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 a lookbehind. V článku uvedieme zaradenie modelu zodpovedajúcej triedy jazykov do Chomského hierarchie, vlasnosti tejto triedy a výsledky z oblasti priestorovej zložitosti.

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

1 Úvod

Regulárne výrazy vznikli v 60tych rokoch v teórii jazykov ako ďalší model na vyjadrenie regulárnych jazykov. Z takéhoto 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 [Foundation, 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 $\backslash$, 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.

* `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 určnené 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 posledné podslovo zo vstupu, ktoré matchovali k-te zátvorky.

Lookahead

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

Keď v regexe prídeme na pozíciu lookaheadu, zoberieme regex vo vnútri. V slove sa snažíme matchovať ľubovoľný prefix zostávajúcej časti slova. Ak sa to podarí, pokračujeme v regexe ďalej a v slove od pozície, kde lookahead začínal (tzn. akokeby v regexe nikdy nebol).

Má aj negatívnu verziu – negatívny lookahead (?! ...). Vykonáva sa rovnako ako lookahead, ale má otočnú akceptáciu. Teda ak neexistuje prefix, ktorý by vedel matchovať, 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.

Aj lookbehind má negatívnu verziu – negatívny lookbehind (?<! ...) – a pracuje analogicky 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 akonáhle na ne narazíme. Ostatné zoradíme v tabuľke:

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

1.5 Triedy a množiny

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

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

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 $\mathscr{L}_{LERE}$ 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 ho vyjadriť ako 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 tej 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.

Definujeme **krok výpočtu** ako reláciu $\vdash$ na konfiguráciách ... TODO!!!

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]

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

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

VIII. špeciálny ukazovateľ pre spätné referencie – zjavenie
$(r_1 \dots \lceil \backslash k \dots r_n, w_1 \dots \overset{k}{w}_a \dots \overset{k'}{w}_b \dots \lceil w_j \dots w_m)$

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

[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.

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 $a \leq c < b$ 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 pre spätné referencie – 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:

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

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

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

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:

Veta 1. *Nech $\alpha \in \mathscr{L}_{ERE}$ a $w \in L(\alpha)$. Potom existuje akceptačný výpočet, ktorý má najviac $5 \cdot |\alpha|^2 \cdot |w|$ 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 $|\alpha| + |w|$.

V krokoch VIII. a X. sa žiaden ukazovateľ nepohne. Oba kroky sa vyskytujú 1x ku každej spätnej referencii, ktorých je najviac $|\alpha|$, takže tieto konfigurácie sa objavia najviac $2 \cdot |\alpha|$-krát.

Výpočet môže predĺžiť skákanie ukazovateľa dozadu. V krokoch VI.(2),VII.(2) sa na 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. Vieme, že regex opakovaný operáciou $*$ je dlhý nanajvýš $|\alpha|$ znakov a podľa úvahy ho treba opakovať najviac $|w|$-krát.

Teda dokopy spravíme najviac $|\alpha| + |w| + 2 \cdot |\alpha| + |\alpha| \cdot |w| \leq 5 \cdot |\alpha| \cdot |w|$ krokov. □

Veta 2. *Nech $\alpha \in \mathscr{L}_{LERE}$, $w \in L(\alpha)$, $r = |\alpha|$ a $v = |w|$. Potom existuje akceptačný výpočet, ktorý má nanajvýš $O(r^2 w^3)$ konfigurácií.*

Dôkaz. Spočítajme, koľko je všetkých konfigurácií pre regex α a slovo w.

Ukazovateľ v regexe môže mať $(r+1)$ rôznych pozícií. Ukazovateľ v slove môže mať $(v+1)$ rôznych pozícií. V slove môžu byť niekedy 2 ukazovatele. Spolu máme $(r+1)(w+1)+(r+1)(w+1)^2 = (r+1)(w+1)(w+2)$ možností.

V konfiguráciách sú v poschodových symboloch zapísané informácie potrebné k výpočtu. Pre každé spätné referencie sú to 2 zápisy a pre každý lookaround 1. Dokopy je to najviac $2r$ zápisov. Každá informácia buď v regexe ešte nie je zapísaná alebo má w možností, kde zapísaná môže byť. Pre informácie máme dokopy $2r(w+1)$ možností.

Všetkých možných konfigurácií je dohromady

$$(r+1)(w+1)(w+2)\cdot 2r(w+1) = O(r^2w^3) \quad (1)$$

Späť k akceptačnému výpočtu. Nech sú všetky akceptačné výpočty dlhšie ako (1). Potom tento výpočet musí nutne obsahovať 2 rovnaké konfigurácie. Keď vymažeme úsek medzi rovnakými konfiguráciami a napojíme postupnosť, dostaneme opäť validný akceptačný výpočet. Postup opakujeme, pokým výpočet obsahuje nejaké 2 rovnaké konfigurácie. Keď skončíme, sú všetky konfigurácie vo výpočte rôzne a preto má najviac (1) konfigurácií. $\square$

3 Vlastnosti lookaroundu

Na začiatok sme zisťovali, čo robia samotné operácie lookaroundu.

Veta 3. *$\mathscr{R}$ 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 deterministický konečný automat A_i, ktorý ho akceptuje.

Konečné automaty vieme vhodne pospájať dohromady. Spravíme konštrukciu pre prienik regulárnych jazykov, ale mierne upravenú tak, že akonáhle automat pre L_2 v pozitívnom lookaheade 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. 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.

Odhliadnuc teraz od lookaroundu, 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. $\square$

Veta 4. *$\mathscr{L}_{CF}$ nie je uzavretá na pozitívny lookaround.*

Vieme totiž vygenerovať jazyk $a^nb^nc^n$ prienikom jazykov $a*b^nc^n$ a a^nb^nc*.

Veta 5. *$\mathscr{L}_{CS}$ je uzavretá na pozitívny lookaround.*

Veta 6. *Trieda jazykov nad Regex s pozitívnym a negatívnym lookaroundom je ekvivalentná $\mathscr{R}$.*

4 Chomského hierarchia

Veta 7. $\mathscr{R} \subsetneq \mathscr{L}_{ERE} \subsetneq \mathscr{L}_{LERE} \subseteq \mathscr{L}_{nLERE} \subsetneq \mathscr{L}_{CS}$

Dôkaz. Všetky $\subseteq$ triviálne platia.

$\mathscr{L}_{ERE} \subsetneq \mathscr{L}_{LERE}$: Nerovnosť dokazuje jazyk $L = \{a^iba^{i+1}ba^ik \mid k = i(i+1)k' \text{ pre nejaké } k' > 0, i > 0\}$. $L \notin Eregex$ podľa pumpovacej lemy z [Carle and Nadendran, 2009] a tu je regex z $LEregex$ pre L: $\alpha = (_1 a*)_1\, b\, (_2 \backslash 1a)_2\, b(?{=}(\backslash 1)*\$)(\backslash 2)*$

$\mathscr{L}_{LERE}, \mathscr{L}_{nLERE} \subsetneq \mathscr{L}_{CS}$: Triedy $\mathscr{L}_{LERE}$ a $\mathscr{L}_{nLERE}$ sú neporovnateľné s $\mathscr{L}_{CF}$. K jazyku $L_1 = \{ww \mid w \in \{a,b\}^*\} \notin \mathscr{L}_{CF}$ existuje regex z $LEregex$: $\alpha = ((a|b)*)\backslash 1$. Ani jedna z tried neobsahuje jazyk $L_2 = \{a^nb^n | n \in \mathbb{N}\} \in \mathscr{L}_{CF}$. $\square$

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 $\alpha = (?!\ (aa+)(\backslash 1)+\$\ )$, pričom $L(\alpha) = \{a^p \mid p \text{ je prvočíslo}\}$.

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

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 $\beta(?{<=}\,\hat{}\,\alpha)$.

Napak 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 8. *$\mathscr{L}_{LERE}$ 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 $\hat{}\backslash 1.*$
- s ˆ – pred ˆ pridáme $\hat{}\backslash 1$

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 $L(\gamma) = L(\alpha)L(\beta)$. □

Veta 9. *Nech $\alpha \in LEregex$ nad unárnou abecedou $\Sigma = \{a\}$, že neobsahuje lookahead s \$ ani lookbehind s ˆ 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. TODO!!! □

Veta 10. *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[4] a η akceptačnú konfiguráciu. Ak q_0 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 $\forall q \in K,\ \forall y \in \Sigma$ a kde $\xi = \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$.

- Ak $(p, z, 0) \in \delta(q, y)$, potom $\xi_i = (\backslash k\, xpz \backslash k+1)$ 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. □

6 Priestorová zložitosť

Veta 11. *$\mathscr{L}_{LERE} \subseteq NSPACE(\log n)$, 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

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

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 narazí na otváraciu indexovateľnú zátvorku, na ktorú sa odkazujú spätné referencie, 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 $2z + l_a + l_b \leq |\alpha|$. Spolu máme $2 + 2z + l_a + l_b + 1 \leq |\alpha| + 3$, čo je konštanta.

Kroky výpočtu I., IV., V., VI., VII.(1) nepotrebujú pomocné stavy. Ostatné kroky zapisujú, prepisujú a porovnávajú adresy. Zápis aktuálnej adresy (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. □

Dôsledok Savitchovej vety:

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

Veta 13. *$\mathscr{L}_{nLERE} \subseteq DSPACE(\log^2 n)$, kde n je veľkosť vstupu.*

Dôkaz. TODO!!! □

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 akceptuje slovo len vtedy, ak slovo *word* vyhovuje regexu *regex*.

Veta 14. *$L(regex\#word) \in NSPACE(r \log w)$, kde $r = |regex|$, $w = |word|$ a $regex \in LEregex$.*

Dôkaz. Myšlienka dôkazu je podobná ako v dôkaze 11. 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 11, 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. □

Veta 15. *$L(regex\#word) \in DSPACE(n \log^2 n)$, kde $regex \in LEregex$ a n je dĺžka vstupu.*

Dôkaz. TODO!!!zdroj Savitchovej vety Nech $r = |regex|$, $w = |word|$.

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

```
bool TESTUJ(C1, C2, i)
    if (C1 == C2) then return true
    if (i > 0 ∧ C1 ⊢ C2) then return true
    if (i <= 1) return false
    iteruj cez vsetky konfiguracie C3
        if (TESTUJ(C1, C3, ⌊i/2⌋)
                ∧ TESTUJ(C3, C2, ⌈i/2⌉)) then
            return true
    return false
```

Konfigurácie budú zodpovedať formálnemu modelu a ako v predošlom dôkaze budú na páske zaznamenané ako niekoľko adries – aktuálna pozícia a pracovná pozícia v regexe, aktuálna a pracovná pozícia v slove, začiatok a koniec podslova pre k-te indexovateľné zátvorky pre $\forall k$ (z), začiatok každého lookaheadu (l_a) a lookbehindu (l_b) – spolu $2 \log r + (2z + l_a + l_b) \cdot \log w \leq O(n \log n)$ pamäte.

Turingov stroj T

□

Pod’akovanie

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*.

[Foundation, 2012] Foundation, P. S. (2012). *Regular expressions operations*.

[Tóthová, 2013] Tóthová, T. (2013). Moderné regulárne výrazy. Bachelor’s thesis, FMFI UK Bratislava.