Sem vložte zadání Vaší práce.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Zpracování POSIX regulárních výrazů

Oleksandr Zaporozhchenko

Katedra teoretické informatiky
Vedoucí práce: Ing. Ondřej Guth, Ph.D.

29. května 2020

Poděkování

Rád bych zde poděkoval Ing. Ondřeji Guthovi, Ph.D. za vedení mé bakalářské práce, cenné rady a čas, který mi věnoval při zpracování daného tématu. Dále děkuji svým rodičům a bratrovi, kteří mě podporovali a pomáhali při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 29. května 2020 . . . .

Odkaz na tuto práci

Zaporozhchenko, Oleksandr. *Zpracování POSIX regulárních výrazů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Rozšířením tradiční syntaxe regulárních výrazů o zpětné reference vznikne mocný prostředek, kterým lze popsat jazyky silnější než regulární. Tato bakalářská práce se zabývá problémem zpracování regulárních výrazů se zpětnými referencemi. V práci je implementován v jazyce C++ algoritmus založený na konstrukci vícepáskového Turingova stroje a algoritmus od Schmida, který řeší verzi tohoto problému parametrizovanou stupněm aktivních proměnných (*active variable degree*). V závěru je program testován a porovnán s již existujícími nástroji pro práci s regulárními výrazy na vytvořených sadách testů. Mimo jiné je v práci předložen alternativní důkaz věty o NP-úplnosti zkoumaného problému.

Klíčová slova regulární výraz, zpětná reference, Turingův stroj, referenční slovo, problém zpracování, důkaz NP-úplnosti, parametrizovaná složitost, aktivní proměnná, implementace, C++, vytvoření testovacích sad, grep, Perl

Abstract

This bachelor's thesis deals with the matching problem of regular expressions with backreferences (regex, for short), which is a feature available in most modern matching engines. It allows the user to specify even non-regular languages. The chosen algorithm based on the construction of a multi-tape Turing machine and Schmid's algorithm for this problem parametrized by the active variable degree were implemented in C++. All the implementations are tested on created test sets and compared with already existing applications. This paper gives, among other things, an alternative proof of the NP-completeness of the matching problem of regex.

Keywords regex, backreference, Turing machine, ref-word, regex matching problem, NP-completeness proof, parametrized complexity, active variable, implementation, C++, test set creation, grep, Perl

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je těžké najít textový editor, který by nepodporoval regulární výrazy. Mezi nejčastější využití tohoto formalismu pro popis formálních jazyků patří vyhledávání a nahrazování v textu. V praxi používaná syntaxe regulárních výrazu se značně liší od formálního algebraického zápisu. Jeden z nejznámějších standardů IEEE POSIX [1] definuje konstrukce, které umožňují nejen zapsat výraz jednodušším způsobem, ale i popsat jazyky silnější než regulární. Tato práce je zaměřena na jeden z takových prostředků syntaxe, který se nazývá *zpětná reference* (anglicky *backreference*).

Cílem teoretické části práce je popsat tzv. problém zpracování regulárních výrazů se zpětnými referencemi a formalizovat známé algoritmy pro řešení tohoto problému. Mezi cíle praktické části patří implementace vybraného algoritmu, vytvoření testovacích sad a porovnání výpočetních časů implementace s existujícími nástroji pro práci s regulárními výrazy. Jedním z nedostatků většiny dostupných implementací je fakt, že některé regulární výrazy se zpětnými referencemi nejsou podporovány, resp. je není možné zapsat v požadované notaci. Přínosem praktické části je také nový přístup k dokazování NP-úplnosti daného problému založený na pojmu referenčního slova. Na nesprávnost existujícího důkazu [2, s. 289] poukazoval ve své práci Schmid [3, s. 83].

Práce má následující strukturu. V první kapitole jsou uvedeny různé způsoby definování regulárních výrazu se zpětnými referencemi (regexy a semiregexy) a jejich výpočetní síla. Hodnota regexu je v práci definována pomocí tzv. *referenčních slov* (anglicky *ref-words*) [4]. V kapitole je také prozkoumán vztah regexů a referenčního slova, které daný regex generuje. Druhá kapitola se zabývá problémem zpracování regexů, výpočetními modely, které rozpoznávají Regex jazyky, a algoritmy pro převod regexu na vícepáskový Turingův stroj a memory automat. Součásti této kapitoly je také návrh bezkontextové gramatiky generující Regex jazyky a abstraktního syntaktického stromu reprezentujícího libovolný regex. Třetí kapitola je věnována problému zpracování z hlediska teorie parametrizované složitosti. Pro regex je v této kapitole zaveden parametr složitosti *avd* (anglicky *active variable degree* [5]) a je také popsána efektivní metoda konstrukce memory automatu, v němž počet paměťových prvků záleží pouze na parametru *avd* vstupního regexu. Dále jsou implementovány dva algoritmy založené na konstrukci memory automatu a jeden na konstrukci TS. V poslední kapitole jsou tyto algoritmy otestovány na vytvořených datových sadách. Na konci práce jsou změřeny a porovnány výpočetní časy implementací autora s nástrojem *grep* a regulárními výrazy v jazyce *Perl*.

Kapitola 1

Regulární výrazy se zpětnými referencemi

V práci jsou používány základní pojmy teorie formálních jazyků a automatů, jejichž znalost se u čtenáře předpokládá. Standardní definice z teorie deterministické syntaktické analýzy jsou také vzhledem k povaze textu vynechány. V případě nejasností autor práce odkazuje čtenáře na publikace [6], [7] a [8].

1.1 Standardy regulárních výrazů

Regulární výrazy jsou v současné době používány ve většině programovacích jazyků a textových editorů. Mezi nejpoužívanější standardy definující regulární výrazy patří IEEE POSIX a PCRE (Perl Compatible Regular Expressions). V těchto standardech jsou definovány mimo jiné zpětné odkazy a další konstrukce, kterými lze popsat jazyky silnější než regulární. V základních regulárních výrazech POSIX (BRE) se objevuje výraz tvaru $\backslash i$. Tato konstrukce označuje zpětnou referenci na podvýraz, který se nachází v i-té závorce zleva (závorky se číslují podle pozice otevírací závorky). Zatímco počet číslovaných skupin zachycení v regulárním výrazu podle standardu POSIX BRE je omezen devíti [1, s. 233], ve výrazech podle PCRE může existovat až 100 číslovaných skupin zachycení (i když se třeba způsob zápisu liší). Standard PCRE zavádí také pojmenované zpětné odkazy [9], které mají stejný sémantický význam jako číslované odkazy.

1.2 Syntaxe regulárních výrazů se zpětnými referencemi

Regulární výrazy se zpětnými referencemi je možné definovat několika způsoby. Jeden rozdíl spočívá v tom, jak je označen podřetězec, na nějž se ve výrazu odkazuje. Další způsob definování, v němž je podřetězec vázán na závorku, je popsán v [10]. Jelikož závorky jsou očíslovány vzestupně zleva doprava, není možné v těchto výrazech zopakovat „definici“ závorky.

Definice 1.1 (Freydenberger–Schmid)**.** Buď X konečná množina proměnných, buď Σ abeceda ($X \cap \Sigma = \varnothing$). Potom *regulární výraz se zpětnými referencemi* (nad Σ a X), dále zkracováno na *regex*, je každý řetězec získaný aplikací následujících pravidel v konečně mnoha krocích. Množina všech řetězců definovaných tímto způsobem je značena $RV_{\Sigma,X}$. Množina proměnných, které se vyskytují ve výrazu α, se značí $var(\alpha)$.

1. $a \in RV_{\Sigma,X}$ a $var(a) = \varnothing$ pro všechna $a \in \Sigma \cup \{\varnothing, \varepsilon\}$.
2. Jestliže α, $\beta \in RV_{\Sigma,X}$, pak také $(\alpha) \in RV_{\Sigma,X}$, $\alpha^* \in RV_{\Sigma,X}$ ($var(\alpha^*) = var(\alpha)$), $\alpha + \beta \in RV_{\Sigma,X}$ a $\alpha \cdot \beta \in RV_{\Sigma,X}$ ($var(\alpha + \beta) = var(\alpha \cdot \beta) = var(\alpha) \cup var(\beta)$).
3. (*definice proměnné*) $x\{\alpha\} \in RV_{\Sigma,X}$ a $var(x\{\alpha\}) = var(\alpha) \cup \{x\}$ pro $x \in X \setminus var(\alpha)$ a $\alpha \in RV_{\Sigma,X}$.
4. (*zpětná reference*) $x \in RV_{\Sigma,X}$ a $var(x) = \{x\}$ pro všechna $x \in X$. [5]

Příklad 1.1. Buď $X = \{x\}$ množina proměnných, buď $\Sigma = \{a, b\}$ abeceda. Výraz $(xx\{a\})^*$ je validním regexem z množiny $RV_{\Sigma,X}$. Naopak ani řetězec $x\{x\{a^*\}\}$, ani $x\{xa\}\ x$ nepatří do $RV_{\Sigma,X}$, protože oba výrazy lze zapsat jako $x\{\varphi\}$ a $var(\varphi) = \{x\}$, což neodpovídá bodu 3 definice regexu.

Definice 1.2 (Câmpeanu–Salomaa–Yu semiregex)**.** Buď Σ abeceda. Potom *semiregulární výraz se zpětnými referencemi* (nad Σ), dále zkracováno na *semiregex*, je každý řetězec získaný aplikací následujících pravidel v konečně mnoha krocích. Množina všech řetězců definovaných tímto způsobem je značena RV_Σ^{CSY}.

- $\varnothing$, ε, $a \in RV_\Sigma^{CSY}$ pro všechna $a \in \Sigma$.
- Jestliže α, $\beta \in RV_\Sigma^{CSY}$, pak také $(\alpha) \in RV_\Sigma^{CSY}$, $(\alpha + \beta) \in RV_\Sigma^{CSY}$, $(\alpha \cdot \beta) \in RV_\Sigma^{CSY}$ a $(\alpha)^* \in RV_\Sigma^{CSY}$.
- (*zpětná reference*) $\backslash n \in RV_\Sigma^{CSY}$, kde $n \in \mathbb{N}$ a hodnota $\backslash n$ odpovídá hodnotě dílčího výrazu v n-té závorce. [10]

Příklad 1.2. Řetězec $({}_1({}_2a^*)_2 + \varepsilon)_1\backslash 1({}_1({}_3b^*)_3 + \varepsilon)_1\backslash 1$ není validním semiregexem, protože třetí otevírací závorka zleva může být označena pouze číslem 3. Naopak množina $RV_{\Sigma,X}$ obsahuje například řetězec $\alpha = (x\{a^*\} + \varepsilon)\ x\ (x\{b^*\} + \varepsilon)\ x$. Regex α popisuje formální jazyk $L = \{a^{3n} \mid n \in \mathbb{N}_0\} \cup \{\varepsilon\} \cup \{a^{2n}b^{2k} \mid n, k\ \in \mathbb{N}_0\} \cup \{b^{2n} \mid n \in \mathbb{N}_0\}$. Řetězcem z RV_Σ^{CSY}, který reprezentuje jazyk L, je například $(({}_2a^*)_2\backslash 2({}_3b^*)_3\backslash 3) + \varepsilon + (({}_5a^*)_5\backslash 5\backslash 5) + (({}_7b^*)_7\backslash 7)$.

Ve většině případů regexy umožňují popsat nějaký formální jazyk jednodušším způsobem. Jelikož existuje formální jazyk L, který lze popsat regexem z $RV_{\Sigma,X}$, ale žádný semiregex z RV_{Σ}^{CSY} nereprezentuje L, a každý semiregex lze triviálně převést na ekvivalentní regex [11, s. 38–40], lze považovat regex za silnější formalismus (toto je hlavní důvod, proč se v této práci používají regexy).

Podobně jako pro klasické regulární výrazy [12] lze pro regexy definovat hvězdnou výšku (anglicky *star-height*). Hodnota výšky určuje maximální počet do sebe vnořených iterací ve výrazu.

Definice 1.3. Buď Σ abeceda, buď X množina proměnných. Hvězdná výška $\lambda(\varphi)$ pro libovolný regex φ z $RV_{\Sigma,X}$ je definována takto:

- $\lambda(a) = 0$ pro $a \in \Sigma \cup \{\varepsilon, \varnothing\}$,
- $\lambda(x) = 0$ pro $x \in X$,
- Pro α, $\beta \in RV_{\Sigma,X}$ $\lambda(\alpha\beta) = \lambda(\alpha + \beta) = max\{\lambda(\alpha), \lambda(\beta)\}$ a $\lambda(\alpha^*) = \lambda(\alpha) + 1$,
- Pro $\alpha \in RV_{\Sigma,X}$, $x \in X \setminus var(\alpha)$ $\lambda(x\{\alpha\}) = \lambda(\alpha)$.

Příklad 1.3. $\lambda(\ (x\ x\{a^*\})^*) = \lambda(x\ x\{a^*\}) + 1 = max\{\lambda(x), \lambda(x\{a^*\})\} + 1 = max\{0, \lambda(a^*)\} + 1 = max\{0, \lambda(a) + 1\} + 1 = max\{0, 1\} + 1 = 2$.

1.3 Hodnota regexu a REGEX jazyky

V této sekci je zkoumána vyjadřovací síla regexů a definována třída Regex jazyků. Nejprve se zavede pojem *hodnoty regexu*. Podobně jako u regulárních výrazů, hodnota je definována jako formální jazyk, který daný výraz popisuje. Při definování bude vycházeno z [4] a [13].

Definice 1.4. Buď Σ abeceda, buď X konečná množina proměnných a buď $\Gamma = \{[_x,]_x \mid x \in X\}$. Konečná posloupnost znaků z $\Sigma \cup X \cup \Gamma$ se nazývá *referenční slovo*. Referenční jazyk je konečná množina referenčních slov. [4]

Je možné podívat se na regexy jako na generátory referenčních slov, což znamená, že pro každý regex lze definovat referenční jazyk.

Definice 1.5. Buď Σ abeceda, buď X konečná množina proměnných a buď $\Gamma = \{[_x,]_x \mid x \in X\}$. Lze definovat *referenční jazyk* pro libovolný regex α následovně:

1. $R(\varnothing) = \varnothing$,
2. $R(a) = \{a\}$ pro $a \in \Sigma \cup \{\varepsilon\}$,
3. $R(x) = \{x\}$ pro $x \in X$,

4. $R(x\{\alpha\}) = \{[_x\} \cdot R(\alpha) \cdot \{]_x\}$ pro $x \in X$ a $\alpha \in RV_{\Sigma,X}$,

5. $R(\alpha \cdot \beta) = R(\alpha) \cdot R(\beta)$, kde $\alpha, \beta \in RV_{\Sigma,X}$,

6. $R(\alpha + \beta) = R(\alpha) \cup R(\beta)$, kde $\alpha, \beta \in RV_{\Sigma,X}$,

7. $R(\alpha^*) = R(\alpha)^*$, kde $\alpha \in RV_{\Sigma,X}$. [13]

Pozorování 1.1. Nechť $\Sigma' = \Sigma \cup X \cup \Gamma$. Provede-li se substituce všech definic $x\{\beta\}$ v nějakém regexu $\alpha \in RV_{\Sigma,X}$ na výrazy $[_x\beta]_x$, hodnota výsledného výrazu α' se rovná referenčnímu jazyku $R(\alpha)$ (α' je regulárním výrazem nad Σ' dle definice). [13, s. 4]

Příklad 1.4. Regex $\alpha = (b^*c)^* \, x\{a^*c\} \, x$ generuje například referenční slovo $\left(b^9a\right)^3 [_x a^2 c]_x \, x$.

Referenční slovo $r = [_x aa[_y c]_x aaa]_y$ nelze generovat žádným regexem. Pokud by nějaký regex β generoval r, potom podřetězec $[_x aa[_y c]_x$ byl vygenerován pomocí pravidla 4 z definice 1.5. To znamená, že podřetězec $aa[_y c$ byl vygenerován z nějakého regexu α_0, což není možné. Jediné pravidlo, pomocí něhož lze získat znak $[_y$, je pravidlo 4. Pak ale $aa[_y c$ obsahuje také znak $]_y$, což není pravda.

Z předchozího příkladu vyplývá, že ne každé referenční slovo lze vygenerovat pomocí regexu. Pod pojmem „referenční slovo“ je dále myšlen pouze řetězec z podmnožiny validních slov, který je definován následovně.

Definice 1.6. Buď Σ abeceda, buď X konečná množina proměnných a buď $\Gamma = \{[_x,]_x \mid x \in X\}$. Referenční slovo r nad $\Sigma \cup X \cup \Gamma$ je *validní*, pokud existuje $\alpha \in RV_{\Sigma,X}$ tak, že platí $r \in R(\alpha)$.

„Výskyt proměnné $x \in X$ v referenčním slově funguje jako ukazatel na nejbližší podřetězec tvaru $[_x\alpha]_x$, který se nachází vlevo“. [13] V další definici je ukázáno, jak lze formálně převést referenční slovo nad $\Sigma \cup X \cup \Gamma$ na slovo nad Σ (v původním textu se této operaci říká *dereference*). Zjednodušeně lze níže uvedené kroky popsat následovně:

Každá proměnná x se substituuje za příslušný podřetězec, na nějž daná proměnná odkazuje (nebo za ε, pokud se nalevo nevyskytuje podřetězec $[_x \dots]_x$), a všechny znaky z $\Gamma = \{[_x,]_x \mid x \in X\}$ z řetězce budou odstraněny.

Definice 1.7. Buď r referenční slovo nad $\Sigma \cup X \cup \Gamma$, buď $var(x)$ počet znaku x v r a buď $X' = \bigcup\limits_{x \in X} \bigcup\limits_{i=1}^{i \le var(x)} \{x_i\}$. Výraz r' je r, v němž jsou proměnné očíslované ($r' \in (\Sigma \cup X' \cup \Gamma)^*$). Pro r' jsou definovány homomorfismy $deref_0^{r'} : (\Sigma \cup X' \cup \Gamma)^* \to (\Sigma \cup \Gamma)^*$ a $deref_1^{r'} : (\Sigma \cup \Gamma)^* \to \Sigma^*$ následovně:

1. $deref_0^{r'}(a) = a$ pro $a \in \Sigma \cup \Gamma$,

2. $deref_0^{r'}(x_i) = deref_1^{r'}\left(deref_0^{r'}(\alpha)\right)$, pokud platí $x_i \in X' \wedge$ $\left(\exists \alpha, a, b, c \in (\Sigma \cup X \cup \Gamma)^*\right)\left[r' = a[_x\alpha]_x b x_i c \wedge [_x,]_x \notin \alpha \wedge [_x,]_x \notin b\right]$,

3. $deref_0^{r'}(x) = \varepsilon$ jinak,

4. $deref_1^{r'}(a) = a$ pro $a \in \Sigma$,

5. $deref_1^{r'}(y) = \varepsilon$ pro $y \in \Gamma$.

Definice 1.8. Pro referenční slovo r hodnota (anglicky *dereference)* je definována takto:

$$D(r) = deref_1^{r'}\left(deref_0^{r'}(r')\right), \tag{1.1}$$

kde r' je r s očíslovanými proměnnými. Dvě referenční slova r_1 a r_2 jsou *ekvivalentní*, právě když $D(r_1) = D(r_2)$. Hodnota referenčního jazyka R je definována takto:

$$D(R) = \{D(r) \mid r \in R\} \tag{1.2}$$

Pro regex α *hodnota* je definována následovně:

$$L(\alpha) = D(R(\alpha)) \tag{1.3}$$

Třída Regex jazyků je definována takto:

$$\mathbb{L}_{REGEX} = \bigcup_{\alpha \in RV_{\Sigma,X}} L(\alpha) \tag{1.4}$$

Příklad 1.5. Regex $\alpha = x\,(bc)^*\,x\{ya\ y\{ba^* + a\}\ c\}x$ generuje referenční jazyk

$$R(\alpha) = \{x\,(bc)^n\,[_x ya[_y ba^k]_y c]_x x \mid n, k\ \in \mathbb{N}_0\} \cup \{x\,(bc)^n\,[_x ya[_y a]_y c]_x x \mid n \in \mathbb{N}_0\}.$$

Referenční slovo $xbcbc[_x ya[_y ba]_y c]_x x \in R(\alpha)$ odpovídá řetězci

$$D(x_1 bcbc[_x y_1 a[_y ba]_y c]_x x_2) = D\left(bcbc[_x \underbrace{a[_y ba]_y c}_{\alpha}]_x \underline{x_2}\right) = bcbcabacabac.$$

Hodnota regexu α se rovná

$$L(\alpha) = D(R(\alpha)) = \{(bc)^n\left(aba^k c\right)^2 \mid n, k\ \in \mathbb{N}_0\} \cup \{(bc)^n\,(aac)^2 \mid n \in \mathbb{N}_0\}.$$

Pro porovnání hodnoty referenčního slova se vstupním řetězcem lze použít algoritmus 1, který vychází přímo z definice dereference 1.8. Tento algoritmus se chová tak, že projde každý znak c referenčního slova jednou a porovná hodnotu $deref_1^{r'}\left(deref_0^{r'}(c)\right)$ s odpovídajícími znaky vstupního řetězce. Celkový počet porovnávaných znaků nemůže překročit délku řetězce w. Pro nějaký symbol referenčního slova algoritmus porovná maximálně $\mathcal{O}(|w|)$ znaků (řádky 12–15), celkem tedy algoritmus spotřebuje čas $\mathcal{O}(|r| \cdot |w|)$.

Algoritmus 1: Porovnání hodnoty referenčního slova s řetězcem

Vstup : referenční slovo r nad $\Sigma \cup X \cup \Gamma$, vstupní řetězec $w \in \Sigma^*$
Výstup: $D(r) = w$?

1 $pos : \Gamma \to \mathbb{N}$; $(\forall x \in \Gamma)\ pos(x) \leftarrow -\infty$
2 $j \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** $|r|$ **do**
4   **if** $r[i] \in \Sigma$ **then if** $j \geq |w| \ \vee\ r[i] \neq w[j+1]$ **then return** false
5     $j \leftarrow j + 1$
6   **else if** $r[i] \in \Gamma$ **then**
7         $pos(x) \leftarrow j$
8   **else**
9         $start \leftarrow [_{r[i]}$; $end \leftarrow]_{r[i]}$
10   **if** $pos(start) = -\infty \ \vee\ pos(end) \leq pos(start)$ **then** continue
11   **else**
12   **for** $k \leftarrow pos(start) + 1$ **to** $pos(end)$ **do**
13   **if** $j \geq |w| \ \vee\ w[k] \neq w[j+1]$ **then return** false
14                 $j \leftarrow j + 1$
15   **end**
16 **end**
17 **if** $j = |w|$ **then return** true
18 **else return** false

Z příkladu 1.5 vyplývá, že některá referenční slova je možné převést na jednodušší (kratší), pokud budou vynechány například „nedefinované" proměnné.

Definice 1.9. Buď r referenční slovo nad $\Sigma \cup X \cup \Gamma$. Referenční slovo r obsahuje *referenci na nedefinovanou proměnnou* $x \in X$, právě když platí alespoň jedno z následujících tvrzení:

$$\left(\exists a, b \in (\Sigma \cup X \cup \Gamma)^*\right)\Big[\ r = axb \wedge\ ]_x \notin a\ \Big] \tag{1.5}$$

$$\left(\exists \alpha, a, b, c \in (\Sigma \cup X \cup \Gamma)^*\right)\Big[\, r = a[_x \alpha]_x bxc \;\wedge\; [_x,]_x \notin \alpha \;\wedge\; [_x,]_x \notin b \;\wedge$$
$$D(\alpha) = \varepsilon \,\Big] \tag{1.6}$$

Příklad 1.6. Buď $\Sigma = \{c, d\}$ abeceda, buď $X = \{x, y\}$ množina proměnných. $\Gamma = \{[_x,]_x, [_y,]_y\}$. Referenční slovo $r = c[_x y]_x ddx$ nad $\Sigma \cup X \cup \Gamma$ má dvě zpětné reference na nedefinovanou proměnnou. Reference na $y \in X$ je nedefinovaná, protože neexistuje žádná definice této proměnné zleva (je splňen bod 1.5, protože existuje rozklad $r = ayb$ takový, že $a = c[_x$ a $]_y \notin a$). Zpětná reference x je také nedefinovaná, protože je splněn bod 1.6, což znamená, že zleva od tohoto výrazu existuje definice proměnné x, ale hodnota tohoto výrazu se rovná ε (hodnota nedefinované zpětné reference y je prázdné slovo dle pravidla 3 z definice hodnoty). Nedefinované proměnné lze z řetězce odstranit, aniž by se změnila hodnota referenčního slova: $D(c[_x y]_x ddx) = D(c[_x]_x dd) = cdd$.

Definice 1.10. Referenční slovo r obsahuje *zbytečnou definici proměnné* x, právě když platí alespoň jedno z následujících tvrzení:

$$\left(\exists \alpha, a, b \in (\Sigma \cup X \cup \Gamma)^*\right)\Big[\, r = a[_x \alpha]_x b \;\wedge\; [_x,]_x \notin \alpha \;\wedge\; x \notin b \,\Big] \tag{1.7}$$

$$\left(\exists \alpha, a, b, c \in (\Sigma \cup X \cup \Gamma)^*\right)\Big[\, r = a[_x \alpha]_x b]_x cx \;\wedge\; [_x,]_x \notin \alpha \;\wedge\; x \notin b \,\Big] \tag{1.8}$$

Příklad 1.7. Referenční slovo $r = c[_x c]_x [_x d]_x [_y x]_y$ nad $\Sigma \cup X \cup \Gamma$ má dvě zbytečné definice. První definice ve výrazu ($[_x c]_x$) je zbytečná, protože jediná zpětná reference, která se vyskytuje napravo, odkazuje na definici nacházející bezprostředně po dané definici ($[_x d]_x$), což splňuje bod 1.8. Poslední definice je zbytečná, protože napravo od ní neexistuje žádná reference na proměnnou y (je splněn bod 1.7). Závorky, které ohraničují zbytečnou definici, je možné z referenčního slova odstranit, přičemž hodnota výsledného řetězce zůstane stejná: $D(c[_x c]_x [_x d]_x [_y x]_y) = D(cc[_x d]_x x) = ccdd$.

Z definice 1.8 plyne, že referenční slovo může „dosvědčit“ příslušnost slova k hodnotě regexu. V této práci je poprvé zformulováno a dokázáno tvrzení, že pro každé slovo w, které je popsáno nějakým regexem α, existuje referenční slovo, jehož délka je polynomiální vzhledem k $|\alpha|^2 \cdot |w|$.

Lemma 1.1. Buď $\alpha \in RV_{\Sigma,X}$, a buď $w \in L(\alpha)$. Potom nutně existuje referenční slovo r nad $\Sigma \cup X \cup \Gamma$ ($\Gamma = \{[_x,]_x \mid x \in X\}$) takové, že $r \in R(\alpha)$, $D(r) = w$ a platí $|r| \leq |\alpha| + |\alpha|^2 \cdot |w|$.

Důkaz. Trvzení $w \in L(\alpha) \iff \Big(\exists r \in R(\alpha)\Big)\ D(r) = w$ plyne z definice hodnoty regexu. Platí totiž $L(\alpha) = D(R(\alpha)) = \{D(r) \mid r \in R(\alpha)\}$. Potom v referenčním jazyce $R(\alpha)$ existuje nějaké r, jehož hodnota se rovná w ($D(r) = w$).

Druhé tvrzení se rozdělí do dvou případů.

- Nechť α neobsahuje žádnou iteraci. Z pozorování 1.1 vyplývá, že α generuje referenční jazyk R, který je také regulárním jazykem nad $\Sigma' = \Sigma \cup X \cup \Gamma$. Potom délka libovolného referenční slova $r \in R(\alpha)$ nemůže přesáhnout délku regexu ($|r| \leq |\alpha|$). Pokud by délka r byla větší, potom musí nějaké dva podřetězce φ, γ ($\varphi \neq \gamma$) z r odpovídat stejnému podřetězci π z α. To ale je možné, jedině když se π nachází v iteraci, což je spor s předpokladem.

- Předpokládejme, že α obsahuje alespoň jednu iteraci. Nechť S je souborem všech podřetězců α tvaru β^*. Jelikož $r \in R(\alpha)$, existuje nějaké přiřazení podřetězců z r výrazům s_i ($s_i^* \in S$). Potom lze z r odstranit podřetězec γ, který odpovídá nějakému s_i ($s_i^* \in S$), pokud hodnota tohoto podřetězce je ε. Podle definice může γ obsahovat pouze zbytečné definice, které mají prázdnou hodnotu, a nedefinované reference. Po odstranění všech takových podřetězců z r, jejichž hodnoty jsou prázdné, hodnota výsledného referenčního slova r' zůstane stejná, tedy platí $D(r') = D(r) = w$. Jelikož byly z r odstraněny pouze podřetězce odpovídající nějaké iteraci a ε patří do hodnoty regulárního výrazu β^* pro libovolné β, slovo r' bude patřit do referenčního jazyka $R(\alpha)$.

  Jelikož $r' \in R(\alpha)$, existuje nějaké přiřazení podřetězců z r' výrazům s_i ($s_i^* \in S$). Potom r_0 je r', kde všechny takové řetězce jsou opatřeny kulatými závorkami. Nechť tedy $\mathcal{G}$ je strom, pro který platí:

  - V kořeni $\mathcal{G}$ je uložen klíč r_0.
  - Vrchol a s klíčem $\pi(a)$ je synem b (dále značeno $s(a, b)$) právě tehdy, když v $\pi(b)$ existuje podřetězec tvaru $\gamma = (\pi(a))$ a γ se nenachází unitř kulatých závorek v $\pi(b)$.
  - V listech stromu jsou uloženy klíče neobsahující kulaté závorky.
  - Hloubka stromu $\mathcal{G}$ je menší než $\lambda(\alpha)+1$, kde $\lambda(\alpha)$ je hvězdná výška α. Předpokládejme, že strom má hloubku $l \geq \lambda(\alpha)+1$, potom musí existovat alespoň jeden uzel na n-té úrovni, kde $n = \lambda(\alpha) + 2$. To znamená, že kořen stromu má klíč π obsahující podřetězec γ, kde γ se nachází uvnitř $\lambda(\alpha) + 1$ kulatých závorek. Z toho plyne, že γ odpovídá s_i ($s_i^* \in S$) a s_i se nachází uvnitř $\lambda(\alpha) + 1$ do sebe vnořených iterací v regexu α. Potom z definice pro hvězdnou výšku α musí platí $\lambda(\alpha) \geq \lambda(\alpha) + 1$, což není možné.

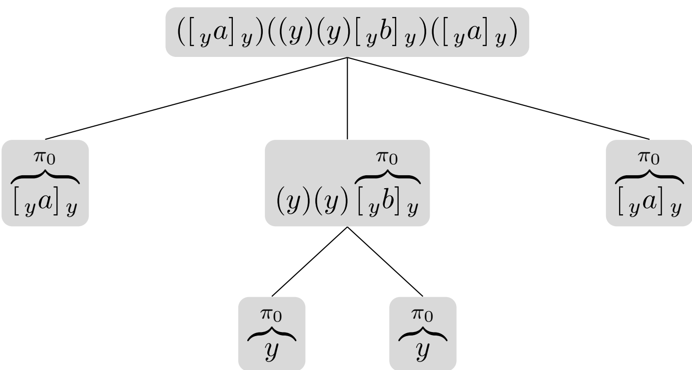
– Počet listu (a tedy i šířka stromu) je menší nebo roven $|w|$. Pokud by strom měl $|w| + 1$ listů, potom alespoň jeden z listů má klíč π takový, že $D(\pi) = \varepsilon$, což není možné (všechny takové řetězce byly z r' odstraněny).

Kořen $\mathcal{G}$ potom má maximálně $\lambda(\alpha) \cdot |w|$ potomků. Množina potomků pro vrchol a je značena $P(a)$. Délka klíče pro každý vrchol a se rovná $|\pi_0(a)| + \sum_{s(i,a)}(|\pi(i)| + 2)$, kde $\pi_0(a)$ je $\pi(a)$ po odstranění všech podřetězců v kulatých závorkách. Jelikož $\pi_0(a)$ odpovídá podsekvenci α neobsahující iteraci, délka $\pi_0(a)$ pro každý vrchol a nepřesáhne $|\alpha|$ a je splněna nerovnost $|\pi(a)| \leq |\alpha| + \sum_{s(i,a)}(|\pi(i)| + 2)$. V r_0 počet závorek je roven počtu potomků kořene stromu φ. Potom pro délku referenčního slova r' platí:

$$\begin{aligned}|r'| = |\pi(\varphi)| - 2|P(\varphi)| &= \pi_0(\varphi) + \sum_{i \in P(\varphi)} (|\pi_0(i)| + 2) - 2|P(\varphi)| \\ &= \pi_0(\varphi) + \sum_{i \in P(\varphi)} |\pi_0(i)| \leq |\alpha| + |\alpha| \cdot \lambda(\alpha) \cdot |w| \leq |\alpha| + |\alpha|^2 \cdot |w|.\end{aligned}$$

$\square$

Pro lepší pochopení důkazu lemmatu 1.1 kroky, které se používají pro regex obsahující iteraci, jsou dále demonstrovány na konkretním příkladě.



Obrázek 1.1: Ukázka grafu $\mathcal{G}$ pro $r' = [_y a]_y yy[_y b]_y[_y a]_y \in R((y^*\ y\{a + b\})^*)$

Příklad 1.8. Nechť $\Sigma = \{a, b\}$ je abeceda a $X = \{y\}$ je množina proměnných. Regex $\alpha = (y^*\ y\{a + b\})^*$ patří do množiny $RV_{\Sigma,X}$. Hvězdná výška je rovna $\lambda(\alpha) = 2$. Soubor iterací výrazu se rovná $S = ((y^*\ y\{a + b\})^*, y^*)$. Referenční slovo $r = y^{1000}[_y a]_y yy[_y b]_y[_y a]_y \in R(\alpha)$ má hodnotu $w = D(r) = aaaba$. Každá z první tisíce referencí na y odpovídá y ($y^* \in S$) a je nedefinovaná, proto lze podřetězec y^{1000} z r odstranit. Pro výsledné referenční slovo $r' = [_y a]_y yy[_y b]_y[_y a]_y$ platí tvrzení $D(r') = D(r)\ \wedge\ r' \in R(\alpha)$. Potom řetězec

$r_0 = ([\,_y a]\,_y)((y)(y)[\,_y b]\,_y)([\,_y a]\,_y)$ je r', kde odpovídající řetězce jsou opatřeny kulatými závorkami, jak je popsáno v důkazu. Graf $\mathcal{G}$ pro r' je zobrazen na obrázku 1.1. Graf má hloubku $\lambda(\alpha)$ a počet listů je $4 \leq |w|$. Délka referenčního slova r' je rovna $11 \leq |\alpha| + |\alpha|^2 \cdot |w| = 616$, i když $|r| > 616$.

Kapitola 2

Algoritmy pro zpracování regexů

Tato kapitola je věnována problému zpracování regexů. Výše uvedený rozhodovací problém lze definovat takto.

Problém 2.1 (Zpracování regexu)**.** Instance: Regex $\alpha \in RV_{\Sigma,X}$, vstupní řetězec $w \in \Sigma^*$.

Otázka: Platí $w \in L(\alpha)$?

Věta 2.1. Problém 2.1 je NP-úplný.

Těžkost tohoto problému lze ukázat převodem z problému vrcholového pokrytí [2, s. 289]. I když autor práce používá odlišnou notaci regulárních výrazu se zpětnými referencemi, vyjadřovací síla daného formalismu je rovna vyjadřovací síle regexu [11, lemma 23 na s. 39]. Aho též popisuje jednoduchý nedeterministický algoritmus, pomocí nějž lze vyřešit problém 2.1 v polynomiálním čase. Základní myšlenka spočívá v nalezení k podřetězců z w, kde k je celkový počet definic a referencí v α. Tímto se problém převede na problém zpracování klasických regulárních výrazů. Algoritmus však nebude nefungovat pro regexy obsahující dílčí výrazy tvaru $(w_1x\{w_2\}w_3)^*$ (resp. $(w_1xw_2)^*$), kde $w_1, w_2, w_3 \in RV_{\Sigma,X}$ [3, s. 83].

Příklad 2.1. Pro regex $\alpha = (x\{a\}\ bx)^*$ a $w = abaaba$ algoritmus Aho nalezne dva podřetězce, které odpovídají výrazu a v definici proměnné x a tyto podřetězce se musí rovnat (definice a reference na tutéž proměnnou). Jelikož počet podřetězců a v α je 4, celkový počet takových dvojic je roven 6. Po odstranění těchto podřetězců z w algoritmus vrátí `true`, pokud $w' \in h(\ (\varepsilon b \varepsilon)^*) = h(b^*)$.

- $w_0 = \overbrace{a}\, b \overbrace{a}\, aba$. $w' = baba \notin h(b^*)$.
- $w_1 = \overbrace{a}\, ba \overbrace{a}\, ba$. $w' = baba \notin h(b^*)$.
- $w_2 = \overbrace{a}\, baab \overbrace{a}$. $w' = abba \notin h(b^*)$.

- $w_3 = ab\overbrace{a}\overbrace{a}ba$. $w' = abba \notin h(b^*)$.
- $w_4 = ab\overbrace{a}ab\overbrace{a}$. $w' = abab \notin h(b^*)$.
- $w_5 = aba\overbrace{a}b\overbrace{a}$. $w' = abab \notin h(b^*)$.

Algoritmus vrátí nesprávný výsledek, protože $abaaba \in L(\alpha)$. Problém spočívá v tom, že nelze předem určit počet podřetězců z w, které je potřeba přiřadit nějakým proměnným v regexu, pokud vstupní regex obsahuje definice nebo zpětné reference v iteraci.

Dá se však ukázat, že problém je v NP, i když regex obsahuje definici proměnné v iteraci. V této práce je poprvé popsán přístup k dokazování založený na pojmu referenčního slova.

Věta 2.2. Problém 2.1 patří do třídy NP.

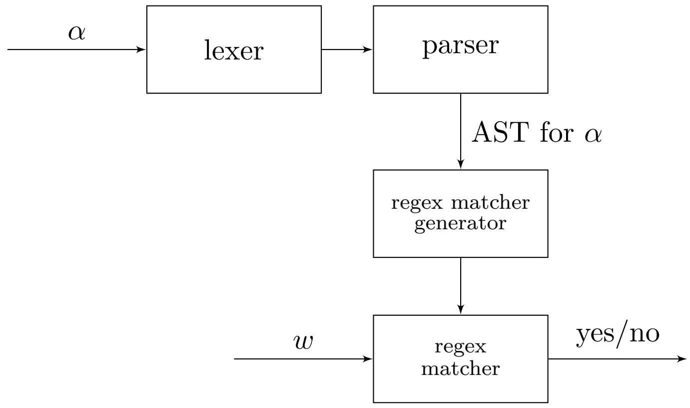
Důkaz. Jednou z možností, jak by mohl vypadat certifikát kladné odpovědí, je nějaké referenční slovo r nad $\Sigma \cup X \cup \Gamma$ ($\Gamma = \{[_x,]_x \mid x \in X\}$). Pokud $w \in L(\alpha)$, podle lemmatu 1.1 musí nutně existovat nějaké referenční slovo, pro něž platí $w = D(r)$, $r \in R(\alpha)$ a $|r| \leq |\alpha| + |\alpha|^2 \cdot |w|$, což znamená, že délka tohoto certifikátu je polynomiální vzhledem k vstupu.

Lze také ukázat, jak tento polynomiální certifikát dosvědčí, že w odpovídá regexu α. K tomu stačí ověřit, že referenční slovo patří k referenčnímu jazyku $R(\alpha)$ a hodnota $D(r)$ se rovná w. Z pozorování 1.1 vyplývá, že rozhodovací problém jestli platí $r \in R(\alpha)$ je možné převést na problém zpracování regulárních výrazů, který lze vyřešit v čase $\mathcal{O}(|\alpha| \cdot |r|)$ ([2, s. 282–285]). Zbývá ověřit, jestli se hodnota $D(r)$ rovná w. Hodnotu $D(r)$ tedy lze porovnat se vstupním slovem pomocí algoritmu 1 v čase $\mathcal{O}(|r| \cdot |w|)$. Jelikož ověření lze provést deterministicky v polynomiálním čase vzhledem k délce vstupu, problém 3.1 patří do třídy NP. □

Všechny algoritmy pro zpracování regexů, kterým je věnován zbytek této kapitoly, se skládají ze dvou části: konstrukce automatu, který přijímá jazyk $L(\alpha)$, a simulace jeho běhu pro vstupní slovo w (viz obrázek 2.1). V dalších sekcích jsou popsány výpočetní modely přijímající libovolný Regex jazyk a algoritmy pro převod regexů na tyto modely.

2.1 Parsování regexů

Algoritmy pro převod nezpracovávají přímo na vstupu znaky regexu, nýbrž strukturu tohoto výrazu. Cílem této sekce je ukázat, jak by mohla vypadat struktura regexu, a popsat následující fáze „překladu“ výrazu: lexikální, syntaktickou a sémantickou analýzu.



Obrázek 2.1: Struktura prostředku pro zpracování regexu

Lexém	**Token**	**Atribut**
znak abecedy	*atom*	ASCII hodnota
ε nebo $\varnothing$	*atom*	‚ε‘ nebo ‚$\varnothing$‘
proměnná	*var*	název proměnné
$+$	*union*	-
$*$	*iter*	-
{ nebo }	{ nebo }	-
(nebo)	(nebo)	-

Tabulka 2.1: Lexikální elementy a jejich reprezentace

2.1.1 Lexikální analýza

Cílem lexikální analýzy je konverze vstupní posloupnosti na lexikální symboly (dále jen *tokeny*). Jazyk tokenů lze popsat regulární gramatikou a lexikální analyzátor (též *lexer*) je pak tvořen deterministickým konečným automatem. Pokud lexer nerozpozná nějaký vstupní symbol, vznikne lexikální chyba.

V tabulce 2.1 je uvedeno, který token a atribut vrátí lexer pro jednotlivé lexémy.

2.1.2 Návrh gramatiky

Aby bylo možné přijímat jazyk regexů, je vhodné pro něj zkonstruovat gramatiku. Bezkontextová gramatika generující regexy z $RV_{\Sigma,X}$ by mohla vypadat následovně:

$G = (\{A, B, C, D\}, \{atom, var, \{, \}, (,), union, iter\}, P, A)$, kde:

$$\begin{aligned} P = \; & \{A \to B \; union \; A \mid B, \\ & B \to CB \mid C, \\ & C \to D \; iter \mid D, \\ & D \to atom \mid var \mid var\{A\} \mid (A)\} \end{aligned}$$

Uvedená gramatika podporuje všechny operace a operátory mají rozdílnou prioritu. Priorita ovlivňuje v jakém pořadí je výraz vyhodnocován.

2.1.3 Syntaktická analýza

Cílem syntaktické analýzy je určit gramatickou strukturu vstupu na základě předem dané gramatiky. Syntaktický analyzátor (též *parser*) dostane na vstup řetězec tokenů a hledá derivační strom k tomuto řetězci. Pokud žádný derivační strom nenajde, vznikne syntaktická chyba.

V této práci se používá $LL(1)$ analýza jako metoda pro syntaktický analyzátor. Před zpracováním řetězce tokenů je nutné ověřit, zda daná gramatika je LL(1) (v opačném případě hrozí nekonečná rekurze). Gramatika G z sekce 2.1.2 není LL(1), protože tato gramatika má *first-first konflikty* v rozkladové tabulce pro neterminální symboly A, B, C a terminály $atom, var, ($. K odstranění konfliktů lze použít levou faktorizaci. Výsledná ekvivalentní gramatika G' je uspořádanou čtveřicí:

$(\{A, A', B, B', C, C', D, F'\}, \{atom, var, \{, \}, (,), union, iter\}, P', A)$, kde:

$$P' = \{A \to BA', \tag{2.1}$$
$$A' \to union \; A, \tag{2.2}$$
$$A' \to \varepsilon, \tag{2.3}$$
$$B \to CB', \tag{2.4}$$
$$B' \to B, \tag{2.5}$$
$$B' \to \varepsilon, \tag{2.6}$$
$$C \to DC', \tag{2.7}$$
$$C' \to iter, \tag{2.8}$$
$$C' \to \varepsilon, \tag{2.9}$$
$$D \to atom, \tag{2.10}$$
$$D \to var \; F', \tag{2.11}$$
$$D \to (A), \tag{2.12}$$
$$F' \to \{A\}, \tag{2.13}$$
$$F' \to \varepsilon \; \} \tag{2.14}$$

Rozkladová tabulka pro tuto gramatiky je znázorněna pomocí tabulky 2.2.

	atom	**var**	$\{$	$\}$	(	)	**union**	**iter**	ε
A	2.1	2.1			2.1				
A'				2.3		2.3	2.2		2.3
B	2.4	2.4			2.4				
B'	2.5	2.5		2.6	2.5	2.6	2.6		2.6
C	2.7	2.7			2.7				
C'	2.9	2.9		2.9	2.9	2.9	2.9	2.8	2.9
D	2.10	2.11			2.12				
F'	2.14	2.14	2.13	2.14	2.14	2.14	2.14	2.14	2.14

Tabulka 2.2: Rozkladová tabulka pro gramatiku generující regexy

2.1.4 Reprezentace regexu abstraktním syntaktickým stromem

Pro reprezentaci regexu se nabízí použít syntaktický strom, jehož vnitřní uzly představují nějaký operátor (zřetězení, sjednocení, iteraci nebo definici proměnné). Koncové uzly reprezentují jednoduché operandy (atomy a proměnné). Pro regexy stačí pět typů uzlů:

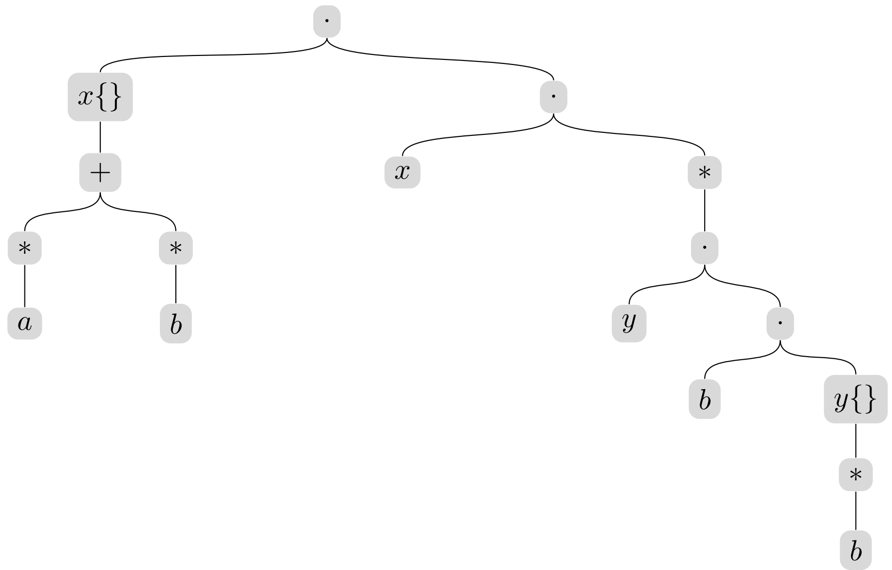
- unární operátor (iterace)
- binární operátor (zřetězení a sjednocení)
- definice proměnné
- atom
- zpětná reference

Na obrázku 2.2 je zobrazen příklad AST pro regex $x\{a^*+b^*\}\ x\ (yb\ y\{b^*\})^*$.

2.1.5 Sémantická analýza

Sémantická analýza se obvykle provádí zároveň se syntaktickou analýzou. Úkolem sémantické analýzy je sestrojit abstraktní syntaktický strom, který uchovává všechny podstatné informace obsažené v regexu s ohledem na jeho strukturu.

Sémantický analyzátor může být vytvořen podle L-atributové gramatiky se vstupní gramatikou G' (viz sekci 2.1.3). Všem neterminálům je přiřazen syntetizovaný atribut *snode*, jehož hodnotou bude ukazatel na vytvořený strom. Dědičný atribut *dnode* u neterminálních symbolů A', B', C' obsahuje ukazatel na vytvořený podstrom pro levý operand. Pro terminální symbol *atom* syntetizovaný atribut *svalue* odpovídá hodnotě elementárního výrazu a atribut



Obrázek 2.2: Ukázka AST pro výraz $x\{a^* + b^*\}\ x\ (yb\ y\{b^*\})^*$

svar pro symbol *var* obsahuje název proměnné. Posloupnost sémantických akcí pro atributovou gramatiku je definována pomocí tabulky 2.3.

2.2 Vícepáskový lineárně omezený Turingův stroj

V této sekci je definován vícepáskový lineární omezený Turingův stroj a jsou popsány jeho vlastnosti (vycházeno z [6] a [7]). Pod pojmem „Turingův stroj“ je v textu myšlena nedeterministická verze tohoto výpočetního modelu.

Definice 2.1. *Turingův stroj s k páskami*, zkráceně *TS(k)*, je formálně definován jako sedmice $\mathcal{M} = (Q, \Sigma, G, \delta, q_0, B, F)$, kde:

- Q je konečná neprázdná množina stavů,
- G je konečná neprázdná pracovní abeceda,
- Σ je konečná vstupní abeceda ($\Sigma \subseteq G$),
- $\delta : (Q \setminus F) \times G^k \to \wp\left(Q \times (G \times \{-1, 0, 1\})^k\right)$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $B \in (G \setminus \Sigma)$ je prázdný symbol,
- $F \subseteq Q$ je konečná množina koncových stavů.

Syntaktické pravidlo	Sémantická pravidla
$A \rightarrow BA'$	$A'.dnode = B.snode$ $A.snode = A'.snode$
$A' \rightarrow union\ A$	$A'.snode = new\ BinOp(union, A'.dnode, A.snode)$
$A' \rightarrow \varepsilon$	$A'.snode = A'.dnode$
$B \rightarrow CB'$	$B'.dnode = C.snode$ $B.snode = B'.snode$
$B' \rightarrow B$	$B'.snode = new\ BinOp(concat, B'.dnode, B.snode)$
$B' \rightarrow \varepsilon$	$B'.snode = B'.dnode$
$C \rightarrow DC'$	$C'.dnode = D.snode$ $C.snode = C'.snode$
$C' \rightarrow iter$	$C'.snode = new\ UnOp(iter, C'.dnode)$
$C' \rightarrow \varepsilon$	$C'.snode = C'.dnode$
$D \rightarrow atom$	$D.snode = new\ Atom(atom.svalue)$
$D \rightarrow var\ F'$	$F'.dnode = new\ Var(var.svar)$ $D.snode = F'.snode$
$D \rightarrow (A)$	$D.snode = A.snode$
$F' \rightarrow \{A\}$	$F'.snode = new\ Def(F'.dnode, A.snode)$
$F' \rightarrow \varepsilon$	$F'.snode = F'.dnode$

Tabulka 2.3: Sémantická pravidla pro vytvoření AST

Definice přechodové funkce nám říká, že je-li vícepáskový TS v nekoncovém stavu a čtecí hlavy na páskách ukazují na nějaké symboly z pracovní abecedy, poté přejde do dalšího stavu, na každou pásku zapíše nějaký symbol a každou čtecí hlavu posune vlevo, vpravo nebo zůstane na místě.

Definice 2.2. Nechť $\mathcal{M} = (Q, \Sigma, G, \delta, q_0, B, F)$ je Turingův stroj s k páskami. *Konfigurace* $\mathcal{M}$ je trojice $(q, \langle w_1, \ldots, w_k\rangle, \langle i_1, \ldots, i_k\rangle)$, kde:

- q je aktuální stav,
- w_i je obsah i-té pásky,
- i_j je pozice čtecí hlavy na j-té pásce.

Počáteční konfigurace $\mathcal{M}$ pro vstup w je konfigurace

$$\Big(q_0, \langle w, B^{|w|}, \ldots, B^{|w|}\rangle, \langle 0, \ldots, 0\rangle\Big)$$

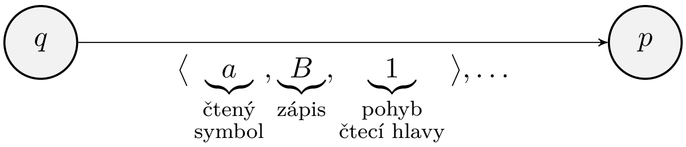
Přechodem $\mathcal{M}$ se nazývá binární relace $\vdash_{\mathcal{M}}$ na množině konfigurací definovaná následovně:

$$\begin{aligned} &(q, \langle w_0 \dots w_{i_1-1} w_{i_1} w_{i_1+1} \dots w_n, \dots \rangle, \langle i_1, \dots \rangle) \vdash_{\mathcal{M}} \\ &(p, \langle w_0 \dots w_{i_1-1} x_1 w_{i_1+1} \dots w_n, \dots \rangle, \langle i_1 + j_1, \dots \rangle) \\ &\iff (p, \langle (x_1, j_1), \dots \rangle) \in \delta (q, \langle w_{i_1}, \dots \rangle) \end{aligned} \tag{2.15}$$

Jazyk přijímaný $\mathcal{M}$ je množina

$$L(\mathcal{M}) = \Big\{ w \;\Big|\; (\exists p \in F)(\exists x_1, \dots, x_{k-1} \in G^*)(\exists i_1, \dots, i_k \in \mathbb{N}_0) \Big[(q_0, (w, \varepsilon, \dots, \varepsilon), (0, \dots, 0)) \vdash_{\mathcal{M}}^* \Big(p, \Big(B^{|w|}, x_1, \dots, x_{k-1}\Big), (i_1, \dots, i_k)\Big) \Big] \Big\} \tag{2.16}$$

V této práci se používá graf přechodů pro znázornění přechodové funkce automatu. Hrany grafu reprezentují přechody a jsou ohodnoceny posloupností k trojic. Každá trojice odpovídá operacím, které se provádějí s i-tou páskou (kde i je pozice trojice v posloupnosti), a skládá se ze čteného symbolu, symbolu k zápisu a pohybu čtecí hlavy. Na obrázku 2.3 je zobrazen přechod ze stavu q do p, po němž automat přečte z 0. pásky symbol a, zapíše na ni blank symbol a posune čtecí hlavu vpravo.



Obrázek 2.3: Znázornění přechodové funkce $TS(k)$ pomocí grafu přechodů

Věta 2.3. Pro každý k-páskový TS existuje ekvivalentní jednopáskový TS.

Důkaz věty 2.3 je podrobně uveden v [6, s. 337–338].

Definice 2.3. *Lineárně omezený Turingův stroj*, zkráceně LOTS, je Turingův stroj, který nemůže překročit délku k-násobku vstupního slova pro nějaké $k \geq 1$.

Algoritmus 2 lze použít pro simulaci výpočtu vícepáskového TS pro řetězec w. Funkce *accepts* má tři vstupní parametry, které jednoznačně určují aktuální konfiguraci TS. Na začátku se zavolá tato funkce se vstupními parametry, které odpovídají počáteční konfigurace automatu. Funkce vrátí hodnotu *true*, pokud je na vstupu koncová konfigurace. Jinak pro každý přechod algoritmus vytvoří novou konfiguraci a zavolá rekurzivně funkci *accepts*.

Algoritmus 2: Simulace vícepáskového TS

Vstup : $TS(k)$ $M = (Q, G, B, \Sigma, \delta, q_0, F)$, vstupní slovo w
Výstup: $w \in L(M)$?

$tapes[0, \dots, k-1][1, \dots, |w|]$; $tapes[0] \leftarrow w$
$pos[0 \dots, k-1]$; $(\forall i \in \{0, \dots, k-1\})$ $pos[i] \leftarrow 0$
$curState \leftarrow q_0$
def `accepts`($curState$, $tapes$, pos)
  if $curState \in F \;\wedge\; tapes[0] = B^{|w|+2}$ **then return** true
  $transitions \leftarrow$ `getTrans`($curState$, $tapes$, pos)
  for $i \leftarrow 1$ **to** $|transitions|$ **do**
    $tapesCopy \leftarrow$`getCopy`($tapes$)
    $posCopy \leftarrow$`getCopy`(pos)
    $stateCopy \leftarrow$`getCopy`($curState$)
    `execTrans`($transitions[i]$, $stateCopy$, $tapesCopy$, $posCopy$)
    if `accepts`($stateCopy$, $tapesCopy$, $posCopy$) **then**
      return true
  end
  return false

return `accepts`($curState$, $tapes$, pos)

2.3 Algoritmus pro převod regexu na vícepáskový LOTS

V této sekci je ukázáno, jak lze pro libovolný regex α sestrojit vícepáskový lineárně omezený TS, který přijímá jazyk $L(\alpha)$. Základní myšlenka konstrukce byla převzata z [10, s. 8], kde je popsán převod semiregexů na vícepáskový TS. Autor práce zmíněný algoritmus rozšířil na množinu regexů. Při modifikaci byl brán zřetel především na to, že se v regexech oproti semerigexům může vyskytnout několik definic téže proměnné $(\dots x\{\dots\} \dots x\{\dots\} \dots)$.

Algoritmus 3 dostane AST pro regex α na vstupu. Potom výstupem algoritmu je $TS(k+1)$ $(k = |var(\alpha)|)$, který má následující vlastnosti:

- Na počátku výpočtu je na 0. pásce zapsán vstupní řetězec. Této pásce se také říká *vstupní*. Ostatní (pracovní) pásky jsou vyplněny prázdnými symboly. Počáteční konfigurace tedy je

$$\left(q_{start}, \langle BwB, B^{|w|+2}, \dots, B^{|w|+2}\rangle, \langle 1, \dots, 1\rangle\right)$$

- Pro nějaké $i \in \{1, \dots, k\}$ i-tá páska odpovídá nějaké proměnné x v α. V kroku výpočtu, kdy symboly na vstupní pásce odpovídají dílčímu výrazu v definici proměnné x, automat odstraní symboly z i-té pásky a začne kopírovat znaky ze vstupní pásky na i-tou, dokud nenarazí na poslední symbol odpovídající tomuto výrazu. Pokud se čtecí hlava na vstupní pásce ukazuje na začátek řetězce, který odpovídá zpětné referenci na x, TS posune čtecí hlavu na i-té pásce na začátek a začne porovnávat obsah i-té pásky se vstupní.

Nechť num je zobrazení z $var(\alpha)$ do množiny $\{1, \ldots, k\}$ takové, že pro každé dvě různé proměnné x, y v α platí $num(x) \neq num(y)$. Potom i-tá páska odpovídá proměnné označené číslem i. Nechť tedy $T[1, \ldots, k]$ je tabulka, která nabývá hodnot z $\{0, 1\}$. Potom $T[i] = 1$, právě když se na i-tou pásku kopíruje obsah 0. pásky, jinak $T[i] = 0$.

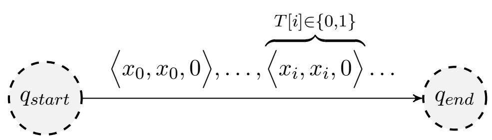
Algoritmus prochází rekurzivně uzly stromu a přidává nové stavy a odpovídající přechody pro daný typ uzlu. Konstrukce vícepáskového Turingova stroje pro atomy a regulární operace vychází z podobné myšlenky jako Thompsonův algoritmus pro převod regulárních výrazů na NKA. Výsledný Turingův stroj má počáteční stav q_{start} a jediný koncový stav q_{end}. Postup konstrukce je popsán pro jednotlivé uzly takto:

1. Prvním typem uzlu je elementární výraz. Může nastat jeden z následujících případů:

   a) Uzel reprezentuje prázdný řetězec. Potom se přidají „ε-přechody“ ze stavu q_{start} do stavu q_{end}. Pod pojmem „ε-přechod“ je myšleno přechod, při kterém stroj přečte a zapíše tentýž symbol na každou pásku a čtecí hlavy na každé pásce zůstanou na místě. Počet přechodů je roven počtu $(k+1)$-členných variací s opakováním z $|G|$ prvků ($|G|^{k+1}$).

      Pro $\forall x_0, \ldots, x_k \in G$ se přechodová funkce modifikuje následovně (viz obrázek 2.4):

$$\begin{aligned}\delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \leftarrow & \delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \cup \\ & \{(q_{end}, \langle x_0, 0\rangle, \ldots, \langle x_k, 0\rangle)\}.\end{aligned}$$



Obrázek 2.4: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím výraz ε

   b) Dalším typem uzlu je prázdný regulární výraz. Pro tento uzel se do množiny stavů přidá nový stav p. Přidají se „ε-přechody“ ze stavu q_{start} do stavu p. Stav p zde zastává funkci „nulového“ stavu (přechodová funkce δ neobsahuje žádný přechod ze stavu p do nějakého dalšího stavu).

Algoritmus 3: Převod regexu na vícepáskový LOTS

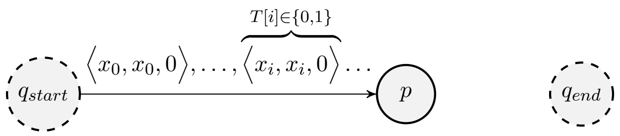
Vstup : AST reprezentující regex $\alpha \in RV_{\Sigma,X}$ s kořenem φ
Výstup: $(k+1)$-páskový LOTS M takový, že $L(M) = L(\alpha)$
`/*` k `je počet proměnných v` α $(k = |var(\alpha)|)$ `*/`

$num : X \to \{1, \ldots, k\}$; $(\forall x, y \in X)$ $num(x) = num(y) \Rightarrow x = y$
$(\forall i \in \{1, \ldots, k\})$ $T[i] \leftarrow 0$
$Q \leftarrow \{q_{start}, q_{end}\}$; $F \leftarrow \{q_{end}\}$; $G \leftarrow \Sigma \cup \{B\}$
def `build-TM(`β, q_{start}, q_{end}`)`
  if $type(\beta) = atomic$ **then**
    if $\beta.value = \varepsilon$ **then** uprav δ podle (1a)
    else if $\beta.value = \varnothing$ **then** $p \notin Q$; $Q \leftarrow Q \cup \{p\}$
    uprav δ podle (1b)
    else uprav δ podle (1c)
  else if $type(\beta) = union$ **then**
    $q^l_{start}, q^r_{start}, q^l_{end}, q^r_{end} \notin Q$; $Q \leftarrow Q \cup \{q^l_{start}, q^r_{start}, q^l_{end}, q^r_{end}\}$
    uprav δ podle (2a)
    `build-TM(`$\beta.left$, q^l_{start}, q^l_{end}`)`; `build-TM(`$\beta.right$, q^r_{start}, q^r_{end}`)`
  else if $type(\beta) = concatenation$ **then**
    $q^l_{start}, q^r_{start}, q^l_{end}, q^r_{end}, q_{mid} \notin Q$;
     $Q \leftarrow Q \cup \{q^l_{start}, q^r_{start}, q^l_{end}, q^r_{end}, q_{mid}\}$
    uprav δ podle 2b
    `build-TM(`$\beta.left$, q^l_{start}, q^l_{end}`)`; `build-TM(`$\beta.right$, q^r_{start}, q^r_{end}`)`
  else if $type(\beta) = iteration$ **then**
    $q^\alpha_{start}, q^\alpha_{end} \notin Q$; $Q \leftarrow Q \cup \{q^\alpha_{start}, q^\alpha_{end}\}$
    uprav δ podle (3a)
    `build-TM(`$\beta.inner$, q^α_{start}, q^α_{end}`)`
  else if $type(\beta) = definition$ **then**
    $p, q^\alpha_{start}, q^\alpha_{end} \notin Q$; $Q \leftarrow Q \cup \{p, q^\alpha_{start}, q^\alpha_{end}\}$
    uprav δ podle (3b)
    $T[num(\beta.var)] \leftarrow 1$
    `build-TM(`$\beta.inner$, q^α_{start}, q^α_{end}`)`
    $T[num(\beta.var)] \leftarrow 0$
  else if $type(\beta) = reference$ **then**
    $p, q \notin Q$
    $Q \leftarrow Q \cup \{p, q\}$
    uprav δ podle (4)

`build-TM(`φ, q_{start}, q_{end}`)`
$\mathcal{M} = (Q, \Sigma, G, \delta, q_{start}, B, F)$
return $\mathcal{M}$

Pro $\forall x_0, \ldots, x_k \in G$ se provede úprava přechodové funkce automatu následovně (viz obrázek 2.5):

$$\begin{aligned}\delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \leftarrow & \delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \cup \\ & \{(p, \langle x_0, 0 \rangle, \ldots, \langle x_k, 0 \rangle)\}.\end{aligned}$$



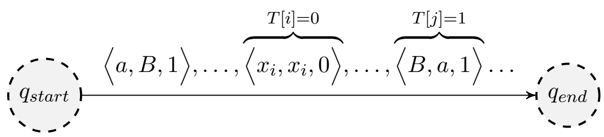
Obrázek 2.5: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím prázdný regulární výraz

c) Posledním elementárním výrazem je symbol abecedy. Ze stavu q_{start} po přečtení symbolu abecedy ze vstupní pásky, automat překopíruje tento symbol na všechny „otevřené“ pásky ($T[i] = 1$), odstraní ho ze vstupní pásky a přejde do stavu q_{end} (viz obrázek 2.6). Uvedené kroky lze formálně popsat takto:

$n \leftarrow \Big| \{i \in \{1, \ldots, k\} \mid T[i] = 0\} \Big|$

Pro $\forall x_1, \ldots, x_n \in G$ a nějaký symbol abecedy a se přidají přechody:

$$\begin{aligned}\delta(q_{start}, \langle a, \ldots, \overbrace{x_1}^{T[i]=0}, \ldots, \overbrace{B}^{T[j]=1}, \ldots \rangle) \leftarrow & \\ & \delta(q_{start}, \langle a, \ldots, x_1, \ldots, B, \ldots \rangle) \cup \\ & \{(q_{end}, \langle B, 1 \rangle, \ldots, \langle x_1, 0 \rangle, \ldots, \langle a, 1 \rangle \ldots)\}.\end{aligned}$$



Obrázek 2.6: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím výraz a

2. Další typ vrcholu je binární operátor. Pro takovýto případ se do množiny stavů pro každý operand přidají „počáteční“ a „koncový“ stav (například stavy q^l_{start} a q^l_{end} pro levý operand). Algoritmus začne rekurzivně procházet pravého a levého potomka a přidávat pro ně nové stavy a přechody do automatu.

a) Pokud při průchodu algoritmus narazí na uzel reprezentující sjednocení dvou výrazů, jednoduše se přidají „ε-přechody“ mezi stavem q_{start} a „počátečními“ stavy pro potomky a z „koncových“ stavů do stavu q_{end} (viz obrázek 2.7).

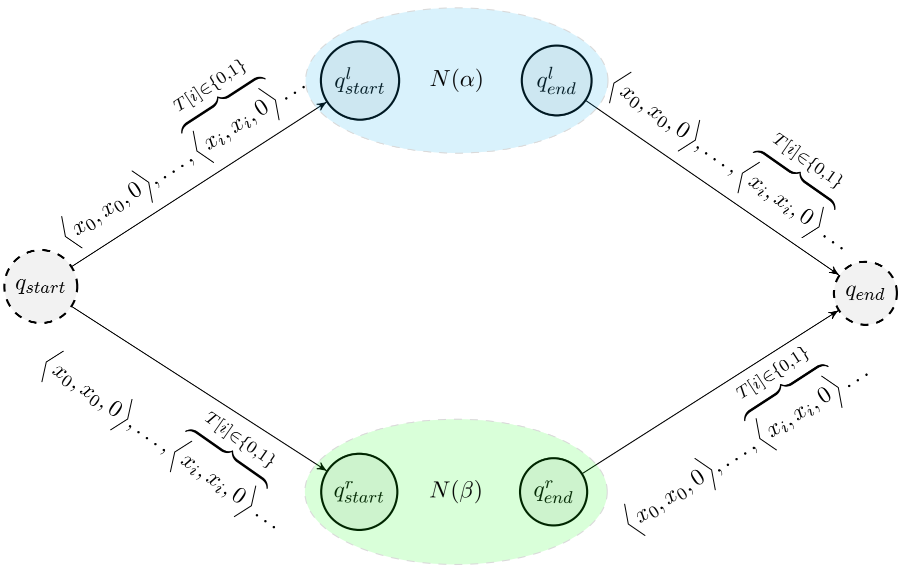
Pro $\forall x_0, \dots, x_k \in G$ se přechodová funkci modifikuje následovně:

$$\delta(q_{start}, \langle x_0, \dots, x_k \rangle) \leftarrow \delta(q_{start}, \langle x_0, \dots, x_k \rangle) \cup \{(q^l_{start}, \langle x_0, 0 \rangle, \dots, \langle x_k, 0 \rangle)\}$$

$$\delta(q_{start}, \langle x_0, \dots, x_k \rangle) \leftarrow \delta(q_{start}, \langle x_0, \dots, x_k \rangle) \cup \{(q^r_{start}, \langle x_0, 0 \rangle, \dots, \langle x_k, 0 \rangle)\}$$

$$\delta(q^l_{end}, \langle x_0, \dots, x_k \rangle) \leftarrow \delta(q^l_{end}, \langle x_0, \dots, x_k \rangle) \cup \{(q_{end}, \langle x_0, 0 \rangle, \dots, \langle x_k, 0 \rangle)\}$$

$$\delta(q^r_{end}, \langle x_0, \dots, x_k \rangle) \leftarrow \delta(q^r_{end}, \langle x_0, \dots, x_k \rangle) \cup \{(q_{end}, \langle x_0, 0 \rangle, \dots, \langle x_k, 0 \rangle)\}$$



Obrázek 2.7: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím výraz $\alpha + \beta$

b) Další typ uzlu, na který lze při průchodu narazit, je zřetězení. Pro tento typ uzlu se do množiny stavů automatu přidá nový stav q_{mid}.

Pak algoritmus vytvoří „ε-přechody“ mezi stavy, jako je znázorněno na obrázku 2.8.

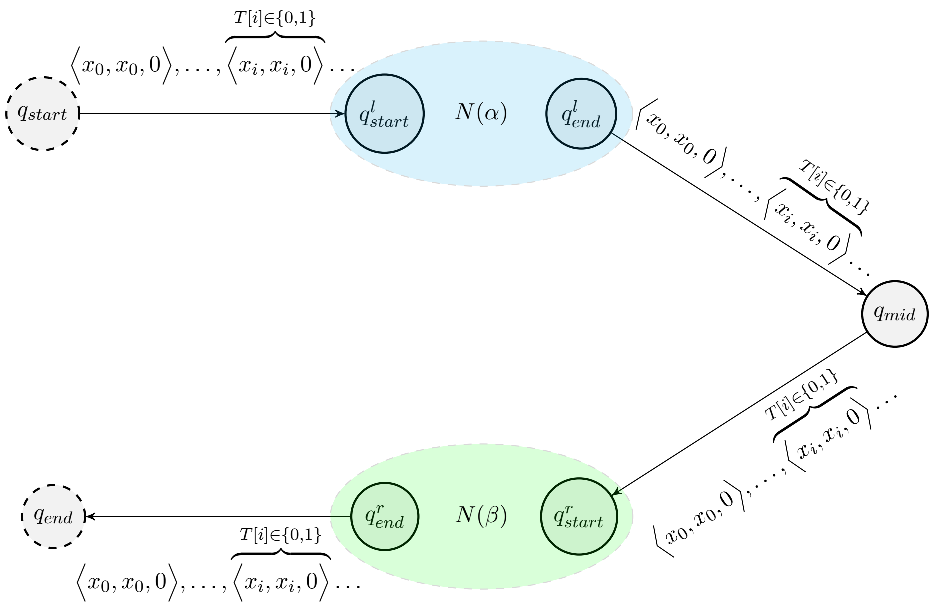
Pro $\forall x_0, \ldots, x_k \in G$ se přechodová funkce upraví takto:

$$\begin{aligned}\delta(q_{start}, \langle x_0, \ldots, x_k\rangle) \leftarrow & \delta(q_{start}, \langle x_0, \ldots, x_k\rangle) \cup \\ & \{(q^l_{start}, \langle x_0, 0\rangle, \ldots, \langle x_k, 0\rangle)\}\end{aligned}$$

$$\begin{aligned}\delta(q_{mid}, \langle x_0, \ldots, x_k\rangle) \leftarrow & \delta(q_{mid}, \langle x_0, \ldots, x_k\rangle) \cup \\ & \{(q^r_{start}, \langle x_0, 0\rangle, \ldots, \langle x_k, 0\rangle)\}\end{aligned}$$

$$\begin{aligned}\delta(q^l_{end}, \langle x_0, \ldots, x_k\rangle) \leftarrow & \delta(q^l_{end}, \langle x_0, \ldots, x_k\rangle) \cup \\ & \{(q_{mid}, \langle x_0, 0\rangle, \ldots, \langle x_k, 0\rangle)\}\end{aligned}$$

$$\begin{aligned}\delta(q^r_{end}, \langle x_0, \ldots, x_k\rangle) \leftarrow & \delta(q^r_{end}, \langle x_0, \ldots, x_k\rangle) \cup \\ & \{(q_{end}, \langle x_0, 0\rangle, \ldots, \langle x_k, 0\rangle)\}\end{aligned}$$



Obrázek 2.8: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím výraz $\alpha \cdot \beta$

3. Při průchodu lze také narazit na unární operátor nebo definice proměnné. Potom se do množiny stavů přidají nové stavy q^{α}_{start} a q^{α}_{end} pro potomka.

a) Jediným unárním operátorem, na nějž lze narazit při průchodu, je iterace. Pro takovýto případ algoritmus vytvoří „ε-přechody" mezi stavy, jako je znázorněno na obrázku 2.9, a začne rekurzivně procházet uzlem potomka.

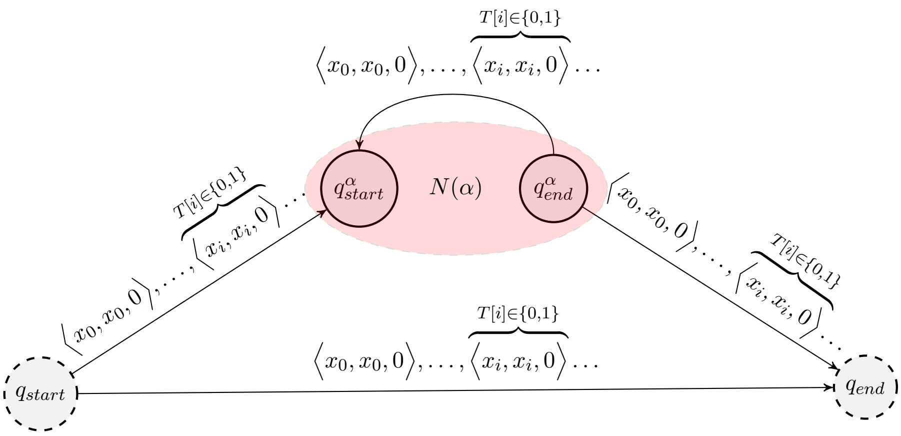
Pro $\forall x_0, \ldots, x_k \in G$ se přidají přechody takto:

$$\delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \leftarrow \delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \cup \{(q_{end}, \langle x_0, 0 \rangle, \ldots, \langle x_k, 0 \rangle)\}$$

$$\delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \leftarrow \delta(q_{start}, \langle x_0, \ldots, x_k \rangle) \cup \{(q^{\alpha}_{start}, \langle x_0, 0 \rangle, \ldots, \langle x_k, 0 \rangle)\}$$

$$\delta(q^{\alpha}_{end}, \langle x_0, \ldots, x_k \rangle) \leftarrow \delta(q^{\alpha}_{end}, \langle x_0, \ldots, x_k \rangle) \cup \{(q^{\alpha}_{start}, \langle x_0, 0 \rangle, \ldots, \langle x_k, 0 \rangle)\}$$

$$\delta(q^{\alpha}_{end}, \langle x_0, \ldots, x_k \rangle) \leftarrow \delta(q^{\alpha}_{end}, \langle x_0, \ldots, x_k \rangle) \cup \{(q_{end}, \langle x_0, 0 \rangle, \ldots, \langle x_k, 0 \rangle)\}$$



Obrázek 2.9: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím výraz α^*

b) Dalším typem uzlu je definice proměnné. Jelikož pro libovolnou proměnnou y je vyhrazena právě jedna $num(y)$-tá páska a definice proměnné se může opakovat ve výrazů několikrát, je nutné nejprve všechny neprázdné symboly z pásky vymazat. Stroj postupně posouvá čtecí hlavu na příslušné pásce a maže symboly, dokud nenarazí na první *blank* symbol (viz obrázek 2.10). Pak algoritmus

„povolí“ zápis na pásku a rekurzivně projde uzlem potomka. Po skončení se „zakáže“ zápis na pásku a automat přejde do koncového stavu q_{end}.

Pro $\forall x_0, \dots, x_{k-1} \in G$ se přechodová funkce modifikuje následovně:

$$\delta(q_{start}, \langle x_0, \dots, \overbrace{B}^{num(y)}, \dots \rangle) \leftarrow \delta(q_{start}, \langle x_0, \dots, B, \dots \rangle) \cup \{(p, \langle x_0, 0 \rangle, \dots, \langle B, -1 \rangle, \dots)\}$$

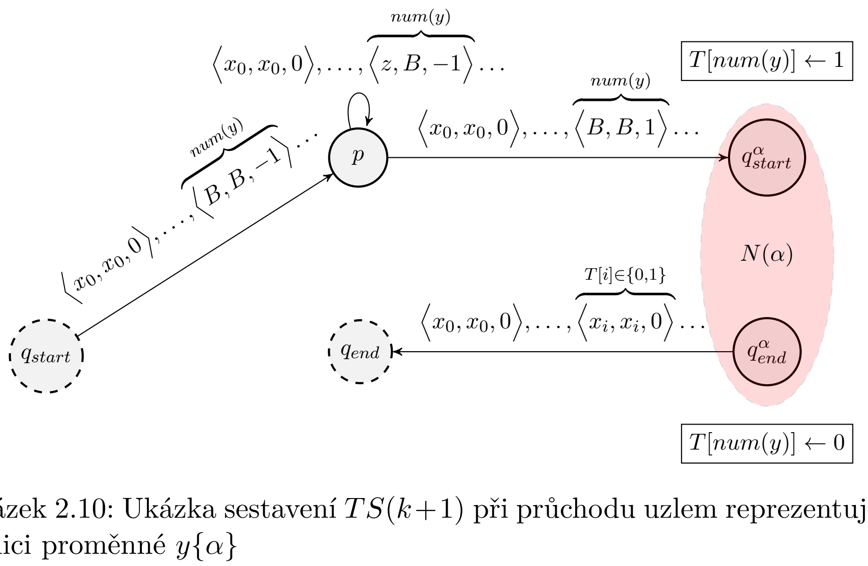
$$\delta(p, \langle x_0, \dots, \overbrace{B}^{num(y)}, \dots \rangle) \leftarrow \delta(p, \langle x_0, \dots, B, \dots \rangle) \cup \{(q^{\alpha}_{start}, \langle x_0, 0 \rangle, \dots, \langle B, 1 \rangle, \dots)\}$$

Pro $\forall x_0, \dots, x_{k-1} \in G$, $\forall z \in \Sigma$ algoritmus vytvoří přechody takto:

$$\delta(p, \langle x_0, \dots, \overbrace{z}^{num(y)}, \dots \rangle) \leftarrow \delta(p, \langle x_0, \dots, z, \dots \rangle) \cup \{(p, \langle x_0, 0 \rangle, \dots, \langle B, -1 \rangle, \dots)\}$$

Pro $\forall x_0, \dots, x_k \in G$:

$$\delta(q^{\alpha}_{end}, \langle x_0, \dots, x_k \rangle) \leftarrow \delta(q^{\alpha}_{end}, \langle x_0, \dots, x_k \rangle) \cup \{(q_{end}, \langle x_0, 0 \rangle, \dots, \langle x_k, 0 \rangle)\}$$



Obrázek 2.10: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím definici proměnné $y\{\alpha\}$

4. Poslední typ uzlu, na který lze narazit při průchodu, je zpětná reference. Podobně jako v předchozím případě stroj postupně posouvá čtecí hlavu

na příslušné pásce, dokud nenarazí na první *blank* symbol (viz obrázek 2.11). Pak začne porovnávat obsah vstupní pásky s $num(y)$-tou a přejde do koncového stavu pouze v případě, že se obsahy obou pásek shodují.

Pro $\forall x_0, \ldots, x_{k-1} \in G$ se přidají následující přechody:

$$\begin{aligned}\delta(q_{start}, \langle x_0, \ldots, \overbrace{B}^{num(y)}, \ldots \rangle) \leftarrow & \delta(q_{start}, \langle x_0, \ldots, B, \ldots \rangle) \cup \\ & \{(p, \langle x_0, 0 \rangle, \ldots, \langle B, -1 \rangle, \ldots)\}\end{aligned}$$

$$\begin{aligned}\delta(p, \langle x_0, \ldots, \overbrace{B}^{num(y)}, \ldots \rangle) \leftarrow & \delta(p, \langle x_0, \ldots, B, \ldots \rangle) \cup \\ & \{(q, \langle x_0, 0 \rangle, \ldots, \langle B, 1 \rangle, \ldots)\}\end{aligned}$$

$$\begin{aligned}\delta(q, \langle x_0, \ldots, \overbrace{B}^{num(y)}, \ldots \rangle) \leftarrow & \delta(q, \langle x_0, \ldots, B, \ldots \rangle) \cup \\ & \{(q_{end}, \langle x_0, 0 \rangle, \ldots, \langle B, 0 \rangle, \ldots)\}\end{aligned}$$

Pro $\forall x_0, \ldots, x_{k-1} \in G$, $\forall z \in \Sigma$ se přechodová funkce modifikuje následovně:

$$\begin{aligned}\delta(p, \langle x_0, \ldots, \overbrace{z}^{num(y)}, \ldots \rangle) \leftarrow & \delta(p, \langle x_0, \ldots, z, \ldots \rangle) \cup \\ & \{(p, \langle x_0, 0 \rangle, \ldots, \langle z, -1 \rangle, \ldots)\}\end{aligned}$$

Pro $\forall x_0, \ldots, x_{k-2} \in G$, $\forall z \in \Sigma$:

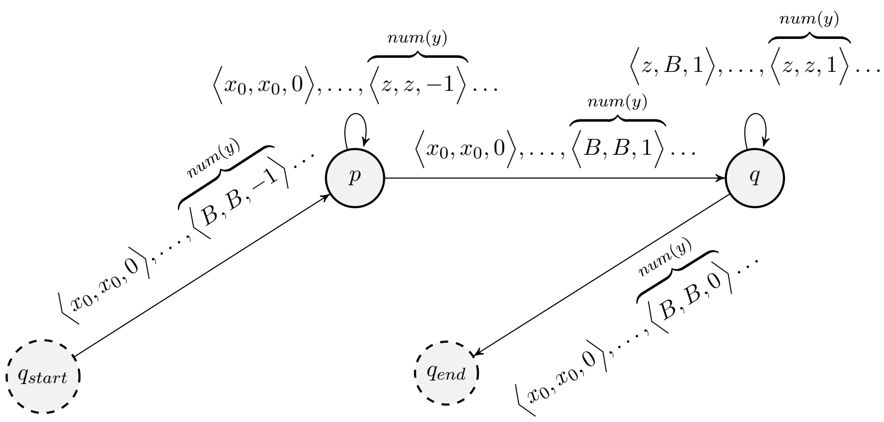
$$\begin{aligned}\delta(q, \langle z, \ldots, \overbrace{z}^{num(y)}, \ldots \rangle) \leftarrow & \delta(q, \langle z, \ldots, z, \ldots \rangle) \cup \\ & \{(q, \langle B, 1 \rangle, \ldots, \langle z, 1 \rangle, \ldots)\}\end{aligned}$$

Věta 2.4. Časová složitost algoritmu 3 je $\mathcal{O}(|\alpha| \cdot |\Sigma|^{k+1})$.

Důkaz. Inicializace algoritmu (kroky 1 až 3) trvá $\mathcal{O}(k)$. Algoritmus rekurzivně prochází každý vrchol syntaktického stromu nejvýše jednou, tedy se celkem provede $\mathcal{O}(|\alpha|)$ volání funkce `build-TM`. V každém volání se do množiny stavů přidají maximálně 5 stavů (krok 15) a vytvoří se nejvýše $2|G|^k + |G|^{k+1} + |\Sigma| \cdot |G|^k \leq 3|G|^{k+1}$ přechodů (krok 24). Jelikož $|G| = |\Sigma| + 1$, jedno volání funkce potrvá $\mathcal{O}(|\Sigma|^{k+1})$. Celkově tedy algoritmus spotřebuje čas $\mathcal{O}(|\alpha| \cdot |\Sigma|^{k+1})$. □

Věta 2.5. Pro regex α výstupem algoritmu 3 je lineárně omezený TS s k páskami $\mathcal{M}$ takový, že $L(\mathcal{M}) = L(\alpha)$.

Důkaz. Buď $\mathcal{M}$ výstupní Turingův stroj, buď φ kořen AST pro α. Ekvivalenci tohoto výpočetního modelu a regexu lze dokázat indukcí podle typu uzlu φ.



Obrázek 2.11: Ukázka sestavení $TS(k+1)$ při průchodu uzlem reprezentujícím zpětnou referenci na proměnnou y

1. Pro elementární výraz je tvrzení triviální.

   Nechť je φ uzel reprezentující regex ε. $L(\varepsilon) = \{\varepsilon\}$. Výstupem algoritmu pro φ je $\mathcal{M} = (\{q_{start}, q_{end}\}, \varnothing, \{B\}, \delta, q_{start}, B, \{q_{end}\})$. Jazyk přijímaný $\mathcal{M}$ je $\{w \mid (q_{start}, (BwB), (1)) \vdash_{\mathcal{M}} \left(q_{end}, \left(B^{|w|+2}\right), (1)\right)\} = \{\varepsilon\}$.

   Nechť φ reprezentuje prázdný regulární výraz. $L(\varnothing) = \varnothing$. Výstupem je $\mathcal{M} = (\{q_{start}, p, q_{end}\}, \varnothing, \{B\}, \delta, q_{start}, B, \{q_{end}\})$. Jelikož v přechodové funkci automatu neexistuje žádný přechod do koncového stavu, $L(\mathcal{M}) = \varnothing$.

   Poslední elementární výraz, na něhož může algoritmus při průchodu narazit, je symbol abecedy a. $L(a) = a$. Výstupem algoritmu pro φ je $\mathcal{M} = (\{q_{start}, q_{end}\}, \{a\}, \{B, a\}, \delta, q_{start}, B, \{q_{end}\})$. Jazyk přijímaný $\mathcal{M}$ je $\{w \mid (q_{start}, (BwB), (1)) \vdash_{\mathcal{M}} \left(q_{end}, \left(B^{|w|+2}\right), (2)\right)\} = \{a\}$.

2. Nechť φ reprezentuje regulární operaci.

   Kořen φ reprezentuje regex $\beta \cdot \gamma$. Hodnota výrazu je $L(\beta \cdot \gamma) = L(\beta)L(\gamma)$. Nechť $\mathcal{M}_\beta$ a $\mathcal{M}_\gamma$ jsou automaty pro β a γ. Výstupní TS $\mathcal{M}$ přijímá jazyk $L(\mathcal{M}_\beta)L(\mathcal{M}_\gamma) = L(\beta)L(\gamma)$.

   Nechť φ reprezentuje regex $\beta + \gamma$. $L(\beta + \gamma) = L(\beta) \cup L(\alpha)$. Nechť $\mathcal{M}_\beta$ a $\mathcal{M}_\gamma$ jsou automaty pro β a γ. Výstupní TS $\mathcal{M}$ přijímá jazyk $L(\mathcal{M}_\beta) \cup L(\mathcal{M}_\gamma) = L(\beta) \cup L(\gamma)$.

   Nechť je φ uzel reprezentující regex β^*. Hodnota výrazu je $L(\beta^*) = L(\beta)^*$. Buď $\mathcal{M}_\beta$ automat pro regex β. Jazyk přijímaný výstupním TS $\mathcal{M}$ je $\{\varepsilon\} \cup L(\mathcal{M}_\beta) \cup L(\mathcal{M}_\beta)L(\mathcal{M}_\beta) \cup \ldots = L(\mathcal{M}_\beta)^* = L(\beta)^*$.

3. Nechť φ reprezentuje definici proměnné $x\{\beta\}$. $L(x\{\beta\}) = L(\beta)$. Buď $\mathcal{M}_\beta = (Q, \Sigma, G, \delta_{\mathcal{M}_\beta}, start_{\mathcal{M}_\beta}, B, F)$ automat pro regex β. Výstupní au-

tomat $\mathcal{M}$ je sedmice $(\{q_{start}, p, q_{end}\} \cup Q, \Sigma, G, \delta_{\mathcal{M}}, q_{start}, B, \{q_{end}\})$. Jazyk přijímaný automatem $\mathcal{M}$ je $\{w \mid \left(q_{start}, \left(BwB, B^{|w|+2}\right), (1,1)\right) \vdash^*_{\mathcal{M}} \left(q_{end}, \left(B^{|w|+2}, BwB\right), (|w|+1, |w|+1)\right)\} = L(\mathcal{M}_\beta) = L(\beta)$. Na konci výpočtu na pracovní pásce pro proměnnou x je zapsáno vstupní slovo w.

4. Nechť φ reprezentuje zpětnou referenci na x. Výstupní automat $\mathcal{M}$ je sedmice $(\{q_{start}, p, q, q_{end}\}, \Sigma, G, \delta_{\mathcal{M}}, q_{start}, B, \{q_{end}\})$. Jazyk přijímaný $\mathcal{M}$ je

$$\begin{aligned}\{w \mid (q_{start}, (BwB, BwB), (1,1)) \vdash^*_{\mathcal{M}} \\ \left(q_{end}, \left(B^{|w|+2}, BwB\right), (|w|+1, |w|+1)\right)\}.\end{aligned}$$

   Automat přijme pouze slova, která odpovídají obsahu pracovní pásky pro proměnnou x. Pokud je páska prázdná, proměnná nebyla ve výrazu definována. Potom $L(\mathcal{M})$ se rovná hodnotě $L(x)$.

Jelikož automat $\mathcal{M}$ použije maximálně $|w| + 2$ buněk na každé pásce (celkem $(k+1)(|w|+2)$ pro $k = |var(\alpha)|$), výstupní TS je lineárně omezený. $\square$

Jelikož pro každý regex lze sestrojit ekvivalentní vícepáskový Turingův stroj pomocí algoritmu 3 a pro každý k-páskový TS existuje ekvivalentní jednopáskový TS (věta 2.3), platí následující tvrzení.

Věta 2.6. Každý jazyk z množiny $\mathbb{L}_{REGEX}$ je kontextovým jazykem.

Algoritmus, který přijme na vstup regex α a vstupní slovo w, zkonstruuje ekvivalentní vícepáskový TS $\mathcal{M}$ a nasimuluje výpočet $\mathcal{M}$ pro vstup w, se v této práci nazývá `simpleTM`.

Příklad převodu regexu na vícepáskový TS lze najít v příloze A.1.

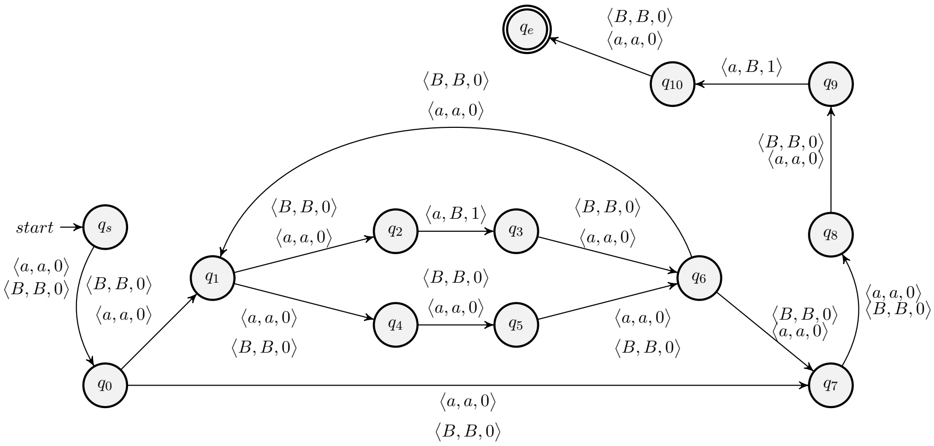
Lze však ukázat, že se výpočet vícepáskového Turingova stroje, který byl zkonstruován pomocí algoritmu `simpleTM`, může zacyklit (i když ve vstupním regexu není žádná proměnná). Tento případ je potřeba ošetřit.

Příklad 2.2. Buď $\alpha = (a+\varepsilon)^* a$. Výstupem algoritmu pro tento regex je TS $\mathcal{M}$ s jednou páskou a přechodovou funkcí δ (viz obrázek 2.12).

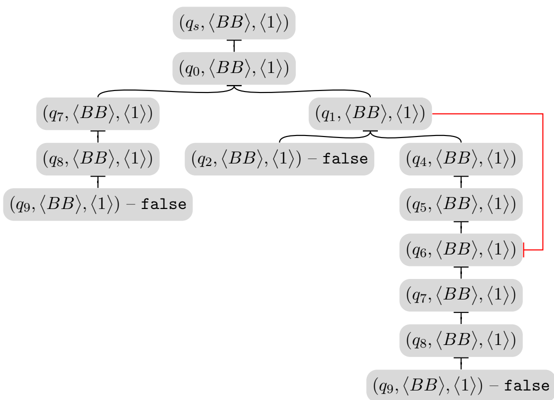
Automat se zacyklí například pro vstup $w = \varepsilon$. Posloupnost přechodů $\mathcal{M}$ z počáteční konfigurace $(q_s, \langle BB \rangle, \langle 1 \rangle)$ je zobrazena na obrázku 2.13. Lze si všimnout, že tato situace nastává, když v AST existuje vrchol reprezentující iteraci a hodnota výrazu potomka daného vrcholu obsahuje prázdné slovo ε. Potom se při výpočtu může vzniknout cyklus v posloupnosti přechodů

$$(q^\alpha_{start}, \langle w_0, \dots \rangle, \langle i_0, \dots \rangle) \vdash^*_{\mathcal{M}} (q^\alpha_{start}, \langle w_0, \dots \rangle, \langle i_0, \dots \rangle),$$

kde q^α_{start} je počáteční stav pro potomka (viz obrázek 2.9).



Obrázek 2.12: Přechodová funkce TS přijímajícího jazyk $L((a+\varepsilon)^*a)$



Obrázek 2.13: Posloupnost přechodů $\mathcal{M}$ pro $w = \varepsilon$

Je však možné upravit algoritmus 2 pro simulaci vícepáskového TS tak, aby podobné situace nevznikaly. Nejprve pro každý takový stav q^{α}_{start} uložíme konfiguraci pásek pokaždé, když algoritmus zavolá funkci `accepts` se vstupním parametrem $curState = q^{\alpha}_{start}$. Pokud se nová konfigurace rovná uložené, algoritmus tuto větev zahodí a funkce vrátí `false`. Pomocí kontroly opakování konfigurací je docíleno toho, aby se při simulaci konfigurace neopakovaly.

Pozorování 2.1. Výstup algoritmu 3 je $TS(k)$ $\mathcal{M}$, který má následující vlastnost. Pro libovolnou konfiguraci $\varphi = (q, \langle w_1, \ldots, w_k\rangle, \langle i_1, \ldots, i_k\rangle)$ platí

$$|\{p \mid \varphi \vdash_{\mathcal{M}} p\}| \leq 2.$$

Věta 2.7. Algoritmus `simpleTM` pracuje v čase $\mathcal{O}(|\alpha| \cdot |\Sigma|^{k+1} \cdot |w|^{4k})$, kde $k = |var(\alpha)|$.

Důkaz. Množina všech konfigurací vícepáskového TS $\mathcal{M}$, který je výstupem algoritmu 3, je $C_{\mathcal{M}} = \{(q, \langle w_0, w_1, \ldots, w_k \rangle, \langle i_0, i_1, \ldots, i_k \rangle) \mid w_0 \in S_w, \forall i \in \{1, \ldots, k\}\ w_i \in F_w\}$ (kde S_w (resp. F_w) je množina suffixů (resp. podřetězců) w). Pro počet konfigurací stroje platí $|C_{\mathcal{M}}| = \mathcal{O}(|Q| \cdot |w|^k \cdot (|w|^2)^k \cdot |w|^k) = \mathcal{O}(|Q| \cdot |w|^{4k})$ (kde Q je množina stavů $\mathcal{M}$). Z pozorování 2.1 vyplývá, že simulace $\mathcal{M}$ spotřebuje čas $\mathcal{O}(|Q| \cdot |w|^{4k})$. Jelikož platí $Q = \mathcal{O}(|\alpha|)$ (pro každý vrchol AST se vytvoří maximálně 5 stavů) a konstrukce $TS(k)$ spotřebuje čas $\mathcal{O}(|\alpha| \cdot |\Sigma|^{k+1})$, celkem zpracování regexu α trvá $\mathcal{O}(|\alpha| \cdot |\Sigma|^{k+1} \cdot |w|^{4k})$. □

2.4 Memory automat

V této sekci je popsán další výpočetní model přijímající REGEX jazyky. *Memory automat* (zkráceně $\mu KA(k)$) lze chápat jako nedeterministický konečný automat rozšířený o k adresovatelných paměti. Při definování pojmů týkajících se memory automatů je vycházeno z [5, sekce 2.2].

Definice 2.4. Buď $k \in \mathbb{N}_0$ a buď $\Gamma = \{i, [_i,]_i \mid i \in \{0, \ldots, k-1\}\}$. $\mu KA(k)$ je formálně definován jako pětice $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná neprázdná množina vnitřních stavů,
- Σ je neprázdná vstupní abeceda,
- $\delta : (Q \times (\Sigma \cup \{\varepsilon\} \cup \Gamma)) \to \wp(Q)$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů. [5]

Definice 2.5. Konfigurace $\mu KA(k)$ je trojice $(q, w, (\langle r_0, s_0 \rangle, \ldots \langle r_{k-1}, s_{k-1} \rangle))$, kde:

- q je aktuální stav,
- w je obsah vstupní pásky,
- r_i je obsah i-té paměti,
- $s_i \in \{\mathtt{O}, \mathtt{C}\}$ je stav i-té paměti. [5]

Sémantický význam jednotlivých přechodů je zaveden pomocí následující definice.

Definice 2.6. Buď c_i, c_j jsou konfigurace automatu. Binární relace $\vdash_{\mathcal{M}}$ na množině konfigurací automatů je definovaná následovně:

$c_i \vdash_{\mathcal{M}} c_j$, právě když platí alespoň jedno z následujících tvrzení:

- $c_i = (q, vw, (\langle r_0, s_0\rangle, \dots, \langle r_{k-1}, s_{k-1}\rangle))$;
  $c_j = (p, w, (\langle r'_0, s_0\rangle, \dots, \langle r'_{k-1}, s_{k-1}\rangle))$;
  $\exists p \in \delta(q, x)$, kde ($x \in \Sigma \cup \{\varepsilon\} \;\wedge\; v = x$) nebo ($x \in \{0, \dots, k-1\} \;\wedge\; s_x = \mathtt{C} \;\wedge\; v = r_x$) a platí
  $$\forall i \in \{0, \dots, k-1\}\Big(s_i = \mathtt{O} \implies r'_i = r_i v\Big),$$
  $$\forall i \in \{0, \dots, k-1\}\Big(s_i = \mathtt{C} \implies r'_i = r_i\Big).$$
- $c_i = (q, vw, (\langle r_0, s_0\rangle, \dots, \langle r_{k-1}, s_{k-1}\rangle))$;
  $c_j = (p, w, (\langle r'_0, s_0\rangle, \dots, \langle r'_j, s_j\rangle, \dots, \langle r'_{k-1}, s_{k-1}\rangle))$;
  $\exists p \in \delta(q, x)$, kde ($x = [_j \;\wedge\; s'_j = \mathtt{O} \;\wedge\; r'_j = \varepsilon$) nebo ($x =]_j \;\wedge\; s'_j = \mathtt{C} \wedge r'_j = r_j$). [5]

Počáteční konfigurace $\mu KA(k)$ $\mathcal{M}$ pro vstup w je konfigurace

$$(q_0, w, (\langle \varepsilon, \mathtt{C}\rangle \dots \langle \varepsilon, \mathtt{C}\rangle))\,.$$

Konfigurace $(p, \varepsilon, (\langle r_0, s_0\rangle, \dots, \langle r_{k-1}, s_{k-1}\rangle))$ je *koncová*, právě když p je koncový stav. Memory automat přijímá slovo w, pokud existuje posloupnost konfigurací z počáteční (pro tento vstup) do nějaké koncové.

Jazyk přijímaný $\mu KA(k)$ $\mathcal{M}$ je množina všech slov, které daný automat přijímá.

Podobně jako je tomu u vícepáskového TS, simulace výpočtu memory automatu pro vstupní slovo w odpovídá prohledávání grafů konfigurací tohoto automatu.

2.5 Převod regexu na ekvivalentní memory automat

Tato sekce je věnována algoritmu pro převod regexu na memory automat. Postup konstrukce byl převzat z [5, sekce 2.3].

Prvním krokem je konstrukce orientovaného grafu s ohodnocenými hranami $\mathcal{H}(\alpha)$ pro vstupní regex α, která je popsána pomocí následující definice.

Definice 2.7. Buď $\alpha \in RV_{\Sigma,X}$, a buď $\mathcal{N}(\alpha)$ množina uzlů syntaktického stromu, který reprezentuje α. Orientovaný graf s ohodnocenými hranami $\mathcal{H}(\alpha)$ je trojice (V, E, f), kde V je množina vrcholů, E je množina hran a f je zobrazení z E do $\Sigma \cup X \cup \Gamma \cup \{\varepsilon\}$ ($\Gamma = \{[_x,]_x \mid x \in X\}$). Pro každý uzel t z $\mathcal{N}(\alpha)$ v množině vrcholu V jsou dva vrcholy t_{in}, t_{out} (pro zřetězení V obsahuje navíc vrchol t_{mid}).

- Pokud uzel t reprezentuje zřetězení, množina hran obsahuje hrany $e_1 = (t_{in}, l_{in})$, $e_2 = (l_{out}, t_{mid})$, $e_3 = (t_{mid}, r_{in})$ a $e_4 = (r_{out}, t_{out})$, kde l (resp. r) je uzel reprezentující levého (resp. pravého) potomka. Hrany e_1, e_2, e_3, e_4 jsou ohodnoceny ε.

- Pokud uzel t reprezentuje sjednocení, množina hran obsahuje hrany $e_1 = (t_{in}, l_{in})$, $e_2 = (t_{in}, r_{in})$, $e_3 = (l_{out}, t_{out})$ a $e_4 = (r_{out}, t_{out})$. Hrany e_1, e_2, e_3, e_4 jsou ohodnoceny ε.

- Pokud uzel t reprezentuje iteraci, množina hran obsahuje hrany $e_1 = (t_{in}, i_{in})$, $e_2 = (t_{in}, t_{out})$, $e_3 = (t_{out}, t_{in})$ a $e_4 = (i_{out}, t_{out})$, kde i je uzel reprezentující potomka. Hrany e_1, e_2, e_3, e_4 jsou ohodnoceny ε.

- Pokud uzel t reprezentuje iteraci, množina hran obsahuje hrany $e_1 = (t_{in}, i_{in})$, $e_2 = (t_{in}, t_{out})$, $e_3 = (t_{out}, t_{in})$ a $e_4 = (i_{out}, t_{out})$, kde i je uzel reprezentující potomka. Hrany e_1, e_2, e_3, e_4 jsou ohodnoceny ε.

- Pokud uzel t reprezentuje elementární výraz, množina hran obsahuje hranu $e = (t_{in}, t_{out})$ a hrana je ohodnocena hodnotou tohoto výrazu.

- Pokud uzel t reprezentuje definici na proměnnou x, množina hran obsahuje hrany $e_1 = (t_{in}, i_{in})$, $e_2 = (i_{out}, t_{out})$, kde i je uzel reprezentující potomka. $f(e_1) = [_x$ a $f(e_2) =]_x$.

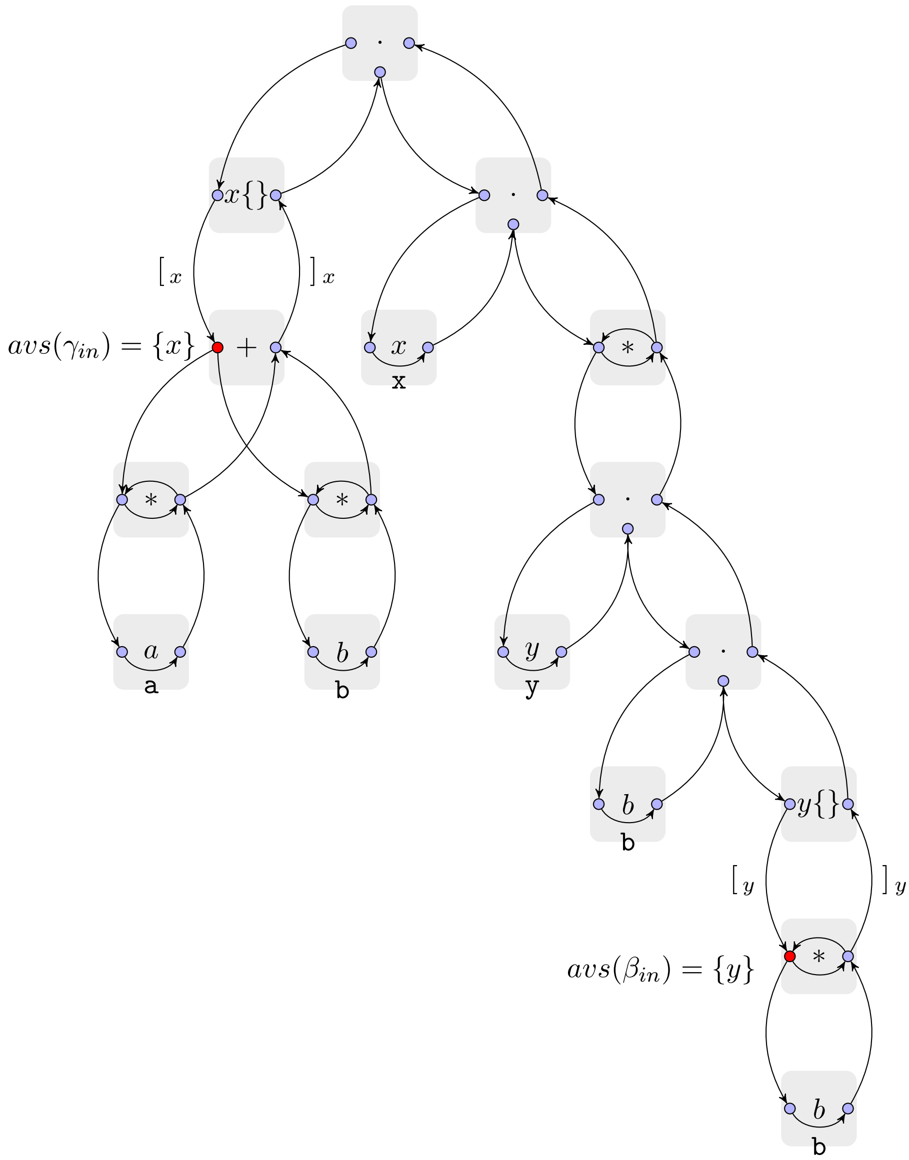
- Pokud uzel t reprezentuje zpětnou referenci na $x \in X$, množina hran obsahuje hranu $e = (t_{in}, t_{out})$ $(f(e) = x)$. [5]

Příklad 2.3. Nechť $\alpha = x\{a^* + b^*\}\ x\ (yb\ y\{b^*\})^*$ je regex z $RV_{\{a,b\},\{x,y\}}$. Graf $\mathcal{H}(\alpha)$ je zobrazen na obrázku 2.14. Všimněte si, že se na graf $\mathcal{H}(\alpha)$ lze podívat jako na graf přechodové funkce NKA s ε-přechody $\mathcal{M} = (V(\mathcal{H}(\alpha)), \{a, b\} \cup \{x, y\} \cup \{[_x, [_y,]_x,]_y\}, \delta, \varphi_{in}, \{\varphi_{out}\})$ přijímajícího referenční jazyk $R(\alpha)$. Počáteční (resp. jediný koncový) stav automatu je φ_{in} (resp. φ_{out}), kde φ je kořen AST reprezentujícího regex α.

Nechť num je zobrazení z $var(\alpha)$ do množiny $\{0, \ldots, k-1\}$ takové, že pro každé dvě různé proměnné x, y v α platí $num(x) \neq num(y)$. Z grafu $\mathcal{H}(\alpha)$ lze vytvořit graf přechodové funkce δ substitucí každého výskyt proměnné x v ohodnocení nějaké hrany za $num(x)$. Potom i-tá paměť odpovídá proměnné označené číslem i a výsledný automat $\mathcal{M} = (V(\mathcal{H}(\alpha)), \Sigma, \delta, \varphi_{in}, \{\varphi_{out}\})$ (kde φ je kořen AST pro α) přijímá jazyk $L(\alpha)$ [5, s. 5].

Příklad převodu regexu na memory automat pomocí uvedeného algoritmu lze najít v příloze A.2.

Jelikož konstrukce memory automatu pro regex je podobná konstrukci TS, výsledný μKA se také může zacyklit, pokud v AST existuje vrchol reprezentující iteraci a hodnota výrazu potomka obsahuje ε (viz příklad 2.2). Při simulaci



Obrázek 2.14: Graf $\mathcal{H}(x\{a^* + b^*\}\ x\ (yb\ y\{b^*\})^*)$

memory automatu je kontrolováno, jestli se konfigurace neopakují, obdobným způsobem, jako je tomu u TS.

Algoritmus, který přijme na vstup regex α a vstupní slovo w, zkonstruuje ekvivalentní memory automat $\mathcal{M}$ a nasimuluje výpočet $\mathcal{M}$ pro vstup w, se v této práci nazývá `simpleMemory`.

Věta 2.8. Časová složitost algoritmu `simpleMemory` je $\mathcal{O}(|\alpha| \cdot |w|^{k+1})$, kde $k = |var(\alpha)|$.

Důkaz. Konstrukce grafu $H(\alpha)$ spotřebuje čas $\mathcal{O}(|\alpha|)$ [5, lemma 3]. Simulace výsledného memory automatu potrvá $\mathcal{O}(|Q| \cdot |2|^k \cdot |w|^{k+1})$, kde $|Q| = \mathcal{O}(|\alpha|)$ [5, lemma 2]. $\square$

Kapitola 3

Parametrizovaná složitost zpracování regexů

Problém zpracování regexů patří mezi problémy, jejichž parametrizovaná složitost dosud nebyla podrobně zkoumána. Věta 2.8 nám poskytuje algoritmus pracující v čase $\mathcal{O}(|I|^{k+1})$, kde k je součástí vstupu I. Lze proto tvrdit, že parametrizovaný problém 2.1 patří do třídy XP.

Cílem této kapitoly je prozkoumat společnou vlastnost (parametr) vstupů a vlastnost regexů, na níž je tento parametr závislý. V této kapitole bude vycházeno z [5].

3.1 Množina aktivních proměnných

Časovou složitost algoritmu pro převod regulárních výrazů se zpětnými referencemi na memory automat, je možné zlepšit několika způsoby. Jednou možností je omezení paměťových prvků, které se používají pro zpracování zpětných referencí.

Příklad 3.1. Buď $X = \{x, y\}$ množina proměnných, buď $\Sigma = \{a, b\}$ abeceda. Pro regex $\alpha = x\{a^*\}\ x\ (ya\ y\{b^*\})^* \in RV_{\Sigma,X}$ podle algoritmu `simpleMemory` lze sestrojit memory automat se dvěma pamětmi. Je očividné, že jazyk $L(\alpha)$ lze přijat i μKA(1) tak, že se jediná paměť současně využije ke zpracování obou zpětných referencí. K tomu však nestačí pouze substituovat proměnnou y za x, protože se hodnota regexu $x\{a^*\}\ x\ (xa\ x\{b^*\})^*$ nerovná $L(\alpha)$.

Vlastnost regexu, pomocí níž lze omezit počet pásek při převodu na TS, je definována na orientovaném grafu s ohodnocenými hranami $\mathcal{H}(\alpha)$. Konstrukce tohoto grafu je popsána v sekci 2.4.

Definice 3.1. Buď $\varphi \in \mathcal{N}(\alpha)$ kořen AST. Binární relace $\triangleright_{def} \subseteq X \times V$ a $\triangleright_{call} \subseteq v \ \times X$ jsou definovány takto:

$$\begin{aligned} x \triangleright_{def} \beta \iff & \text{ v } \mathcal{H}(\alpha) \text{ existuje cesta } (\varphi_{in}, e_1, \ldots, e_n, \beta) \text{ taková, že} \\ & \Big(\exists k \ \in \{1, \ldots, n\}\Big)\Big[f(e_k) = [\,_x\Big] \end{aligned} \tag{3.1}$$

$$\begin{aligned} \beta \triangleright_{call} x \iff & \text{ v } \mathcal{H}(\alpha) \text{ existuje cesta } (\beta, e_1, \ldots, e_n, \varphi_{out}) \text{ taková, že} \\ & \Big(\exists k \ \in \{1, \ldots, n\}\Big)\Big[f(e_k) = x\Big] \ \wedge \ \Big(\forall i < k\Big)\Big[f(e_i) \neq [\,_x\Big] \end{aligned} \tag{3.2}$$

Pro $\beta \in V$ se množina $avs(\beta) := \{x \mid x \triangleright_{def} \beta \triangleright_{call} x\}$ nazývá množina aktivních proměnných. Parametr *avd* (anglicky *active variable degree*) pro regex α je definován takto: [5]

$$avd(\alpha) := max\Big\{|avs(\varphi)| \ \Big| \ \varphi \in v \ \ \wedge \Big(\exists e = (\gamma, \varphi) \in E\Big)\Big[f(e) = [\,_x\Big]\Big\} \tag{3.3}$$

Příklad 3.2. Nechť $\alpha = x\{a^* + b^*\}\ x\ (yb\ y\{b^*\})^*$ je regex z $RV_{\{a,b\},\{x,y\}}$. Graf $\mathcal{H}(\alpha)$ je zobrazen na obrázku 2.14. K určení parametru *avd* stačí najít aktivní proměnné pouze pro dva vrcholy γ_{in}, β_{in} z $V(\mathcal{H}(\alpha))$ (označeny červeně), kde γ, $\beta \in \mathcal{N}(a)$ jsou uzly reprezentující výraz uvnitř definice. Jelikož množina aktivních proměnných pro oba vrcholy je jednoprvková, $avd(\alpha) = 1$.

Pozorování 3.1. Buď $x \in X$, a buď $G := \Sigma \cup X \cup \{\varepsilon\} \cup \Gamma$. Nedeterministický konečný automat $\mathcal{M}_{x,\triangleright_{def}} = (\{q, p\}, G, \delta_0, q, \{p\})$ přijímá jazyk $L(\mathcal{M}_{x,\triangleright_{def}}) = \{w \mid \ |w|_{[\,_x} \geq 1\}$. NKA $\mathcal{M}_{x,\triangleright_{call}}$ je pětice $(\{q, p, \varnothing\}, G, \delta_1, q, \{p\})$ a $L(\mathcal{M}_{x,\triangleright_{call}}) = \{uxv \mid \ |u|_{[\,_x} = 0\}$. Přechodové funkce δ_0 a δ_1 jsou znázorněny pomocí grafů přechodů na obrázku 3.1.

Nechť α je regex z $RV_{\Sigma,X}$ a $\mathcal{H}(\alpha) = (V, E, f)$. Nechť $\delta : v \ \times G \to \wp(V)$ je zobrazení definované takto:

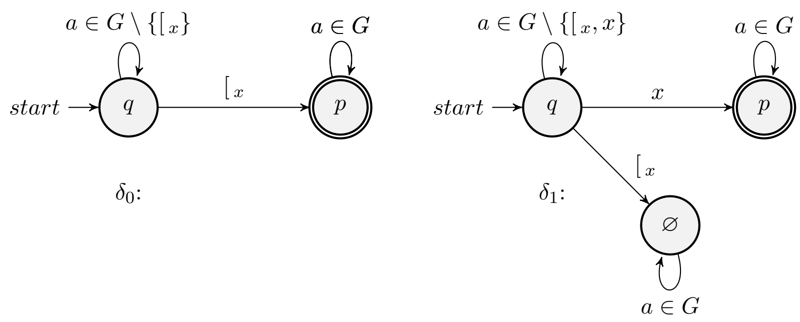
$$(\forall p, q \in V)(\forall x \in G)\ q \in \delta(p, x) \iff (p, q) \in E \ \wedge \ f((p, q)) = x$$

Potom pro libovolné $\beta \in V$ a $x \in X$ platí následující tvrzení:

$$x \triangleright_{def} \beta \iff L(\mathcal{R}_0(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{def}}) \neq \varnothing$$

$$\beta \triangleright_{call} x \iff L(\mathcal{R}_1(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{call}}) \neq \varnothing,$$

kde φ je kořen AST reprezentujícího α, $\mathcal{R}_0(\alpha) = (V, G, \delta, \varphi_{in}, \{\beta\})$ a $\mathcal{R}_1(\alpha) = (V, G, \delta, \beta, \{\varphi_{out}\})$. $\mathcal{R}(\alpha) = (V, G, \delta, \varphi_{in}, \{\varphi_{out}\})$ je NKA přijímající referenční jazyk daný regexem α. Potom jazyk přijímaný $\mathcal{R}_1(\alpha)$ (resp. $\mathcal{R}_0(\alpha)$) obsahuje podřetězce, které může přijat $\mathcal{R}(\alpha)$ s počátečním (resp. jediným koncovým) stavem β. $\mathcal{M}_{x,\triangleright_{def}}$ přijímá všechny řetězce, v nichž je alespoň jedna otevírací závorka definice proměnné x a $\mathcal{M}_{x,\triangleright_{call}}$ přijímá všechny řetězce, v nichž je alespoň jedna reference na x a před danou referencí neexistuje žádná definice na tuto proměnnou. Průnik jazyků $L(\mathcal{R}_0(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{def}})$ je prázdný jazyk, právě když v $\mathcal{R}(\alpha)$ z počátečního stavu do stavu β neexistuje posloupnost přechodů, pomocí níž automat přečte řetězec obsahující otevírací závorku pro definici proměnné x (toto odpovídá binární relaci $x \not\triangleright_{def} \beta$). Průnik jazyků $L(\mathcal{R}_1(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{call}})$ není prázdný jazyk, právě když v $\mathcal{R}(\alpha)$ z počátečního stavu do stavu β existuje posloupnost přechodů, pomocí níž automat přečte řetězec, v němž existuje alespoň jedna reference na x před nějakou definicí $(\ldots x \ldots [_x \ldots]_x)$, což odpovídá binární relaci $\beta \triangleright_{call} x$. [5]



Obrázek 3.1: Grafy přechodů automatů $\mathcal{M}_{x,\triangleright_{def}}$ a $\mathcal{M}_{x,\triangleright_{call}}$

Parametr *avd* nám neformálně říká, že automat přijímající jazyk daný α musí v nějakém kroku výpočtu mít vyhrazenou paměť alespoň pro $avd(\alpha)$ proměnných.

Věta 3.1. Algoritmus 4 spočítá pro regex $\alpha \in RV_{\Sigma,X}$ parametr *avd* v čase $\mathcal{O}(|X| \cdot |\alpha|^2)$. [5]

Důkaz předchozí věty je podrobně uveden v [5, s. 7]. V kroku 12 se dvakrát použije algoritmus pro skládání automatu [7, s. 54–56] a pro dva výsledné NKA $\mathcal{M}_1$ a $\mathcal{M}_2$ pro průnik odpovídajících jazyků je potřeba ověřit, jestli nejsou přijímané jazyky $L(\mathcal{M}_1)$ a $L(\mathcal{M}_2)$ prázdné (neexistuje žádná posloupnost přechodů z počáteční konfigurace do koncové).

3.2 Algoritmus `avdMemory`

V této sekci je popsán způsob, jak lze převést regex na ekvivalentní memory automat tak, že počet pamětí záleží pouze na parametru *avd* tohoto regexu. Uvedený algoritmus byl převzat z [5, kapitola 3].

Nechť $k = avd(\alpha)$, kde α je vstupní regex, a nechť je dána množina $\Gamma = \{x,]_x, [_x \mid x \in var(\alpha)\}$. Jelikož parametr *avd* může být menší než počet proměnných ve výrazu, výsledný automat bude fungovat tak, že jedna paměť je sdílena mezi více proměnnými.

Algoritmus 4: Výpočet parametru avd

Vstup : $\alpha \in RV_{\Sigma,X}$
Výstup: $avd(\alpha)$

Nechť φ je kořen AST reprezentujícího α, $G \leftarrow \Sigma \cup X \cup \{\varepsilon\} \cup \Gamma$
Sestroj graf $\mathcal{H}(\alpha) = (V, E, f)$
$\delta \; : \; v \; \times G \to \wp(V)$
$(\forall p, q \in V)(\forall a \in G) \; p \in \delta(q, a) \iff e = (q, p) \in E \; \wedge \; f(e) = a$
$avd \leftarrow 0$
for $\beta \in V$ **do**
  if $(\gamma, \beta) \in E \; \wedge \; f((\gamma, \beta)) = [_x$ **then**
    $avs(\beta) \leftarrow \varnothing$
    for $x \in X$ **do**
      $\mathcal{R}_0(\alpha) = (V, G, \delta, \varphi_{in}, \{\beta\})$
      $\mathcal{R}_1(\alpha) = (V, G, \delta, \beta, \{\varphi_{out}\})$
      if $L(\mathcal{R}_0(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{def}}) \neq \varnothing \; \wedge \; L(\mathcal{R}_1(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{call}}) \neq \varnothing$ **then**
        $avs(\beta) \leftarrow avs(\beta) \cup \{x\}$
    end
    if $|avs(\beta)| > avd$ **then** $avd \leftarrow |avs(\beta)|$
end
return avd

Množina stavů výsledného automatu Q je vytvořena z množiny vrcholů grafu $\mathcal{H}(\alpha)$ následovně:

- Každý stav $q \in Q$ je dvojicí $(v, \langle i_0, \dots, i_{k-1} \rangle)$, kde $v \in V(\mathcal{H}(\alpha))$ a $i_j \in var(\alpha) \cup \{\bot\}$ pro $\forall j \in \{0, \dots, k-1\}$. Posloupnost $i_0, \dots, i_{k-1}$ se nazývá *memory list*. Tato modifikace množiny stavů nám říká, že nachází-li se automat ve stavu q, j-tá paměť je vyhrazena pro proměnnou x, právě když $i_j = x$. Pokud je j-tá paměť volná, platí $i_j = \bot$.
- Počáteční stav automatu je $q_0 = (\varphi_{in}, \langle \bot, \dots, \bot \rangle)$, kde φ je kořen AST pro α.
- Množina koncových stavů $F \subseteq \{\varphi_{out}\} \times (var(\alpha) \cup \{\bot\})$ se skládá ze stavů, do nichž vede alespoň jeden přechod.

Buď $q = (v, \langle i_0, \dots, i_{k-1} \rangle), p = (w, \langle j_0, \dots, j_{k-1} \rangle) \in Q$. Přechodová funkce je definována následovně.

1. Pro libovolnou hranu $p \in \delta(q, t)$ (kde $t \in \Sigma \cup \{\varepsilon\} \cup \Gamma$) a proměnnou $x \in X$ platí:

$$\forall l \in \{0, \dots, k-1\} (w \not\triangleright_{call} x \implies j_l \neq x) \qquad (3.4)$$

Ve stavu p by neměla být vyhrazena paměť pro proměnnou x, protože ze stavu w lze přečíst pouze referenční slovo, v němž bude před každou následující referencí existovat nová definice této proměnné (plyne z definice relace $\triangleright_{call}$).

2. Pro $a \in \Sigma \cup \{\varepsilon\}$

$$p \in \delta(q, a) \iff (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, w)) = a \tag{3.5}$$

a pro každé $l \in \{0, \ldots, k-1\}$ $j_l = i_l$, pokud $w \triangleright_{call} i_l$ (jinak $j_l = \perp$ podle pravidla 1).

Pro všechny přechody, při nichž automat neprovádí operace s pamětí, seznam proměnných $i_0, \ldots, i_k$ se jenom překopíruje ze stavu q do p (příp. se uvolní odpovídající paměť dle pravidla 1).

3. Pro $x \in X$ a $\varphi \in \{[_x,]_x\}$

$$p \in \delta(q, \varepsilon) \iff (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, w)) = \varphi \ \wedge \ w \not\triangleright_{call} x \tag{3.6}$$

a pro každé $l \in \{0, \ldots, k-1\}$ $j_l = i_l$, pokud $w \triangleright_{call} i_l$ (jinak $j_l = \perp$ podle pravidla 1).

Pokud ze stavu w lze přečíst pouze referenční slovo, v němž bude před každou následující referencí existovat nová definice této proměnné, není potřeba ani vyhrazovat paměť pro x.

4. Pro $x \in X$

$$p \in \delta(q, \varepsilon) \iff (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, w)) = x \ \wedge \ (\nexists s) i_s = x \tag{3.7}$$

a pro každé $l \in \{0, \ldots, k-1\}$ $j_l = i_l$, pokud $w \triangleright_{call} i_l$ (jinak $j_l = \perp$ podle pravidla 1).

S x-přechodem ze stavu q lze zacházet jako s ε-přechodem, pokud v q není vyhrazena žádná paměť pro x.

5. Pro $x \in X$

$$\begin{gathered} p \in \delta(q, s) \iff (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, x)) = \varphi \ \wedge \ (\exists s) i_s = x \\ p \in \delta(q,]_s) \iff (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v,]_x)) = \varphi \ \wedge \ (\exists s) i_s = x \end{gathered} \tag{3.8}$$

a pro každé $l \in \{0, \ldots, k-1\}$ $j_l = i_l$, pokud $w \triangleright_{call} i_l$ (jinak $j_l = \perp$ podle pravidla 1). Automat ze stavu q (v tomto stavu s-tá paměť je vyhrazena pro x) čte referenci nebo „uzavírá“ paměť vyhrazenou pro x.

6. Pro $x \in X$

$$\begin{aligned} p \in \delta(q, [_p) \iff & (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, w)) = [_x \wedge \\ & (\exists p)(i_p = \bot \ \wedge \ j_p = x) \end{aligned} \tag{3.9}$$

$$\begin{aligned} p \in \delta(q, [_p) \iff & (v, w) \in E(\mathcal{H}(\alpha)) \ \wedge \ f((v, w)) = [_x \wedge \\ & (\exists p)(i_p = x \ \wedge \ j_p = x) \end{aligned} \tag{3.10}$$

a pro každé $l \in \{0, \ldots, k-1\}$ $j_l = i_l$, pokud $w \triangleright_{call} i_l$ (jinak $j_l = \bot$ podle pravidla 1).

Pokud ve stavu q je vyhrazena paměť pro x, automat při čtení definice ponechá *memory list* beze změn, jinak najde nějakou volnou paměť p a nastaví $s_p \leftarrow \mathtt{O}$ (tvrzení 3.9).

Výsledný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ je $\mu KA(k)$, kde $k = avd(\alpha)$. Příklad převodu regexu na memory automat pomocí uvedeného algoritmu lze najít v příloze A.2.

Algoritmus, který přijme na vstup AST pro regex α a vstupní slovo w, zkonstruuje ekvivalentní memory automat $\mathcal{M}$ pomocí výše popsaného algoritmu a nasimuluje výpočet $\mathcal{M}$ pro vstup w, se v této práci nazývá `avdMemory`.

Věta 3.2. Časová složitost `avdMemory` je $\mathcal{O}(|X|^k \cdot |w|^{k+1} \cdot |\alpha|^2)$, kde $k = avd(\alpha)$.

Důkaz. Konstrukce memory automatu se skládá ze tří části: výpočet avd, konstrukce grafu $\mathcal{H}(a)$ a vytvoření stavů a přechodů pomocí výše popsaných pravidel. Výpočet parametru avd spotřebuje čas $\mathcal{O}(|X| \cdot |\alpha|^2)$ (věta 3.1). Konstrukce grafu $\mathcal{H}(\alpha)$ spotřebuje čas $\mathcal{O}(|\alpha|)$ [5, lemma 3]. Pro každý vrchol z $V(\mathcal{H}(\alpha))$ se vytvoří $\mathcal{O}(|X|^k)$ stavů, kde $k = avd(\alpha)$, a pro každou hranu se do přechodové funkce δ přidá $\mathcal{O}(k)$ přechodů (pravidlo 6). Jelikož počet hran a vrcholů v $\mathcal{H}(\alpha)$ je $\mathcal{O}(|\alpha|)$ (plyne z definice grafu), celkem převod regexu na memory automat potrvá $\mathcal{O}(|X| \cdot |\alpha|^2 + |\alpha| \cdot (k + |X|^k)) = \mathcal{O}(|X|^k \cdot |\alpha|^2)$.

Simulace výsledného automatu je procházení grafem konfigurací. Celkem výpočet memory automatu $\mathcal{M}$ pro vstup w spotřebuje čas $\mathcal{O}(|Q| \cdot |w|^{k+1}) = \mathcal{O}(|\alpha| \cdot |X|^k \cdot |w|^{k+1})$ [5, lemma 3]. □

Kapitola **4**

Realizace

Pro implementaci byl autorem zvolen programovací jazyk C++. Bylo rozhodnuto realizovat tři algoritmy pro zpracování regexů: `simpleTM`, `simpleMemory` a `avdMemory`. V následujících sekcích jsou popsány implementace jednotlivých částí prostředku pro práci s regexy.

4.1 Implementace lexeru a parseru

Zdrojové soubory `lexer.h` a `lexer.cpp` obsahují realizaci lexikálního analyzátoru. Lexer vytváří pro parser tokeny, které jsou definovány pomocí výčtového typu `Token`. Jako promenné lexer rozeznává latinské kapitálky. Abeceda regexu je omezena na 27 malých písmen latinské abecedy. Pro jednoduchost se používají symboly `?` pro ε a `0` pro prázdný regulární výraz. Například `X{a*+?}X` je potom zápis regexu $x\{a^* + \varepsilon\}\ x$. Všimněte si, že se pro lexer používá objekt typu **istringstream**, do něhož nejprve je zapsán textový řetězec reprezentující vstupní regex.

Zavoláním metody `getToken()` dojde k načtení tokenu, jehož hodnota se uloží do proměnné `val`.

Parser dostává na vstup tokeny z lexeru (metoda `getNextToken()`) a vytváří vnitřní reprezentaci regexu. Realizaci parseru lze najít ve zdrojových souborech `parser.h` a `parser.cpp`. Překlad definovaný LL(1) atributovou gramatikou z sekce 2.1.5 je realizován pomocí tzv. *metody rekurzivního sestupu* s parametry. Například pro neterminál A' je vytvořena funkce `ParseARest` (viz obrázek 4.1), jejíž vstupní parametr `left` odpovídá dědičnému atributu *dnode*. Výstupní parametr odpovídá syntetizovanému atributu *snode*. Hodnoty obou dvou atributů jsou ukazatele na vytvořené abstraktní syntaktické stromy, které jsou realizovány pomocí základní třídy `NodeAST` a odvozených tříd. Metoda `ExpandError` je volána, pokud dojde k chybě při expanzi. Vznikne-li chyba při srovnání, metoda `MatchError` vypíše odpovídající chybovou zprávu na standardní výstup.

```
// A' -> + A | eps
unique_ptr<NodeAST> ParseARest(unique_ptr<NodeAST> left) {
    if (m_CurTok == tokenUnion) {
        getNextToken();
        auto right = move(ParseA());
        return make_unique<UnionAST>(move(left), move(right));
    } else if (m_CurTok == tokenEOF || m_CurTok == ')'
|| m_CurTok == '}') {
        return move(left);
    } else {
        ExpandError("A'", (Token) m_CurTok);
        return nullptr;
    }
}
```

Obrázek 4.1: Funkce z metody rekurzivního sestupu pro neterminál A'

Dále následuje popis jednotlivých tříd, které jsou definovány v `ast.h` a `ast.cpp`.

- Abstraktní třída `NodeAST` reprezentuje uzel syntaktického stromu. Tato třída obsahuje abstraktní metody `constructTM` a `constructAvdFA`, jež se používají při konstrukci TS a memory automatu (viz sekce 4.3–4.4).
- Třída `AtomAST` definuje uzel stromu reprezentující elementární regulární výraz. Jediným atributem je ASCII hodnota znaku (`m_Val`).
- Pomocí třídy `ConcatenationAST` (resp. `UnionAST`) je definován uzel reprezentující zřetězení (resp. sjednocení) dvou výrazu. Hodnoty atributů `m_LHS` a `m_RHS` obsahují ukazatele na uzly levého a pravého podstromu. Analogicky atribut `m_Expr` třídy `IterationAST` označuje uzel potomka.
- Od třídy `VarAST` jsou odvozeny třídy `DefinitionAST` (uzel reprezentující definici proměnné) a `BackRefAST` (uzel reprezentující zpětnou referenci). Metoda `getVar` vrátí hodnotu atributu `m_Var` (název proměnné). Atribut `m_Expr` třídy `DefinitionAST` obsahuje ukazatel na uzel podstromu.

4.2 Reprezentace vícepáskového TS

Způsob, jakým je automat reprezentován, byl převzat z [14].

Zdrojové soubory `tape.h`, `tape.cpp` obsahují realizaci pásky TS pomocí třídy `Tape`. Tato třída definuje atributy `m_Head` (pozice čtecí hlavy) a `m_Cells` (obsah pásky). Typ pohybu čtecí hlavy je definován pomocí výčtového typu `ShiftType` (názvy ve výčtu jsou `left, noShift` a `right`). Posuv čtecí hlavy se provede zavoláním `moveHead` se vstupním parametrem typu `ShiftType`. Metoda `readSymbol` vrátí znak, na něhož ukazuje čtecí hlava. Zápis znaku na pásku je realizován pomocí metody `writeSymbol(char)`.

`Automaton` (`automaton.h`) je šablonová abstraktní třída se šablonovým parametrem `T` (datový typ pro stav), ze které jsou odvozené třídy `NDTM` a `AvdFA`. Tato třída definuje následující atributy:

- `m_InitialState : T` je počáteční stav automatu,
- `m_FinalStates : set<T>` je množina koncových stavů,
- `m_CurState : T` je aktuální stav,
- `m_StateCnt : int` je počet stavů,
- `m_Input : set<char>` je vstupní abeceda.

Pro simulátor TS stavy automatu jsou reprezentovány celými čísly. Přidání nového stavu odpovídá inkrementaci proměnné `m_StateCnt`. Číslo nového stavu odpovídá hodnotě `m_StateCnt` před inkrementací.

Třída `NDTM` reprezentuje vícepáskový nedeterministický TS. Přechodová funkce automatu je reprezentována jako mapa, kde klíčem je pár (q, w) (q je stav automatu a w jsou symboly, na něž ukazují čtecí hlavy pásek). Prvkem mapy je vektor párů (q, o), kde o označuje operace prováděny s odpovídající páskou ($o = (c, shift)$, kde c je symbol, který se má zapsat, a $shift$ je typ pohybu hlavy). Taková reprezentace přechodové funkce byla zvolena z důvodu co největší podobnosti s formální definicí. Mezi atributy také patří `m_Tapes` (vektor pásek automatu) a `m_Blank` (blank symbol). Zavoláním metody `addTransition` se do přechodové funkce přidá přechod, který odpovídá vstupním parametrům.

Metoda `initialize(w : string)` nastaví počáteční konfiguraci pro řetězec `w` tak, že zavolá `loadTape`, která inicializuje obsah pracovních pásek hodnotou $B^{|w|+2}$, kde B je blank symbol TS, a nastaví pozice čtecích hlav na 1. Po volání předchozí metody obsah vstupní pásky bude odpovídat řetězci BwB.

Výpočet TS je realizován metodou `accepts()` (viz obrázek 4.2), která vrátí `true`, pokud existuje posloupnost přechodů z aktuální konfigurace do nějaké koncové. Nejprve je ověřeno, jestli se aktuální konfigurace už předtím ve výpočtu vyskytla (`hasCycle()`). Pokud ne, konfigurace pásek pro aktuální stav se uloží do paměti konfigurací (atribut `m_ConfigurationsMemory`). Pokud ano, je potřeba výpočet pro tuto větev ukončit a vrátit `false`, jinak se automat zacyklí (viz příklad 2.2). Po kontrole opakování konfigurací se pro každý možný přechod z aktuální konfigurace vytvoří hluboká kopie automatu (`clone()`), pro vytvořenou kopii se provede odpovídající přechod (`execTransition()`) a zavolá se na ni metoda `accepts()`. Pokud ani jedno z volání `accepts` nevrátí `true`, zjistí se, není-li aktuální konfigurace koncová.

```
bool accepts() {
    auto trans = m_Transitions[make_pair(m_CurState,
this->readSymbols())];
    if (this->checkCycle()) return false;
    for (int i = 0; i < trans.size(); i++) {
        auto tm = this->clone();
        tm->execTransition(trans[i]);
        if (tm->accepts()) return true;
    }
    return m_Tapes[0]->isEmpty() &&
m_FinalStates.find(m_CurState) != m_FinalStates.end();
}
```

Obrázek 4.2: Definice metody `accepts()` z `ndtm.cpp`

4.3 Implementace algoritmu `simpleTM`

Programovou realizaci prostředku pro práci s regexy (anglicky *regex matcher*) lze najít ve zdrojových souborech `matcher.h` a `mather.cpp`. Ve funkci `main` se vytvoří instance parseru a simulátoru TS s počátečním a koncovým stavem a zavolá se konstruktor třídy `Matcher` se dvěma parametry `parser` (ukazatel na parser) a `option = "0"` (volba algoritmu). Metoda `ParseA()` vrátí ukazatel na kořen AST pro vstupní regex (`m_Root`). Počet pásek automatu se nastaví na počet proměnných, které parser byl schopen rozpoznat, zvětšený o jednu. Přidání stavů a přechodů se do instance třídy `NDTM` provede zavoláním na ukazatel na `m_Root` virtuální metody `constructTM` (viz deklaraci v `ast.h`), která bude očekávat následující parametry:

- `tm` je ukazatel na TS,
- `tapes` je reference na `vector<bool>`, odpovídá tabulce $T[1, \ldots, k]$ v algoritmu 3,
- `memory` je reference na `map<char, int>`, odpovídá zobrazení *num*,
- `start` je počáteční stav pro uzel,
- `end` je koncový stav pro uzel.

Každá z tříd odvozených od `NodeAST` implementuje `constructTM` takovým způsobem, aby implementace této metody odpovídala funkci `buildTM` z algoritmu `simpleTM`.

Definice metody `match` se vstupním parametrem `w` se skládá z nastavení počáteční konfigurace simulátoru (pomocí `initialize(w)`) a simulace výpočtu metodou `accepts`.

4.4 Implementace algoritmů `simpleMemory` a `avdMemory`

Analogicky jako v předchozí sekci se z funkce main zavolá konstruktor `Matcher` s parametry `parser` a `option` (volba algoritmu: `"1"` pro `simpleMemory` a `"2"` pro `avdMemory`).

Vytvoří se instance třídy `AvdFA` (reprezentuje NKA, přijímající referenční jazyk vstupního regexu). Konstrukce daného automatu se provede zavoláním na ukazatel na `m_Root` virtuální metody `constructAvdFA` (viz deklaraci v `ast.h`), která bude očekávat následující parametry:

- `automaton` je ukazatel na NKA,
- `avd` je reference na `vector<int>` (vektor, do něhož se uloží počáteční stavy pro uzly reprezentující výraz uvnitř definice: pro tyto stavy je potřeba spočítat množinu aktivních proměnných při volbě algoritmu `avdMemory`),
- `in` je počáteční stav pro uzel,
- `out` je koncový stav pro uzel.

Výsledný μKA je reprezentován pomocí šablonové třídy `MemoryAutomaton`. Tato třída definuje následující atributy:

- `m_Tape : string` je vstupní páska,
- `m_Pos : int` je pozice čtecí hlavy,
- `m_Memory : vector<pair<bool, string>>` je paměť automatu (stav a obsah paměti),
- `m_CurState : T` je aktuální stav,
- `m_StateCnt : map<T, vector<pair<string, T>>>` reprezentuje přechodovou funkci automatu,
- `m_ConfigurationMemory : map<T, pair<int, vector<string>>>` je paměť konfigurací (pomocí této paměti lze zabránit vzniku cyklů v memory automatu).

Při volbě `"1"` se program zavoláním metody `simpleMemory` (se vstupním parametrem `vars`, který odpovídá množině proměnných regexu) na instanci třídy `AvdFA` vytvoří objekt `MemoryAutomaton` (stavy jsou opět reprezentovány datovým typem `int`).

Při volbě `"2"` program vypočte parametr *avd* regexu.

Pomocí metod `constructR0` a `constructR1` je implementován algoritmus pro skládání automatu (viz krok 12 algoritmu 4). Metoda `constructR0` se vstupními parametry `state` a `var` vrátí ukazatel na automat přijímající jazyk $L(\mathcal{R}_0(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{def}})$, kde $\mathcal{R}_0(\alpha)$ je NKA $\mathcal{R}$ přijímající referenční jazyk α s jediným koncovým stavem `state`, a `constructR1` vrátí ukazatel na automat přijímající $L(\mathcal{R}_1(\alpha)) \cap L(\mathcal{M}_{x,\triangleright_{call}})$, kde $\mathcal{R}_1(\alpha)$ je $\mathcal{R}$ s počátečním stavem `state`. Virtuální metoda `accepts` v třídě `AvdFA` vrací `true`, pokud automat přijímá neprázdný jazyk. Jedná se o prohledávání grafu konfigurací automatu pomocí algoritmu DFS.

Pomocí předchozích metod se vypočte mohutnost množiny aktivních proměnných pro každý stav z vektoru `avd`.

Parametr *avd* program předá jako vstupní parametr metodě `avdMemory`. Tato metoda implementuje algoritmus pro převod regexu na memory automat, kde počet pamětí je roven $avd(\alpha)$. Výstupem funkce je instance třídy `MemoryAutomaton` se šablonovým parametrem typu `MemoryState`, který reprezentuje stav memory automatu rozšířený o *memory list*.

Kapitola 5

Testování

Testování proběhlo ve virtuálním stroji s operačním systémem Ubuntu 19.10 v konfiguraci s 2 GB operační paměti a dvěma procesory. Hostitelský stroj měl šestijádrový procesor Intel Core i7-8750H @ 2.20 GHz a 16 GB RAM.

Spustitelný soubor `regexmatcher` se vytvoří příkazem `make compile`. Při kompilaci je použít kompilátor GNU C++ verze 9.2.1, který se spustí s následujícímu přepínači: `-std=c++14 -Wall -pedantic -Wno-long-long -O0 -ggdb`. Pro generování dokumentace ve formátu HTML zadejte příkaz `make doc`.

Program lze spustit z příkazové řádky, kde vstupní regex je druhý parametr. Vstupní řetězec je zapsán jako třetí argument spouštěného programu. Prvním parametrem je volba algoritmu. Například zadáním příkazu

```
./regexmatcher 0 X{a+b}X+? bbaa
```

program vytvoří simulátor TS přijímajícího $L(x\{a+b\}\ x+\varepsilon)$ pomocí algoritmu `simpleTM`, provede simulaci výpočtu pro vstupní slovo *bbaa* a vytiskne na standardní výstup `yes`, pokud vstupní řetězec odpovídá regexu (jinak vytiskne `no`).

5.1 Vytvoření testových souborů

Pro účely testování bylo vytvořeno šest testovacích sad (vstup a referenční výstup). Soubor <název_sady>.in má $2n$ řádků a obsahuje celkem n testů. Každý test se skládá ze dvou řádků: na prvním je zapsán regex, na druhém je vstupní slovo. <název_sady>.out se skládá z n řádků, kde na i-tém řádku je referenční řešení pro i-tý test. Následuje popis jednotlivých testovacích sad:

- První sada `simple` obsahuje 20 testů. Regexy z této sady jsou omezeny dvouprvkovou abecedou $\{a, b\}$ a jednoprvkovou množinou proměnných $\{x\}$.

- `simpleReg` obsahuje 20 testů. Regexy z této sady jsou omezeny stejným způsobem jako v sadě `simple`. Vstupní slovo v testech je většinou delší. Tyto testy mohou zpomalit hlavně simulátor TS nebo memory automatu kvůli většímu počtu volání funkce `accepts`.

- Sada s názvem `nVar` se skládá z 10 testů. Regexy z této sady jsou omezeny dvouprvkovou abecedou $\{a, b\}$, přičemž parametr *avd* daných regexů je maximálně 2. Pomocí těchto testů lze porovnat matcher založený na algoritmech `simpleMemory` (resp. `avdMemory`) s implementací `simpleTM`.

- Testovací soubor `nSigma.in` obsahuje 10 testů. Regexy z tohoto souboru jsou omezeny jednoprvkovou množinou proměnných $\{x\}$. Počet různých symbolů abecedy p v libovolném regexu je větší než 5. Takové regexy by mohly zpomalit konstrukci TS hlavně kvůli tzv. *ε-přechodům*, kde se do přechodové funkce přidá $\mathcal{O}(|\alpha| \cdot |p|^k)$ přechodů (k je počet pásek).

- Testovací sada s názvem `hard` se skládá z 12 testů. Regexy z tohoto souboru mají větší délku a obsahují velký počet proměnných a symbolů abecedy. Vstupní slova jsou také v těchto testech delší. Účelem dané sady je co nejvíc zatížit každou část prostředku pro zpracování regexů.

- Poslední sada `avd` je určena pro porovnání algoritmů `simpleMemory` a `avdMemory`. Regexy z této sady mají parametr *avd* menší nebo roven 2 a vstupní slova jsou mnohem delší než v ostatních sadách.

Pro spuštění testů, které ověří správnost implementací, zadejte `./test.sh <název_sady>`.

5.2 Porovnání s nástroji pro práci s regulárními výrazy

Pro porovnání byly zvoleny tyto nástroje:

- program *grep* verze 3.3, který používá syntaxi regexů odpovídající standardu POSIX ERE. Všimněte si, že tato utilita nepodporuje ani celou množinu semiregexů (konkrétně semiregexy, v nichž se zpětná reference na i-tou závorku nemůže nacházet před i-tou uzavírací závorkou). Navíc existuje omezení na počet číslovaných skupin zachycení (nemůže být větší než 9).

- *Perl* verze 5.28.1, kde regulární výrazy odpovídají standardu PCRE. Na regexy v jazyce Perl nejsou kladena omezení na počet proměnných. Perl však nepodporuje například regexy, v nichž se opakují závorkové skupiny se stejným jménem (např. `(?<A>a*)(?<A>b*)\g{A}`). Problém také tvoří

	`simpleTM`	`simpleMemory`	`avdMemory`	`grep`	`Perl`
`simple`	0,141	0,063	0,098	0,031	0,036
`simpleReg`	6,348	0,568	0,69	–*	0,014
`nVar`	–**	2,137	45,733	–*	–*
`nSigma`	1,687	0,11	0,273	0,017	0,023
`avd`	–**	1,677	1,572	–*	–*
`hard`	–**	1,831	16,392	–*	–*

Tabulka 5.1: Naměřené časy pro implementace autora a konkurenční implementace pro jednotlivé sady v sekundách

nedefinované reference na proměnné (např. řetězec `aa` neodpovídá regexu `((?<A>a)|(?<B>b))\g{A}\g{B}`).

Pro testovací sady je vytvořen vstupní soubor, v němž vstupní regexy odpovídají syntaxi použité v nástroji, ale mají tentýž sémanticky význam jako regexy v původním vstupním souboru. Pro některé regexy nebylo možné najít ekvivalentní výraz v požadované notaci.

Pro měření času je použít příkaz `time`. Pomocí skriptu `testTime.sh` (nebo příkazem `make testTime <název_sady>`) lze porovnat čas běhu vytvořeného prostředku pro práci s regexy s konkurenčními nástroji pro danou sadu. Naměřené časy (`real time`) pro jednotlivé testovací sady a nástroje pro práci s regexy jsou zobrazeny v tabulce 5.1.

5.3 Zhodnocení výsledků testování

Implementace autora jsou výrazně pomalejší než implementace v jazyce Perl pro testovací sadu `simpleReg`, kde vstupní řetězec je mnohem delší než regex. Je toto ovlivněno hlavně neefektivním sekvenčním algoritmem pro simulaci vícepáskového TS a memory automatu. Oproti programu *grep* implementace `simpleTM`, `simpleMemory` a `avdMemory` podporují například regexy s definicemi a referencemi v iteraci (například `(X{a+b*}X)*` z testovací sady `simpleReg`). Z tabulky lze z jistotou tvrdit, že obě konkurenční implementace kladou poměrně striktní omezení na zpětné odkazy a zachytávající skupiny.

Co se týče porovnání implementací autora, `simpleMemory` je efektivnější pro většinu testovacích sad kromě `avd`. Pokud na vstupu je delší řetězec a kratší regex s parametrem *avd* menším než 3 (což odpovídá specifikaci testovací sady `avd`), algoritmus `avdMemory` může být efektivnější (i když při konstrukci je potřeba spočítat *avd* regexu a výsledný memory automat má větší

*pro některé regexy neexistuje ekvivalent v daném nástroji (resp. ekvivalentní výraz je mnohem složitější)

**testy trvají řádově desítky minut

počet přechodů a stavů). Naopak algoritmus `simpleTM` se ve srovnání s ostatními implementacemi ukázal mnohem pomalejší: testy z sad `nVar`, `hard` a `avd` mohou trvat řádově desítky minut.

Závěr

Cílem teoretické části této práce bylo zformulovat problém zpracování regexů a podrobně popsat algoritmy pro zpracování těchto výrazů. Mezi přínosy práce lze řadit důkaz věty o NP-úplnosti problému založený na pojmu referenčního slova.

Praktická část této práce si kladla za cíl implementovat a otestovat jeden z algoritmů. Výsledkem práce je funkční implementace dvou algoritmů založených na konstrukci memory automatu (`simpleMemory` a `avdMemory`) a jednoho na konstrukci TS (`simpleTM`). Z výsledků měření lze považovat algoritmus `simpleMemory` za efektivnější. Oproti konkurenčním implementacím vypracovaná konzolová aplikace neklade striktní omezení na vstupní regex. Autorem implementovaný prostředek pro zpracování regexů se však ve srovnání s existujícími nástroji pro práci s regulárními výrazy ukázal méně výkonný na třech z šesti vytvořených testovacích sad.

V budoucnosti by bylo zajímavé paralelizovat simulaci výpočtu TS a memory automatu, která z výsledků testování nejvíc zpomalovala běh celé implementace. Větší pozornost by mohla být věnována také parametrizovanému problému zpracování regexů.

Literatura

[1] *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008). IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*. IEEE and The Open Group, 2018, ISBN 978-1-504-44963-2, 228–243 s. Dostupné z: `https://pubs.opengroup.org/onlinepubs/9699919799/`

[2] AHO, A. V.: Algorithms for Finding Patterns in Strings. In *Algorithms and Complexity*, editace J. van LEEUWEN, Handbook of Theoretical Computer Science [online], Elsevier, 1990, ISBN 978-0-444-88071-0, s. 255–300, [cit. 10. 1. 2020]. Dostupné z: `http://www.sciencedirect.com/science/article/pii/B9780444880710500102`

[3] SCHMID, M. L.: Inside the Class of REGEX Languages. In *Developments in Language Theory*, editace H.-C. YEN; O. H. IBARRA, International Conference on Developments in Language Theory [online], Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31653-1, s. 73–84, [cit. 22. 2. 2020]. Dostupné z: `https://link.springer.com/chapter/10.1007/978-3-642-31653-1_8`

[4] SCHMID, M. L.: Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing. In *Developments in Language Theory, International Conference on Developments in Language Theory [online]*, ročník 249, editace A. M. SHUR; M. V. VOLKOV, Springer International Publishing, 2014, ISBN 978-3-319-09698-8, s. 142–153, [cit. 22. 2. 2020]. Dostupné z: `https://link.springer.com/chapter/10.1007/978-3-319-09698-8_13`

[5] SCHMID, M. L.: Regular Expressions with Backreferences: Polynomial-Time Matching Techniques. [online], 2019, [cit. 12. 1. 2020], `1903.05896`. Dostupné z: `http://arxiv.org/abs/1903.05896`

[6] HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D.: *Introduction to Automata Theory, Languages, and Computation*. Boston: Addison-Wesley, třetí vydání, 2006, ISBN 978-0-321-45536-9.

[7] ŠESTÁKOVÁ, E.: *AUTOMATY A GRAMATIKY. Sbírka řešených příkladů*. Praha: Česká technika – nakladatelství ČVUT, 2017, ISBN 978-80-01-06306-4.

[8] AHO, A. V.; LAM, M. S.; SETHI, R.; aj.: *Compilers: Trinciples, Techniques, and Tools*. Boston: Pearson / Addison-Wesley, druhé vydání, 2007, ISBN 978-0-321-48681-3.

[9] HAZEL, P.: pcrepattern man page [online]. University of Cambridge Computing Service, October 2016, [cit. 23. 5. 2020]. Dostupné z: `https://www.pcre.org/original/doc/html/pcrepattern.html`

[10] CÂMPEANU, C.; SALOMAA, K.; YU, S.: A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, ročník 14, č. 06, 2003: s. 1007–1018, ISSN 0129-0541.

[11] BERGLUND, M.; van der MERWE, B.: Regular Expressions with Backreferences Re-examined. In *Prague Stringology Conference 2017*, editace J. HOLUB; J. ŽĎÁREK, Prague: CTU in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2017, ISBN 978-80-01-06193-0, s. 30–41.

[12] EGGAN, L. C.: Transition graphs and the star-height of regular events. *Michigan Mathematical Journal*, ročník 10, č. 4, 1963: s. 385–397, ISSN 0026-2285. Dostupné z: `https://doi.org/10.1307/mmj/1028998975`

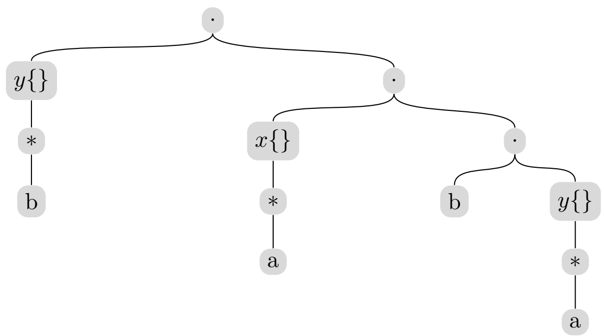
[13] FREYDENBERGER, D. D.; SCHMID, M. L.: Deterministic regular expressions with back-references. *Journal of Computer and System Sciences*, ročník 105, 2019: s. 1–39, ISSN 0022-0000. Dostupné z: `https://www.sciencedirect.com/science/article/pii/S0022000018301818`

[14] Multitape Nondeterministic Turing Machine simulator [online]. GeeksforGeeks, Aug 2019, [cit. 27. 4. 2020]. Dostupné z: `https://www.geeksforgeeks.org/multitape-nondeterministic-turing-machine-simulator`

Příloha A

Ukázka fungování algoritmů pro zpracování regexů

Nechť $\Sigma = \{a, b\}$ je abeceda a $X = \{x, y\}$ je množina proměnných. Regex $\alpha = y\{b^*\}\ x\{a^*\}\ b\ x$ je validní řetězec z množiny $RV_{\Sigma,X}$. Abstraktní syntaktický strom pro α je zobrazen na obrázku A.1.



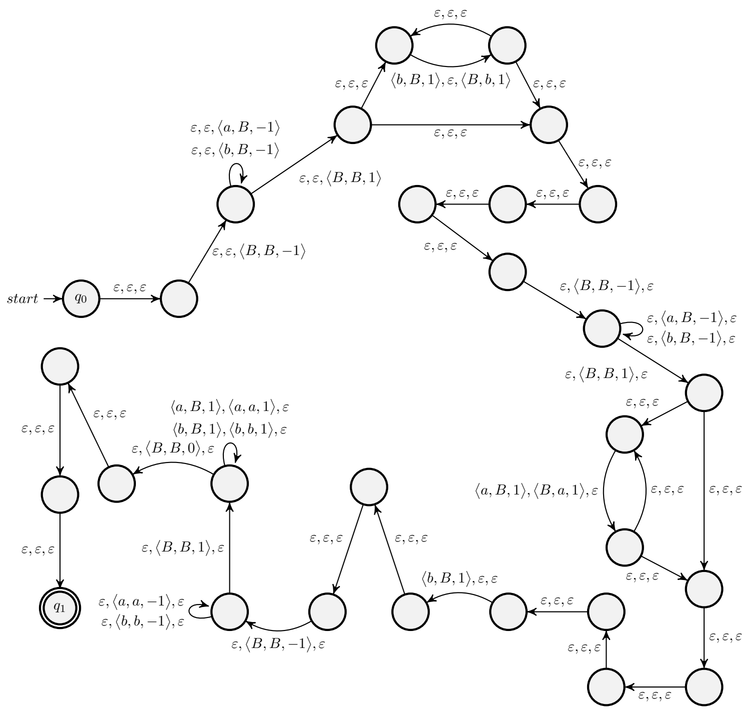
Obrázek A.1: AST pro regex $y\{b^*\}\ x\{a^*\}\ b\ x$

A.1 Ukázka převodu regexu na vícepáskový TS

Počet proměnných ve výrazu je roven 2. Zobrazení *num* lze definovat například takto: $num(x) := 1$; $num(y) := 2$, což znamená, že první (resp. druhá) páska je vyhrazena pro proměnnou x (resp. y). Výstupní $TS(3)$ je sedmici

$$(Q, \{a, b, B\}, B, \{a, b\}, \delta, q_0, \{q_1\}),$$

kde Q je množina stavu a δ je znázorněna pomocí grafu přechodů na obrázku A.2. Pro přehlednost některé hrany jsou označeny symboly ε. Například hrana, která je označená řetězcem $\langle a, B, 1\rangle, \varepsilon, \langle a, a, 1\rangle$, znamená přechod, při kterém automat přečte a zapíše tentýž symbol na první pásku a čtecí hlava této pásky zůstane na místě (konfigurace první pásky se nezmění).



Obrázek A.2: Přechodová funkce $TS(3)$ přijímajícího jazyk $L(y\{b^*\}\ x\{a^*\}\ b\ x)$

A.2 Ukázka převodu regexu na memory automat

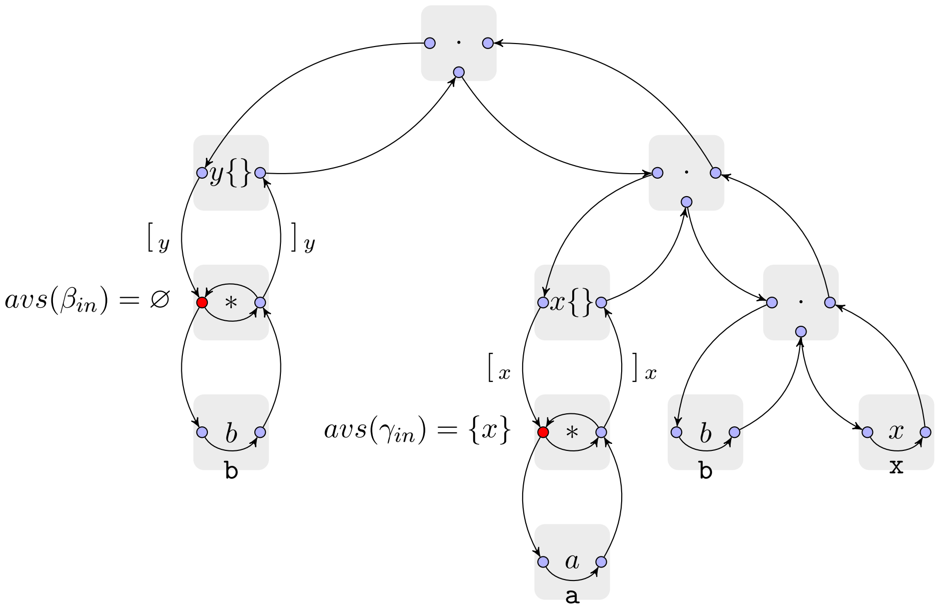
Nejprve sestrojme graf $\mathcal{H}(\alpha)$ podle definice (viz obrázek A.3).

- Algoritmus `simpleMemory`. Zobrazení num lze definovat například takto: $num(x) := 1;\ num(y) := 2$, což znamená, že první (resp. druhá) paměť automatu je vyhrazena pro proměnnou x (resp. y). Přechodová funkce $\mu KA(2)\ \mathcal{M} = (V(\mathcal{H}(\alpha)), \Sigma, \delta, \varphi_{in}, \{\varphi_{out}\})$ přijímajícího hodnotu daného regexu je zobrazena na obrázku A.4.

- Algoritmus `avdMemory`. Parametr avd je roven

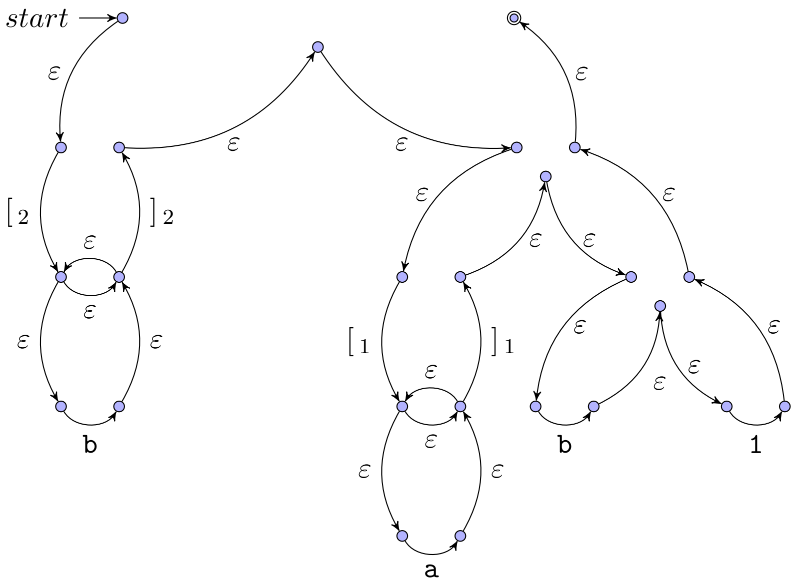
  $$max\{|avd(\gamma_{in})|, |avd(\beta_{in})|\} = 1.$$

  Pro proměnnou y automat nepoužije žádnou paměť (pro každý vrchol $v \in V(\mathcal{H}(\alpha))$ platí $v \not\triangleright_{call} y$). Při konstrukci memory automatu pro hrany ohodnocené $[_y$ a $]_y$ algoritmus přidá ε-přechody (podle pravidla 3 na s. 43). Výsledný $\mu KA(1)\ \mathcal{M} = (Q', \Sigma, \delta', (\varphi_{in}, \langle\perp\rangle), \{(\varphi_{out}, \langle\perp\rangle)\})$

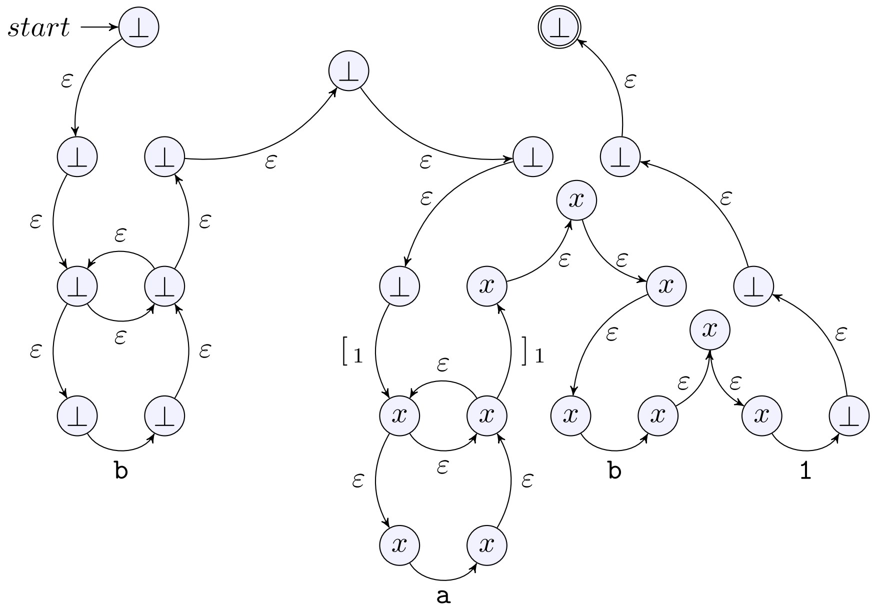


Obrázek A.3: Graf $\mathcal{H}(y\{b^*\}\ x\{a^*\}\ b\ x)$

přijímá hodnotu regexu α (δ' je znázorněna pomocí grafu přechodové funkce na obrázku A.5). Q' je množina stavů, kde každý stav obsahuje *memory list*.



Obrázek A.4: Přechodová funkce $\mu KA(2)$ přijímajícího $L(y\{b^*\}\ x\{a^*\}\ b\ x)$



Obrázek A.5: Přechodová funkce $\mu KA(1)$ přijímajícího $L(y\{b^*\}\ x\{a^*\}\ b\ x)$

Příloha B

Seznam použitých zkratek

AST Abstract Syntax Tree

DFS Algoritmus *Depth-first search*

LOTS Lineárně omezený Turingův stroj

μ**KA** Memory automat

NKA Nedeterministický konečný automat

TS Turingův stroj

PCRE Perl Compatible Regular Expressions

POSIX BRE POSIX Basic Regular Expressions

POSIX ERE POSIX Extended Regular Expressions

Příloha **C**

Obsah přiloženého CD

Obsah CD je dostupný také na `https://gitlab.fit.cvut.cz/zaporole/regex_matcher`.

- `BP_Zaporozhchenko_Oleksandr_2020.pdf` ... text práce ve formátu PDF
- `README.md` ... stručný popis obsahu CD
- `src`
  - `Doxyfile` ... konfigurační soubor nástroje Doxygen
  - `impl` ... zdrojové kódy implementace
  - `Makefile`
  - `test.sh` ... skript určený pro testování správnosti implementací
  - `testGrep.sh` ... skript určený pro spuštění testů v programu grep
  - `testPerl.sh` ... skript určený pro spuštění testů v jazyce Perl
  - `testTime.sh` ... skript určený pro měření času běhu
  - `tests` ... datové sady použité pro testování
  - `thesis` ... zdrojová forma práce ve formátu LaTeX