Moderné regulárne výrazy

Tatiana Tóthová*

Školiteľ: Michal Forišek†

Katedra informatiky, FMFI UK, Mlynská Dolina 842 48 Bratislava

Abstrakt: Regulárne výrazy implementované v súčasných programovacích jazykoch ponúkajú omnoho viac operácií ako pôvodný model z teórie jazykov. Už konštrukciou spätných referencií bola prekročená hranica regulárnych jazykov. Náš model obsahuje naviac konštrukcie lookahead, lookbehind a ich negatívne verzie. V článku uvádzame zaradenie modelu zodpovedajúcej triedy jazykov do Chomského hierarchie, vlastnosti tejto triedy a výsledky z oblasti priestorovej zložitosti.

Kľúčové slová: regulárny výraz, regex, lookahead, lookbehind, spätné referencie, negatívny lookaround

1 Úvod

Regulárne výrazy vznikli v 60tych rokoch v teórii jazykov ako ďalší model na vyjadrenie regulárnych jazykov. Z ich popisu ľudský mozog rýchlejšie pochopil o aký jazyk sa jedná, než zo zápisu konečného automatu, či regulárnej gramatiky. Ďalšou výhodou bol kratší a kompaktný zápis.

Vďaka týmto vlastnostiam boli implementované ako vyhľadávací nástroj. Postupom času sa iniciatívou používateľov s vyššími nárokmi pridávali nové konštrukcie na uľahčenie práce. Nástroj takto rozvíjali až do dnešnej podoby. My sa budeme opierať o špecifikáciu regulárnych výrazov v jazyku Python [Python documentation, 2012].

Ako čoskoro zistíme, nové regulárne výrazy vedia reprezentovať zložitejšie jazyky ako regulárne, preto je dobré ich nejako odlíšiť. V literatúre sa zaužíval výraz „regex" z anglického *regular expression*, ktorý budeme používať aj my.

1.1 Základná definícia

Regulárne výrazy sú zložené zo znakov a metaznakov. Znak a predstavuje jazyk $L(a) = \{a\}$. Metaznak alebo skupina metaznakov určuje, aká operácia sa so znakmi udeje. Základné operácie sú zreťazenie (je definované tým, že regulárne výrazy idú po sebe, bez metaznaku), Kleeneho uzáver ($(0-\infty)$-krát zopakuj výraz, metaznak $*$) a alternácia (vyber výraz naľavo alebo napravo, metaznak $|$). Naviac sa využíva metaznak \, ktorý robí z metaznakov obyčajné znaky a okrúhle zátvorky na logické oddelenie regulárnych výrazov.

Pre regulárny výraz α a slovo $w \in L(\alpha)$ hovoríme, že α vyhovuje slovu w resp. α matchuje slovo w. Tiež budeme hovoriť, že α generuje jazyk $L(\alpha)$.

1.2 Nové jednoduché konštrukcie

- $+$ – Kleeneho uzáver opakujúci $(1-\infty)$-krát
- $\{n,m\}$ ($\{n\}$) – opakuj regulárny výraz aspoň n a najviac m-krát (opakuj n-krát)
- $[a_1a_2\dots a_n]$ – predstavuje ľubovoľný znak z množiny $\{a_1,\dots,a_n\}$
- $[\hat{}a_1a_2\dots a_n]$ – predstavuje ľubovoľný znak, ktorý nepatrí do množiny $\{a_1,\dots,a_n\}$
- . – predstavuje ľubovoľný znak
- ? – 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 – metaznak označujúci začiatok slova
- $ – metaznak označujúci koniec slova
- (?# komentár) – komentár sa pri vykonávaní regexu úplne ignoruje

Všetky tieto konštrukcie sú len „kozmetickou" úpravou pôvodných regexov – to isté vieme popísať pôvodnými regulárnymi výrazmi, akurát je to dlhšie a menej prehľadné.

Rozdiely medzi minimalistickou a greedy verziou operácií vníma iba používateľ, pretože ak existuje zhoda regexu so slovom, v oboch prípadoch sa nájde. Viditeľné sú až pri výstupnej informácii pre používateľa, ktorú môže použiť ďalej a ktorá je v teoretickom modeli nepotrebná.

*tothova166@uniba.sk

†forisek@dcs.fmph.uniba.sk

[1] všetky spomenuté operácie sú implementované greedy algoritmom

1.3 Zložitejšie konštrukcie

Spätné referencie

Najprv potrebujeme očíslovať všetky zátvorky v regexe. Číslujú sa všetky, ktoré nie sú tvaru (?...). Poradie je určované zľava doprava podľa otváracej zátvorky.

Spätné referencie umožňujú odkazovať sa na konkrétne zátvorky. Zápis je $\backslash k$ a môže sa nachádzať až za k-tymi zátvorkami. Skutočná hodnota $\backslash k$ sa určí až počas výpočtu – predstavuje podslovo zo vstupného slova, ktoré matchovali k-te zátvorky. Ak je takých viac, platí vždy to posledné. Regex $\alpha\underset{k}{(}\beta\underset{k}{)}\gamma\backslash k\delta$ na w:

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

$$\text{platí} \quad w_k = x_i \dots x_{j-1} = x_l \dots x_{m-1}$$

Lookahead

Zapísaný formou (?=...), vnútri je validný regex.

Keď v regexe prídeme na pozíciu lookaheadu, zoberieme regex vo vnútri a snažíme sa v slove matchovať ľubovoľný prefix zostávajúcej časti slova. Ak sa to podarí, pokračujeme vo vonkajšom regexe ďalej a v slove od pozície, kde lookahead začínal (tzn. akokeby v regexe nikdy nebol). Regex $\alpha(?\!=\!\beta)\gamma$:

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

Má aj negatívnu verziu – negatívny lookahead (?! ...). Vykonáva sa rovnako ako lookahead, ale má opačnú akceptáciu. Teda ak neexistuje vyhovujúci prefix, akceptuje.

Lookbehind

Zapísaný formou (?<=...), vnútri je validný regex.

Pri výpočte zoberieme regex vnútri lookbehindu a snažíme sa vyhovieť ľubovoľnému sufixu už matchovanej časti slova. Ak vyhovieme, pokračujeme v slove a regexe akoby tam lookbehind vôbec nebol. Regex $\alpha(?\!<\!=\!\beta)\gamma$ matchuje slovo w:

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

Aj lookbehind má negatívnu verziu – negatívny lookbehind (?<! ...) – a má otočenú akceptáciu podobne ako negatívny lookahead.

Lookahead a lookbehind (spolu nazývané jedným slovom **lookaround**) sú v rôznych implementáciách rôzne obmedzované, aby výpočet netrval príliš dlho. V teórii tieto obmedzenia ignorujeme a prezentujeme model v plnej sile – výsledky tak prezentujú hornú hranicu toho, čo implemetnácie dokážu.

1.4 Priorita

Pri interakcii toľkých operácií je nutné vedieť ich priority. Existujú také, ktoré sa správajú ako znak, čomu zodpovedajú [ ],[^], . a každá spätná referencia. Špeciálne sú lookahead a lookbehind – tie sa vykonajú hneď ako na ne narazíme. Ostatné zoradíme v tabuľke:

priorita	3	2	1	0
operácia	()	$* + ? \{\}$	zreťazenie	$\vert$

1.5 Triedy a množiny

Kvôli porovnávaniu a vytvoreniu hierarchie sme rozdelili operácie do niekoľkých množín:

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

Eregex – *Regex* so spätnými referenciami

LEregex – *Eregex* s pozitívnym lookaroundom

nLEregex – *LEregex* s negatívnym lookaroundom

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

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

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

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

Trieda $\mathscr{L}_{ERE}$ už bola hlbšie preskúmaná a výsledky čerpáme z článkov [Câmpeanu et al., 2003] a [Carle and Nadendran, 2009].

2 Formalizácia modelu

Pri zložitejších dôkazoch sa ukázala potreba lepšieho formalizmu, než len množiny operácií. Kvôli jednoduchosti sme vybrali len potrebné operácie – zreťazenie, alternáciu, Kleeneho $*$, spätné referencie a pozitívny a negatívny lookaround – a pokúsili sa popísať model, ktorý pracuje v krokoch podobne ako Turingov stroj.

Základným prvkom je **konfigurácia**. Je to dvojica regex $r_1 \dots r_n$ a vstupné slovo $w_1 \dots w_m$, pričom v oboch reťazcoch sa navyše nachádza ukazovateľ pozície $\lceil$ (ako hlava Turingovho stroja): $(r_1 \dots \lceil r_i \dots r_n, w_1 \dots \lceil w_j \dots w_m)$. Špeciálne rozoznávame počiatočnú konfiguráciu $(\lceil r_1 \dots r_n, \lceil w_1 \dots w_m)$ a akceptačnú konfiguráciu $(r_1 \dots r_n \lceil, w_1 \dots w_m \lceil)$.

Znaky slova $(w_1 \dots w_m)$ v konfigurácii budú niesť nejakú informáciu naviac, preto použijeme poschodové symboly. Na najspodnejšom poschodí bude uchovaná informácia o skutočnom znaku na zodpovedajúcej pozícii v slove a nebude sa meniť. Obsah vrchných poschodí bude špecifikovaný v kroku výpočtu. Z celého poschodového symbolu budeme zobrazovať vždy len tú informáciu, ktorú práve potrebujeme, tzn. w_j bude predstavovať iba znak na najspodnejšom poschodí. Nezabúdajme však, že na ostatných poschodiach môže mať zapísané čokoľvek, čo možno neskôr v inom kroku využijeme.

Najprv si definujeme potrebné pojmy indexovateľnosti a alternovateľnosti. *Indexovateľné zátvorky* sú také, kde za otváracou zátvorkou nenasleduje ? (t.j. všetky prípady okrem lookaroundu). Tieto zátvorky budeme číslovať. *Alternovateľný regex* je taký, ktorý sa môže vyskytovať v alternácii. Sú 3 prípady: regex sa môže nachádzať naľavo od |, napravo od | alebo je z oboch strán ohraničený |. Ak alternácia nie je uzavretá zátvorkami, ľavý a pravý krajný regex siaha až ku kraju slova, pretože alternácia je operácia s najmenšou prioritou. Inak sú pre nich hranicou zátvorky uzatváracie alternáciu.

Vďaka definovaniu týchto pojmov vidíme, že vieme algoritmicky zistiť, ktoré zátvorky sú indexovateľné a ktoré regexy sú alternovateľné.

Definícia 1. ***Krok výpočtu*** *je relácia $\vdash$ na konfiguráciách definovaná nasledovne:*

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

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

II. *zápis adresy zátvorky (*
Nech (je indexovateľná, k-ta v poradí:
$(r_1 \dots \lceil (\dots r_n, w_1 \dots \lceil w_j \dots w_m)$

$$(1) \vdash (r_1 \dots (\lceil \dots r_n, w_1 \dots \lceil \overset{k}{w_j} \dots w_m)$$

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

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

III. *zápis adresy zátvorky)*
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. *výber možnosti v alternácii*
Nech podslová $\alpha_1, \alpha_2, \dots, \alpha_A$ regexu α sú všetkými členmi zobrazenej alternácie:
$(r_1 \dots \lceil \alpha_1 | \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

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

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

$$\vdots$$

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

V. *skok z dokončenej možnosti na koniec alternácie*
Nech podslová $\alpha_1, \alpha_2, \dots, \alpha_A$ regexu α sú všetkými členmi zobrazenej alternácie, potom pre všetky možnosti:
$(r_1 \dots \alpha_1 \lceil | \alpha_2 | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$,
$(r_1 \dots \alpha_1 | \alpha_2 \lceil | \dots | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$,
$\vdots$
$(r_1 \dots \alpha_1 | \alpha_2 | \dots | \alpha_{A-1} \lceil | \alpha_A \dots r_n, w_1 \dots \lceil w_j \dots w_m)$

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

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

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

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

VII. *skok Kleeneho $*$ za regexom v ( )*
$(r_1 \dots \underset{k}{(} \dots \underset{k}{)} \lceil * \dots r_n, w_1 \dots \overset{k}{w_a} \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$[2]

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

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

[2] 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.

VIII. *špeciálny ukazovateľ – zjavenie*

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

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

IX. *porovnávanie spätnej referencie*

$(r_1 \dots \lceil\backslash k \dots r_n, w_1 \dots \overset{k}{w_a} \dots \top w_c \dots \overset{k'}{w_b} \dots \lceil w_j \dots w_m)$, *kde* $a \le c < b$ *a zároveň* $w_c = w_j$[3]

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

X. *špeciálny ukazovateľ – zmiznutie*

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

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

XI. *lookahead – začiatok a jeho záznam*

Nech (?= ...) je k-ty pozitívny lookahead v poradí:

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

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

XII. *lookahead – koniec, skok a vymazanie záznamu*

Nech) patrí ku k-temu pozitívnemu lookaheadu v poradí:

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

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

XIII. *lookbehind – začiatok, jeho záznam a skok*

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

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

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

XIV. *lookbehind – koniec a vymazanie záznamu*

Nech) patrí ku k-temu pozitívnemu lookbehindu v poradí:

$(r_1 \dots (?\!<\!= \dots \lceil) \dots r_n, w_1 \dots \lceil\overset{\overleftarrow{k}}{w_j} \dots w_m)$

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

XV. *negatívny lookahead*

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. *negatívny lookbehind*

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:*

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

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

Akceptačný výpočet je postupnosť konfigurácií $(\lceil R, \lceil W) \vdash^* (R\lceil, W\lceil)$. Ak existuje akceptačný výpočet pre daný regex R a slovo W hovoríme, že regex R matchuje slovo W respektívne slovo W vyhovuje regexu R. **Jazyk** vyhovujúci danému regexu je množina slov, pre ktoré existuje akceptačný výpočet.

Vďaka týmto definíciám sme schopný odhadnúť dĺžku výpočtu:

Lema 1. *Nech* $\alpha \in \mathscr{L}_{ERE}$ *a* $w \in L(\alpha)$*. Potom existuje akceptačný výpočet, ktorý má najviac* $3 \cdot |\alpha| \cdot |w|^2$ *konfigurácií.*

Dôkaz. Vo väčšine krokov výpočtu sa posúvame dopredu – buď v regexe alebo v slove alebo v oboch. Takéto kroky vedú k postupnosti dĺžky najviac $|\alpha| + |w|$.

Výpočet môže predĺžiť skákanie ukazovateľa dozadu. V krokoch VI.(2),VII.(2) sa pri Kleeneho $*$ rozhodneme urobiť ďalšiu iteráciu. Zamyslime sa nad samotným akceptačným výpočtom. Ak existuje, potom existuje aj taká jeho verzia, kde každé opakovanie regexu pomocou Kleeneho $*$ matchuje aspoň 1 znak – prázdne iterácie môžeme vyhodiť, lebo ukazovateľ v slove zostal na mieste a konfigurácia skoku je rovnaká, takže postupnosť sa priamo naviaže. Jediná prázdna iterácia môže zostať ako posledná, pokiaľ sa jedná o krok VII.(2) a na zátvorky pred $*$ sa odkazuje spätná referencia – vtedy môže nastať prípad, kedy potrebujeme mať posledné matchované podslovo prázdne. Vieme, že regex opakovaný operáciou $*$ je dlhý nanajvýš $|\alpha|$ znakov a podľa úvahy ho treba opakovať najviac $|w|$-krát[4]. Medzi ná-

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

[4] Keď v každej iterácii matchujeme aspoň 1 písmenko, potrebujeme najviac $|w| - 1$ návratov. Zarátaním možnej poslednej práznej iterácie dostávame $|w|$ iterácií.

vratmi používame nenávratové kroky, ktorých je najviac $|w|+|\alpha|$, takže spolu má akceptačný výpočet najviac $|w|\cdot(|w|+|\alpha|)$ konfigurácií.

V krokoch VIII. a X. sa žiaden ukazovateľ nepohne. Oba kroky sa vyskytujú 1x ku každej spätnej referencii, ktorých je najviac $|\alpha|$. Každá spätná referencia sa môže nachádzať v poli pôsobnosti nejakej $*$, teda môže byť opakovaná najviac toľkokrát, koľko je iterácií a tých je dohromady najviac $|w|$. Tieto kroky nastanú najviac $(|\alpha|\cdot|w|)$-krát.

Z toho vyplýva, že najkratší akceptačný výpočet má najviac $|w|\cdot(|w|+|\alpha|)+|w|\cdot|\alpha| \leq 3\cdot|w|^2\cdot|\alpha|$ konfigurácií. $\square$

Spočítajme počet všetkých konfigurácií pre triedu regexov *LEregex*. Vstupné slovo a regex máme, menia sa iba polohy ukazovateľov a informácií. Ukazovateľ v regexe môže mať $(r+1)$ pozícií a pre ukazovateľ v slove existuje $(w+1)$ rôznych pozícií. Čo sa týka ukazovateľa pre spätné referencie a informácií v poschodových symboloch, každý prvok z tejto množiny môže mať $(w+1)$ pozícií alebo sa v slove nenachádza. Mohutnosť tejto množiny je $m = 1+2\cdot$ (počet spätných ref.) + (počet lookaroundov) $\leq 2r+1$. Počet všetkých možných konfigurácií je dohromady:

$$(r+1)(w+1)(w+2)^m \leq (r+1)(w+2)^{2r+2} \quad (1)$$
$$= O(rw^{2r+2})$$

Keby sme priamo pokračovali v dôkaze lemy 1, odhad by nebol lepší ako odhad 1. Nasledujúca lema to ukazuje všeobecnejšie.

Definujme si **hĺbku vnorenia lookaroundov** ako počet lookaroundov vnorených v sebe. Lookaround obsahujúci regex z *Eregex* má hĺbku 1. Lookaround obsahujúci regex z *LEregex*, kde maximum zo všetkých hĺbok lookaroundov je $h-1$ má hĺbku h.

Lema 2. *Majme triedu regexov takú, že funkcia $f(r)$ určuje ich maximálnu hĺbku vnorenia lookaroudov, kde r je dĺžka regexu. Potom počet konfigurácií v akceptačnom výpočte pre slovo dĺžky w je najviac $O(rw^2(rw)^{f(r)})$.*

Dôkaz. Dokážeme si to matematickou indukciou vzhľadom na hĺbku vnorenia lookaroundov h.

Báza indukcie: Nech $h=1$. V regexe je $k \leq r$ lookaroundov, ktoré vnútri nemajú žiaden lookaround. Keďže lookaroundu nie sú priradené žiadne písmenká zo vstupu, môžeme ho brať ako samostatný regex, ktorý v najhoršom prípade matchuje celé w. Vnútri každého lookaroundu je regex z *Eregex*, pre ktorý podľa lemy 1 existuje akceptačný výpočet s najviac $3rw^2$ konfiguráciami. Mimo všetkých lookaroundov je tiež regex, pre ktorý platí táto lema. V najhoršom prípade sa lookaround nachádza vnútri regexu, ktorý je iterovaný $*$ a môže byť spustený najviac w-krát. Každý lookaround teda pridá najviac $3rw^3$ konfigurácií. Spolu $3rw^2 + r\cdot 3rw^3 = O(r^2w^3) = O(rw^2(rw)^1)$.

Indukčný krok: Nech tvrdenie platí pre $h-1$, ukážeme, že platí pre h. Regex obsahuje $k \leq r$ lookaroundov s hĺbkou vnorenia najviac h. Zoberme si ľubovoľný z týchto lookaroundov. Jeho vnútorný regex obsahuje lookaroundy s hĺbkou vnorenia najviac $h-1$ a podľa indukčného predpokladu takýto regex pridáva $O(rw^2(rw)^{h-1})$ konfigurácií. Pozrime sa teraz na celý regex. Keď ignorujeme lookaroundy, akcetačný výpočet má $3rw^2$ konfigurácií. Každý lookaround pridá $O(rw^2(rw)^{h-1})$ konfigurácií a môže byť spustený najviac w-krát, pokiaľ sa nachádza v poli pôsobnosti nejakej $*$. Lookaroundov je najviac r, teda dohromady má akceptačný výpočet najviac $3rw^2 + rw\cdot O(rw^2(rw)^{h-1}) = O(rw^2(rw)^h)$ konfigurácií, čo sme chceli dokázať. $\square$

Pre triedu *LEregex* je hĺbka vnorenia lookaroundov maximálne dĺžka regexu – $f(r)=r$, teda akceptačný výpočet môže mať najviac $O(rw^2(rw)^r) = O(r^rw^{r+2})$ konfigurácií. Tento odhad je veľmi podobný odhadu 1. Druhý extrém je trieda regexov, ktorej lookaroundy majú hĺbku vnorenia konštantnú, teda $f(r)=O(1)$. V takom prípade má akceptačný výpočet najviac $O(rw^2(rw)^{O(1)})$ konfigurácií.

3 Vlastnosti lookaroundu

Na zoznámenie s novou operáciou sme zisťovali správanie sa tried Chomského hierarchie.

Veta 1. *$\mathscr{R}$ je uzavretá na negatívny a pozitívny lookaround.*

Dôkaz. Nech $L_1, L_2, L_3 \in \mathscr{R}$. Chceme ukázať, že $L_1(?\!=\!L_2)L_3$, $L_1(?\!<\!=\!L_2)L_3$, $L_1(?!\,L_2)L_3$, $L_1(?\!<\!!\,L_2)L_3$ $\in \mathscr{R}$. Pre každé L_i, $i \in \{1,2,3\}$ existuje determinický konečný automat A_i, ktorý ho akceptuje.

Máme zreťazenie L_1 a L_3. Preto prepojíme akceptačný stav A_1 s počiatočným stavom A_3. Výsledný automat sa nedeterministicky rozhoduje, či z akceptačného stavu A_1 pokračuje ďalej v A_1 alebo A_3. Teraz k nim vhodne napojíme automat A_2. Spravíme konštrukciu pre prienik regulárnych jazykov, ale mierne upravenú.

V pozitívnom lookaheade A_2 a A_3 začínajú naraz. Akonáhle A_2 akceptuje, vo výpočte bude pokračovať už len samotný A_3, kým dočíta slovo. Podobne pre pozitívny lookbehind – A_1 začne sám a nedeterministicky v nejakom kroku začne výpočet aj A_2. Akceptovať musia spolu.

Pre negatívne formy musíme navyše upraviť akceptáciu. Ak A_2 pre lookahead akceptuje, celý automat sa zasekne a zamietne. A_2 musí dočítať slovo bez dosiahnutia akceptačného stavu alebo sa zaseknúť. Pre lookbehind v každom kroku A_1 spúšťame ďalší A_2 a držíme si množinu stavov, v ktorých sa všetky nachádzajú. Úspech je, ak A_1 akceptuje a množina stavov pre automaty A_2 neobsahuje akceptačný stav. □

Veta 2. *$\mathscr{L}_{CF}$ nie je uzavretá na pozitívny lookaround.*

Vieme totiž vygenerovať jazyk $L = \{a^n b^n c^n | n \in \mathbb{N}\}$ pomocou jazykov $L_1 = \{a * b^n c^n | n \in \mathbb{N}\}$ a $L_2 = \{a^n b^n c * | n \in \mathbb{N}\}$ takto $L = (?{=}L_1)L_2 = L_1(?{<=}L_2)$.

Takýto výsledok bol očakávateľný, pretože lookaround vytvára uzavretosť na prienik a to trieda bezkontextových jazykov nie je.

Veta 3. *$\mathscr{L}_{CS}$ je uzavretá na pozitívny lookaround.*

Dôkaz. Nech jazyky $L_1, L_2, L_3 \in \mathscr{L}_{CS}$, potom ukážeme, že $L_1(?{=}L_2)L_3,\ L_1(?{<=}L_2)L_3 \in \mathscr{L}_{CS}$. Ku každému L_i existuje Turingov stroj T_i, ktorý ho akcpetuje. Zostrojíme nedeterministický Turingov stroj T pre lookahead.

T bude mať vstupnú read–only pásku. T si nedeterministicky rozdelí vstupné slovo w na w_1, w_2, w_3 tak, že $w = w_1 w_3$ a $w_3 = w_2 x$ pre nejaké x ($w_1 = x w_2$ v prípade lookbehindu). Na jednu pracovnú pásku prepíše w_1 a bude simulovať T_1. Keď akceptuje, pásku vymaže, zapíše tam w_2 a bude simulovať T_2. Ak aj ten akceptuje, pásku vymaže, zapíše tam w_3 a bude simulovať T_3. Ak T_3 akceptuje, bude akceptovať aj T. Zrejme akceptuje požadovaný jazyk. □

Teraz zistíme, čo robí lookaround s regexami (zatiaľ bez spätných referencií) a napíšeme nejaké jeho vlastnosti.

Veta 4. *Trieda jazykov nad Regex s pozitívnym a negatívnym lookaroundom je ekvivalentná $\mathscr{R}$.*

Dôkaz. Trieda jazykov nad *Regex* pokrýva triedu regulárnych jazykov a tá je na lookaround uzavretá (veta 1). Keďže pracujeme s množinou operácií, treba overiť, či nejaká ich kombinácia nie je náhodou silnejšia. Ak regex umiestníme 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.

Operácia ? veľa nespraví – lookaround tam buď bude 1x alebo nebude vôbec. $+$ je prípad $*$ s jedným lookaroundom istým. Teda chceme overiť, že $(L_1(?{=}L_2)L_3) * L_4,\ L_4(L_1(?{<=}L_2)L_3)* \in \mathscr{R}$ pre $L_1, \dots, L_4 \in \mathscr{R}$. K týmto jazykom vieme zostrojiť konečné automaty podobným spôsobom ako vo vete 1.

Ku kostre z automatov A_1, A_3, A_4 pripojíme A_2 pre lookahead tak, že v každej iterácii zároveň s A_3 spustíme aj A_2, pričom A_2 môže skončiť kedykoľvek. Budeme mať množinu stavov A_2, čo budú stavy všetkých spustených automatov. Ak nejaký stav v množine prejde do akceptačného stavu, ten už nepíšeme. Jediná požiadavka je, aby množina stavov A_2 bola po dočítaní slova prázdna.

Automat A_2 pre lookbehind ku kostre pripájame tak, že kedykoľvek počas behu sa môže spustiť 1 jeho inštancia. Podmienka je, že ak sme v počiatočnom stave A_3, množina stavov A_2 musí obsahovať aspoň 1 akceptačný stav. Ten sa potom môžeme rozhodnúť odstrániť alebo nechať, ak predpokladáme, že je to ďalšia inštacia A_2 v rovnakom stave taká, že sa vymaže neskôr. □

Nasledujúca veta hovorí, že do lookaheadu stačí dávať z každého jazyka jeho prefixovú podmnožinu[5]. Akonáhle lookahead nájde prefix, akceptuje a ku zvyšku slova sa nikdy neprepracuje. Podobne to platí pre lookehind a sufixovú podmnožinu jazyka.

Samozrejme toto platí pre lookaheady bez \$ a lookbehindy bez ^.

Veta 5. *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 platí pre lookbehind s $L_s = \{vu \mid u \in L\}$.*

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

[5] T.j. z jazyka L stačí $L_p = \{u \mid u \in L \wedge \nexists v \in L : u = vx\}$.

$\supseteq$: 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. Nech $v \neq \varepsilon$. 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 $w \in L(\alpha')$. □

Dôsledok 1. *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. □

4 Chomského hierarchia

Zaradíme zadefinované triedy do Chomského hierarchie.

Veta 6. $\mathscr{R} \subsetneq \mathscr{L}_{ERE} \subsetneq \mathscr{L}_{LERE} \subseteq \mathscr{L}_{nLERE} \subsetneq \mathscr{L}_{CS}$

Dôkaz. Všetky $\subseteq$ okrem poslednej triviálne platia. Posledná platí, lebo vieme na lineárne ohraničenom Turingovom stroji T simulovať ľubovoľný regex $\in \mathscr{L}_{nLERE}$. Opierať sa budeme o formálny model a definíciu 1. Ľavú stranu konfigurácie (regex) bude mať T v stave a pravú stranu aj s poschodovými symbolmi bude simulovať na vstupnej páske. Kroky odvodenia pre negatívny lookaround bude simulovať spustením nového Turingového stroja pre vnútorný regex – tento Turingov stroj bude mať ohraničený vstup (konkrétne podslovo) a opačnú akceptáciu.[6] Keďže regex je konečný, T bude spúšťať konečný počet nových Turingových strojov, preto vieme dopredu povedať, koľko stôp na vstupnej páske budeme potrebovať.

$\mathscr{R} \subsetneq \mathscr{L}_{ERE}$: $\alpha = (a|b)*\backslash 1, L_1 = L(\alpha) = \{ww \mid w \in \{a,b\}^*\} \notin \mathscr{R}$

$\mathscr{L}_{ERE} \subsetneq \mathscr{L}_{LERE}$: Nerovnosť dokazuje jazyk $L_2 = \{a^iba^{i+1}ba^ik \mid k = i(i+1)k'$ pre nejaké $k' > 0, i > 0\}$. $L \notin Eregex$ podľa pumpovacej lemy z [Carle and Nadendran, 2009] a tu je regex z $LEregex$ pre L_2: $\alpha = \underset{1}{(}a*\underset{1}{)}b\underset{2}{(}\backslash 1a\underset{2}{)}b(?{=}(\backslash 1)*\$)(\backslash 2)*$

$\mathscr{L}_{LERE}, \mathscr{L}_{nLERE} \subsetneq \mathscr{L}_{CS}$: Triedy $\mathscr{L}_{LERE}$ a $\mathscr{L}_{nLERE}$ sú neporovnateľné s $\mathscr{L}_{CF}$. Vyššie spomínaný jazyk $L_1 \notin L_{CS}$. Ani jedna z tried neobsahuje jazyk $L_2 = \{a^nb^n | n \in \mathbb{N}\} \in \mathscr{L}_{CF}$. □

Intuitívne by malo platiť aj $\mathscr{L}_{LERE} \subsetneq \mathscr{L}_{nLERE}$, pretože negatívny lookaround pridáva uzavretosť na komplement. Jazyk dokazujúci nerovnosť by mohol byť napríklad regex $\alpha = (?!\ (aaa*)\backslash 1(\backslash 1)*\$\ )$, kde $L(\alpha) = \{a^p \mid p \text{ je prvočíslo}\}$. Avšak na dokázanie $L(\alpha) \notin \mathscr{L}_{LERE}$ zatiaľ nemáme šikovné prostriedky a preto to zostáva netriviálnym otvoreným problémom.

5 Vlastnosti triedy $\mathscr{L}_{LERE}$

Keďže sa ukázalo, že množina $LEregex$ je silnejšia ako skúmaná $\mathscr{L}_{ERE}$, intenzívnejšie sme sa povenovali jej vlastnostiam.

Očividne operácia lookahead/lookbehind pridala uzavretosť na prienik. Nech $\alpha, \beta \in LEregex$, potom $L(\alpha) \cap L(\beta) = L(\gamma)$, kde $\gamma = (?{=}\alpha\$)\beta$ alebo $\beta(?{<=}\,\hat{}\,\alpha)$.

Naopak ohrozila uzavretosť na základnú operáciu – zreťazenie. Pri zreťazení 2 jazykov, ktorých regexy nutne musia obsahovať lookahead resp. lookbehind nastáva problém. Nemôžeme tieto regexy len tak položiť za seba. Ak sa napríklad v prvom z jazykov nachádza lookahead, počas výpočtu môže zasahovať aj do časti vstupu, ktorú matchuje druhý regex a tým zmeniť výsledok celého výpočtu. Nakoniec sa ukázalo:

Veta 7. *$\mathscr{L}_{LERE}$ je uzavretá na zreťazenie.*

Dôkaz. Nech $\alpha, \beta \in LEregex$. Jazyku $L(\alpha)L(\beta)$ bude zodpovedať regex

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

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 $\hat{}\backslash 1.*$

[6] Regexy majú tú vlastnosť, že vieme pre ne zostrojiť taký Turingov stroj, ktorý sa nezacyklí.

- s ^ – pred ^ pridáme ^\1

Slovami vyjadrené, regex γ najprv rozdelí vstupné slovo na 2 podslová w_1, w_2 patriace do príslušných jazykov $L(\alpha), L(\beta)$. Potom spustí ešte raz regex α upravený tak, že jeho lookaheady sú „skrotené", pretože ich na konci donúti matchovať w_2. Rovnako lookbehindy v β' donúti na začiatku matchovať w_1, až potom normálne pokračuje ich výpočet.

Zrejme $L(\gamma) = L(\alpha)L(\beta)$. □

Otvoreným problémom je, či platí aj uzavretosť na $*$. Podľa komplikovanosti zreťazenia sa domnievame, že uzavretosť neplatí.

Pre model $\mathscr{L}_{ERE}$ existuje pumpovacia lema. Ukázalo sa, že *LEregex* túto vlastnosť nezachováva.

Veta 8. *Nech $\alpha \in LEregex$ nad unárnou abecedou $\Sigma = \{a\}$, že neobsahuje lookahead s \$ ani 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:*

(i) $|y| \geq 1$

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

Dôkaz. Pokiaľ $\alpha \in Eregex$, tak pre α platí pumpovacia lema z [Câmpeanu et al., 2003, Lemma 1], t.j. $w = a_0 b a_1 b \ldots a_m$ pre nejaké m a $a_0 b^j a_1 b^j \ldots a_m \in L(\alpha)\ \forall j$. My pracujeme nad unárnym jazykom, teda na poradí nezáleží: $x = a_0 a_1 \ldots a_m,\ y = b^m, k = 1$ a $xy^j \in L(\alpha)\ \forall j$.

Pokiaľ α neobsahuje spätné referencie, potom podľa vety 4 generuje regulárny jazyk a pre ten existuje pumpovacia lema. Podľa nej splníme podmienky tejto vety. Nech α obsahuje aspoň 1 spätnú referenciu.

Definujme teraz konštantu N. Dostatočne dlhé slovo je také, kedy s istotou vieme povedať, že aspoň jedna Kleeneho $*$ ($+$) spravila viac ako 1 iteráciu. Nestačí nastaviť $N = |\alpha|$, lebo operácie $\{n\}, \{n,m\}$ a spätné referencie môžu slovo predĺžiť namiesto $*$. Preto nech d je súčet dĺžky regexu v k-tych zátvorkách pre všetky \k v α, plus n-krát dĺžky regexov opakovaných $\{n\}$, plus m-krát dĺžky regexov opakovaných $\{n,m\}$. Potom $N = |\alpha| + d$ je dostatočne veľká konštanta – predlžovať slovo môže len $*, +$.

Zoberme teraz tú $*$[7], ktorá iterovala aspoň 2x. Tá generuje nejake a^s.

Podľa predošlých úvah α musí obsahovať spätné referencie. Niektoré spätné referencie sa môžu odkazovať na našu vybranú $*$, na tieto spätné referencie sa môžu odkazovať na ďalšie spätné referencie, atď. V konečnom dôsledku síce $*$ generuje a^s, ale dokopy je generované $a^{ms} = a^n$. Nazvime tieto miesta, závislé od vybranej $*$, generovacie miesta.

Tiež vieme, že α musí obsahovať nejaký lookaround. Ten môže ovplyvňovať nejaké miesto generovania (prípadne aj viac). Máme 3 prípady interakcie:

1. Žiaden lookaround nezasahuje do generujúcich miest. Nech $w = a^t$, potom $x = a^{t-n},\ y = a^n,\ k = 1$ a platí $xy^j \in L(\alpha)\ \forall j$.

2. Lookahead bez \$ a/alebo lookbehind bez ^ zasahuje do generujúceho miesta alebo sa môže nachádzať vnútri iterovaniej $*$. Podľa vety 5 vieme, že v lookaheade stačí prefixová podmnožina, čo nad unárnou abecedou dáva jazyk s 1 slovom. Toto slovo obmedzuje iterovanie zdola – slovo v lookaheade určuje minimálnu dĺžku slova od daného miesta. (Podobne pre lookbehind.) w už túto podmienku spĺňa, teda máme generovacie miesta bez obmedzení a to je predošlý prípad.

3. generovacie miesto je v oblasti pôsobenia lookaheadu s \$ a/alebo lookbehindu s ^. Takýto lookaround tvorí prienik. Keďže slovo je dostatočne dlhé, musí byť iterujúca $*$ aj v takýchto lookaroundoch.

Rozoberme si prípad $*$ a 1 lookaroundu. $*$ generuje a^s, lookaround generuje a^l. Lookaround robí prienik jazykov, takže v danom úseku sú dobré len slová tvaru a^b, kde $b = j \cdot nsn(s,l)$[8].

Všeobecnejšie, nech máme 1 lookaround a $*$ s niekoľkými spätnými referenciami. Potom sčítame to, čo generujú $*$ so spätnými referenciami – spolu nejaké a^r. Opäť výsledné slovo bude prienik s lookaroundom (ten nech generuje a^l), teda a^b, kde $b = j \cdot nsn(r,l)$ – teda násobok najmenšieho spoločného násobku.

Týmto spôsobom vieme spočítať koeficient spoločného generovaného prvku – postupne sčítavame $*$ a spätné refrencie a keď sa vyskytne lookaround spravíme najmenší spoločný násobok ich generovaných prvkov. Potom nech v je výsledný koeficient, $x = a^{t-1}, y = a, k = v$ a platí $xy^{jk} \in L(\alpha)$. □

Podmienka „neobsahuje lookahead s \$ ani lookbehind s ^vnútri iterácie" je opodstatnená. Takáto kombinácia je vôbec ťažko predstaviteľná a preto sa pre

[7] $+$ je tiež v podstate $*$

[8] najmenší spoločný násobok

ňu ťažko dokazujú tvrdenia. Napríklad regex

$$((?=\underbrace{\underbrace{(?=(a^m)*\$)}_{a^{km},\ k\in\mathbb{N}}\underbrace{(a^{m+1})*a\{1,m-1\}\$ \mid a^m\$)}_{\text{vie } a* \text{ okrem } a^{m+1},\ a^{m(m+1)l},\ l\in\mathbb{N}}}_{a^{km} \text{ také, že nevie } a^{m(m+1)l},\ l\in\mathbb{N}}a^m)+$$

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 $i \in \{1,\dots,z\}$.

A teraz dôkaz, že všeobecná pumpovacia lema určite neexistuje:

Veta 9. *Jazyk všetkých platných výpočtov Turingovho stroja patrí do $\mathscr{L}_{LERE}$.*

Dôkaz. Takýto jazyk pre konkrétny Turingov stroj M obsahuje slová, ktoré sú tvorené postupnosťou konfigurácií oddelených oddeľovačom #. Každá postupnosť zodpovedá akceptačnému výpočtu na nejakom slove. Jazyk obsahuje akceptačné výpočty na všetkých slovách, ktoré sú v jazyku $L(M)$.

Turingov stroj má konečný zápis, preto je možné regex pre takýto jazyk vytvoriť. Konštrukcia regexu: $\alpha = \beta(\gamma)*\eta$, kde β predstavuje počiatočnú konfiguráciu[9] a η akceptačnú konfiguráciu. Ak q_0 (počiatočný stav M) je akceptačný stav, potom na koniec α pridáme $|(\#q_0.*\#)$.

$\gamma = \gamma_1 \mid \gamma_2 \mid \gamma_3$. Prvok γ_i generuje validnú konfiguráciu a zároveň kontroluje pomocou lookaheadu, či nasledujúca konfigurácia môže podľa δ-funkcie nasledovať. Rozpíšeme si iba jednu možnosť:

$$\gamma_1 = (\underset{k}{(}.*\underset{k}{)}xqy\underset{k+1}{(}.*\underset{k+1}{)}\#)(?=\xi\#)$$

platí pre $\forall q \in K,\ \forall x,y \in \Sigma$ a kde $\xi = \xi_1 \mid \xi_2 \mid \dots \mid \xi_n$.

- Ak $(p,z,0) \in \delta(q,y)$, potom $\xi_i = (\backslash k\, xpz \backslash k+1)$ pre nejaké i
- Ak $(p,z,1) \in \delta(q,y)$, potom $\xi_i = (\backslash k\, xzp \backslash k+1)$ pre nejaké i
- Ak $(p,z,-1) \in \delta(q,y)$, potom $\xi_i = (\backslash k\, pxz \backslash k+1)$ pre nejaké i

γ_2 a γ_3 sú podobné ako γ_1, ale matchujú krajné prípady, kedy je hlava Turingovho stroja na ľavom alebo pravom konci pásky.

Zrejme $L(\gamma)$ je požadovaný jazyk. $\square$

[9]Musí byť previazaná s nasledujúcou konfiguráciou, aby spĺňala δ-funkciu. Spraví sa to pomocou lookaheadu, podobne ako v γ_1.

6 Priestorová zložitosť

Veta 10. $\mathscr{L}_{LERE} \subseteq NSPACE(\log n)$*, kde n je veľkosť vstupu.*

Dôkaz. Nech $\alpha \in LEregex$. Zostrojíme nedeterministický Turingov stroj T akceptujúci $L(\alpha)$, ktorý bude mať vstupnú read–only pásku a 1 jednosmerne nekonečnú pracovnú pásku, na ktorej zapíše najviac $\log n$ políčok.

Výpočet Turingovho stroja bude prebiehať podľa postupnosti konfigurácií formálneho modelu. Nemôžeme nič zapísať na vstupnú pásku a máme k dispozícii menej priestoru ako je dĺžka vstupu. Využijeme to, že pre vstup dĺžky n vieme uložiť ľuboboľnú pozíciu na vstupe do adresy dĺžky $\log n$. Ukážeme, že takýchto adries potrebujeme konečný počet. Potom ich vieme písať nad seba do niekoľkých stôp pásky a mať tak zapísaných najviac $\log n$ políčok na páske.

Celý regex α bude uložený v stave aj s ukazovateľom. Budú existovať stavy pre všetky možné pozície ukazovateľa v regexe a medzi stavmi budú tzv. metaprechody podľa definície kroku výpočtu na regexe. Medzi každými dvoma stavmi prepojenými metaprechodom môže byť potrebných až niekoľko prechodov cez pomocné stavy (napríklad keď v stave vidíme otváraciu indexovateľnú zátvorku, na ktorú sa odkazujú spätné referencie, T musí zapísať aktuálnu pracovnú adresu v slove ako začiatok podslova).

Adresy budú zaznamenávať všetky ostatné informácie v konfigurácii – aktuálnu pracovnú pozíciu na vstupe (ukazovateľ v slove), pomocný ukazovateľ na spätné referencie, začiatok a koniec podslova zodpovedajúceho k-tym zátvorkám pre $\forall k$ (počet z), začiatok každého lookaheadu (l_a) a lookbehindu (l_b). K tomu bude potrebná 1 pomocná adresa – aktuálna pozícia hlavy na vstupe. Z definície α je konečnej dĺžky a pre počty daných operácií platí $2z + l_a + l_b \leq |\alpha|$. Takže dokopy potrebujeme $2 + 2z + l_a + l_b + 1 \leq |\alpha| + 3$ adries, čo je konštanta.

Kroky výpočtu I., IV., V., VI., VII.(1) nepotrebujú pomocné stavy. Ostatné kroky zapisujú, prepisujú a porovnávajú adresy. Zápis aktuálnej adresy (to je len kopírovanie znakov z inej stopy), vynulovanie záznamu a porovnávanie niekoľkých stôp vyžaduje 1 prechod cez pracovnú pásku a žiadnu prídavnú pamäť.

Preto T akceptuje $L(\alpha)$ a spĺňa pamäťové požiadavky. $\square$

Dôsledok Savitchovej vety [Savitch, 1970]:

Veta 11. *$\mathscr{L}_{LERE} \subseteq DSPACE(\log^2 n)$, kde n je veľkosť vstupu.*

Veta 12. *$\mathscr{L}_{nLERE} \subseteq DSPACE(\log^2 n)$, kde n je veľkosť vstupu.*

Dôkaz tejto vety uvedieme neskôr.

V praxi je bežné, že užívateľ zadáva nielen vstupný text, ale aj samotný regex. Preto sme sa rozhodli analyzovať jazyk, ktorý dostane na vstup oboje – slovo *regex#word*– a to je v jazyku len vtedy, ak slovo *word* vyhovuje regexu *regex*.

Veta 13. *$L(regex\#word) \in NSPACE(r \log w)$, kde $r = |regex|$, $w = |word|$ a $regex \in LEregex$.*

Dôkaz. Myšlienka dôkazu je podobná ako v dôkaze 10. Rozdiel je v tom, že regex nepoznáme dopredu. Z čoho vyplýva, že si ho nemôžeme uchovať v stave. Preto pribudnú ďalšie 2 adresy – pracovná pozícia v regexe (ukazovateľ) a aktuálna pozícia v regexe. Ďalším dôsledkom je, že síce počet adries ohraničíme zhora číslom $r+3$, ale už to viac nie je konštanta. Preto adresy nemôžeme ukladať na viacerých stopách pod sebou, ale musia byť vedľa seba oddelené oddeľovačmi. Pre rovnako pohodlné porovnávanie a zapisovanie si môžeme dovoliť pridať 1 pracovnú pásku, na ktorú si 1 z porovnávaných adries zapíšeme – tá bude mať vždy najviac $\log w$ zapísaných políčok.

Turingov stroj bude fungovať ako v dôkaze 10, ale odhad zapísanej pamäte bude $(r+3) \cdot \log w + 2 \log r$. Všetky pozície v slove vieme adresovať od oddeľovača #, preto zaberú $\log w$ pamäte. Na záver pribudla pracovná a aktuálna pozícia v regexe, z nich každá potrebuje $\log r$ políčok. Dokopy Turingov stroj zapíše $O(r \log w)$ pamäte. □

Zo Savitchovej vety vyplýva $DSPACE(r^2 \log^2 w)$. My vieme dokázať lepšiu zložitosť:

Veta 14. *$L(regex\#word) \in DSPACE(n \log^2 n)$, kde $regex \in Eregex$ a n je dĺžka vstupu.*

Dôkaz. Nech $r = |regex|$, $w = |word|$.

Myšlienka je podobná dôkazu Savitchovej vety[Ďuriš, 2003]. Turingov stroj T bude testovať, či sa dá dostať z konfigurácie C_1 do konfigurácie C_2 na i krokov:

```
1 bool TESTUJ(C1, C2, i)
2     if (C1 == C2) then return true
3     if (i > 0 ∧ C1 ⊢ C2) then return true
4     if (i <= 1) return false
5     iteruj cez všetky konfigurácie C3
6         if (TESTUJ(C1, C3, ⌊i/2⌋)
              ∧ TESTUJ(C3, C2, ⌈i/2⌉)) then
              return true
7     return false
```

Konfigurácie budú zodpovedať formálnemu modelu a ako v predošlom dôkaze budú na páske zaznamenané v podobe niekoľkých adries za sebou – ukazovateľ v regexe, ukazovateľ v slove, začiatok a koniec podslova pre k-te indexovateľné zátvorky pre $\forall k$ (z), začiatok každého lookaheadu (l_a) a lookbehindu (l_b). Globálne si budeme pamätať ešte aktuálnu pozíciu v regexe a v slove, kvôli orientácii a prípadnému kopírovaniu adries. Spolu to zaberie $\log r + (1 + 2z + l_a + l_b) \cdot \log w + \log r + \log w \leq O(r \log w)$ pamäte.

Turingov stroj T začne volaním inštancie $TESTUJ(C_0, C_a, c)$, kde C_0 je počiatočná konfigurácia, C_a je akceptačná konfigurácia a c je číslo z lemy 1. Ak akceptačný výpočet existuje, potom existuje aj taký, ktorý má nanajvýš c konfigurácií.

Procedúra $TESTUJ$ je rekurzívna. Preto bude na pracovnej páske stroja T zásobník. Pre každú inštanciu procedúry bude mať uložené konfigurácie C_1, C_2, C_3, c a informáciu, či sa vrátil z prvého alebo druhého volania (potrebný 1 bit informácie). Hodnotu c vieme zapísať do priestoru $\log c = O(\log r + \log w)$, teda záznam pre 1 inštanciu procedúry zaberie $3r \log w + \log c = O(r \log w)$ pamäte.

Keďže parameter i je vždy o polovicu menší, hĺbka rekurzie bude $\log c$. Z toho vyplýva, že zásobník bude potrebovať $O((\log r + \log w) \cdot r \cdot \log w) = O(n \log^2 n)$ pamäte. Ešte treba overiť, že úkony na riadkoch 2–4 zvládne T vykonať tiež v rámci pamäťového limitu.

Riadok 2 je porovnanie rovnosti adries – ktoré symboly už porovnal si môže značiť poschodovými symbolmi. Riadok 4 je triviálny. Riadok 3 je zložitý kvôli overeniu $C_1 \vdash C_2$. K tomu potrebuje nasledovné kontroly:

ukazovateľ – či je správne posunutý ukazovateľ (týka sa aj špeciálneho, ak je nastavený). To znamená, že buď má byť posunutý o konkrétny počet políčok alebo má byť v jeho okolí konkrétny symbol.

adresy – všetky adresy (mimo ukazovateľov) musia byť rovnaké. Okrem tých, ktorým je v tomto kroku nastavovaná nová hodnota. Tá musí byť korektne nastavená (t.j. rovnaká ako ukazovateľ v slove).

zátvorky – pre korektné skoky v regexe v krokoch II.(2) a VII.(2) musí byť medzi starou a novou pozíciou ukazovateľa korektne uzátvorkovaný výraz.

alternovateľnosť – pokiaľ sa jedná o skok v alternácii (IV.,V.), treba skontrolovať prvý alebo posledný alternovateľný regex.

indexovateľnosť – ak zátvorka nie je indexovateľná, tak sme narazili na lookaround.

Indexovateľnosť a ukazovateľ sa skontrolujú bez použitia pomocnej pamäte. Adresy využívajú porovnávanie, ale to sme už popísali, že si vieme vypomôcť poschodovými symbolmi. Alternovateľnosť využíva algoritmus na kontrolu zátvoriek – zisťuje, či je alternácia uzavretá zátvorkami (ak hej, ktorými) alebo nie. Počet zátvoriek je najviac $\frac{r}{2}$. Používame algoritmus, kde zátvorke (priradíme 1 a) hodnotu -1. Pri každom výskyte sa hodnoty sčítavajú, 0 je dobre uzátvorkovaný výraz.[10] Kontrola sa vykoná a súčet po nej už nepotrebujeme, preto ho môžeme dočasne zapísať na koniec zásobníka a vzápätí vymazať. Zapísaná pamäť tak bude $r + O(n\log^2 n) = O(n\log^2 n)$. $\square$

Dôsledok 2. *Nech $\mathscr{U}$ je trieda regexov, pre ktoré platí, že počet konfigurácií v akceptačnom výpočte pre regex α a slovo w je najviac $f(n)$, kde $n = |\alpha| + |w| + 1$. Potom $L(regex\#word) \in DSPACE(n \cdot \log n \cdot \log(f(n)))$, kde $regex \in \mathscr{U}$.*

Dôkaz. Vyplýva z dôkazu vety 14 – hĺbka vnorenia funkcie $TESTUJ$ je logaritmus z horného ohraničenia dĺžky akceptačného výpočtu. $\square$

Dôsledok 3. *Trieda regexov s hĺbkou vnorenia lookaroundov zhora ohraničenou konštantou h patrí do $DSPACE(n\log^2 n)$.*

Dôkaz. Podľa lemy 2 je počet konfigurácií v akceptačnom výpočte najviac $O((rw)^h)$. Potom hĺbka rekurzie funkcie $TESTUJ$ je

$$\log((rw)^h) = h\log r + h\log w = O(\log r + \log w)$$

A to je ten istý odhad ako v dôkaze vety 14. $\square$

[10] Ak počítame sprava doľava, hodnoty zátvoriek prenásobíme (-1), aby sme pri prvej zátvorke nemali súčet rovný -1.

Tu už nasleduje sľubovaný dôkaz vety 12. Budeme čerpať z dôkazu predošlej vety 14.

Dôkaz. Nech $\alpha \in nLEregex$ a $r = |\alpha|$. Zostrojíme Turingov stroj T, ktorý bude akceptovať $L(\alpha)$ a na pracovných páskach zapíše najviac $O(\log^2 n)$ políčok.

Pokiaľ α neobsahuje negatívny lookaround, tvrdenie triviálne vyplýva z vety 11. Nech teda obsahuje aspoň jeden negatívny lookaround a nech k je najvyšší počet negatívnych lookaroundov vnorených do seba (nezáleží na tom, či sú to lookaheady alebo lookbehindy).

T bude skonštruovaný ako Turingov stroj vo vete 14, pričom musíme dodefinovať správanie v prípade negatívneho lookaroundu. V definícii 1 v bodoch XV. a XIV. je napísané, že ak splníme istú podmieku, negatívny lookaround možno preskočiť a pokračovať ďalej vo výpočte. Podmienka začína „neexistuje výpočet", čo naznačuje, že musíme vyskúšať všetky možnosti postupnosti konfigurácií – teda mať deterministický algoritmus, ktorý akceptuje práve vtedy, keď akceptačný výpočet existuje.

Vhodným algoritmom je procedúra $TESTUJ$. Zakaždým, keď T bude overovať podmienku $C_1 \vdash^? C_2$ a jedná sa o prechod XV. alebo XIV. z definície 1, stane sa nasledovné. T spustí na novej páske novú procedúru $TESTUJ(C_0', C_a', c')$, kde C_0' je počiatočná a C_a' akceptačná konfigurácia z definície daného kroku a $c' \leq c$ je hodnota z lemy 1 pre regex vo vnútri tohto negatívneho lookaroundu. Túto procedúru treba spustiť niekoľkokrát po sebe – pre každé p, čo prichádza do úvahy. Pokiaľ niektorý z behov procedúry $TESTUJ$ skončí úspešne, znamená to, že existuje akceptačný výpočet tam, kde nechceme, aby existoval – podmienka negatívneho lookaroundu neplatí a výsledok je $C_1 \nvdash C_2$. Ak všetky behy skončia s výsledkom *false*, výsledok je $C_1 \vdash C_2$.

Vynechali sme detail „spustenie pre každé p, čo prichádza do úvahy". Tu je treba zadať hranice podslova, na ktorom procedúra pracuje, a neprekročiť ich. Jednoducho zakomponujeme do procedúry kontrolu, či sú všetky adresy a ukazovateľ pre toto spustenie v povolenom intervale. Tieto hodnoty budú globálne a zapíšu sa pri prvom volaní na začiatok zásobníka. Pre hlavný beh procedúry to budú hodnoty 0 a $n+2$ (t.j. interval $\langle 0, n+2))$.

Popísali sme správanie T pre prípady, keď operácie negatívneho lookaroundu nie sú vnorené. Zoberme si prípad, keď ich α obsahuje niekoľko vnorených. T má na 1. páske rozpracovanú hlavnú vetvu $TESTUJ$, teraz pracuje na 2. páske na negatívnom lookarounde a narazí na ďalší. Uvedomme si, že pre T je to rovnaká situácia, akokeby pracoval stále na 1. páske. Zopakuje postup popísaný vyššie – niekoľkokrát spustí $TESTUJ$ na 3. páske pre vhodné hranice slov a ak výsledkom každého behu bude *false*, vráti sa na 2. pásku. Nech regex α má k vnorených negatívnych lookaroundov, potom T bude potrebovať $k+1$ pracovných pások.

Pre spočítanie zapísaných políčok na páskach si najprv popíšme konfigurácie. Regex poznáme dopredu. To znamená, že pre každú polohu ukazovateľa v regexe vieme pridať 1 znak do pracovnej abecedy, ktorý toto nastavenie predstavuje (t.j. $r+1$ špeciálnych symbolov). Zároveň pre výpočty na negatívnych lookaroundoch nám stačí ich vnútorný regex s ukazovateľom. Takýchto podslov je konečne veľa, preto aj pre tie vieme mať nové symboly v abecede.

Adresy, ktoré v konfiguráciách potrebujeme sú: pracovná pozícia v slove, začiatok a koniec podslova pre k-te zátvorky pre $\forall k$ (z), začiatok každého lookaheadu (l_a) a lookbehindu (l_b). Spolu $1+2z+l_a+l_b \leq 2r+1$ adries a to je konštanta. Konštantný počet adries vieme umiestniť nad seba do konštantného počtu stôp na páske (ako v dôkaze 10) a takto nimi zaberieme $\log n$ políčok (symbol pre stav regexu bude v samostatnej najvrchnejšej stope).

Jedna inštancia procedúry $TESTUJ$ potrebuje 3 konfigurácie a konštantný počet políčok. Hĺbka vnorenia rekurzie je na každej páske najviac $\log c = O(\log r + \log w) = O(\log n)$. Každé prvé volanie potrebuje navyše aktuálnu pozíciu hlavy na vstupe a hranice podslova, na ktorom pracuje. Dokopy bude na každej páske zapísaných najviac $\log n \cdot O(\log n) + 3\log n = O(\log^2 n)$ políčok. $\square$

Poďakovanie

Ďakujem školiteľovi za cenné rady a pripomienky.

Literatúra

[Carle and Nadendran, 2009] Carle, B. and Nadendran, P. (2009). On extended regular expressions. In *Language and Automata Theory and Applications*, volume 3, pages 279–289. Springer.

[Câmpeanu et al., 2003] Câmpeanu, C., Salomaa, K., and Yu, S. (2003). A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018.

[Ehrenfeucht and Zeiger, 1975] Ehrenfeucht, A. and Zeiger, P. (1975). Complexity measures for regular expressions. *Computer Science Technical Reports*, 64.

[Ellul et al., 2013] Ellul, K., Krawetz, B., Shallit, J., and wei Wang, M. (2013). Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics u (v) w, x–y*.

[Python documentation, 2012] Python documentation (2012). *Regular expression operations*. Python Software Foundation. `http://docs.python.org/2/library/re.html`.

[Savitch, 1970] Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192.

[Tóthová, 2013] Tóthová, T. (2013). Moderné regulárne výrazy. Bachelor's thesis, FMFI UK Bratislava. `https://github.com/Tatianka/bak`.

[Ďuriš, 2003] Ďuriš, P. (2003). Výpočtová zložitosť (materiály k prednáške). `http://www.dcs.fmph.uniba.sk/zlozitost/data/zlozitost_duris.pdf`.