Univerzita Komenského, Bratislava
Fakulta Matematiky, Fyziky a Informatiky

Moderné regulárne výrazy

Bakalárska práca

2013
Tatiana Tóthová

Univerzita Komenského, Bratislava
Fakulta Matematiky, Fyziky a Informatiky

Moderné regulárne výrazy

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky
Školiteľ:	RNDr. Michal Forišek, PhD.

Bratislava, 2013
Tatiana Tóthová

83491340

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

Poďakovanie

Tatiana Tóthová

Abstrakt

Abstrakt po slovensky

Kľúčové slová: napíšme, nejaké, kľúčové, slová

Abstract

Abstract in english

Key words: some, key, words

Obsah

Úvod

Bla bla úvodné info o regulárnych výrazoch: - vzniklo ako teória - niekto implementoval do textových editorov, neskôr Thomson do Unixu - odtiaľ sa rozšírili do programovacích jazykov - programovacie jazyky rozširujú svoju funkcionalitu, to platí aj pre časť regulárnych výrazov. Rozšírením o nové konštrukcie už nemusia patriť do triedy regulárnych jazykov. Mnohé konštrukcie sú len kozmetické úpravy a pomôcky, ktoré nezosilnia daný model. Zaujímavé sú tie, ktoré už pomôžu vytvoriť (akceptovať) jazyky z vyšších tried Chomského hierarchie. - Zaradením triedy jazykov aktuálneho modelu do Chomského hierarchie môže dopomôcť pri implementácii jednotlivých operácií. Trieda regulárnych jazykov vystačí s ľahko naprogramovateľnými konečnými automatmi, avšak vyššie triedy vyžadujú backtracking, ktorý samozrejme znamená väčšiu časovú zložitosť.

Používané pojmy a skratky

L_1L_2 – zreťazenie 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, zároveň trieda jazykov tvorená regexami

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

$\mathscr{L}_{CS}$ – trieda bezkontextových jazykov

DKA/NKA – deterministický/nedeterministický konečný automat

LBA – lineárne ohraničený Turingov stroj

regex – regulárny výraz, ktorý môže vytvoriť najviac regulárny jazyk (základná definícia)

TS – Turingov stroj

e-regex – regex so spätnými referenciami

le-regex – e-regex s operáciami lookahead a lookbehind

nle-regex – le-regex s operáciami negatívny lookahead a negatívny lookbehind

Eregex – trieda jazykov tvorená e-regexami

LEregex – trieda jazykov tvorená le-regexami

nLEregex – trieda jazykov tvorená nle-regexami

lookaround – spoločný názov pre lookahead a lookbehind

nsn – najmenší spoločný násobok

matchovať – keď regex matchuje slovo, vyhlási zhodu, znamená to, že patrí do jeho jazyka

1 Definície a známe výsledky

...

1.1 Základná definícia regulárnych výrazov

Keďže implementované regulárne výrazy sa už natoľko líšia od počiatočného teoretického modelu, zaužíval sa pre ne názov **regexy**. Budeme ho používať aj my a prípadnými predponami budeme rozlišovať, ktorú množinu operácií práve myslíme.

Pojem **regex** bude slúžiť na pomenovanie regulárnych výrazov, ktoré pokrývajú triedu regulárnych jazykov. Z tohto dôvodu nebudem vytvárať zvlášť pomenovanie pre túto triedu, bolo by to zbytočné. Pre ozrejmenie uvediem základnú definíciu regexu z článku [CSY03]. Niektoré konštrukcie sú oproti teoretickému modelu nové, ale dôkaz toho, že pokrýva stále rovnakú triedu jazykov, je triviálny.

Základná forma regexov

(1) Pre každé $a \in \Sigma$, a je regex a $L(a) = \{a\}$. Poznamenajme, že pre každé $x \in \{(,), \{, \}, [,], \$, |, \backslash, ., ?, *, +\}$, $\backslash x \in \Sigma$ a je regexom a $L(\backslash x) = \{x\}$. Naviac aj $\backslash n$ a $\backslash t$ patria do Σ a oba sú regexami. $L(\backslash n)$ a $L(\backslash t)$ popisujú jazyky skladajúce sa z nového riadku a tabulátora.

(2) Pre regexy e_1 a e_2

$(e_1)(e_2)$ (zreťazenie),

$(e_1)|(e_2)$ (alternácia), a

$(e_1)*$ (Kleeneho uzáver)

sú regexy, kde $L((e_1)(e_2)) = L(e_1)L(e_2)$, $L((e_1)|(e_2)) = L(e_1) \cup L(e_2)$ a $L((e_1)*) = (L(e_1))^*$. Okrúhle zátvorky môžu byť vynechané. Ak sú vynechané, alternácia, zreťazenie a Kleeneho uzáver majú vyššiu prioritu.

(3) Regex je tvorený konečným počtom prvkov z (1) a (2).

Skrátnená forma

(1) Pre každý regex e: $(e)+$ je regex a $(e)+ \equiv e(e)*$.

(2) Znak ' . ' znamená ľubovolný znak okrem $\backslash n$.

Triedy znakov

(1) Pre $a_{i_1}, a_{i_2}, \ldots, a_{i_t} \in \Sigma,\ t \geq 1,\ [a_{i_1} a_{i_2} \ldots a_{i_t}] \equiv a_{i_1}|a_{i_2}|\ldots|a_{i_t}$.

(2) Pre $a_i, a_j \in \Sigma$ také, že $a_i \leq a_j,\ [a_i - a_j]$ je regex a $[a_i - a_j] \equiv a_i|a_{i+1}|\ldots|a_j$.

(3) Pre $a_{i_1}, a_{i_2}, \ldots, a_{i_t} \in \Sigma,\ t \geq 1,\ [\hat{}\, a_{i_1} a_{i_2} \ldots a_{i_t}] \equiv b_{i_1}|b_{i_2}|\ldots|b_{i_s}$, kde $\{b_{i_1}|b_{i_2}|\ldots|b_{i_s}\} = \Sigma - \{a_{i_1}, a_{i_2}, \ldots, a_{i_t}\}$.

(4) Pre $a_i, a_j \in \Sigma$ také, že $a_i \leq a_j,\ [a_i - a_j]$ je regex a $[\hat{}\, a_i - a_j] \equiv b_{i_1}|b_{i_2}|\ldots|b_{i_s}$, kde $\{b_{i_1}|b_{i_2}|\ldots|b_{i_s}\} = \Sigma - \{a_i|a_{i+1}|\ldots|a_j\}$.

(5) Zmes (1) a (2) alebo (3) a (4).

Ukotvenie

(1) Znak pre začiatok riadku ˆ.

(2) Znak pre koniec riadku \$.

Vo formálnom texte bude prirodzené pomenovávať ľubovoľný model regexov pomocou písmen gréckej abecedy.

Rada by som spomenula aj iné konštrukcie, ktoré v definícii chýbajú. V konečnom dôsledku síce modelu silu nepridajú, ale budem s nimi rátať v ďalšom výskume, nakoľko sa bavíme o moderných regulárnych výrazoch.

$*?, +?, ??$ definované ako $*, +, ?$, ale snažia sa matchovať čo najmenej znakov ($*, +, ?$ sú greedy)

$\{m\}$ – matchuje práve m kópií predošlého regexu; $m \in \mathbb{N}$ je konštanta

$\{m, n\}$ – matchuje aspoň m a najviac n kópií predošlého regexu, matchuje čo najviac; $m, n \in \mathbb{N}$ sú konštanty

$\{m, n\}?$ – ako $\{m, n\}$, ale matchuje čo najmenej

(?# . . .) komentár, pri matchovaní sa obsah ignoruje

Ukážme si, že to naozaj platí. Pre ľahší dôkaz uvediem aj formálne definície.

Definícia 1.1.1 (Greedy iterácia)**.**

$$L_1 \circledast L_2 = \{uv \mid u \in L_1^* \wedge v \in L_2 \wedge u \textit{ je najdlhšie také}\}$$

Definícia 1.1.2 (Minimalistická iterácia).

$$L_1 *? L_2 = \{uv \mid u \in L_1^* \wedge v \in L_2 \wedge u \textit{ je najkratšie také}\}$$

V sekcii 2.1 dokážem, že tieto iterácie pokrývajú rovnakú triedu jazykov ako Kleeneho $*$. Ostatné operácie sa ľahko prepíšu na už definované regexy, preto triedu jazykov nad regexami nerozšíria o žiaden nový jazyk. Poďme sa teda venovať tým zložitejsím a zaujímavejším operáciám.

1.2 Spätné referencie

Spätná referencia (angl. *backreference*) je označovaná ako $\backslash m$, kde $m \in \mathbb{N}$ je konštanta. Predstavuje reťazec, ktorý matchoval regex vnútri m–tých okrúhlych zátvoriek. Okrúhle zátvorky číslujeme zľava doprava podľa poradia ľavej zátvorky. Samozrejme, $\backslash m$ môže ukazovať ná zátvorky, ktoré obsahujú regex s inými spätnými referenciami. Vždy však budeme predpokladať, že spätná referencia s číslom m sa bude nachádzať až za pravou zátvorkou s číslom m.

Regexy rozšírené o spätné referencie budeme nazývať **e-regex**. Triedu jazykov nad e-regexami budem nazývať **Eregex**, presná definícia sa nachádza v článku [CSY03]. (Autori ju pôvodne nazvali extended regex resp. EREG, avšak pre lepšiu prehľadnosť v tejto práci som názov upravili.) Narába s regexami uvedenými iba v úvodnej definícii, ale myslíme, že je zrejmé, že nové operácie nepridajú modelu silu, preto môžeme pracovať s touto rozšírenou množinou operácií.

Dospelo sa k nasledujúcim výsledkom:

- V rámci Chomského hierarchie je trieda Eregex vlastnou podmnožinou $\mathscr{L}_{CS}$. Existujú jazyky z $\mathscr{L}_{CF}$ aj $\mathscr{L}_{CS}$, ktoré do nej nepatria.
- Čo sa týka uzáverových vlastností, trieda je uzavretá na homomorfizmus a nie je uzavretá na komplement, inverzný homomorfizmus, konečnú substitúciu, shuffle s regulárnym jazykom. Neuzavretosť na prienik sa podarilo dokázať až v článku [CN09].
- Nekonečné jazyky z Eregex sa dajú pumpovať, dokázali sa už 2 verzie pumpovacej lemy ([CSY03, Lemma 1], [CN09, Lemma 3]).

1.3 Lookaround

Tieto operácie nás v práci budú zaujímať. Je to niečo nové a málo skúmané. Tu si uvedieme ich popis a definície.

Lookaround je spoločný názov pre dve operácie – lookahead a lookbehind. Ako fungujú?

Zoberme si lookahead. Myšlienka spočíva v tom, že si chceme povedať, čo má za daným slovom nasledovať. Teda nazrieme dopredu a overíme, či to tam je. Táto operácia 'nevyjedá' písmenká, čiže keď lookahead dokončí a uspeje, pokračuje sa v ďalšom matchovaní akokeby tam nebol, teda presne od toho miesta, kde on začal pracovať. Presnejšie: slovo sa matchuje podľa regexu. Keď narazíme na lookahead, zapamätáme si toto miesto. Matchujeme podľa lookaheadu. Ak uspeje, potom sa v slove vrátime na zapamätané miesto a pokračujeme v matchovaní regexom za lookaheadom.

Lookbehind pracuje analogicky, ale pozerá sa dozadu – teda chceme vedieť, čo danému slovu predchádza. V praxi by sa toto riešilo backtrackingom, lebo nevieme vopred ako ďaleko dozadu sa budeme potrebovať pozrieť. A keďže autori programovacích jazykov nechceli príliš spomaľovať výpočty a asi nepovažovali lookbehind za taký dôležitý, určili si, že lookbehind môže obsahovať iba také regexy, z ktorých je jasne určiteľné aký dlhý reťazec bude potrebovať na výpočet (takže sa backtrackingu vyhli). My sme však v teoretickom prostredí a primárne nás bude zaujímať veľkosť triedy, ktorú budeme vedieť popísať aj pomocou týchto operácií, preto sme sa rozhodli na tieto obmedzenia zabudnúť. Je však jasné, že reálny model bude akousi podmnožinou toho nášho.

Definícia 1.3.1 (Pozitívny lookahead)**.**

$$L_1(? = L_2)L_3 = \{uvw \mid u \in L_1 \wedge v \in L_2 \wedge vw \in L_3\}$$

Operáciu $(? = \dots)$ *nazývame pozitívny lookahead alebo len lookahead.*

Definícia 1.3.2 (Pozitívny lookbehind)**.**

$$L_1(? <= L_2)L_3 = \{uvw \mid uv \in L_1 \wedge v \in L_2 \wedge w \in L_3\}$$

Operáciu $(? <= \dots)$ *nazývame pozitívny lookbehind alebo len lookbehind.*

Negatívna verzia funguje rovnako, ale otáča akceptačnú požiadavku – namiesto toho, aby sme písali, čo nasleduje, definujeme práve to, čo nechceme aby nasledovalo. Teda negatívny lookahead 'zbehne', ak nie je schopný matchovať vstup.

Definícia 1.3.3 (Negatívny lookahead)**.**

$$L_1(?!L_2)L_3 = \{uv \mid u \in L_1 \wedge v \in L_3 \wedge \textit{neexistuje také } x, y, \textit{ že } v = xy \textit{ a } x \in L_2\}$$

Operáciu $(?!\dots)$ *nazývame negatívny lookahead.*

Definícia 1.3.4 (Negatívny lookbehind).

$$L_1(?<!L_2)L_3 = \{uv \mid u \in L_1 \wedge v \in L_3 \wedge \textit{neexituje také } x, y, \textit{ že } u = xy \textit{ a } y \in L_2\}$$

Operáciu $(?<!\dots)$ *nazývame negatívny lookbehind.*

Nachádza sa tu ešte jeden rozdiel od reálneho modelu, na ktorý by sme chceli upozorniť. Regulárne výrazy sa začali využívať na vyhľadávanie v texte. V tomto kontexte má lookaround iný význam ako v teoretickom prostredí. Využíva sa to, že nevyjedá písmenká a preto sa dá použiť v takých prípadoch, kedy si chceme označiť slovo w, ale vieme, že hľadáme slovo v, pričom w je podslovo v. Teda v takom prípade môže lookahead ďaleko presahovať slovo, na ktoré sme vyhlásili zhodu. Z toho vyplýva, že my by sme na vstup nechceli dostať w, ale v. V teoretickom prostredí nás však zaujíma slovo, ktoré do jazyka patrí a na jeho okolie sa nepozeráme, v podstate ho ani nemáme k dispozícii. Preto sa zdá, že budeme na vstupe očakávať w a lookahead či lookbehind nebudú môcť presahovať za hranice slova. Táto úprava však neuškodí, nakoľko vieme z daného w spraviť v jednoduchým pridaním $(.*)$ na správne miesta.

2 Naše výsledky

2.1 Minimalistická iterácia

Veta 2.1.1. $L_1 \circledast L_2 = L_1 *? L_2 = L_1^* L_2$

Dôkaz. $\subseteq$: Nech $w \in L_1 \circledast L_2$. Potom z definície $w = uv$ vieme, že $u \in L_1^*$ a $v \in L_2$, teda $uv \in L_1^* L_2$. Analogicky ak $x = yz \in L_1 *? L_2$, potom $yz \in L_1^* L_2$.

$\supseteq$: Majme $w \in L_1^* L_2$ a rozdeľme na podslová u, v tak, že $u \in L_1^*, v \in L_2$ a $w = uv$. Takéto rozdelenie musí byť aspoň jedno. Ak je ich viac, vezmime to, kde je u najdlhšie. Potom $uv \in L_1 \circledast L_2$. Ak zvolíme u najkratšie, tak zasa $uv \in L_1 *? L_2$. □

Dôsledok 2.1.2. *Trieda $\mathcal{R}$ je uzavretá na operácie $\circledast$ a $*?$.*

Kleeneho $*$ uvedená v definícii regexov je príliš nedeterministická na ľahké prevedenie do praxe, preto reálny model používa v prípade operácie $*$ algoritmus pre greedy iteráciu. Vidíme však, že to nevadí, lebo pokrývame stále tú istú triedu jazykov. Takisto po pridaní minimalistickej iterácie. Z teoretického hľadiska je existencia dvoch operácií s rovnakou funkcionalitou zbytočná. Ak zhoda existuje, regex matchuje s použitím ľubovoľnej z nich. Ale riešenie (rozdelenie slova) vyzerá inak, čo má v praxi zmysel pri jeho ďalšom použití. Preto je existencia oboch operácií opodstatnená.

2.2 Lookaround

2.2.1 Chomského hierarchia

Operácie máme zadefinovné, poďme sa pozrieť, čo urobia s jazykmi z tried Chomského hierarchie.

Lema 2.2.1. *Trieda $\mathcal{R}$ je uzavretá na lookahead.*

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

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

Zostrojím NKA A pre L. Myšlienka spočíva v tom, že najprv necháme simulovať A_1 a keď akceptuje, simulujeme naraz A_2 a A_3. Konštrukcia je podoná ako prienik dvoch regulárnych jazykov (karteziánsky súčin stavov) s tým rozdielom, že A_2 môže skončiť skôr ako A_3.

Konštrukcia. $A = (K, \Sigma, \delta, q_0, F)$, kde $K = K_1 \cup K_2 \times K_3 \cup K_3$ (predp. $K_1 \cap K_3 = \emptyset$), $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$, $q_0 = q_{01}$, $F = F_3 \cup F_2 \times F_3$, δ funkciu definujeme nasledovne:

$$\begin{aligned}
\forall q \in K_1, \forall a \in \Sigma &\;:\; \delta(q,a) \ni \delta_1(q,a) \\
\forall q \in F_1 &\;:\; \delta(q,\varepsilon) \ni [q_{02}, q_{03}] \\
\forall p \in K_2, \forall q \in K_3, \forall a \in \Sigma_2 \cap \Sigma_3 &\;:\; \delta([p,q], a) \ni [\delta(p,a), \delta(q,a)] \\
\forall p \in F_2, \forall q \in K_3 &\;:\; \delta([p,q], a) \ni \delta(q,a)
\end{aligned}$$

$L(A) = L$.

$\supseteq$: Máme $w \in L$ a chceme preň nájsť výpočet na A. Z definície L vyplýva $w = xyz$, kde $x \in L_1, y \in L_2$ a $yz \in L_3$, teda existujú akceptačné výpočty pre x, y, yz na DKA A_1, A_2, A_3. Z toho vyskladáme výpočet pre w na A nasledovne. Výpočet pre x bude rovnaký ako na A_1. Z akceptačné stavu A_1 vieme na ε prejsť do stavu $[q_{02}, q_{03}]$, kde začne výpočet pre y. Ten vyskladáme z A_2 a A_3 tak, že si ich výpočty napíšeme pod seba a stavy nad sebou budú tvoriť karteziánsky súčin stavov v A (keďže A_2 aj A_3 sú deterministické, tieto výpočty na y budú rovnako dlhé). $y \in L_2$, teda A_2 skončí v akceptačnom stave. Podľa δ funkcie v A vieme pokračovať len vo výpočte na A_3, teda doplníme zvyšnú postupnosť stavov pre výpočet z. Keďže $yz \in L_3$ a $F_3 \subseteq F$ (resp. $F_2 \times F_3 \subseteq F$ pre $z = \varepsilon$), automat A akceptuje.

$\subseteq$: Nech $w \in L(A)$, potom existuje akceptačný výpočet na A. Z toho vieme w rozdeliť na x, y a z tak, že x je slovo spracovávené od začiatku po prvý príchod do stavu $[q_{02}, q_{03}]$, y odtiaľto po posledný stav reprezentovaný karteziánskym súčinom stavov a zvyšok bude z. Nevynechali sme žiadne znaky a nezmenili poradie, teda $w = xyz$. Do $[q_{02}, q_{03}]$ sa A môže prvýkrát dostať len vtedy, ak bol v akceptačnom stave A_1. Prechod do $[q_{02}, q_{03}]$ je na ε, takže $x \in L_1$. Práve tento stav je počiatočný pre A_2 aj A_3. Ak $z = \varepsilon$, tak akceptačný stav A je z $F_2 \times F_3$ a $y \in L_2, y \in L_3$ a aj $yz \in L_3$. Z toho podľa definície vyplýva, že $xyz = w \in L$. Ak $z \neq \varepsilon$, potom je akceptačný stav A z F_3. Podľa δ funkcie sa z karteziánskeho súčinu stavov do normálneho stavu dá prejsť len tak, že A_2 akceptuje, teda $y \in L_2$. A_3 akceptuje na konci, čo znamená $yz \in L_3$. Znova podľa definície operácie lookahead $xyz = w \in L$. □

Lema 2.2.2. *Trieda $\mathcal{R}$ je uzavretá na lookbehind.*

Dôkaz. Opäť chceme ukázať, že $L = L_1(?<= L_2)L_3 \in \mathcal{R}$ pre $L_1, L_2, L_3 \in \mathcal{R}$. Postupujeme podobne ako pri lookahead. Urobíme karteziánsky súčin stavov A_1 a A_2 tak, že A_2

môže začať výpočet v ľubovoľnom stave A_1 - celkový NKA si potom nedeterministicky zvolí jeden moment tohto napojenia. A_1 a A_2 musia naraz akceptovať, potom sa začne simulácia A_3. □

Veta 2.2.3. *Regulárne jazyky sú uzavreté na lookaround.*

Veta 2.2.4. *$\mathscr{L}_{CF}$ nie je uzavretá na operácie lookahead a lookbehind.*

Dôkaz. Nech $L_1, L_2, L_3, L_4 \in \mathscr{L}_{CF}$. $L_1 = \{a^n b^n \mid n \geq 1\}, L_2 = \{a * b^n c^n \mid n \geq 1\}, L_3 = \{a^n b^n c * \mid n \geq 1\}, L_4 = \{ab^n c^n \mid n \geq 1\}$. Potom $d(? = L_1)L_2 = \{da^n b^n c^n \mid n \geq 1\}$ a $L_3(? <= L_4)d = \{a^n b^n c^n d \mid n \geq 1\}$, čo nie sú bezkontextové jazyky. □

Veta 2.2.5. *$\mathscr{L}_{CS}$ je uzavretá na operáciu lookahead a lookbehind.*

Dôkaz. Uzavretosť na lookahead:

Nech $L_1, L_2, L_3 \in \mathscr{L}_{CS}$. Pre $L = L_1(? = L_2)L_3$ zostrojíme LBA A z LBA A_1, A_2, A_3 pre dané kontextové jazyky. Najprv sa pozrime na štruktúru vstupu – prvé je slovo z L_1 a za ním nasleduje slovo z L_3, pričom jeho prefix patrí do L_2. Preto, aby A mohol simulovať dané lineárne ohraničené automaty, je potrebné označiť hranice jednotlivých slov.

Na začiatku výpočtu A prejde pásku a nedeterministicky označí 2 miesta – koniec slov pre A_1 a A_2. Následne sa vráti na začiatok a simuluje A_1. Ak akceptuje, A pokračuje a presunie sa za označený koniec vstupu pre A_1. Inak sa zasekne. V tomto bode sa začína vstup pre A_2 aj A_3, teda slovo až do konca prepíše na 2 stopy. Najprv na hornej simuluje A_2. Pokiaľ A_2 neskončí v akceptačnom stave, A sa zasekne. Inak sa vráti na označené miesto a simuluje A_3 na spodnej stope až do konca vstupu. Akceptačný stav A_3 znamená akceptáciu celého vstupného slova.

Uzavretosť na lookbehind sa dokáže analogicky. □

2.2.2 Regexy

Keď už sme lepšie zoznámení s lookaroundom, mali by sme posúdiť ako zapadá medzi regexy. Nebude to také jednoduché, keďže celý model regexov je množina operácií a jednopísmenkových jazykov (konečná abeceda), z ktorých postupne vyskladávame zložitejšie jazyky. V Chomského hierarchii sme pracovali iba s akousi finálnou formou jazyka, o ktorom sme vedeli zopár vlastností. Teraz zoberieme množinu operácií a pridáme k nim ďalšie dve. Otázkou je, čo spôsobí ich kombinovanie. Najzaujímavejšie vyzerajú tie, ktoré vedia ovplyvniť samotné vlastnosti lookaroundu, nakoľko sme ukázali, že regulárne jazyky sú naň uzavreté.

Lema 2.2.6. *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. Keďže $L_1, L_2, L_3, L_4 \in \mathcal{R}$, tak pre ne existujú DKA A_1, A_2, A_3, A_4, kde $A_i = (K_i, \Sigma_i, \delta_i, q_{0i}, F_i)$ pre $\forall i$. Z nich zostrojíme NKA A pre L. Výpočet bez lookaheadov by vyzeral tak, že by sme simulovali A_1, potom po jeho akceptácii A_3 a odtiaľ by sa išlo v rámci iterácie naspäť na A_1. Zároveň z A_1 by sa dalo na ε prejsť na A_4, čo by znamenalo koniec iterácie (ošetruje aj nulovú iteráciu). Pozrime sa na to, ako a kam vsunúť lookahead. Problém je, že pri každej ďalšej iterácii pribúda nový, teda ďalší A_2. Vieme ich však simulovať všetky naraz, keď vezmeme do úvahy, že vždy pracujeme nad konečnou abecedou a K_2 je konečná. Z toho vyplýva, že aj $\mathcal{P}(K_2)$ je konečná, a teda vieme zostrojiť automat, ktorý si bude simulovať prechody medzi množinami stavov A_2.

Konštrukcia. $A = (K, \Sigma, \delta, q_0, F) : K = (K_1 \cup K_3 \cup K_4) \times \mathcal{P}(K_2)$, kde $K_1 \cap K_3 \cap K_4 = \emptyset$ (množiny v stavoch možno reprezentovať napr. 0-1 reťazcom dĺžky $|K_2|$, kde 1 na i–tom mieste symbolizuje, že nejaká inštancia A_2 je v i–tom stave), $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4$, $q_0 = (q_{01}, \emptyset)$, $F = F_4 \times \emptyset$, δ–funkcia:

- $\forall q \in K_i \ i = 1, 3, 4\ ,\ \forall U \in \mathcal{P}(K_2), \forall a \in \Sigma : \delta((q_i, U), a) \ni (\delta_i(q_i, a), V)$,
  kde $\forall q \in U :\ \delta_2(q, a) \in V'$ a $V = V' \setminus F_2$
- $\forall q_A \in F_1, \forall U \in \mathcal{P}(K_2) : \delta((q_A, U), \varepsilon) \ni (q_{03}, U \cup \{q_{02}\})$
- $\forall q_A \in F_3, \forall U \in \mathcal{P}(K_2) : \delta((q_A, U), \varepsilon) \ni (q_{01}, U)$
- $\forall U \in \mathcal{P}(K_2) : \delta((q_{01}, U), \varepsilon) \ni (q_{04}, U)$

Automat A akceptuje až keď akceptuje A_4. Je zrejmé, že ak v simulácii A_i príde písmenko, ktoré do Σ_i nepatrí, automat sa zasekne.

$L(A) = L(\alpha)$.

$\subseteq$: Nech $w \in L(A)$, potom preň existuje akceptačný výpočet na A. Podľa stavov vieme určiť počet iterácií, podslová z L_1, L_3, L_4 a takisto vznikajúce a akceptujúce výpočty na A_2 – každý takýto výpočet totiž začína s výpočtom na A_3 a keďže A_2 je deterministický, existuje práve jeden výpočet, ktorý musí byť akceptačný. Teda vieme povedať, že $w = x_1y_1x_2y_2 \dots x_ny_nz$, kde n je počet iterácií, $\forall i = 1, 2, \dots, n :\ x_i \in L_1, y_i \in L_3$ a $z \in L_4$. Zároveň vieme, že v mieste, kde začína y_i takisto začína podreťazec slova w, ktorý patrí do L_2. Z toho vidíme ako vyzerá matchovanie regexu α.

$\supseteq$: Majme $v \in L(\alpha)$, teda vieme nájsť zhody podslov v pre všetky L_i (rovnaká dekompozícia slova ako v predošlej inklúzii). Keďže poradie jazykov je rovnaké ako ε–ové prepojenie A_i v A (akceptačný–počiatočný, počiatočný–počiatočný pri L_4), vieme správne poprepájať akceptačné výpočty jednotlivých A_i do celkového výpočtu automatu A. $\square$

Lema 2.2.7. *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. Opäť dôkaz pre lookbehind vyzerá podobne ako pre lookahead, A simuluje najprv A_4, potom $A_1, A_3, A_1, A_3, \ldots$ až po koniec slova. Ale musíme zmeniť simulovanie A_2. Stále potrebujeme množinu jeho stavov, keďže každou iteráciou má jeden pribudnúť, ale pridanie počiatočného stavu nie je viazané na konkrétny krok – ten môže teraz pribudnúť hocikedy (nedeterministicky). To, čo je fixné je akceptačný stav. Ten zase musí prísť zarovno s akceptáciou A_1.

Celý výpočet upravíme tak, že A_4 sa bude simulovať ako prvý (keďže sa pozmenil celý jazyk L), kedykoľvek môže pribudnúť počiatočný stav A_2 a vždy keď akceptuje A_1, aspoň jeden A_2 musí akceptovať. Aspoň jeden preto, že to nemusí byť práve jeden. V akceptačnom stave sa môžu naraz nachádzať hoci aj 2 inštancie A_2, ale v tom prípade nechceme z množiny stavov vymazávať jeho akceptačný stav. Preto tento krok necháme na nedeterminizmus. V prípade, že by sme sa nedeterministicky rozhodli pre viacero výpočtov lookbehindov ako je iterovaní a automat akceptuje, pretože sa všetkým A_2 podarilo v správnom kroku skončiť (t.j. viaceré skončili naraz), nevadí nám to. Samotný lookbehind totiž môže matchovať vedieť viaceré podslová končiace v jednom bode, čo sa vie prejaviť práve takto.

δ–funkcia by vyzerala takto:

- $\forall U \in \mathcal{P}(K_2) : \delta((q_{04}, U), \varepsilon) \ni (q_{01}, U)$
- $\forall q \in K_i \; i = 1, 3, 4$, $\forall U \in \mathcal{P}(K_2), \forall a \in \Sigma : \delta((q_i, U), a) \ni (\delta_i(q_i, a), V)$,
  kde $\forall q \in U : \; \delta_2(q, a) \in V$
- $\forall q_A \in F_1 : \delta((q_A, U), \varepsilon) \ni (q_{03}, U), (q_{03}, U')$
  pre všetky také $U \in \mathcal{P}(K_2)$, že $\exists q_A \in F_2$ a $q_A \in U$; $U' = U \setminus q_A$
- $\forall q_A \in F_3, \forall U \in \mathcal{P}(K_2) : \delta((q_A, U), \varepsilon) \ni (q_{01}, U)$

$F = F_3 \times \emptyset$. $L(A) = L(\alpha)$ dokážeme podobne ako v leme 2.2.6. □

Veta 2.2.8. *Trieda nad regexami s lookaheadom je $\mathcal{R}$.*

Dôkaz. Samotné regexy pokrývajú triedu regulárnych jazykov a tá je na lookaround uzavretá 2.2.3. Keďže pracujeme s množinou operácií, treba overiť, či nejaká ich kombinácia nie je náhodou silnejšia. Ak regex umiestnime do lookaroundu, či pred alebo za neho, vždy to bude regulárny jazyk a celý regex bude tiež definovať regulárny jazyk. Teda nás zaujíma vloženie lookaroundu dovnútra inej operácie. V tomto prípade prichádza do úvahy $*, +$ a ?, ktoré menia počet lookaroundov a vynútia ich simuláciu na rôznych častiach slova.

? veľa nespraví – lookaround tam buď bude 1x alebo nebude vôbec. $+$ je prípad $*$ s jedným lookaroundom istým. No a podľa liem 2.2.6 a 2.2.7 vieme, že ani táto kombinácia nestačí na zložitejší jazyk. □

2.2.3 Vlastnosti lookaroundu.

Tu ukážem, že dávať do lookaheadu prefixový jazyk nemá zmysel. Vytvorme čisto jazyk všetkých rôznych prefixov, aké obsahuje. Do lookaheadu stačí vložiť regex pre tento jazyk a celkový matchovaný jazyk to nezmení. Samozrejme, to isté platí aj pre lookbehind a sufixové jazyky.

Veta 2.2.9. *Nech L je ľubovoľný jazyk a $L_p = L \cup \{uv \mid u \in L\}$. Nech α je ľubovoľný regulárny výraz taký, že obsahuje $(? = L_p)$. Potom ak prepíšeme tento lookahead na $(? = L)$ (nazvime to α'), bude platiť $L(\alpha') = L(\alpha)$. Analogicky to platí aj pre lookbehind.*

Dôkaz. $\subseteq$: triviálne, $L \subseteq L_p$.

$\supseteq$: Majme $w \in L(\alpha)$ a nech x je také podslovo w, ktoré sa zhodovalo práve s daným lookaheadom. Potom $x \in L_p$, teda $x = uv$, kde $u \in L$. Ak $v = \varepsilon$, $x \in L$ a máme čo sme chceli. Takže $v \neq \varepsilon$. Ale celá zhoda lookaheadu sa môže zúžiť len na u, keďže $u \in L_p$, a bude to platná zhoda s w. Čo znamená, že $w \in L(\alpha')$. $\square$

Dôsledok 2.2.10. *Nech α je regulárny výraz, ktorý obsahuje nejaký taký lookahead $(? = L)$ (lookbehind $(? <= L)$), že $\varepsilon \in L$. Nech je α' regulárny výraz bez tohto lookaheadu (lookbehindu). Potom $L(\alpha') = L(\alpha)$.*

Dôkaz. Uvedomme si, že lookaround nie je fixovaný na dĺžku vstupu - musí sa zhodovať s nejakým podslovom začínajúcim sa (končiacim sa) na konkrétnom mieste. Tým pádom akonáhle si môže regulárny výraz vnútri tejto operácie vybrať ε, bude hlásiť zhodu vždy. $\square$

2.2.4 Spätné referencie

Skúsme teraz zložitejší model. Pridáme k e-regexom pozitívny lookaround a túto množinu nazveme **le-regex**. Triede jazykov nad le-regexami budeme hovoriť **LEregex**.

Veta 2.2.11. *Eregex $\subsetneq$ LEregex*

Dôkaz. $\subseteq$ vyplýva z definície.

Jazyk $L = \{a^i b a^{i+1} b a^k \mid k = i(i+1)k' \ pre\ nejak\acute{e}\ k' > 0, i > 0\}$ nepatrí do triedy Eregex [CN09, Lemma 2], ale patrí do LEregex:

$$\alpha = (a*)b(\backslash 1a)b(? = (\backslash 1) * \$)(\backslash 2) * \$^1$$

$\square$

[1]Nie je nutné dávať \$ na koniec le-regexu, nakoľko pracujeme v teoretickom prostredí a nehľadáme slovo v texte. Na vstup dostávame vždy len samotné slovo a teda vieme povedať, že na jeho konci bude isto aj koniec riadku (\$). Znak sme uviedli, pre ľahšie pochopenie, budeme uvádzať aj vo všetkých ďalších prípadoch.

Veta 2.2.12. *$LEregex \subseteq \mathscr{L}_{CS}$*

Dôkaz. Vieme, že $Eregex \in \mathscr{L}_{CS}$ [CSY03, Theorem 1], teda ľubovoľný e-regex vieme simulovať pomocou LBA. Ukážem, že ak pridáme operáciu lookahead/lookbehind, vieme to simulovať tiež.

Nech α je le-regex. Potom A je LBA pre α, ktorý ignoruje lookaround (t.j. vyrábame LBA pre e-regex). Teraz vytvorím LBA B pre e-regex vnútri lookaroundu. Z nich vytvoríme LBA C pre úplný le-regex α tak, že bude simulovať A a keď príde na rad 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 (lokahead/lookbehind). Pokiaľ B akceptoval, vráti sa naspäť k výpočtu na A. C akceptuje práve vtedy, keď A.

Všimnime si, že samotný lookaround môže obsahovať le-regex, t.j. vnorený lookaround. Tých však môže byť iba konečne veľa, keďže každý le-regex musí mať konečný zápis. Čo znamená, že aj stôp bude konečne veľa a naznačeným postupom si vieme postupne vybudovať LBA, ktorý bude simulovať α. □

Veta 2.2.13. *LEregex je uzavretá na prienik.*

Dôkaz. Nech $L_1, L_2 \in LEregex$, potom $L_1 \cap L_2 = (? = L_1\$)\, L_2\$$. □

Veta 2.2.14. *Jazyk všetkých platných výpočtov Turingovho stroja patrí do LEregex.*

Dôkaz. Máme Turingov stroj $A = (K, \Sigma, q_0, \delta, F)$. Zostrojíme le-regex α, ktorý bude matchovať všetky jeho platné výpočty. Vieme, že každý TS pracuje nad konečnou abecedou, má konečne veľa stavov a konečne definovanú δ–funkciu. Keďže to všetko treba zahrnúť do le-regexu, vďaka konečnosti máme potrebné predpoklady na jeho vytvorenie. Stav bude ukazovať pozíciu hlavy a # bude oddeľovať jednotlivé konfigurácie (bude sa nachádzať aj na začiatku a na konci celého slova/výpočtu).

Prejdime k samotnému výpočtu. Na to, aby po sebe nasledovali len prijateľné konfigurácie využijeme lookahead a spätné referencie. To zabezpečí, aby sa v každom kroku výpočtu menilo len malé okolie stavu (spätné referencie zvyšok okopírujú) a aby sa menilo práve podľa δ–funkcie. Keď máme zaručenú správnu postupnosť konfigurácií, vieme použiť $*$ a matchovať tak ľubovoľne dlhý výpočet. Netreba zabudnúť, že prvá konfigurácia je počiatočná a posledná akceptačná. Toto sú špeciálne prípady, preto ich ošetríme zvlášť, a takisto prípad, kedy je počiatočný stav zároveň aj akceptačným.

Konštrukcia. $\alpha = \beta(\gamma) * \eta$, kde:

- $\gamma = \gamma_1 \mid \gamma_2 \mid \gamma_3$
    - $\star$ $\gamma_1 = ((\underset{k}{(} .* \underset{k}{)} xqy \underset{k+1}{(} \; .* \; \underset{k+1}{)} \#)(? = \xi\#)$ platí pre $\forall q \in K$, $\forall y \in \Sigma$ a kde $\xi = \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$. Ak

- ∘ $(p, z, 0) \in \delta(q, y)$, potom $\xi_i = (\backslash kxpz\backslash k + 1)$ pre nejaké i
- ∘ $(p, z, 1) \in \delta(q, y)$, potom $\xi_i = (\backslash kxzp\backslash k + 1)$ pre nejaké i
- ∘ $(p, z, -1) \in \delta(q, y)$, potom $\xi_i = (\backslash kpxz\backslash k + 1)$ pre nejaké i

⋆ $\gamma_2 = (qy\underset{k}{(}\,.*\,\underset{k+1}{)}\,\#)(? = \xi\#)$ platí pre $\forall q \in K$, $\forall y \in \Sigma \cup \{B\}$[2] a kde $\xi = \xi_1 \mid \xi_2 \mid \dots \mid \xi_n$. Teda hlava ukazuje na začiatok slova. Dodefinovanie ξ_i je podobné. Ak

- ∘ $(p, z, 0) \in \delta(q, y)$, potom $\xi_i = (pz\backslash k)$ pre nejaké i
- ∘ $(p, z, 1) \in \delta(q, y)$, potom $\xi_i = (zp\backslash k)$ pre nejaké i
- ∘ $(p, z, -1) \in \delta(q, y)$, potom $\xi_i = (pBz\backslash k)$ pre nejaké i

⋆ $\gamma_3 = (\underset{k}{(}\,.*\,\underset{k}{)}\,xqy\#)(? = \xi\#)$ platí pre $\forall q \in K$, $\forall y \in \Sigma \cup \{B\}$ a kde $\xi = \xi_1 \mid \xi_2 \mid \dots \mid \xi_n$. Teda hlava je na konci slova, opäť je treba ošetriť blankový prípad. Ak

- ∘ $(p, z, 0) \in \delta(q, y)$, potom $\xi_i = (\backslash kxpz)$ pre nejaké i
- ∘ $(p, z, 1) \in \delta(q, y)$, potom $\xi_i = (\backslash kxzpB)$ pre nejaké i
- ∘ $(p, z, -1) \in \delta(q, y)$, potom $\xi_i = (\backslash kpxz)$ pre nejaké i

- $\beta = (\#q_0x\underset{k}{(}\,.*\,\underset{k}{)}\,\#)(? = \theta\#)$ je definovaná rovnako ako γ_2, ale iba pre q_0.
- $\eta = ((.*)q_A(.*)\#)$ platí pre $\forall q_A \in F$

Ak je počiatočný stav akceptačným, pridáme na koniec α regex '$|(\#q_0.*\#)$' . □

Poznámka. Práve sme si ukázali aká silná je kombinácia lookaroundu a spätných referencií. Jazyk platných výpočtov TS je celkom zložitý vďaka závislostiam z δ–funkcie a nedá sa pumpovať. Vidíme to, keď si predstavíme DTS, ktorý akceptuje nekonečný jazyk. Tým, že existujú akceptačné výpočty na týchto slovách, získavame nekonečný jazyk platných výpočtov. Nevieme však nadĺžiť ani slovo – pokazilo by to výpočet, ani výpočet – vďaka determinizmu vieme, že ak sa TS zacyklí, tak sa z toho už nedostane.

Tento poznatok nielenže zvýrazňuje rozdiely medzi triedami Eregex a LEregex, ale zároveň nám komplikuje situáciu, pretože dôkaz, že nejaký jazyk nepatrí do triedy LEregex je bez pumpovacej lemy veľmi zložitý. Preto sme sa skúsili pozrieť na unárne jazyky, či by aspoň na nich pumpovanie fungovalo.

Vyslovíme vetu, ktorá je veľmi blízko pumpovacej leme, ale ňou nie je, nakoľko sme nevedeli dokázať pumpovanie pre jeden prípad. A to keď sa lookahead, obsahujúci na konci znak \$ (lookbehind so znakom ˆ na začiatku je analogický prípad), objaví vnútri iterácie. Nenašli sme argument ako zaručiť, že pri pumpovaní vzniká slovo z daného

[2] B označuje blank – prázdne políčko na páske. Z definície TS $B \notin \Sigma$.

jazyka. Problém tkvie v tom, že pri každej pridanej iterácii sa pridá aj ďalší lookahead a my musíme zaručiť, že matchuje slovo za ním až do konca.

Veta 2.2.15. *Nech α taký je le-regex nad unárnou abecedou $\Sigma = \{a\}$, že neobsahuje lookahead s \$ a lookbehind s ^ vnútri iterácie. Existuje konštanta N taká, že ak $w \in L(\alpha)$ a $|w| > N$, potom existuje dekompozícia $w = xy$ s nasledujúcimi vlastnosťami:*

1. $|y| \geq 1$
2. $\exists k \in \mathbb{N},\ k \neq 0;\ \forall j = 1, 2, \ldots : xy^{kj} \in L(\alpha)$

Dôkaz. Pokiaľ $\alpha \in Eregex$, tak pre α platí pumpovacia lema [CSY03, Lemma 1], t.j. $w = a_0ba_1b\ldots a_m$ pre nejaké m a $a_0b^ja_1b^j\ldots a_m \in L(\alpha)\ \forall j$. My pracujeme nad unárnym jazykom, teda na poradí nezáleží: $x = a_0a_1\ldots a_m,\ y = b^m, k = 1$ a $xy^j \in L(\alpha)\ \forall j$.

Zaoberajme sa le-regexami, ktoré obsahujú aspoň jeden lookaround. Ale ešte predtým si povedzme, aké je to dostatočne dlhé slovo. Chceme, aby sme s určitosťou vedeli, či nejaká $*/+$ iteruje aspoň 2x. Zoberieme dĺžku le-regexu $|\alpha|$. Skreslovať ju môžu 3 operácie – spätné referencie, $\{n\}$ a $\{n, m\}$. Nech teda d je súčet dĺžok le-regexov vnútri l–tých zátvoriek pre všetky $\backslash l$, ktoré v α sú a sú pripočítané toľkokrát, koľkokrát sa tam jednotlivé $\backslash l$ nachádzajú. Ďalej k d pripočítame za každé $\{n\}$ n–krát dĺžku le-regexu, ktorý má kopírovať, a takisto za každú operáciu $\{n, m\}$ pripočítame takýto le-regex m–krát. Pre $N = |\alpha| + d$ vieme povedať, že ak $|w| > N$, potom jediná operácia, ktorá ho vie predĺžiť do takejto dĺžky môže byť iba Kleeneho $*$, prípadne $+$. $+$ je však v podstate prípad $*$, preto budeme pracovať s touto operáciou.

Máme dostatočne dlhé slovo, rozoberieme si teraz pípady, ktoré môžu nastať.

1. Žiadny lookaround nezasahuje do $*$, ktorá iteruje. V tom prípade iteruje normálne, bez obmedzení, nejaké $a^s,\ s < |\alpha| \leq N < |w|$. Potom ak $w = a^t$, tak $x = a^{t-s}, y = a^s, k = 1$ a $xy^j \in L(\alpha)\ \forall j$.
2. Lookaround zasahujúci do iterujúcej $*$ nie je ohraničený, t.j. ak je to lookahead, neobsahuje na konci \$, a ak je to lookbehind, neobsahuje na začiatku ^. Potom ho podľa vety 2.2.9 vieme prepísať tak, že bude matchovať len najkratšie slovo z jeho jazyka. Ak získame konečný jazyk prefixov/sufixov, tak síce ovplyvňuje dĺžku matchovaných slov, ale iba tú minimálnu. Teda $*$ iteruje bez obmedzení, čo je predošlý prípad. Pokiaľ je jazyk stále nekonečný, prípad je podobný tomu nasledujúcemu.
3. Našli sme iterujúcu $*$ a zasahuje do nej lookaround matchujúci až po koniec resp. začiatok slova. Buď je to lookbehind (nachádzajúci sa za $*$), za ktorým už v α nie je žiadne iterovanie alebo je to lookahead (nachádzajúci sa pred $*$), prípadne oba alebo ich môže byť viac. Každopádne, keďže je slovo dostatočne dlhé, aj samotný

lookaround musí niečo pumpovať. Vyriešme to pre prípad jedného lookaroundu a potom uvidíme, že tie ostatné z toho vyplývajú.

Máme $w = a^e$. V lookarounde sa pumpuje nejaké a^f, v $*$ mimo neho a^g (ak je na výber viac možností, stačí vybrať jednu, napr. tú najkratšiu). Potom lookaround matchuje a^{e+h*f} a $*$ mimo neho a^{e+i*g}, pre ľubovoľné $h, i \in \mathbb{N}_0$. Nech $b = nsn(f, g)$, prienik týchto slov je $a^{e+b*j} \in L(\alpha),\ \forall j \in \mathbb{N}_0$. Tvrdíme $b < e$. A naozaj, keďže $a^e \in L(\alpha)$ a zároveň lookaround aj $*$ určite niečo pumpovali, ich najmenšia možná zhoda je práve b.

Pracujeme s le-regexami, takže sa niekde musia objaviť aj spätné referencie[3]. Uvedomme si, že lookaroundy sa môžu vyskytovať niekde medzi nimi, takže keď napríklad $*$ pumpuje a^m, celkovo sa v slove pumpuje a^{n*m}, kde n je počet referencií ukazujúcich na túto $*$. Tým, že sa lookaroundy budú nachádzať medzi nimi, môže nastať, že pre jeden sa ďalej v slove pumpuje a^{n_1*m} a pre druhý a^{n_2*m}, kde $n_1 \neq n_2$. Ak $*$ vie pumpovať a^m, tak vie aj $a^{n!*m}$, čo by vyriešilo všetky možné násobky, ktoré ovplyvňujú rôzne lookaroundy. Podľa toho $k = n!$, pre n ako najvyšší počet spätných referencií na nejakú iteráciu, ktorú využívame.

Celkovo máme $x = a^{e-b}$, $y = a^b$ a $k = n!$ ak je určené, inak $k = 1$. Z toho $xy^{kj} = a^{e-b}a^{bkj} = a^{e+bj'} \in L(\alpha)\ \forall j \in \mathbb{N},\ j' = kj - 1$.

V prípade viacerých lookaroundov riešime postupne – vždy vyrátané b bude v ďalšom kole novým g. Podobne riešime vnorené lookaroundy – postupujeme zvnútra von (musí ich byť konečne veľa a ten najvnútornejší z nich isto bude obsahovať už len Eregex).

$\square$

Poznámka 1. Spomínaný prípad, pre ktorý je problematické túto vetu dokázať má problém aj s definovaním dostatočne veľkého slova. Pozrime sa napríklad na le-regex $\beta = ((? = (a^m) * \$)a^{m+1}) * a\{1, m-1\}\$$, aké slová obsahuje?

- $a^{m+1+(m-1)}$
- $a^{2m+2+(m-2)}$

  $\vdots$
- $a^{(m-1)(m+1)+1}$

avšak $a^{m(m+1)} \notin L(\beta)$, lebo nám chýba zvyšok 0. Teda vidíme, že le-regex využíva iteráciu $(m-1)$–krát a napriek tomu je konečný.

[3] Keby sa v α nenachádzali, podľa vety 2.2.3 máme regulárny jazyk, pre ktorý pumpovacia lema existuje.

Poznámka 2. Zoberme si triedu jazykov nad takýmito le-regexami. Veta nám hovorí čosi o zložitosti jej jazykov. Napriklad $L = \{a^m \mid m\ je\ prvočíslo\}$ do nej nepatrí, lebo sa nedá vôbec pumpovať – medzi prvočíslami nenájdeme žiadne násobky. Z toho už vyplýva, že je vlastnou podmnožinou $\mathscr{L}_{CS}$ a nie je uzavretá na komplement ($L^c = (aaa*)\backslash 1(\backslash 1)* \in Eregex$).

2.3 Negatívny lookaround

Definícia 2.3.1. *Triedu nad leregexami obohatenými o negatívny lookaround budeme nazývať nLEregex.*

Veta 2.3.2. *Trieda nLEregex je uzavretá na prienik.*

Dôkaz. Podobne ako veta 2.2.13. □

Veta 2.3.3. *Trieda nLEregex je uzavretá na komplement.*

Dôkaz. Nech $L_1 \in nLEregex$, potom $L_1^c = (?!L_1\$)\,.*\$$. □

Záver

Literatúra

[CN09] BENJAMIN CARLE and PALIATH NARENDRAN. 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` [Online; accessed 19-March-2013].

[Cox07] Russ Cox. *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*, 2007. `http://swtch.com/~rsc/regexp/regexp1.html` [Online; accessed 30-December-2012].

[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 19-March-2013].

[doc12] Python documentation. *Regular expressions operations*, 2012. `http://docs.python.org/3.1/library/re.html` [Online; accessed 30-December-2012].