \{\alpha \blacktriangleright\}$$

$\delta$  : (v tvare: stav, znak čítaný na vstupnej páske)

Kvôli prehľadnosti popíšeme len hlavné kroky algoritmu.

$\delta(\beta \blacktriangleright a\xi, a) = \{(\beta a \blacktriangleright \xi, 1)\} \forall a \in \Sigma$  – Ak je v regexe znak, matchujeme ho s tým na vstupe.

$\delta(\beta \blacktriangleright (\gamma)\xi, a) = \{(\beta(\blacktriangleright \gamma)\xi, 0)\}$  – Ak sú to  $k$ -te zátvorky a regex obsahuje  $\backslash k$ , zapíšeme aktuálnu adresu ako adresu začiatku pre  $\backslash k$ .

$\delta(\beta(\gamma \blacktriangleright)\xi, a) = \{(\beta(\gamma) \blacktriangleright \xi, 0)\}$  – Ak sú to  $k$ -te zátvorky a regex obsahuje  $\backslash k$ , zapíšeme aktuálnu adresu ako adresu konca pre  $\backslash k$ . Používame polootevorený interval – znak na koncovej adrese do podslova nepatrí.

$\delta(\beta \blacktriangleright (\gamma) * \xi, a) = \{(\beta(\gamma) * \blacktriangleright \xi, 0)\}$  – Pri Kleeneho  $*$  máme možnosť vykonať aj 0 iterácií, teda zátvorky preskočiť. Ak sú to  $k$ -te zátvorky a regex obsahuje  $\backslash k$ , zapíšeme aktuálnu adresu ako adresu začiatku aj konca pre  $\backslash k$ .

$\delta(\beta(\gamma) \blacktriangleright * \xi, a) = \{(\beta(\blacktriangleright \gamma) * \xi, 0), (\beta(\gamma) * \blacktriangleright \xi, 0)\}$  – Kleeneho  $*$ : buď opakujeme alebo pokračujeme ďalej.

<sup>10</sup>T.j. všetky znaky a metaznaky použité v regexe  $\alpha$ .

$\delta(\beta \blacktriangleright \backslash k\xi, a) = \{(q_{najdi\_zaciatok(k)}, 0)\}$  – Najprv si zapamätáme aktuálnu pozíciu na vstupe (ďalej označovanú ako 'aktuálna pracovná pozícia'). Stav  $q_{najdi\_zaciatok(k)}$  zodpovedá presunu hlavy na vstupnej páske na začiatočnú pozíciu podslova. Tu začneme algoritmus porovnávania podslov podľa definície spätnej referencie – aké podslovo matchujú  $k$ -te zátvorky, také isté musí ležať aj na 'aktuálnej pracovnej pozícii'. Algoritmus bude porovnávať vždy postupne po jednom znaku od začiatočnej pozície (ľavá zátvorka) po (koniec - 1) (pravá zátvorka):

pokiaľ zaciatok != koniec:

```
    zapamätaj si znak
    začiatok++
    presuň hlavu na pozíciu 'aktuálna pracovná pozícia'
    porovnaj znak (ak neseďí, zasekni sa)
    (aktuálna pracovná pozícia)++
    presuň hlavu na pozíciu 'začiatok'
```

presuň hlavu na pozíciu 'aktuálna pracovná pozícia'

Vidíme, že si dočasne potrebujeme zapamätať adresu 'aktuálna pracovná pozícia', ale po skončení tohto algoritmu ju môžeme zahodiť.

Po úspešnom zbehu algoritmu TS prejde do stavu  $\beta \backslash k \blacktriangleright \xi$ .

$\delta(\beta \blacktriangleright (?=\gamma)\xi, a) = \{(\blacktriangleright \gamma, 0)\}$  – Lookahead: zapamätáme si aktuálnu pozíciu na vstupe do zodpovedajúceho slotu na pracovnej páske a spracujeme regex vnútri lookaheadu. Ak uspejeme, presunieme hlavu na vstupnej páske naspäť na zapamätanú pozíciu a pokračujeme v regexe ďalej:  $\delta(\gamma \blacktriangleright, a) = \{(\beta(?=\gamma) \blacktriangleright \xi, 0)\}$ .

$\delta(\beta \blacktriangleright (?<=\gamma)\xi, a) = \{(doľava\_ \gamma, 0)\}$  – Lookbehind: zapamätáme si aktuálnu pozíciu na vstupe do zodpovedajúceho slotu na pracovnej páske. Toto je zarážka pre lookbehind – svoje matchovanie musí skončiť na tejto pozícii tak, že tento znak už neberie do úvahy. Nedeterministicky sa vrátíme o niekoľko políčok doľava ( $\delta(doľava\_ \gamma, a) = \{(doľava\_ \gamma, -1), (\blacktriangleright \gamma, 0)\}$ ) a skúsime matchovať regex v lookbehinde. Keď uspejeme, porovnáme aktuálnu pozíciu so zarážkou. Ak sú rôzne, Turingov stroj sa zasekne. Inak pokračuje vo výpočte ďalej, od tejto pozície:  $\delta(\gamma \blacktriangleright \_overene, a) = \{(\beta(?<=\gamma) \blacktriangleright \xi, 0)\}$ .

**Počet adries:**

Pre **spätné referencie** potrebujeme vždy 2 adresy – na začiatok a koniec. Ak sa náhodou budú  $k$ -te zátvorky opakovať, napr. kvôli Kleeneho  $*$ , v definícii stojí, že sa vždy berie do úvahy posledný výskyt, takže adresy prepisujeme pri každom opakovaní. V prípade, že sa  $k$ -te zátvorky nachádzajú vnútri lookaheadu/lookbehindu, tiež máme pre ne rezervované 2 sloty. Algoritmus porovnávania potrebuje 1 ďalšiu adresu – aktuálnu pracovnú pozíciu. Po jeho dokončení adresu môžeme vymazať (tzn. v ďalšom výpočte prepísať niečím iným).

V prípade **lookaheadu a lookbehindu** spotrebujeme len 1 adresný slot a to tiež len dočasne – dokým sa operácia celá nevykoná. Potom je nám tento údaj zbytočný. Tieto operácie však môžu byť vnorené a tak v najhoršom prípade zaberú  $p \cdot \log n$  priestoru, ak ich počet je  $p$ .

Ak máme 1 rezervovaný slot pre aktuálnu adresu,  $s$  spätných referencií,  $l_a$  lookaheadov a  $l_b$  lookbehindov, najviac spotrebujeme  $(1 + 2s + 1 + l_a + l_b) \log n$  priestoru. Celý regex  $\alpha$  je konečne dlhý, teda počet operácií je konečný. Čo znamená, že  $m = 1 + 2s + 1 + l_a + l_b$  je konštanta a to sme chceli dokázať.

□

**Veta 2.4.2** (Savitch [Sav70]). *Nech  $S(n) \geq \log n$  je páskovo konštruovateľná, potom*

$$NSPACE(S(n)) \subseteq DSPACE(S^2(n))$$

**Dôsledok 2.4.3.**  $\mathcal{L}_{LRE} \subseteq DSPACE(\log^2 n)$ , kde  $n > 1$  je veľkosť vstupu.

Keď už máme deterministický model pre pozitívny lookaround, vieme triedu rozšíriť aj o ten negatívny.

**Veta 2.4.4.**  $\mathcal{L}_{nLRE} \subseteq DSPACE(\log^2 n)$ , kde  $n > 1$  je veľkosť vstupu.

Dôkaz tejto vety uvidíme neskôr. Najprv si zavedieme niekoľko nových pojmov. Keďže sa budeme opierať o dôkaz Savitchovej vety [Žu03], bude to akási forma konfigurácií, inšpirovaná definíciou Turingovho stroja.

**Definícia 2.4.5.** *Konfiguráciou regexu  $\alpha = r_1 \dots r_n$  nazývame dvojicu  $(r, w)$ , kde  $r \in (\lceil \alpha \rceil \cup (\alpha \rfloor) \cup (r_1 \dots \lceil r_i \dots r_n)$   $w \in \Gamma^* \lceil \Gamma^*$  a symbol  $\lceil$  ukazuje, kde sa nachádzame vo výpočte v regexe a v slove.*



**Definícia 2.4.6.** Konfiguráciu  $(r_1 \dots r_n[, w_1 \dots w_m[)$  nazývame **akceptačná**.

**Definícia 2.4.7.** Zátvorka  $($  v regexe  $\alpha$  je **indexovateľná**, ak sa bezprostredne za ňou nenachádza metaznak  $?$  a zároveň sa nenachádza vnútri negatívneho lookaroundu.

Je zakázané odkazovať sa spätnými referenciami na operácie formy  $(? \dots)$ , preto ich ani nechceme a nebudeme brať do úvahy v poradí zátvoriek. Z týchto operácií sa v našom modeli nachádza iba pozitívny a negatívny lookaround. Je povolené odkazovať sa na zátvorky vnútri lookaroundu, takže sa vieme odvolať na podslovo, čo sa zhoduje s pozitívnou formou (ak lookaround považujeme za neindexovateľné zátvorky, stačí ho prepísať do formy  $(?=(\dots))$  a vieme sa referencovať na jeho obsah). Problém nastáva pri jeho negatívnej verzii – podľa definície nesmie nájsť žiadnu zhodu, inak sa výpočet zastaví. Potom po akceptácii nedefinuje žiadne podslovo, na ktoré by sme sa mohli odvolať. To isté platí o ľubovoľných zátvorkách v jeho vnútri. **TODO!!!** ale na samotné matchovanie vnútri potrebuje aj spätné referencie

**Definícia 2.4.8.** Zátvorka  $)$  v regexe  $\alpha$  je **indexovateľná**, ak k nej prislúchajúca otváracia zátvorka je indexovateľná.

**Definícia 2.4.9.** Nech regex  $\alpha \in nLEregex$  obsahuje alternáciu. Potom jeho podslovo  $\beta$  nazývame **alternovateľným**, ak je validným regexom z nle a zároveň zodpovedá jednej z týchto podmienok:

(i) **prvá**

$\beta$  je prefix  $\alpha$  alebo znak pred  $\beta$  je  $($ <sup>11</sup>  
 $\wedge$  za  $\beta$  nasleduje metaznak  $|$

(ii) **stredná**

znak pred  $\beta$  je  $|$   
 $\wedge \beta$  je dobre uzátvorkovaný výraz  
 $\wedge$  za  $\beta$  nasleduje  $|$

(iii) **posledná**

znak pred  $\beta$  je  $|$

<sup>11</sup>Z podmienky hovoriacej, že  $\beta$  musí byť validný regex, vyplýva, že prislúchajúca  $)$  sa bude nachádzať až za metaznakom  $|$

$\wedge \beta$  je suffix  $\alpha$  alebo za  $\beta$  nasleduje ) <sup>12</sup>

**Lema 2.4.10.** Alternácia používa iba alternovateľné regexy.

*Dôkaz.* Ukážeme indukciou na počet alternácií v regexe.

Majme regex  $\alpha \in nLEregex$ , ktorý obsahuje alternáciu. Bez újmy na všeobecnosti nech je to práve 1 alternácia (potenciálne zložená z viacerých metaznakov  $|$ ). Podľa tabuľky priorít vieme, že alternácia má najnižšiu prioritu. Regex  $\alpha$  môže byť v takýchto tvaroch:

1.  $\beta_1 | \dots | \beta_n$
2.  $\alpha_1(\beta_1 | \dots | \beta_n)\alpha_2$

Pre  $\alpha_1, \alpha_2, \beta_i \in nLEregex$   $i \in \{1, \dots, n\}$ . Je zrejmé, že  $\beta_1$  spĺňa podmienku prvého alternovateľného regexu,  $\beta_i$  pre  $i \in \{2, \dots, n-1\}$  spĺňajú podmienku pre stredný alternovateľný regex a  $\beta_n$  spĺňa podmienku pre posledný alternovateľný regex.

Nech platí pre regexy obsahujúce  $m-1$  alternácií, že tieto alternácie používajú iba alternovateľné regexy. Zoberme teraz regex  $\alpha \in nLEregex$  obsahujúci  $m$  alternácií. Opäť vieme  $\alpha$  rozdeliť buď spôsobom 1. alebo 2. na validné regexy z  $nLEregex$ . Každý z  $\beta_1, \dots, \beta_n, \alpha_1, \alpha_2$  obsahuje najviac  $m-1$  alternácií, teda alternácie v týchto regexoch používajú iba alternovateľné regexy. Navyše je zrejmé, že aj regexy  $\beta_1, \dots, \beta_n$  majú tvar alternovateľných regexov.  $\square$

Na základe definovania pojmu alternovateľnosti vieme jednoznačne určiť, ktoré regexy patria ku ktorej alternácii. Dôležité je, že to vieme naprogramovať do Turingovho stroja – pre každú alternáciu bude skúmať, či je ohraničená zátvorkami alebo rozdeľuje celý regex na časti (teda či nastáva prípad 1. alebo 2. z dôkazu predchádzajúcej vety).

V nasledujúcej definícii kroku výpočtu budeme používať v konfiguráciách poschodové symboly. Informáciu o tom, aké písmenko na danej pozícii leží budeme potrebovať neustále – tá bude aj súčasťou poschodovej varianty písmenka. Špeciálna informácia vo vyšších poschodiach bude patriť k spätným referenciám a lookaroundu. Už z definície spätných referencií vieme, že informácia o tom, aké podslovo predstavuje  $\backslash k$  získame až počas výpočtu na konkrétnom slove. Referencia  $\backslash k$  predstavuje posledné podslovo

<sup>12</sup>To isté ako pri podmienke (i) – prislúchajúca ( sa musí nachádzať pred metaznakom  $|$  (susediacim s  $\beta$ )

matchované  $k$ -tymi zátvorkami. Preto informácia o jeho pozícii musí byť súčasťou konfigurácie. Zároveň potrebujeme informáciu o každom lookaheade/lookbehinde – a to pozíciu v slove, kde začal matchovanie, aby sme sa po jeho vykonaní mohli na túto pozíciu vrátiť. Opäť to je informácia, ktorú získame až pri výpočte na konkrétnom vstupe a preto si ju uchováme do poschodového symbolu.

Každý regex má konečný počet operácií, preto aj poschodí v poschodových symboloch bude konečne veľa. Pre každú spätnú referenciu potrebujeme 2 miesta (začiatok a koniec podslova) a pre každý lookaround 1 miesto.

Poschodové symboly sú veľmi prehľadné, čo sa týka zápisu konfigurácií. Z pohľadu výpočtu nejakého Turingovho stroja však môžu byť veľmi nepraktické. Pokiaľ nepoznáme regex dopredu, je to o to ťažšia interpretácia. Preto vo výpočtoch Turingovho stroja necháme vstup nedotknutý a informáciu pre jednotlivé operácie budeme reprezentovať vo forme adries ukazujúcich na konkrétnu pozíciu v slove.

**Definícia 2.4.11.** *Krok výpočtu* je relácia  $\vdash$  na konfiguráciách definovaná nasledovne (najprv napíšeme, čoho sa jednotlivé kroky týkajú a potom uvedieme formálny zápis):

- I. *Prečítame rovnaké písmeko v regexe aj v slove.*
- II. *V regexe ukazujeme na (, ktorá je  $k$ -ta indexovateľná. V slove si zaznačíme do poschodového symbolu, že na tejto pozícii začína podslovo ku  $k$ -tym zátvorkám. V regexe prejdeme za zátvorku.*  
*Pokiaľ za  $k$ -tymi zátvorkami nasleduje \*, môžeme sa rozhodnúť ich celé preskočiť (až za \*). V tom prípade si na poschodový symbol zaznačíme začiatok aj koniec podslova.*
- III. *V regexe ukazujeme na ), ktorá je  $k$ -ta indexovateľná. V slove si zaznačíme do poschodového symbolu, že znak na tejto pozícii už do podslova ku  $k$ -tym zátvorkám nepatrí. V regexe prejdeme za zátvorku.*
- IV. *Narazili sme na začiatok alternácie (začiatok jej prvého alternovateľného regexu). Musíme si zvoliť jednu z jej možností. Buď budeme plynule pokračovať ďalej a spracovávať prvý alternovateľný regex alebo skočíme na začiatok ľubovoľného ďalšieho alternovateľného regexu z tejto alternácie.*
- V. *V alternácii sme práve dokončili matchovanie jednej z možností – ukazujeme na metaznak /. Skočíme v regexe za posledný alternovateľný regex.*

- VI. Ukazujeme na Kleeneho \*, ktorej predchádza písmenko. Máme 2 možnosti. Buď sa rozhodneme pokračovať – preskočíme v regexe \* – alebo spravíme ďalšiu iteráciu – skočíme pred písmenko.
- VII. Ukazujeme na Kleeneho \*, ktorej predchádzajú k-te zátvorky. Možnosti sú rovnaké ako v prípade s písmenkom. Rozdiel je len v tom, ak sa rozhodneme spraviť ďalšiu iteráciu. Potom musíme zmazať predošlý záznam k podslovu ku k-tým zátvorkám a zaznačiť si túto pozíciu ako jeho začiatok.
- VIII. Narazili sme na znak spätných referencií \k. Do slova si na túto pozíciu umiestníme pomocný ukazovateľ T.
- IX. Stále ukazujeme na \k a v slove už máme umiestnený T a písmenko za ním neobsahuje značku konca podslova ku k-tým zátvorkám. Porovnávame písmenká v slove na pozíciách s normálnym a pomocným ukazovateľom. Pokiaľ sa zhodujú, posunieme obe pozície o 1 ďalej.
- X. Stále ukazujeme na \k, v slove je umiestnený T a ukazuje na písmenko zo značkou konca podslova ku k-tým zátvorkám. Odstránime zo slova T a v regexe sa posunieme za \k.
- XI. Narazili sme na pozitívny lookahead. Zaznačíme si do poschodového symbolu v slove, že na tejto pozícii začína a v regexe skočíme do lookaheadu.
- XII. Skončili sme matchovanie lookaheadu, ukazujeme na jeho ). V slove skočíme na zaznačený začiatok a vymažeme túto značku. V regexe skočíme za ).
- XIII. Narazili sme na pozitívny lookbehind. Zaznačíme si do poschodového symbolu v slove, že na tejto pozícii začína. V slove skočíme o niekoľko symbolov nazad (ľubovoľná pozícia medzi začiatkom slova a súčasnou, vrátane týchto 2) a v regexe skočíme do lookbehindu.
- XIV. Skončili sme matchovanie lookbehindu, ukazujeme na jeho ). Zároveň v slove ukazujeme na jeho zaznačený začiatok. Zmažeme túto značku a v regexe prejdeme za ).
- XV. Narazili sme na negatívny lookahead. Pokiaľ neexistuje postupnosť konfigurácií taká, že regex v negatívnom lookaheadu by matchoval nejaký prefix slova začínajúc od súčasnej pozície v slove, potom môžeme lookahead preskočiť.

XVI. Narazili sme na negatívny lookbehind. Pokiaľ neexistuje postupnosť konfigurácií taká, že regex v negatívnom lookbehinde by matchoval nejaký sufix slova končiac na súčasnej pozícii v slove, potom môžeme lookahead preskočiť.

Formálny zápis:

$$I. \forall a \in \Sigma : (r_1 \dots \lceil a \dots r_n, w_1 \dots \lceil a \dots w_m) \vdash (r_1 \dots a \lceil \dots r_n, w_1 \dots a \lceil \dots w_m)$$

$$II. \text{Nech } ( \text{ je indexovateľná, } k\text{-ta v poradí: } (r_1 \dots \lceil (\dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$(1) \vdash (r_1 \dots (\lceil \dots r_n, w_1 \dots \lceil w_j^k \dots w_m)$$

Ak za jej uzatváracou zátvorkou nasleduje \*, t.j.  $\alpha$  je tvaru  $r_1 \dots \lceil (\dots) * \dots r_n$ , potom

$$(2) \vdash (r_1 \dots (\dots) * \lceil \dots r_n, w_1 \dots \lceil w_j^{k'} \dots w_m)$$

$$III. \text{Nech } ) \text{ je indexovateľná, } k\text{-ta v poradí:}$$

$$(r_1 \dots \lceil) \dots r_n, w_1 \dots \lceil w_j \dots w_m) \vdash (r_1 \dots) \lceil \dots r_n, w_1 \dots \lceil w_j^{k'} \dots w_m)$$

$$IV. \text{Nech podslová } \alpha_1, \alpha_2, \dots, \alpha_A \text{ regexu } \alpha \text{ sú všetkými členmi zobrazenej alternácie:}$$

$$(r_1 \dots \lceil \alpha_1 | \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$(1) \vdash \text{ďalší prechod v } \alpha_1$$

$$(2) \vdash (r_1 \dots \alpha_1 | \lceil \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

...

$$(A) \vdash (r_1 \dots \alpha_1 | \alpha_2 | \dots | \lceil \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$V. \text{Nech podslová } \alpha_1, \alpha_2, \dots, \alpha_A \text{ regexu } \alpha \text{ sú všetkými členmi zobrazenej alternácie, potom pre všetky možnosti:}$$

$$(r_1 \dots \alpha_1 \lceil | \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m),$$

$$(r_1 \dots \alpha_1 | \alpha_2 \lceil | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

...

$$(r_1 \dots \alpha_1 | \alpha_2 | \dots | \alpha_{A-1} \lceil | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots \alpha_1 | \alpha_2 | \dots | \alpha_{A-1} | \alpha_A \lceil \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

VI.  $(r_1 \dots a \lceil * \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

$$(1) \vdash (r_1 \dots a * \lceil \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$(2) \vdash (r_1 \dots \lceil a * \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

VII.  $(r_1 \dots (\underset{k}{r_i} \dots \underset{k}{r_l}) \lceil * \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)^{13}$

$$(1) \vdash (r_1 \dots (\underset{k}{r_i} \dots \underset{k}{r_l}) * \lceil \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$$

$$(2) \vdash (r_1 \dots (\lceil \underset{k}{r_i} \dots \underset{k}{r_l}) * \dots r_n, w_1 \dots w_a \dots w_b \dots \lceil \overset{k}{w_j} \dots w_m)$$

VIII.  $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$

$$\vdash (r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \top \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$$

IX.  $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots \top w_c \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$ , kde  $a \leq c < b$  a zároveň  $w_c = w_j^{14}$

$$\vdash (r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots w_c \top \dots \overset{k'}{w_b} \dots w_j \lceil \dots w_m)$$

X.  $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots \top \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$

$$\vdash (r_1 \dots \backslash k \lceil \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots w_j \lceil \dots w_m)$$

XI. *Nech  $(?= \dots)$  je  $k$ -ty pozitívny lookahead v poradí:*

$$(r_1 \dots \lceil (?= \dots) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots (?= \lceil \dots) \dots r_n, w_1 \dots \lceil \overset{k \rightarrow}{w_j} \dots w_m)$$

XII. *Nech  $)$  patrí ku  $k$ -temu pozitívnemu lookaheadu v poradí:*

$$(r_1 \dots (?= \dots \lceil) \dots r_n, w_1 \dots \overset{k \rightarrow}{w_l} \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots (?= \dots) \lceil \dots r_n, w_1 \dots \lceil w_l \dots w_j \dots w_m)$$

<sup>13</sup>Podľa definície spätných referencií platí podsledné podslovo nájdené regexom v  $k$ -tych zátvorkách.

Pri tejto pracovnej pozícii v regexe je zrejmé, že nejde o prvý prechod cez tieto zátvorky a teda existuje také  $a, b$ , že  $k$  je v slove nad  $w_a$  a  $k'$  nad  $w_b$ . Ak nastane prechod (2), pôvodné horné indexy  $k, k'$  miznú a pridáva sa  $k$  nad  $w_j$ .

<sup>14</sup> $w_c$  a  $w_j$  môžu byť poschodové symboly, avšak pri tejto rovnosti poschodia ignorujeme – chceme porovnať iba písmenká v slove, prislúchajúce týmto pozíciám.

XIII. *Nech  $(?<=\dots)$  je  $k$ -ty pozitívny lookbehind v poradí,  $\forall L \in \{0, \dots, j-1\}$ :*

$$(r_1 \dots \lceil (?<=\dots) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots (?<=\lceil \dots) \dots r_n, w_1 \dots \lceil w_{j-L} \dots \overset{k}{w}_j \dots w_m)$$

XIV. *Nech  $\lceil$  patrí ku  $k$ -temu pozitívnemu lookbehindu v poradí:*

$$(r_1 \dots (?<=\dots \lceil) \dots r_n, w_1 \dots \lceil \overset{k}{w}_j \dots w_m)$$

$$\vdash (r_1 \dots (?<=\dots) \lceil \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

XV. *Ak  $\nexists p \in \{j, \dots, m\} : (\lceil r_k \dots r_l, \lceil w_j \dots w_p) \vdash^* (r_k \dots r_l \lceil, w_j \dots w_p \lceil)$ , potom:*

$$(r_1 \dots \lceil (?! r_k \dots r_l) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots (?! r_k \dots r_l) \lceil \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

XVI. *Ak  $\nexists p \in \{1, \dots, j-1\} : (\lceil r_k \dots r_l, \lceil w_p \dots w_{j-1}) \vdash^* (r_k \dots r_l \lceil, w_p \dots w_{j-1} \lceil)$ , potom:*

$$(r_1 \dots \lceil (?<! r_k \dots r_l) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

$$\vdash (r_1 \dots (?<! r_k \dots r_l) \lceil \dots r_n, w_1 \dots \lceil w_j \dots w_m)$$

Prechody XV. a XVI. sa môžu zdať ťažkopádne, ťažko overiteľné. Nie je však šikovný spôsob ako krokmi znázorniť, že sa niečo nedá. Môžeme si to však predstaviť tak, že k vnútornému regexu máme zostrojený deterministický Turingov stroj, ktorý vie postupovať podľa krokov I. – XIV. a má obrátenú akceptáciu. Teda ak by chcel akceptovať, zamietne, a ak sa zasekne/zamietne, tak akceptuje.

Táto úvaha nezahŕňa vnorené negatívne lookaroundy. Potrebný Turingov stroj sa však dá zostrojiť aj tak. Keďže každý regex musí byť konečnej dĺžky, nejaký z tých negatívnych lookaroundov musí byť najviac vnorený – bez vnorených negatívnych lookaroundov. Práve k nemu vieme zostrojiť vyššie popísaný Turingov stroj. Potom Turingov stroj pre negatívny lookaround, ktorý ho obaľuje bude spúšťať tento Turingov stroj a tak ďalej, vieme zostrojovať Turingove stroje zvnútra von. Pri výpočte sa potom budú navzájom volať zvonka smerom dnu.

**Definícia 2.4.12.** *Jazyk generovaný regexom  $\alpha$  je množina*

$$L(\alpha) = \{w \mid \text{platí } (\lceil \alpha, \lceil w) \vdash^* (\alpha \lceil, w \lceil)\}$$

**Poznámka.** Postupnosť konfigurácií  $(\lceil \alpha, \lceil w) \vdash^* (\alpha \lceil, w \rceil)^{15}$  pre daný regex  $\alpha$  a slovo  $w$  nazývame **akceptačný výpočet**. Konfiguráciu  $(\lceil \alpha, \lceil w)$  nazývame počiatočnou a  $(\alpha \lceil, w \rceil)$  akceptačnou konfiguráciou.

**Lema 2.4.13.** *Nech  $\alpha \in \mathcal{L}_{ERE}$  a  $w \in L(\alpha)$ . Potom existuje akceptačný výpočet, ktorý má najviac  $3 \cdot |\alpha| \cdot |w|^2$  konfigurácií.*

*Dôkaz.* Pri prechodoch medzi konfiguráciami I., II., III., IV., V., VI.(1), VII.(1), IX., XI., XIV., XV. a XVI. sa ukazovateľ posúva vždy vpred buď v slove alebo v regexe alebo v oboch. Keby sme využívali iba tieto prechody, akceptačný výpočet má najviac  $|\alpha| + |w|$  konfigurácií.

Existujú prechody, pri ktorých sa žiaden ukazovateľ nepohne: VIII. a X. Tieto prechody nastávajú pri spätných referenciách a jedná sa o objavenie sa alebo zmiznutie pomocného ukazovateľa. Pri každej spätnej referencii nastávajú oba javy práve jedenkrát. Zároveň počet spätných referencií je najviac  $|\alpha|$ , teda tieto 2 konfigurácie sa objavia dokopy najviac  $(2 \cdot |\alpha|)$ -krát.

Akceptačný výpočet, ktorý obsahuje iba doteraz spomenuté kroky výpočtu obsahuje najviac  $|w| + 3|\alpha|$  konfigurácií. Zostali kroky výpočtu, v ktorých ukazovateľ skáče dozadu, rozoberme si ich postupne:

VI.(2), VII.(2) – v prípade Kleeného  $*$  nastáva ďalšie opakovanie. Keďže  $w \in L(\alpha)$ , existuje pre  $w$  akceptačný výpočet. Takýchto výpočtov môže byť viac, napríklad ak sa v  $\alpha$  vyskytuje regex  $(\beta)^*$  taký, že  $\varepsilon \in L(\beta)$ . Potom existuje nekonečne veľa výpočtov, ktoré sa líšia v počte prechodov týmto regexom. Z nich si vyberieme najkratší akceptačný výpočet. Vyberme si ľubovoľnú  $*$  a sled jej iterácií v nejakom momente výpočtu<sup>16</sup>, nech je ich počet  $k$ . Pokiaľ  $k = 0$ , potom je to krok II.(2) a ten sme už zarátali. V prípade  $k = 1$  opäť krok VI.(2) ani VII.(2) nenastali. Nech teda  $k > 1$ . Tvrdíme, že každá iterácia okrem poslednej v tomto slede matchovala aspoň 1 písmenko. Ukážeme to sporom. Nech nejaká  $i$ -ta,  $i < k$ , matchovala prázdne slovo. Potom ale túto operáciu možno vymazať zo sledu, napojiť konfigurácie a máme kratší výpočet, čo je spor. Tvrdenie neplatí pre poslednú iteráciu, pretože pokiaľ je iterovaný regex v zátvorkách,

<sup>15</sup>Bez ohľadu na to, čo obsahujú vyššie poschodia symbolov vo  $w$ .

<sup>16</sup>Táto  $*$  môže mať viac sledov iterácií, pokiaľ je vnútri regexu, ktorý je v poli pôsobnosti inej  $*$ . Ak táto vonkajšia  $*$  vykonáva niekoľko ( $>1$ ) iterácií, potom vnútorná  $*$  je v každej z nich spúšťaná odznova. Teda môže existovať niekoľko jej sledov iterácií.



posledná iterácia ovplyvňuje podslovo, ktoré bude vyjadrené spätnými referenciami a jej vymazanie by mohlo pokaziť akceptačný výpočet. Takýto sled využíva  $(k - 1)$ -krát prechod VI.(2) alebo VII.(2), teda prvé písmenko je zadarmo. Zároveň však platíme navyše za 1 návrat navyše v VII.(2), pokiaľ chceme mať posledné slovo  $\varepsilon$ . Predstavme si preto, že platíme za prvé písmenko a neplatíme za posledný prechod na prázdne slovo – v takomto prípade získame horný odhad počtu konfigurácií, lebo na záver nemusí byť zakaždým  $\varepsilon$  a zároveň podľa tvrdenia prvý prechod musí byť na písmenko.

Odhadnime teraz dĺžku najkratšieho akceptačného výpočtu aj s použitím krokov VI.(2) a VII.(2). Celkovo počet návratov môže byť najviac  $|w|$ , pretože každá iterácia, za ktorú platíme návratom, musí matchovať aspoň 1 písmenko. Medzi 2 návratmi používame nenávratové kroky, ktorých je najviac  $|w| + 3|\alpha|$ . Preto akceptačný výpočet obsahujúci doteraz spomenuté kroky výpočtu bude mať najviac  $|w| \cdot (|w| + 3|\alpha|) \leq 3 \cdot |w|^2 \cdot |\alpha|$  konfigurácií.  $\square$

Príkladom toho, že táto lema poskytuje tesný odhad, je jazyk  $(a * b * c * d * e^*)^*$  a slovo  $edcba$ . V najkratšom akceptačnom výpočte v každom opakovaní  $*$  naozaj nájdeme zhodu práve s jedným písmenom.

V praxi často regex dopredu nepoznáme. Vstupným údajom je text na vyhľadávanie a rovnako aj regex, ktorý znázorňuje požiadavku na vyhľadávanie. Tento problém predstavuje jazyk

$$L(regex\#word) = \{word \mid word \in L(regex) \wedge regex \in \mathcal{U}\}$$

kde  $\mathcal{U} \in \{Regex, Eregex, LRegex, nLRegex\}$ .

**Veta 2.4.14.**  $L(regex\#word) \in NSPACE(n \log n)$ , kde  $regex \in LRegex$ . (Presnejšie  $NSPACE(r \log w)$ , kde  $r = |regex|$  a  $w = |word|$ .)

*Dôkaz.* **TODO!!!** Zostrojíme nedeterministický Turingov stroj  $M$ , ktorý bude akceptovať jazyk  $L(regex\#word)$  a na pracovných páskach zapíše najviac  $O(r \log w)$  políčok.

Na pracovnej páske  $\square$

**Veta 2.4.15.**  $L(regex\#word) \in DSPACE(n \log^2 n)$ , kde  $regex \in LRegex$  a  $n$  je dĺžka vstupu.

*Dôkaz.* Pre účely dôkazu budeme označovať  $w = |word|$  a  $r = |regex|$ .

Zostrojíme deterministický Turingov stroj  $M$ , ktorý bude akceptovať jazyk  $L(regex\#word)$  a na každej pracovnej páske použije najviac  $O(n \log^2 n)$  políčok.  $M$  bude mať vstupnú read-only pásku a 2 pracovné pásky.

Myšlienka je podobná dôkazu Savitchovej vety [Ďu03] –  $M$  bude vykonávať funkciu  $TESTUJ(C_1, C_2, i)$ , ktorá zistí, či sa vieme dostať z konfigurácie  $C_1$  do konfigurácie  $C_2$  na  $i$  krokov.

Podľa lemy 2.4.13 vieme, že ak existuje akceptačný výpočet pre  $w$ , potom existuje aj akceptačný výpočet taký, ktorý má najviac  $3rw^2$  konfigurácií. Na základe tohto výsledku bude  $M$  zisťovať, či sa z počiatočnej konfigurácie  $C_0$  vieme dostať do akceptačnej konfigurácie  $C_a$  na  $3rw^2$  krokov –  $TESTUJ(C_0, C_a, 3rw^2)$ .

Pseudokód procedúry  $TESTUJ$ :

```

1  bool TESTUJ( $C_1, C_2, i$ )
2      if ( $C_1 == C_2$ ) then return true
3      if ( $i > 0 \wedge C_1 \vdash C_2$ ) then return true
4      if ( $i \leq 1$ ) return false
5      iteruj cez všetky konfigurácie  $C_3$ 
6          if ( $TESTUJ(C_1, C_3, \lfloor \frac{i}{2} \rfloor) \wedge TESTUJ(C_3, C_2, \lceil \frac{i}{2} \rceil)$ ) then return true
7      return false

```

Pre podrobnejší popis pseudokódu je nutné, aby sme najprv definovali tvar konfigurácie. Použijeme zápis z definície 2.4.11, v tvare  $(a_1 \dots \lceil a_i \dots a_r, b_1 \dots \lceil b_j \dots b_w)$  respektíve  $(a_1 \dots \lceil a_i \dots a_r, b_1 \dots \lceil b_j \dots b_w)$ , kde  $a_1 \dots a_r = regex$  a  $b_1 \dots b_w = word$ . Reprezentovať ich budeme vo forme 2 resp. 3 adries – pracovná pozícia v regexe ( $i$ ), pracovná pozícia v slove ( $j$ ) a adresa k spätnej referencii ( $l$ ). Namiesto poschodových symbolov si  $M$  bude pre každú konfiguráciu pamätať informáciu, ktoré zátvorky zodpovedajú ktorému podslovu slova  $word$ . Pre každé zátvorky si uloží 2 adresy – začiatok podslova a 1 políčko za koncom podslova (použijeme polootvorený interval  $\langle zač, kon \rangle$ ). Pre každý lookahead a každý lookbehind bude mať vyhradené 1 adresné miesto aby si zapamätal, kam do slova ukazoval, keď naňho narazil.

Popis pseudokódu:

**riadok 2** Ak  $C_1 = C_2$ , potom vieme prejsť z  $C_1$  do  $C_2$  na ľubovoľný počet krokov.

**riadok 3** Platí  $C_1 \neq C_2$ . Ak  $i = 0$ , nevieme prejsť do žiadnej inej konfigurácie ako  $C_1$ , teda vrátime **false**. Nech  $i > 0$ . Skontrolujeme podľa definície 2.4.11, či platí  $C_1 \vdash C_2$ . Konfigurácie sú uložené na páske ako  $m$ -tice, kde  $m = 3 + 2 \cdot (\text{počet zátvoriek}) + (\text{počet lookaheadov}) + (\text{počet lookbehindov})$ :  $C_1 = (d_1, \dots, d_m)$ ,  $C_2 = (e_1, \dots, e_m)$ . Nech  $d_1, e_1$  sú pracovné pozície v regexe,  $d_2, e_2$  sú pracovné pozície v slove a  $d_3, e_3$  pracovné pozície ukazovateľa pre spätné referencie, pričom  $d_3, e_3$  môžu byť nedefinované (na prislúchajúcom mieste medzi oddeľovačmi adries nebude nič zapísané). TS M overí, či je splnená nejaká z týchto podmienok (rímske čísla zodpovedajú tým v definícii 2.4.11):

- I.  $regex[d_1] = word[d_2] \wedge (e_1, \dots, e_m) = (d_1 + 1, d_2 + 1, nedef, d_4, \dots, d_m)$
- II. (1)  $regex[d_1] = '('$   
 $\wedge regex[d_1]$  je indexovateľná  
 $\wedge$  nech  $d_k$  prislúcha k  $regex[d_1]$ ,  $4 \leq k \leq m$ :  $e_k = d_2$  (sedí začiatok podslova)  
 $\wedge (e_1, \dots, e_m) = (d_1 + 1, d_2, nedef, d_4, \dots, d_{k-1}, d_2, d_{k+1}, \dots, d_m)$
- II. (2)  $regex[d_1] = '('$   
 $\wedge$  2 políčka pred pozíciou  $e_2$  je v regexe  $'^*)'$   
 $\wedge$  zátvorky  $regex[d_1]$  a  $regex[e_1 - 2]$  sú k sebe prislúchajúce<sup>17</sup>  
 $\wedge$  nech  $d_k$  prislúcha k  $regex[d_1]$ ,  $4 \leq k \leq m$ :  $e_k = e_{k+1} = d_2$  (začiatok a koniec podslova je nastavený na to isté políčko)  
 $\wedge (e_1, \dots, e_m) = (d_1 + l, d_2, nedef, d_4, \dots, d_{k-1}, e_k, e_{k+1}, d_{k+2}, \dots, d_m)$  pre  $l \in \mathbb{N}$
- III. podobne ako II.(1) – kontrola, či sedí koniec podslova
- IV.  $regex[e_1 - 1] = '|'$   
 $\wedge$  regex medzi  $d_1$  a najbližším metaznakom  $|$  je alternovateľný  
 $\wedge$  všetky regexy ohraničené  $|$  medzi  $d_1$  a  $e_1$  sú alternovateľné  
 $\wedge (e_1, \dots, e_m) = (d_1 + l, d_2, nedef, d_4, \dots, d_m)$  pre  $l \in \mathbb{N}$
- V.  $regex[d_1] = '|'$   
 $\wedge$  všetky regexy ohraničené  $|$  medzi  $d_1$  a  $e_1$  sú alternovateľné  
 $\wedge$  regex medzi  $e_1$  a najbližším metaznakom  $|$  naľavo od neho je alternovateľný  
 $\wedge (e_1, \dots, e_m) = (d_1 + l, d_2, nedef, d_4, \dots, d_m)$  pre  $l \in \mathbb{N}$

<sup>17</sup>To M skontroluje tak, že si overí, že medzi nimi ku každej  $'('$  existuje  $')$  – teda počet výskytov  $'('$  a  $')$  musí byť rovnaký.

- VI. (1)  $regex[d_1] = '*' \wedge (e_1, \dots, e_m) = (d_1 + 1, d_2, nedef, d_4, \dots, d_m)$
- VI. (2)  $regex[d_1] = '*'$   
 $\wedge regex[d_1 - 1] = a, a \in \Sigma$   
 $\wedge (e_1, \dots, e_m) = (d_1 - 1, d_2, nedef, d_4, \dots, d_m)$
- VII. (1)  $regex[d_1] = '*' \wedge (e_1, \dots, e_m) = (d_1 + 1, d_2, nedef, d_4, \dots, d_m)$
- VII. (2)  $regex[d_1] = '*'$   
 $\wedge regex[e_1 - 1] = '('$  a prislúcha k  $regex[d_1]$   
 $\wedge$  nech  $d_k, d_{k+1}$  prislúchajú k týmto zátvorkám, potom  $e_k = d_2, e_{k+1} = nedef$   
 $\wedge (e_1, \dots, e_m) = (d_1 - l, d_2, nedef, d_4, \dots, d_{k-1}, e_k, e_{k+1}, d_{k+2}, \dots, d_m)$  pre  
 $l \in \mathbb{N}$
- VIII.  $regex[d_1] = '\backslash'$   
 $\wedge$  nasleduje číslo  $k, 0 \leq k \leq$  (počet indexovateľných zátvoriek)  
 $\wedge (e_1, \dots, e_m) = (d_1, d_2, d_{2k+2}, d_4, \dots, d_m)$  ( $d_{2k+2}$  je adresa začiatku podslova prislúchajúca ku  $k$ -tým zátvorkám)
- IX. na pozícii  $d_1$  je podslovo  $'\backslash k'$   
 $\wedge$  podobne ako I.– musí platiť  $d_3 < d_{2k+3}$  a správne sa posunú adresy  $d_2, d_3$
- X. na pozícii  $d_1$  je podslovo  $'\backslash k'$   
 $\wedge d_3 = d_{2k+3}$   
 $\wedge (e_1, \dots, e_m) = (d_1, d_2, nedef, d_4, \dots, d_m)$
- XI. na pozícii  $d_1$  je podslovo  $'(=?'$   
 $\wedge$  nech  $d_l$  adresa prislúchajúca k tomuto lookaheadu:  $e_l = d_2$ , teda  
 $(e_1, \dots, e_m) = (d_1 + 3, d_2, nedef, d_4, \dots, d_{l-1}, d_2, d_{l+1}, \dots, d_m)$
- XII.  $regex[d_1] = ')'$  a prislúcha lookaheadu, ktorému patrí adresa  $d_l$   
 $\wedge (e_1, \dots, e_m) = (d_1 + 1, d_l, nedef, d_4, \dots, d_{l-1}, nedef, d_{l+1}, \dots, d_m)$
- XIII. na pozícii  $d_1$  je podslovo  $'(?<='$   
 $\wedge$  nech  $d_l$  adresa prislúchajúca k tomuto lookbehindu:  $e_l = d_2$   
 $\wedge (e_1, \dots, e_m) = (d_1 + 3, d_2 - l, nedef, d_4, \dots, d_{l-1}, d_2, d_{l+1}, \dots, d_m)$ , kde  $l \in \mathbb{N}$
- XIV.  $regex[d_1] = ')'$  a prislúcha lookbehindu, ktorému patrí adresa  $d_l$   
 $\wedge d_2 = d_l$   
 $\wedge (e_1, \dots, e_m) = (d_1 + 1, d_2, nedef, d_4, \dots, d_{l-1}, nedef, d_{l+1}, \dots, d_m)$

Keďže regex neobsahuje negatívny lookbaround, ďalšie riadky z definície netes-

tujeme.

**riadok 4** Po predošlých riadkoch platí  $C_1 \neq C_2 \wedge C_1 \not\prec C_2$ . Ak zároveň  $i \leq 1$ , potom sa z  $C_1$  do  $C_2$  na  $i$  krokov dostať nedokážeme. *TESTUJ* vráti **false**.

**riadok 5** M začne iterovať cez všetky možné konfigurácie – t.j. všetky možné kombinácie adries z množiny  $\{1, \dots, d\}$ , kde  $d = \lceil \log(r+1) \rceil$  pre  $d_1$  a  $d = \lceil \log(w+1) \rceil$  pre ostatné adresy. Vygenerované  $C_3$  bude mať M uložené ako lokálnu premennú.

**riadok 6** M testuje, či je  $C_3$  vo výpočte v strede medzi  $C_1$  a  $C_2$ . Ak obe volané procedúry vrátia **true**, vrátime **true**. Inak sa M vráti na **riadok 5** a vygeneruje ďalšie  $C_3$ .

**riadok 7** Neexistuje vhodná konfigurácia  $C_3$ , teda sa nevieme dostať z  $C_1$  do  $C_2$  na  $i$  krokov. Vrátime **false**.

Podľa vyššie špecifikovaných  $m$ -tíc pre konfigurácie bude úvodné nastavenie nasledovné:

$$C_0 = (1, 1, \text{ndef}, \text{ndef}, \dots, \text{ndef}, \text{ndef}, \dots, \text{ndef})$$

$$C_a = (r+1, w+1, \text{ndef}, \underbrace{\text{ndef}, \dots, \text{ndef}}_{\substack{\text{adresy pre} \\ \text{spätné} \\ \text{referencie}}}, \underbrace{\text{ndef}, \dots, \text{ndef}}_{\substack{\text{adresy pre} \\ \text{lookahead a lookbehind}}})$$

Turingov stroj M rekurzívne volá procedúru *TESTUJ*. Preto na prvej páske bude mať zásobník, kde budú uložené záznamy o jednotlivých volaniach. Druhá páska je pomocná pri vykonávaní konkrétneho volania – M potrebuje konštantný počet adries, aby mohol realizovať porovnávanie (overovanie rovnosti konfigurácií a adries, overovanie konkrétneho rozdielu medzi adresami), zisťovanie príslušnosti zátvoriek (t.j. ktorá je druhá zátvorka k tejto – treba počítat zátvorky (,)), zisťovanie poradia zátvorky-/lookaheadu/lookbehindu a prislúchajúcej adresy, kontrolu alternovateľnosti (to je v podstate počítanie zátvoriek), čítanie čísla  $k$  za metaznakom \ a hľadanie adries pre  $k$ -te zátvorky, ...

M potrebuje vykonávať nasledovné pomocné operácie:

- overovanie rovnosti – pri porovnávaní adries zapíšeme 1

Zrejme ak existuje akceptačný výpočet, M ho nájde. Treba ukázať, že sa pri tom na každej páske zmestí do pamäte  $O(n \log^2 n)$ . Podľa predchádzajúceho odstavca vieme,

že na druhej (pomocnej) páske potrebuje  $O(\log n)$  políčok, čo spĺňa podmienku. Spočítajme veľkosť potrebnú zásobníka.

Adresy vieme zapísať v logaritmickom priestore závislom od dĺžky slova, kam ukazujú. Pre regex to bude  $\log r$  a pre slovo  $\log w$ , pretože vieme adresovať od oddeľovača  $\#$ . Číslo  $i$  je najviac  $3rw^2$  a tiež ho vieme zapísať v logaritmickom tvare, čo zaberie  $\log(3rw^2) = \log 3 + \log r + 2 \log w$  políčok. Platí  $r \leq n, w \leq n$ , lebo  $r + w + 1 = n$ .

Počet zátvoriek, lookaheadov a lookbehindov je v regexe dokopy najviac  $r$ , teda jedna konfigurácia bude potrebovať najviac  $\log r + 2 \log w + 2r \log w$  priestoru.

Jeden záznam procedúry *TESTUJ* obsahuje 3 konfigurácie a číslo  $i$ , čo spolu zaberá  $\log 3 + \log r + 2 \log w + 3 \log r + 6 \log w + 6r \log w = O(r \log w) = O(n \log n)$  priestoru.

Počet záznamov na zásobníku závisí od hĺbky rekurzie. Keďže začíname na hodnote  $i = 3rw^2$  a pri každom volaní je  $i$  zmenšené na polovicu, hĺbka vnorenia bude  $\log(3rw^2) = O(\log n^3) = O(3 \log n) = O(\log n)$ .

Celkovo  $M$  na zásobníkovej páske zapíše  $O(\log n) \cdot O(n \log n) = O(n \log^2 n)$  priestoru.  $\square$

## 2.5 Popisná zložitosť moderných regulárnych výrazov

V tejto kapitole rozoberieme moderné regulárne výrazy z dvoch hľadísk. Najprv nás bude zaujímať, ako pomohli nové konštrukcie pri popise regulárnych jazykov a potom prejdeme na analýzu dĺžky výrazov pre zložitejšie jazyky.

Lookaround môže výrazne pomôcť pri definovaní konečných jazykov, napríklad regex

$$\begin{array}{c}
 ((? = \underbrace{(? = (a^m) * \$)}_{\text{generuje}}) \underbrace{(a^{m+1}) * a\{1, m-1\} \$ | a^m \$)}_{\text{generuje celé } a^*, \text{ okrem:}}) a^m) + \\
 \underbrace{a^{km}, k \in \mathbb{N} \quad a^{m+1}, a^{m(m+1)l}, l \in \mathbb{N}}_{\text{generuje } a^{km} \text{ také, že nevie } a^{m(m+1)l}, l \in \mathbb{N}}
 \end{array}$$

generuje konečný jazyk obsahujúci slová  $a^m, a^{2m}, \dots, a^{(m-1)(m+1)}$ . Hlavný lookahead je spúšťaný každú iteráciu, teda pre slovo  $a^{zm}$  musí matchovať všetky  $a^{im}$  pre  $i \in \{1, \dots, z\}$ .

# Záver

Zhrnutie na záver...

# Literatúra

- [CN09] Benjamin Carle and Paliath Nadendran. On extended regular expressions. In *Language and Automata Theory and Applications*, volume 3, pages 279–289. Springer, April 2009. [http://www.cs.albany.edu/~dran/my\\_research/papers/LATA\\_version.pdf](http://www.cs.albany.edu/~dran/my_research/papers/LATA_version.pdf) [Online; accessed 15-April-2015].
- [CSY03] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. <http://www.worldscientific.com/doi/abs/10.1142/S012905410300214X> [Online; accessed 15-April-2015].
- [EKSW04] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-wei Wang. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.*, 9(2-3):233–256, September 2004. <https://cs.uwaterloo.ca/~shallit/Papers/re3.pdf> [Online; accessed 15-April-2015].
- [EZ76] Andrzej Ehrenfeucht and Paul Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976. [http://digitool.library.colostate.edu///exlibris/dtl/d3\\_1/apache\\_media/L2V4bGlicmlzL2R0bC9kM18xL2FwYWNoZV9tZWRpYS8xNjYzMDQ=.pdf](http://digitool.library.colostate.edu///exlibris/dtl/d3_1/apache_media/L2V4bGlicmlzL2R0bC9kM18xL2FwYWNoZV9tZWRpYS8xNjYzMDQ=.pdf) [Online; accessed 15-April-2015].
- [HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.
- [HURM78] John E. Hopcroft, Jeffrey D. Ullman, Branislav Rovan, and Peter Mi-



- kulecký. *Formálne jazyky a automaty*. 1.vydanie Alfa, Bratislava, 1978.  
Prel. z: Formal languages and their relation to automata.
- [Pyt12] Python Software Foundation. *Regular expression operations*, 2012.  
<https://docs.python.org/2/library/re.html> [Online; accessed 15-April-2015].
- [RF13] Branislav Rován and Michal Forišek. Formálne jazyky a automaty (skriptá), 2013. <http://foja.dcs.fmph.uniba.sk/materialy/skripta.pdf> [Online; accessed 18-April-2015].
- [Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
- [Tó13] Tatiana Tóthová. Moderné regulárne výrazy. Bachelor’s thesis, FMFI UK, 2013. <https://github.com/tatianka/bak> [Online; accessed 15-April-2015].
- [Ďu03] Pavol Ďuriš. Výpočtová zložitosť (materiály k prednáške), 2003. [http://www.dcs.fmph.uniba.sk/zlozitost/data/zlozitost\\_duris.pdf](http://www.dcs.fmph.uniba.sk/zlozitost/data/zlozitost_duris.pdf) [Online; accessed 10-April-2015].