UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

......NÁZOV.......

Diplomová práca

201. **Tatiana Tóthová**

Univerzita Komenského, Bratislava
Fakulta Matematiky, Fyziky a Informatiky

......NÁZOV.......

Diplomová práca

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

Bratislava, 201. **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

.....tu bude poďakovanie.....

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

Úvod

Pár slov na úvod...

1 Úvod do súčasneho stavu problematiky

Myšlienka regulárnych výrazov bola prvýkrát spomenutá vo formálnych jazykoch a automatoch ako iný spôsob popisu regulárnych jazykov. Vtedy pozostávali z operácií zjednotenia, zreťazenia a Kleeneho uzáver. Pre ich jednoduchosť 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 menej znakmi - ostatné otvárali dvere k popisu úplne nových jazykov.

Zmes týchto operácií nazývame moderné regulárne výrazy a zaujíma nás, kam sa až dostali v popisovaní jazykov v rámci Chomského hierarchie. Túto problematiku sme z väčšej časti rozobrali v bakalárskej práci [**?**]. V tejto práci rozbor dokončíme a pozrieme sa na ne aj z hľadiska popisnej zložitosti.

1.1 Základné definície

Formálna definícia je uvedená v [**?**], tu si len neformálne uvedieme, s ktorými operáciami pracujeme a ako fungujú. Každý regulárny výraz sa skladá zo znakov a metaznakov. Znaky sú symboly, ktoré charakterizujú samé seba, teda $L(a) = \{a\}$. Metaznaky popisujú operáciu nad regulárnymi výrazmi. Ak potrebujeme v slove zhodu s nejakým metaznakom, stačí pred neho dať \, teda $L(\backslash x) = \{x\}$. Ak metaznak vyžaduje vstup, je ním posledný znak/metaznak/uzátvorkovaný podvýraz pred ním. Metaznaky vyzerajú nasledovne:

- () - okrúhle zátvorky slúžia na oddeľovanie podvýrazov

- {} - kučeravé zátvorky - používané ako $\{n, m\}$ (opakuj aspoň n a najviac m-krát) a $\{n\} = \{n, n\}$ (opakuj n-krát)
- [ ] - hranaté zátvorky - znaky vnútri tvoria množinu, z ktorej si vyberáme. Vieme použiť aj intervaly, napr. a-z, A-Z, 0-9, ... a kombinovať. Všetky metaznaky vnútri [ ] sa považujú za normálne znaky.
- | - operácia zjednotenia
- \ - robí z metaznakov obyčajné znaky
- . - ľubovoľný znak
- $*$ - Kleeneho uzáver, opakuj ľubovoľný počet krát
- $+$ - opakuj 1 alebo viackrát
- ? - ak samostatne: opakuj 0 alebo 1-krát, ak za operáciou: namiesto greedy implementácie použi minimalistickú, t.j. zober čo najmenej znakov (platí pre $*, +, ?, \{n, m\}$)[1]
- ˆ - začiatok slova; špeciálnym prípadom je výraz $[\hat{}\alpha]$ (kde α je nejaká množina znakov), ktorý špecifikuje ľubovoľný znak, ktorý sa v množine α nenacháza
- \$ - koniec slova

Okrem operácií označených metaznakmi vznikli aj zložitejšie operácie, na ktorých popis treba dlhšie konštrukcie. V prvom rade sa zaviedlo číslovanie okrúhlych zátvoriek. Čísluje sa zľava doprava podľa poradia ľavej (otváracej) zátvorky. Toto číslovanie sa deje automaticky pri každom behu algoritmu, teda už pri písaní výrazu ho môžeme využívať. Popíšme si konečne spomínané zložitejšie operácie:

- **komentár** ozn. (?#*text*) - klasický komentár, určený čitateľom kódu; nemá vplyv na výraz, algoritmus ho ignoruje
- **spätné referencie** ozn. $\backslash k$ - môže sa nachádzať na ľubovoľnom mieste vo výraze ZA k-tou pravou (zatváracou) zátvorkou. Odkazuje sa na k-te zátvorky, 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 čaká, keď vo výraze vidí $\backslash k$.

[1] všetky spomenuté operácie "žerú-znaky a na ich implementáciu sa použil greedy algoritmus

- **lookahead**
  - **pozitívny** ozn. $(? = \alpha)$, kde α je nejaký moderný regulárny výraz - tzv. nazeranie dopredu pracuje tak, že si zapamätáme miesto, na ktorom lookahead začíname vykonávať, potom ho vykonáme. Ak vyhlásil zhodu, vrátime sa naspäť na zapamätané miesto a odtiaľ hľadáme zhodu akokeby tam lookahead nikdy nebol. Inak povedané lookahead nevyžiera písmenká a robí akýsi prienik. Ak neobsahuje znak pre koniec slova, môže končiť kdekoľvek - hneď ako zistil zhodu.
  - **negatívny** ozn. $(?! \ \alpha)$, kde α je nejaký moderný regulárny výraz - hľadá slovo z komplementu jazyka popísaného α popísaným spôsobom
- **lookbehind**
  - **pozitívny** ozn. $(? <= \alpha)$, kde α je nejaký moderný regulárny výraz - tzv. nazeranie dozadu, hľadá slovo z $L(\alpha)$ naľavo od aktuálneho miesta v slove (musí končiť hneď vedľa aktuálneho miesta). Opäť nie je určená hranica slova, môže začínať kdekoľvek, ak nie je vynútený začiatok slova znakom ˆ.

    Ak by sme chceli deterministický algoritmus, vyzeral by nasledovne: symbol v slove, na ktorom stojíme, bude teraz pre nás znak pre koniec slova - endmarker. Najprv vyskúšame symbol naľavo, či patrí do jazyka. Ak nie skúsime čítať o jeden znak viac (2 symboly naľavo), keď neuspejeme, opäť posunieme pomyselný začiatok slova doľava. Ak akceptujeme, môže to byť len na endmarkeri. Ak sme neakceptovali a začiatok slova nejde viac posunúť, zamietneme.
  - **negatívny** ozn. $(? <! \ \alpha)$, kde α je nejaký moderný regulárny výraz - hľadá slovo z komplementu $L(\alpha)$ popísaným spôsobom

Pre lookahead a lookbehind používame spoločný názov <u>lookaround</u>, takisto s prívlastkom pozitívny/negatívny myslíme iba ich pozitívne/negatívne verzie.

Keďže sa týmito operáciami budeme zaoberať podrobnejšie, uvedieme pre upresnenie ich definície.

Definícia 1.1.1 (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 1.1.2 (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 1.1.3 (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.*

Definícia 1.1.4 (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.*

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

Máme veľkú množinu operácií a budeme chcieť pracovať aj s jej podmnožinami, preto si zavedieme názvoslovie pre ich jednoznačnejšie a kratšie určenie.

regex - množina operácií, pomocou ktorých vieme popísať iba regulárne jazyky; presnejšie všetky znaky a metaznaky (bez zložitejších operácií)

e-regex - regexy so spätnými referenciami

le-regex - e-regexy s pozitívnym lookaroundom

nle-regex - le-regexy s negatívnym lookaroundom

Eregex - trieda jazykov nad e-regexami

LEregex - trieda jazykov nad le-regexami

nLEregex - trieda jazykov nad nle-regexami

1.2 Vlastnosti a sila moderných regulárnych výrazov

TODO!!!opraviť referencie na vety z bakalárky

Potrebné vety z bakalárskej práce:

Veta 1.2.1 (Veta 2.2.5.). *Regulárne jazyky sú uzavreté na lookaround.*

Veta 1.2.2 (Veta 2.2.10.). *Trieda nad regexami s pozitívnym lookaroundom je $\mathcal{R}$.*

Veta 1.2.3 (Veta 2.2.14.). $LEregex \subseteq \mathscr{L}_{CS}$

1.3 Popisná zložitosť moderných regulárnych výrazov

V čase, keď sme hľadali články týkajúce sa popisnej zložitosti moderných regulárnych výrazov, nenašli sme nič týkajúce sa danej problematiky. Známe boli len výsledky pre regulárne výrazy s operáciami zjednotenia, zreťazenia a Kleeneho $*$ (RE), prípadne ešte prieniku a komplementu (GRE). Navyše mali ešte znak pre prázdny jazyk $\emptyset$ a prázdne slovo ε.

Spomeňme si najprv ako možno zadefinovať popisnú zložitosť [?] [?]:

Nech E je regulárny výraz, potom

- $|E|$ je jeho celková dĺžka
- $rpn(E)$ je jeho celková dĺžka v poľskej normálnej forme
- $|alpha(E)| = N(E)$ 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 maximum symbolov v zjednotení (duálne k L)

	Alphabetical Symbol	E ∪ F	E · F	E*
N	1	N(E)+N(F)	N(E)+N(F)	N(E)
H	0	max(N(E), N(F))	max(N(E),N(F))	N(E)+1
L	1	max(N(E),N(F))	N(E)+N(F)	N(E)
W	1	N(E)+N(F)	max(N(E),N(F))	N(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 [**?**]:

Definícia 1.3.1. *Nech E je regulárny výraz nad abecedou Σ 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)=ε*
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 origináli 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). 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 **nedeliteľný** (irreducible).*

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

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

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

(a) Ak dĺžka najkratšieho slova v L je n, potom $|alph(E)| \geq n$ pre ľubovoľný regulárny výraz E, kde L(E) = L.

(b) Ak naviac L je konečný a dĺžka najdlhšieho slova v L je n, potom $|alph(E)| \geq n$ 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 $|alph(E)| = n$. 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 Σ, 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

2.1 Vlastnosti a sila moderných regulárnych výrazov

V bakaláskej práci sme zabudli ... ***negatívny lookaround*** TODO!!!

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 $L = L_1(?!L_2)L_3 \in \mathcal{R}$.

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 $L(A_i) = L_i, i \in \{1, 2, 3\}$. Zostrojíme NKA A pre L.

Konštrukcia bude veľmi podobná ako pre pozitívny lookahead, 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[1]. Ak A_2 dosiahne akceptačný stav, negatívny lookahead musí zamietnuť.

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 Σ_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 $\Sigma_2 \cup \Sigma_3$.

Poďme 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

[1] Ak sa zasekne, slovo z L_2 tam určite nebude, lebo neexistuje akceptačný výpočet. Ak sa nezasekne, nevieme povedať o tom slove nič, pokiaľ ho neprejdeme celé.

[2] Inak $A_2' = A_2$.

nový stav[3] $K_2' = K_2 \cup \{q_{ZLE}\}$ 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:

$$H_0 = F_2'$$

$$H_{i+1} = \{q \in K_2' \mid \exists p \in H_i\ \exists a \in \Sigma_2' :\ \delta(q,a) = p\}$$

Zrejme $\exists i \in \mathbb{N} :\ H_{i+1} = H_i = H$. Nás zaujíma množina $K_2' \setminus H$, v ktorej sa nachádzajú všetky stavy, z ktorých sa na žiadne slovo nie je možné dostať do žiadneho akceptačného stavu.

$A_2'' = (K_2'', \Sigma_2'', \delta_2'', q_0, F_2'')$: $K_2'' = K_2' \cup \{q_A, q_Z\}$, $\Sigma_2'' = \Sigma_2'$, $F_2'' = (K_2' \setminus H) \cup \{q_A\}$

$$\begin{aligned}
\forall q \in K_2' \setminus F_2'\ \forall a \in \Sigma_2' &\ :\ \delta_2''(q,a) = \delta_2'(q,a) \\
\forall q \in K_2' \setminus F_2'\ \forall a \in \Sigma_2' &\ :\ \delta_2''(q,\varepsilon) = q_A \\
\forall q \in F_2'\ \forall a \in \Sigma_2' &\ :\ \delta_2''(q,a) = q^4 \\
\forall a \in \Sigma_2' &\ :\ \delta_2''(q_A,a) = q_Z
\end{aligned}$$

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, δ-funkcia ho pošle do stavu q_Z. Podľa toho je zrejmé, že ak existuje akceptačný výpočet a dočítal slovo do konca, je práve jeden[5] [6].

[3]Pre každý nový stav predpokladáme, že je jedinečný - t.j. žiaden taký v množine stavov ešte nie je.

[4]Tento riadok v podstate netreba, A_2'' sa môže rovno zaseknúť, lebo A_2' práve akceptoval.

[5]Okrem prípadu, kedy dočíta slovo a je v stave z $(K_2' \setminus H)$ - vieme ho predĺžiť o krok na ε a skončiť v q_A. Jadro výpočtu ale zostáva rovnaké - jednoznačné.

[6]Zaujímavé je, že aj zamietací výpočet je pre každé slovo jednoznačný

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[7]. 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 δ-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 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 ε do q_A (2.riadok definície δ_2'') a akceptuje.

Z práve dokázaného tvrdenia a vety 1.2.1 už vyplýva aj platnosť našej lemy. □

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 $L = L_1(? <!\ L_2)L_3 \in \mathcal{R}$.

Nemôžme skonštruovať A_2'' také, že bude akceptovať práve vtedy, keď $(? < !L_2)$, pretože nevieme čítať doľava. Tzn. zaručiť, že miesto, kde začína A_2'' výpočet je skutočný začiatok slova. Mechanizmus lookbehindu musí byť riadený zvonku, automatom pre L. Skonštruujeme ho. $C = (K, \Sigma, \delta, q_0, F)$:

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

[7]Sem spadá aj prípad, keď A_2 prečítal neznámy znak a zasekol sa - A_2' zostáva až do konca slova v stave q_{ZLE}.

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

Druhý riadok δ-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 vtedy, keď sa C rozhodne, že A_1 končí výpočet. Ak tam je, nedostane sa k simulovaniu A_3 a tým ani k akceptačným stavom.

$L(C) = L$.

$\subseteq$: 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 $\nexists xy$ 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. □

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

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 vety 1.2.2. □

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. TODO!!! □

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

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

Veta 2.1.8. *Trieda LEregex je uzavretá na zreťazenie.*

Dôkaz. Nech sú le-regexy α, β. Chceme ukázať, že jazyk $L(\alpha)L(\beta) \in LEregex$. Intuitívne nás to vedie k riešeniu $\alpha\beta$, čo ale nemusí byť vždy správne. Problémom sú operácie lookaround, presnejšie každý lookahead v α môže zasahovať do slova z $L(\beta)$ a takisto každý lookbehind z β môže zasahovať do slova z $L(\alpha)$. Navyše, ak lookahead obsahuje \$ a lookbehind ˆ, potom zasahujú do slova z iného jazyka určite. V takom prípade môže le-regex $\alpha\beta$ vynechať niektoré slová z L_1L_2 a tiež pridať nejaké nevhodné slová naviac. Preto treba nájsť spôsob, ako operácie lookaroundu vhodne 'skrotiť', predpokladajme, že α má k označených zátvoriek:

$$(? = \underset{1}{(}\,\alpha\,\underset{1}{)}\ \underset{k+2}{(}\ \beta\ \underset{k+2}{)}\ \$)\alpha'\backslash k+2(? <= \hat{}\,\backslash 1\beta') \tag{2.1}$$

V α, β treba vhodne prepísať označenie zátvoriek (po poradí). α' je α prepísaný tak, že pre každý lookahead:

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

β' je β prepísaný tak, že pre každý lookbehind:

- bez ˆ - na začiatok pridáme $\hat{}\,\backslash 1.*$
- s ˆ - pred ˆ pridáme $\hat{}\,\backslash 1$

Čo teda robí le-regex 2.1? Nech w je vstupné slovo, ktoré chceme matchovať. Lookahead na začiatku ho nejak rozdelí na w_1 a w_2, pričom $w = w_1w_2$, tak, že do $L(\alpha)$ pridelí w_1 a do $L(\beta)$ podslovo w_2. Ešte musíme overiť, či α matchuje w_1 samostatne.

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

Veta 2.1.9. *LEregex je neporovnateľná s $\mathscr{L}_{CF}$.*

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

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

Vieme, že $L \notin Eregex$, preto musí obsahovať nejaký lookaround. Zároveň z $L \notin \mathcal{R}$ a 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 2.1.10. $LEregex \subsetneq \mathscr{L}_{CS}$

Veta 2.1.11. $nLEregex \subseteq \mathscr{L}_{CS}$

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

Nech α je nle-regex. Potom A je LBA pre α, ktorý ignoruje negatívny lookaround (t.j. vyrábame LBA pre le-regex). Teraz vytvoríme LBA B pre le-regex vnútri negatívneho lookaroundu. Z nich vytvoríme LBA C pre úplný nle-regex α tak, že 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 lookarounde. 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 le-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[8], raz sa mu musia minúť. To znamená, že ak nemá v δ-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 lookarounde aj jeho negatívna verzia môže obsahovať nle-regex, t.j. vnorený (negatívny) lookaround. Tých však môže byť iba konečne veľa, keďže každý nle-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 2.1.12. *nLEregex je neporovnateľná s* $\mathscr{L}_{CF}$.

Dôkaz. Opäť použíjeme jazyk $L = \{a^n b^n \mid n \in \mathbb{N}\}$ a budeme dokazovať sporom. Nech $L \in LEregex$.

Z vety **??** vieme, 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. □

Dôsledok 2.1.13. *nLEregex* $\subsetneq \mathscr{L}_{CS}$

Veta 2.1.14. $LEregex \subseteq NSPACE(\log n)$*, 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[9] a pridáme doňho špeciálny znak ►, ktorým si budeme ukazovať, kam sme sa v regexe dopracovali – bude to akýsi smerník

[8] 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 nie, algoritmus odoberá hviezdičke znaky dovtedy, dokým nenájde zhodu alebo odoberie všetky znaky - vtedy sa mu minuli všetky možnosti a môže slovo neakceptovať.

[9] Každý regex je má z definície konečnú dĺžku, to znamená aj konečný počet stavov.

na znak, ktorý práve spracovávame. Teda stav bude vyzerať takto: $q_{\beta \blacktriangleright \xi}$ (pre lepšiu čitateľnosť budeme uvádzať namiesto stavu iba jeho dolný index: $\beta \blacktriangleright \xi$), kde $\beta\xi = \alpha$, časť β sme už namatchovali na vstup a práve sa chystáme pokračovať časťou ξ. 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.

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 $m.\log n \in O(\log n)$. 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 α.

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

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

δ : (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 polootvorený interval – znak na koncovej adrese do podslova nepatrí.

[10] T.j. všetky znaky a metaznaky použité v regexe α.

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

$\delta(\beta \blacktriangleright \backslash k\xi, a) = \{(q_{najdi\_zaciatok(k)}, 0)\}$ – 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 nesedí, 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 zbehutí 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: $\delta(\gamma \blacktriangleright, a) = \{(\beta(? = \gamma) \blacktriangleright \xi, 0)\}$.

$\delta(\beta \blacktriangleright (? <= \gamma)\xi, a) = \{(do\text{ľ}ava\_\gamma, 0)\}$ – 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 vrátime o niekoľko políčok doľava ($\delta(do\text{ľ}ava\_\gamma, a) = \{(do\text{ľ}ava\_\gamma, -1), (\blacktriangleright \gamma, 0)\}$) 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 $p.\log n$ 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 α 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.

$\square$

Veta 2.1.15 (Savitch)**.** *Nech $S(n) \geq \log n$ je páskovo konštruovateľná, potom*

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

Dôsledok 2.1.16. *$LEregex \subseteq DSPACE(\log^2 n)$, kde $n > 1$ je veľkosť vstupu.*

Veta 2.1.17. *$nLEregex \subseteq DSPACE(\log^2 n)$, kde $n > 1$ je veľkosť vstupu.*

Dôkaz. TODO!!! $\square$

Definícia 2.1.18. *$L(regex\#word)$ nazývame jazyk takých slov, kde regex je validný regulárny výraz z R a slovo word $\in L(R)$. Množina $R \in \{Eregex, LEregex, nLEregex\}$ musí byť špecifikovaná.*

Definícia 2.1.19. ***Konfiguráciou*** *regexu $\alpha = r_1 \dots r_n$ nazývame dvojicu (r, w), kde $r \in (\lceil\alpha) \cup (\alpha\lceil) \cup (r_1 \dots \lceil r_i \dots r_n)$ $w \in \Gamma^* \lceil\ \Gamma^*$ a sybmol $\lceil$ ukazuje, kde sa nachádzame vo výpočte v regexe a v slove.*

Definícia 2.1.20. *Konfiguráciu* $(r_1 \dots r_n\lceil, w_1 \dots w_m\lceil)$ *nazývame* ***akceptačná****.*

Definícia 2.1.21. *Zátvorka (v regexe* α *je* ***indexovateľná****, ak sa bezprostredne za ňou nenachádza metaznak ? a zárorveň 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 2.1.22. *Zátvorka) v regexe* α *je* ***indexovateľná****, ak k nej prislúchajúca otváracia zátvorka je indexovateľná.*

Definícia 2.1.23. *Regex* α *je* ***alternovateľný****, ak zodpovedá jednému z týchto tvarov:*

- *prázdny regex*
- $a \in \Sigma$
- $a*$*, kde* $a \in \Sigma$
- (β)*, kde* β *je ľubovoľný regex z rovnakej triedy ako* α
- $(\beta)*$*, kde* β *je ľubovoľný regex z rovnakej triedy ako* α
- $\backslash k$
- $\backslash k*$

Lema 2.1.24. *Alternácia používa iba alternovateľné regexy.*

Dôkaz. Dokážeme sporom. Majme regex $\alpha = \beta|\gamma$, kde β, γ sú všetky členy alternácie a nech β nie je alternovateľná. Môžu nastať 3 prípady:

1. β má na konci $|$. Potom $\beta = \beta'|$ a členy alternácie v α sú β', γ a prázdny regex. To je spor s predpokladom.

2. β končí lookaroundom, ale lookaround sa nedá alternovať.
3. $\beta = \beta_1\beta_2$. Alternácia berie posledný ucelený regex, takže β nemôže byť členom alternácie.

□

TODO!!!: krok výpočtu/krok zhody/posun/...

Definícia 2.1.25. ***Krok výpočtu*** *je relácia* $\vdash$ *na konfiguráciách definovaná nasledovne:*

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. Nech (je indexovateľná, k-ta v poradí:

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

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

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

III. Nech) je indexovateľná, k-ta v poradí:

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

IV. Nech podslová $\alpha_1, \alpha_2, \dots, \alpha_A$ *regexu* α *sú všetkými členmi zobrazenej alternácie:* $(r_1 \dots \lceil \alpha_1 | \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

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

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

$$\dots$$

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

V. $(r_1 \dots \underset{k}{(} r_i \dots r_l \underset{k}{)} \lceil * \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$[11]

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

[11] 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.

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

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

$$\vdash (r_1 \dots \lceil \backslash k \dots r_n, w_1 \cdots \mathsf{T}\, \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$$

VII. $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \cdots \mathsf{T}\, \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$

$$\vdash (r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a}\, \mathsf{T} \dots \overset{k'}{w_b} \dots w_j \lceil \dots w_m)$$

VIII. $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \cdots \mathsf{T}\, w_c \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$ *a zároveň* $w_c = w_j$[12]

$$\vdash (r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots w_c\, \mathsf{T} \dots \overset{k'}{w_b} \dots w_j \lceil \dots w_m)$$

IX. $(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \cdots \mathsf{T}\, \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$

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

X. *Nech* $(? = \dots)$ *je k-ty pozitívny lookahead v poradí:*
$(r_1 \dots \lceil (? = \dots) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

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

XI. *Nech) patrí ku k-temu pozitívnemu lookaheadu v poradí:*
$(r_1 \dots (? = \dots \lceil) \dots r_n, w_1 \dots \overset{k\rightarrow}{w_l} \dots \lceil w_j \dots w_m)$

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

XII. *Nech* $(? < \dots)$ *je k-ty pozitívny lookbehind v poradí,* $\forall L \in \{0, \dots, j-1\}$*:*
$(r_1 \dots \lceil (? <= \dots) \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

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

XIII. *Nech) patrí ku k-temu pozitívnemu lookbehindu v poradí:*
$(r_1 \dots (? <= \dots \lceil) \dots r_n, w_1 \dots \lceil \overset{k\leftarrow}{w_j} \dots w_m)$

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

[12] w_c a w_j môžu byť poschodové symboly, avšak pri tejto rovnosti poschodia ignorujeme – chceme porovnať iba písmenká v slove, prislúchajúce týmto pozíciám.

XIV. *Ak* $\nexists p \in \{j, \ldots, m\} : (\lceil r_k \ldots r_l, \lceil w_j \ldots w_p) \vdash^* (r_k \ldots r_l \lceil, w_j \ldots w_p \lceil)$, *potom:*
$(r_1 \ldots \lceil(?!\ r_k \ldots r_l) \ldots r_n, w_1 \ldots \lceil w_j \ldots w_m)$

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

XV. *Ak* $\nexists p \in \{1, \ldots, j-1\} : (\lceil r_k \ldots r_l, \lceil w_p \ldots w_{j-1}) \vdash^* (r_k \ldots r_l \lceil, w_p \ldots w_{j-1} \lceil)$, *potom:*
$(r_1 \ldots \lceil(?<!\ r_k \ldots r_l) \ldots r_n, w_1 \ldots \lceil w_j \ldots w_m)$

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

Definícia 2.1.26. ***Jazyk*** *generovaný regexom* α *je množina*

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

Poznámka. Postupnosť konfigurácií $(\lceil \alpha, \lceil w) \vdash^* (\alpha \lceil, w \lceil)$ pre daný regex α a slovo w nazývame **akceptačný výpočet**.

Lema 2.1.27. *Nech* $\alpha \in LEregex$ *a* $w \in L(\alpha)$. *Potom existuje akceptačný výpočet, ktorý má najviac* $5 * |\alpha| * |w|$ *konfigurácií.*

Dôkaz. Pri prechodoch medzi konfiguráciami I., II., III., IV., V.(1), VII., VIII., X., XIII., XIV. a XV. 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: VI. a IX. 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 $(2\,|\alpha|)$-krát.

Zostali konfigurácie, v ktorých ukazovateľ skáče dozadu, rozoberme si ich postupne:

V.(2) – v prípade Kleeneho $*$ nastáva ďalšie opakovanie. Keďže $w \in L(\alpha)$, existuje akceptačný výpočet. Takýchto výpočtov môže byť viac, napríklad ak sa v α vyskytuje regex $(\beta)*$ taký, že $\varepsilon \in \beta$. Potom existuje nekonečne veľa výpočtov, ktoré sa líšia v počte prechodov týmto regexom. Preto si vyberieme najkratší akceptačný výpočet. To znamená, že v každom prechode cez regex opakovaný operáciou $*$ musíme nájsť zhodu aspoň s jedným písmenkom z w (t.j. ukazovateľ v slove w sa pohne aspoň o 1 pozíciu doprava). Regex opakovaný Kleeneho $*$ je dlhý najviac $|\alpha|$-1 znakov a opakujeme ho najviac w-krát. Z toho vidíme, že existuje akceptačný výpočet, ktorý má najviac $|\alpha|*|w|$ konfigurácií.

XI., XII. – ukončenie operácie lookahead, začiatok operácie lookbehind. V tomto prípade ukazovateľ skáče dozadu v slove. Podstatné však je, že v regexe sa posúva dopredu. Takýchto prechodov vieme spraviť najviac $|\alpha|$.

Spolu je to najviac $|\alpha| + |w| + 2|\alpha| + |\alpha| * |w| + |\alpha| \leq 5 * |\alpha| * |w|$ konfigurácií.

□

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 písmenkom.

V nasledujúcej časti sa budeme zaujímať akú zložitosť dostaneme, ak dostaneme na vstup aj regex aj slovo, o ktorom chceme zistiť, či patrí do jazyka daného regexu. Údaje budú oddelené oddeľovačom #, o ktorom budeme predpokladať, že sa nenachádza ani v slove ani v regexe.

Veta 2.1.28. *$L(regex\#word) \in NSPACE(r \log w)$, kde $r = |regex|$, $w = |word|$ a $regex \in LEregex$.*

Dôkaz. TODO!!! □

Veta 2.1.29. *$L(regex\#word) \in DSPACE(n \log^2 n)$, kde $regex \in LEregex$ a n je dĺžka vstupu.*

Dôkaz. TODO!!!

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

Zostrojíme Turingov stroj, ktorý bude akceptovať jazyk $L(regex\#word)$ a na každej pracovnej páske použije najviac $O(n \log^2 n)$ políčok. TS bude mať vstupnú read-only pásku a 3 pracovné pásky.

Na prvej páske budú postupne enumerované všetky možné pozície, kam sa môžu odkazovať spätné referencie a kde môžu začínať operácie lookahead a lookbehind. Keďže regex zistíme až zo vstupu, počet adries si bude musieť TS najprv zistiť – hneď na začiatku výpočtu prejde celý regex a umiestni na pásku potrebný počet oddelovačov # (pre každé zátvorky 2, pre lookahead 1, pre lookbehind 2). Potom pred každý oddeľovač umiestni 0, čo je počiatočný stav na tejto páske. Pre každé nastavenie adries spustí procedúru, ktorú si popíšeme neskôr. Cez adresy bude iterovať ako cez a-ciferné číslo (a je počet oddeľovačov = počet potrebných adries) z ciframi z $\{0, \ldots, \log w$. Keďže vie rozlíšiť, kde začína slovo, stačí adresovať od tohto miesta a adresy vieme písať v logaritmickej dĺžke od dĺžky slova. □

2.2 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 le-regex z [**?**, Poznámka 1.] $\beta = ((? = (a^m) * \$)a^{m+1}) * a\{1, m-1\}\$$. 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. Zaujímavé je, že le-regex využíva iteráciu $(m-1)$–krát a napriek tomu je konečný.

Záver

Zhrnutie na záver...