VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NOVÉ STRUKTURY A OPERACE V MATEMATICKÉ INFORMATICE

NEW STRUCTURES AND OPERATIONS IN MATHEMATICAL INFORMATICS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **RICHARD BUREŠ**
AUTHOR

VEDOUCÍ PRÁCE **Prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2018

Abstrakt
TODO abs

Abstract
TODO abs

Klíčová slova
TODO key

Keywords
TODO key

Citace
BUREŠ, Richard. *Nové struktury a operace v matematické informatice*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. RNDr. Alexander Meduna, CSc.

Nové struktury a operace v matematické informatice

Prohlášení

TODO

.....
Richard Bureš
6. května 2018

Poděkování

TODO

Obsah

Kapitola 1

Úvod

TODO definice jak se operuje nad množinami jazyků Několik vlastností, které budeme pozorovat pokud bude možno. Délka Nejdelšího řetězce v jazyku(dále jen BS_L), nebo v rodině(dále jen BS_F), délka nejkratšího řetězce v jazyku(dále jen SS_L), případně v rodině(dále jen SS_F). Velikost nejdelšího jazyka v rodině(dále jen BL), velikost nejkratšího jazyka v rodině(dále jen SL) a nakonec velikost samotné rodiny(dále jen FS).

Kapitola 2

Operace

V této kapitole si představíme existující a definujeme si nové operace nad různými množinami(rodinami) jazyků. Operace si ukážeme, nad nimi zobrazíme příklady a řekneme si jejich známé vlastnosti. Na příkladech existujících a známých operací budeme poté stavět při představování operací nových. Na vlastnosti složitějších operací, jenž bude potřeba dokazovat se podíváme v další kapitole.

2.1 Sjednocení (Union)

Union neboli sjednocení je známá operace kdy sjednocujeme dva jazyky A a B a vzniká nám jazyk C obsahující prvky jazyka A i B. (viz. 2.1)

$$Union(L_1, L_2) = \{w | w \in L_1 \vee w \in L_2\} \tag{2.1}$$

2.1.1 Vlastnosti

Operace uzavřena nad Regulárními a bezkontextovými jazyky. Tedy pokud operaci použijeme nad dvěma regulárními jazyky, výsledkem bude regulární jazyk a obdobně je tomu tak s bezkontextovými. Použijeme-li operaci nad rodinou regulárních jazyků, vznikne nám jiná, větší rodina regulárních jazyků.

Není-li žádný z jazyků v rodině podmnožinou jazyka v téže rodině, tak velikost vzniklé rodiny je dána rovnicí 2.2 (čteme n plus jedna, nad dvěma), kde n je délka původní rodiny. Pokud rodina obsahuje jazyky jenž jsou podmnožinou jazyků v téže rodině, je délka vzniklého jazyka na tomto faktu závislá a proto obecně víme pouze to, že bude menší než hodnota daná rovnicí 2.2. Je-li však mezi jazyky v rodině taková vazba, že provedení operace *Union* nad libovolnými dvěma jazyky rodiny nám vytvoří jazyk patřící do této rodiny, víme že vzniklá rodina jazyků bude totožná s rodinou nad kterou jsme tuto operaci používali. (Viz. podkapitola 2.1.2) Tento vztah budeme nazývat, že je rodina tranzitivně uzavřena nad operací.

$$\binom{n+1}{2} \tag{2.2}$$

2.1.2 Příklad

Opakované použití operace Union s prázdným jazykem:
Mějme rodinu jazyků R_1

$$R_1 = \{L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, L_5 : \{\}\}$$

Použijme nad touto rodinou sjednocení: $R_2 = Union(R_1)$

$$\begin{aligned} R_2 = \{&L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\ &L_5 : \{\}, L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\ &L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\ &L_{11} : \{0, 1, \epsilon\} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{aligned}$$

A použijme-li sjednocení ještě jednou, získáme tranzitivně uzavřenou rodinu:
$R_3 = Union(R_2)$

$$\begin{aligned} R_3 = \{&L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\ &L_5 : \{\}, L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\ &L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\ &L_{11} : \{0, 1, \epsilon\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\ &L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\ &\} \end{aligned}$$

Obdobný výsledek by nastal u jakékoliv konečné rodiny jazyků. Přesněji řečeno, obsahuje li rodina jazyků k jazyků, tak nejpozději při $k - 1$ iteraci se dostaneme k tranzitivně uzavřené rodině.

2.2 Průnik (Intersection)

Intersection neboli průnik je opět další známá operace na které si ukážeme jaké má vlastnosti nad rodinami jazyků. Průnik jazyků L_1 a L_2 nám dává jazyk L_3, jenž obsahuje pouze přítomné v jazyce L_1, a zároveň jazyce L_2. (Viz. 2.3)

$$Intersection(L_1, L_2) = \{w | w \in L_1 \wedge w \in L_2\} \tag{2.3}$$

2.2.1 Vlastnosti

Operace uzavřena nad regulárnímy jazyky avšak nikoliv nad bezkontextovými. Tedy oproti Union, pokud použijeme operaci Intersection nad dvěma jazyky jež jsou bezkontextové, nemůžeme si být jisti, že výsledkem bude jazyk bezkontextový.

Při použití nad rodinou jazyků, bude opět vzniklá rodina větší, za předpokladu, že původní rodina nebyla uzavřená.

2.2.2 Příklad

Na příkladu si ukážeme opakované použití operace Intersection až do vodu kdy se dostaneme do stavu kdy platí, že je operace nad rodinou tranzitivně uzavřená, což bychom si mohly dokázat tak, že operaci použijeme znovu.

Opakované použití operace Intersection

Mějme rodinu jazyků R_1

$$\begin{aligned} R_1 = \{ & \\ & L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\}, \\ \} & \end{aligned}$$

Použitím operace Intersection získáme R_2: $R_2 = Intersection(R_1)$

$$\begin{aligned} R_2 = \{ & \\ & L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\}, \\ & L_2 : \{1, 00\}, L_2 : \{0, 00\}, L_3 : \{00, 11\}, \\ \} & \end{aligned}$$

A dalším použitím Intersection se dostáváme k tranzitivně uzavřené rodině R_3

$$\begin{aligned} R_3 = \{ & \\ & L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\}, \\ & L_4 : \{1, 00\}, L_5 : \{0, 00\}, L_6 : \{00, 11\}, L_7 : \{00\} \\ \} & \end{aligned}$$

2.3 Doplněk (Complement)

Doplněk jazyka si můžeme definovat následujícím příkladem: Představme si, že máme jazyk L_1 patřící do abecedy Σ. Doplněk jazyka L_1, jsou všechny řetězce patřící do množiny řetězců Σ^* a zároveň nepatřící do jazyka L_2

Čistě pro představu si můžeme říci, že $Complement(L_1) = \Sigma^* - L_1$.

2.3.1 Vlastnosti

Doplněk jazyka je operace uzavřená nad jazyky regulárními a neuzavřená nad jazyky bezkontextovými.

Také je zcela zřejmé, že $Complement(Complement(L_1)) = L_1$.

2.4 Rozdíl (Difference)

Rozdíl dvou jazyků je dosti podobný rozdílu dvou množin, tedy pokud $L_3 = Difference(L_1, L_2)$, tak L_3 obsahuje všechny řetězce patřící do jazyka L_1 a zároveň nepatřící do jazyka L_2, což si můžeme znázornit rovnicí 2.4. Mnohem výstižněji to však můžeme popsat rovnicí 2.5

$$Difference(L_1, L_2) = w | w \in L_1 \wedge w \notin L_2 \tag{2.4}$$

$$Difference(L_1, L_2) = Intersection(L_1, Complement(L_2)) \tag{2.5}$$

2.4.1 Vlastnosti

Z rovnice 2.5 nám jasně plyne, že rozdíl je uzavřen nad regulárními a neuzavřen nad bezkontextovými jazyky, neb průnik i doplněk jsou oba uzavřené nad regulárními a neuzavřené nad bezkontextovými.

2.5 Rozdílné sjednocení (Different Union)

Tato operace vychází z operace sjednocení, avšak s tím rozdílem, že povoluje pouze sjednocení nestejných jazyků. (Viz. 2.6)

$$DifferentUnion(L_1, L_2) = \{w | (w \in L_1 \vee w \in L_2) \wedge L_1 \neq L_2\} \tag{2.6}$$

2.5.1 Vlastnosti

Operace je uzavřena nad regulárními, ne však nad bezkontextovými jazyky.

Theorem 1 (Uzavřenost nad regulárními jazyky)**.** *Použití operace DifferentUnion nad libovolnými dvěma regulárními jazyky K a L generuje opět regulární jazyk.*

Důkaz 1. Nejprve musíme zjistit, jak porovnat dva jazyky K a L, což provedeme následovně
$K \neq L \iff K - L \neq \varnothing \wedge L - K \neq \varnothing$
Pravou stranu si dosadíme do rovnice 2.6 a jelikož víme, že operace Sjednocení i Rozdíl jsou uzavřeny nad rekulárními jazyky, tedy operace DifferentUnion je uzavřena nad regulárnímy jazyky. □

Theorem 2 (Uzavřenost nad Bezkontextovými jazyky jazyky)**.** *Předpokládejme že použití operace DifferentUnion nad libovolnými dvěma bezkontextovými jazyky K a L generuje opět bezkontextový jazyk.*

Důkaz 2. Postupujeme stejně jako u důkazu 1, avšak zjišťujeme, že operace Rozdíl není uzavřena nad bezkontextovými jazyky. Tedy ani operace DifferentUnion nemůže být nad těmito jazyky uzavřena, neb nemůžeme tvrdit, že jsme schopni porovnat libovolné dva bezkontextové jazyky a rozhodnout o jejich rovnosti. □

Při použití nad rodinou jazyků nám vzniká opačný efekt než kupříkladu u sjednocení a to, že může vzniknout rodina menší než rodina nad kterou jsme operaci aplikovali. Velikost rodiny zde opět záleží na tom, zda-li rodina obsahuje jazyk jenž je podmnožinou jiného jazyka. Při opakovaném použití se však vždy dostaneme do stavu kdy se rodina začne zmenšovat do bodu kdy obsahuje jeden jazyk a nakonec tedy je tato rodina prázdná. Tato vlastnost platí i pro nekonečně velké rodiny jazyků, u nichž však k nule nikdy nedojdeme a rodina se nám zmenšuje a zároveň zůstává nekonečná. (Různě velká nekonečna obdobně jako množina celých čísel je menší nekonečno než množina čísel reálných.)

Tento efekt nastává proto, že pokud se nemůže jazyk sjednotit sám se sebou a ani se svou podmnožinou, nemá možnost být ve výsledku další rodiny a tedy je "vyloučen".

Další vlastností které je dobré si všimnout, je že jazyky v rodině jazyků se vždy zvětšují a to proto, že se do výsledku dostanou pouze jazyky jenž jsou sjednocením dvou jiných, nebo jazyky jež jsou nadmnožinou jazyka jiného.

2.5.2 Příklad

Na příkladu si můžeme všimnout, kolísání velikosti rodin jazyků. Po prvním použití je vidět, že se velikost výsledné rodiny oproti předchozí zvětší a to díky jazyku L_5 jenž je podmnožinou všech jazyků. Následně se velikost rodin začíná zmenšovat až nakonec dojdeme do stavu, kdy je rodina jazyků prázdná.

Iterativně použitá operace DifferentUnion nad rodinou jazyků
Mějme rodinu jazyků R_1 a postupně aplikujme operaci DifferentUnion, dokud se nedostaneme k prázdné rodině jazyků

$$
\begin{aligned}
R_1 &= \{L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, L_5 : \{\}\} \\
R_2 &= \{ \\
&\quad L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\
&\quad L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\
&\quad L_{11} : \{0, 1, \epsilon\} \\
&\} \\
R_3 &= \{ \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\
&\quad L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\
&\quad L_{11} : \{0, 1, \epsilon\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\
&\quad L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\
&\quad \} \\
R_4 &= \{ \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\
&\quad L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\
&\quad \} \\
R_5 &= \{L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}\} \\
R_6 &= \{\}
\end{aligned}
$$

2.6 Operace "rozdílné"(Operation Diferent)

Operation Diferent, nebo-li rozdílné (neplést s Diference, rozdíl), tato operace přijímá libovolné dva jazyky a provádí nad nimi operaci kterou nejlépe definuje 2.7, pokud bychom ji chtěli definovat do podrobna, byla by popsána rovnicí 2.8

$$Different(L_1, L_2) = Union(L_1, L_2) - Intersection(L_1, L_2) \tag{2.7}$$

$$Different(L_1, L_2) = \{w | (w \in L_1 \wedge w \notin L_2) \vee (w \notin L_1 \wedge w \in L_2)\} \tag{2.8}$$

2.6.1 Vlastnosti

Vzhledem k faktu, že tato operace jde rozložit na několik jednodušších operací, jsme tak schopni odvodit i chování této operace. Vzhledem k tomu, že všechny operace uvedené v 2.7 jsou uzavřené nad regulárními jazyky, můžeme si říci, že celá operace je nad regulárními jazyky uzavřená. Stejným způsobem víme, že tato operace není uzavřená nad bezkontextovými jazyky.

Operace při opakovaném volání nevykazuje žádné speciální vlastnosti, vyjma toho, že jakékoliv použití této operace nad rodinou jazyků nám zaručí existenci prázdného jazyku. Existence prázdného jazyku nám dále zaručí, že v dalším výsledku nám nezmizí žádný jazyk a tedy je výsledná rodina vždy stejná, nebo větší.Po určitém počtu iterací se nakonec dostáváme k rodině, která je nad touto operací tranzitivně uzavřená

2.7 Unikátní konkatenace (Unique Concatenation)

Unique Concatenation, nebo-li Výlučná konkatenace, je operace jenž se chová jako konkatenace dvou jazyků, až na ten fakt, že nepřijímá takové řetězce z K a L, jejichž konkatenace by patřila do K nebo L. Pro zjednodušení si určeme konkatenaci, tak, že konkatenace je operace popsatelná rovnicí 2.9. Unikátní konkatenace poté je definována rovnicí 2.10. Ve zkratce výsledek unikátní konkatenace tytéž řetězce jako konkatenace normální, s výjimkou, že neobsahuje řetězce které již byly obsaženy v jazicích jenž jsme konkatenovali, což je nejlépe popsáno rovnicí 2.11.

$$Concatenation(L_1, L_2) = \{w | w = xy; y \in L_1 \wedge y \notin L_2\} \tag{2.9}$$

$$UniqueConc(L_1, L_2) = \{w | w = xy; y \in L_1 \wedge y \notin L_2 \wedge xy \notin L_1 \wedge xy \notin L_2\} \tag{2.10}$$

$$UniqueConc(L_1, L_2) = Concatenation(L_1, L_2) - L_1 - L_2 \tag{2.11}$$

2.7.1 Vlastnosti

Díky smutné neunikátnosti této operace jsme schopni z jejího vyjádření 2.11 vyvodit, že bude uzavřená nad regulárními a neuzavřená nad bez kontextovými jazyky.

Rodina vzniklá touto operací je velikosti n^2 nebo $(n-1)^2 + 1$ pokud původní rodina obsahovala prázdný jazyk.

2.8 Prefixes

Prefixes je operace, jejíž aplikace na K vytváří L, kde L obsahuje všechny řetězce, které jsou prefixy všech řetězců K. (Viz. 2.12)

$$Prefixes(L) = \{w | x = wy; x \in L \wedge y \in \Sigma_L\} \tag{2.12}$$

2.8.1 Vlastnosti

Theorem 3 (Uzavřenost nad regulárními jazyky)**.** *Použití operace Prefixes nad libovolným regulárním jazykem L generuje opět regulární jazyk.*

Důkaz 3. Mějme konečný automat $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ pokud je operace Prefixes uzavřená nad regulárními jazyky, je tedy uzavřená nad konečnými automaty a tedy jsme schopni vytvořit konečný automat N, který bude přijímat všechny prefixy jazyka $L(M)$.

Automat N vytvoříme tak, že vezmeme automat M a všechny stavy, které které nejsou neukončující označíme jako konečné.

Automat N je vždy Konečný Automat a přijímá všechny prefixy jazyka L(M), což potvrzuje, že je operace nad regulárními jazyky uzavřená. □

2.8.2 Příklad

Nechť existuje jazyk $L = \{123, 456, 101\}$, Aplikací operace Prefixes můžeme tedy vytvořit jazyk $K = Prefixes(L) = \{'', 1, 12, 123, 4, 45, 456, 10, 101\}$. Zde je třeba si všimnout faktu že řetězec "123"je prefixem řetězce "123", neboť sufixem je ”

2.9 Shuffle

Operaci shuffle můžeme definovat jako všechny kombinace které vzniknou pomícháním znaků z řetězců u a v se zachováním pořadí znaků tak jak byly v původním řetězci. V podstatě bychom tuto operaci mohli popsat jako prolnutí dvou řetězců. (viz 2.13) Tuto operaci potom můžeme generalizovat tak na jazyky, viz 2.14. Znalost a pochopení této operace nám pomůže v chápání dalších operací, jako je vkládání(Kapitoly 2.10 a 2.11).

$$
\begin{aligned}
Shuffle(u, v) = \{ & \\
& u_1 v_1 ... u_i v_j | \\
& \quad u = u_1 ... u_i \wedge u \in \Sigma* \\
& \quad v = v_1 ... v_j \wedge v \in \Sigma* \\
& \quad u_p \in u \wedge u_p \in \Sigma \cup \epsilon; \\
& \quad v_q \in v \wedge v_q \in \Sigma \cup \epsilon; \\
& \quad 1 \leq p \leq i \wedge 1 \leq q \leq j \\
\} &
\end{aligned}
\tag{2.13}
$$

$$
Shuffle(K, L) = \{w | w = Shuffle(u, v); u \in K \wedge v \in L\} \tag{2.14}
$$

2.9.1 Vlastnosti

Operace shuffle je uzavřená nad regulárními jazyky a neuzavřená nad jazyky bezkontextovými.

2.9.2 Příklad

Operace Shuffle nad dvěma řetězci:
Nechť existují řetězce u a v, ta že $u =$ ”ab”, $v =$ ”cd” $Shuffle(u, v)$ se poté bude rovnat {”$abcd$”, ”$acbd$”, ”$acdb$”, ”$cabd$”, ”$cadb$”, ”$cdab$”}

2.10 Sekvenční vkládání (Sequential Insertion)

Tato operace je krásně popsána v [6] strana 23-28. Ale ve svém provedení jako takovém je se jedná o jednoduchou operaci, kdy do libovolného řetězce z jazyka K vložíme na libovolné místo libovolný řetězec z jazyka L. Tuto operaci si tedy můžeme popsat rovnicí 2.15.

$$SequentialInserion(K, L) = \{w | w = xyz, xz \in K \wedge x \in L\} \tag{2.15}$$

2.10.1 Vlastnosti

Operace je uzavřená nad regulárními a bezkontextovými jazyky a samozřejmě není komutativní, viz [6] strana 25-27.

2.10.2 Příklad

Operace SequentialInsertion nad dvěma jazyky:
Mějme dva jazyky, $K = abc, def$ a $L = xy$
Použití operace nám poté generuje jazyk:
$SequentialInsertion(K, L) = \{xyabc, axybc, abxyc, abcxy, xydef, dxyef, dexyf, defxy\}$

2.11 Paralelní vkládání (Parallel Insertion)

Paralelní vkládání je operace velice podobná sekvenčnímu, avšak s tím rozdílem, že pokud máme jazyk K do kterého vkládáme, tak nevkládáme pouze na jednu libovolnou pozici, nýbrž vkládáme na všechny pozice. Operace je opět krásně popsána v [6] na straně 24-28. Tuto operaci bychom si také mohli definovat rovnicí 2.16

$$\begin{aligned} ParallelInsertion(K, L) = \{ & \\ & w | w = x_0 u_0 u_1 x_1 ... x_n u_n x_{n+1}, \\ & \quad x_k \in L \wedge u_1 u_1 ... u_n \in K, \\ & \quad n \geq 0 \wedge 0 \leq k \leq n \\ \} & \end{aligned} \tag{2.16}$$

2.11.1 Vlastnosti

Paralelní vkládání je uzavřené nad rekulárními a bezkotextovými jazyky, viz [6] na straně 27-28.

2.11.2 Příklad

Operace ParallelInsetion nad jazyky:
Mějme jazyky $K = \{abc\}$ a $L = \{de\}$.
Použití operace nám poté generuje jazyk: $ParallelInsertion(K, L) = \{de\mathbf{a}de\mathbf{b}de\mathbf{c}\}$

2.12 Protkání (Interlacement)

Tato operace je inspirovaná Shuffle a vkládajícími operacemi. Tato operace je nejvíce podobná operaci Shuffle, kdy nejdříve uvažujme dva řetězce u a v. Protkáním těchto dvou

řetězců dostáváme řetězec, který se skládá vždy ze symbolu řetězce u a následně symbolu z řetězce v. Protkání dvou řetězců si tedy můžeme definovat rovnicí 2.17 s tím, že pokud dva řetězce nemají stejnou délku, nemohou být protkány, jinak se generuje řetězec prázdné délky. Tuto operaci poté následně můžeme generalizovat a použít nad jazyky obdobným způsobem jako jakoukoliv jinou operaci, tedy protkáme každý řetězec z jazyka K s každým řetězcem z jazyka L (viz. rovnice 2.18). Obdobně bychom mohli operaci generalizovat i na rodinu jazyků (viz rovnice 2.19). Iterativní použití této operace nad rodinou jazyků by mohlo mít silně expandující efekt, proto si představme ještě variaci, která nám nebude přijímat dva stejné jazyky, tedy může bude růst výrazně pomaleji, případně může za správných okolností stagnovat, i klesat 2.20

$$\begin{aligned} Interlacement(u,v) = \{ & \\ & w | w = u_1v_1u_2v_2...u_kv_k; \\ & u = u_1u_2...u_k \wedge v = v_1v_2...v_k; \\ & k \geq 1 \wedge k = |u| = |v| \\ \} & \end{aligned} \tag{2.17}$$

$$Interlacement(K,L) = \{w | w = Interlacement(u,v); u \in K \wedge v \in L; w \neq \epsilon\} \tag{2.18}$$

$$Interlacement(R) = \{J | J = Interlacement(K,L); K,L \in R; J \neq \emptyset\} \tag{2.19}$$

$$UniqueInterlacement(R) = \{J | J = Interlacement(K,L); K,L \in R \wedge K \neq L; J \neq \emptyset\} \tag{2.20}$$

2.12.1 Vlastnosti

Theorem 4 (Uzavřenost nad regulárními jazyky)**.** *Libovolné dva regulární jazyky, jsou uzavřené nad operací Interlacement.*

Důkaz 4. Mějme automaty $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$ a jimi definované jazyky $K(M)$ a $L(N)$. Nyní jsme schopni vytvořit jazyk $K^{'}(M^{'}) = Interlacement(K,L)$:

$$\begin{aligned} M^{'} = \{ & \\ & Q = \{0,1\} \times Q_M \times Q_N, \\ & \Sigma = \Sigma_M \cup \Sigma N, \\ & R = \{ \\ & \quad (0, q_M, q_N)\alpha \longrightarrow (1, q_M\alpha, q_N), \\ & \quad (1, q_M, q_N)\alpha \longrightarrow (0, q_M, q_N\alpha); \\ & \quad q_M \in M \wedge q_N \in N \\ & \} \\ & s = (0, s_M, s_N), \\ & F = \{(0, q_M, q_N) | q_M \in F_M \wedge q_N \in F_N\} \\ \} & \end{aligned} \tag{2.21}$$

A jelikož jsme schopni sestrojit konečný automat přijímající tento jazyk, víme, že tento jazyk bude regulární. Tedy dostáváme vztah:

$$Interlacement(K, L) = K^{'}(M^{'})$$

□

Na první pohled by se mohlo zdát, že konečný automat pro tuto výsledek této operace sestrojit nedokážeme, neb se nám sama nabízí možnost zásobníkového automatu, který by "počítal"jestli bychom měli aplikovat pravidla z prvního nebo druhého automatu, ale jelikož zásobník takového automatu by vždy obsahoval jeden symbol (vyjma počátečního), tak jsme toto chování schopni replikovat nad konečnými automaty pomocí zdvojnásobení počtu stavů.

Použití této operace nad jazyky, může generovat prázdný jazyk, z čehož je zřejmé, že né všechny řetězce přispějí do výsledného jazyka, přesněji řetězec z jazyka K takové délky, že neexistuje řetězec v jazyce L stejné délky, nepřispívá do generovaného jazyka a naopak.

Použití operace nad dvěma jazyky vždy generuje jazyk nepřijímají prázdný řetězec.

Použití operace neprázdnou rodinou jazyků vždy generuje větší, nebo alespoň stejně velkou rodinu jazyků, s tím že při iterovaném použití, se výsledek nikdy neustálí a tedy není možné aby byla rodina jazyků nad touto operací tranzitivně uzavřena.

Pokud bychom uvažovali upravenou verzi, nepřijímající dva stejné jazyky (Jak jsou dva jazyky porovnávány jsme si ukazovali v kapitole 2.5), tak budeme li uvažovat rodinu jazyků R dostáváme následující vlastnosti:

1. Použití operace nad libovolnými dvěma nestejnými jazyky z R vrací prázdný jazyk:

   Generovaná rodina je prázdná

2. Obsahuje-li R pouze dva nestejné jazyky, nad nimiž operace negeneruje prázdný jazyk:

   Generovaná rodina bude obsahovat opět dva stejně velké jazyky jazyky ($Interlacement(K, L) a Inter$

   (a) Při iterovaném použití nad takovou rodinou:
   Od druhého použití víme, že rodina R_i bude stejně velká jako rodina R_{i-1}, avšak jazyky v rodině R_i budou větší než v rodině R_{i-1} a to samé platí i o délce nejdelšího řetězce.

3. Obsahuje li rodina R více kompatibilních jazyků, víme pouze, že při iterativním použití bude od kroku 2 růst, s tou výjimkou, že obsahuje li rodina pouze dvojce kompatibilních jazyků, bude stagnovat obdobně jako v bodě 2

2.12.2 Příklad

Použití nad dvěma řetězci:
Nechť existuje řetězec $u = abc$ a řetězec $v = def$, poté řetězce $w = Interlacement(u, v)$ a $w' = Interlacement(v, u)$ jsou:
$w = adbecf$ a $w' = daebfc$

Použití nad dvěma Jazyky:
Nechť existují jazyky $K = \{01, 23, 45\}$ a $L = \{67, 890\}$, poté jazyky $K^{'} = Interlacement(K, L)$ a $L^{'} = Interlacement(L, K)$ jsou:
$K^{'} = \{0617, 2637, 4657\}$ a $L^{'} = \{6071, 6273, 6475\}$

Použití nad rodinou jazyků:
Nechť existuje rodina jazyků $R = \{L_1 : \{01, 23, 45\}, L_2 : \{67, 890\}, L_3 : \{89, ab\}\}$ poté použití operace nad touto rodinou bude vypadat následovně:

$$\begin{aligned} R^{'} = Interlacement(R) = \{&L_{1,1} : \{0011, 0213, 0415, 2031, 2233, 2435, 4051, 4253, 4455\}, \\ &L_{1,2} : 0617, 2637, 4657, L_{2,1} : \{6071, 6273, 6475\}, ..., \\ &L_{3,2} : \{5697, a6b7\}, L_{3,3} : \{8899, aabb, 8a9b, a8b9\}\} \end{aligned}$$

Použití unikátní verze operace nad rodinou jazyků:
Nechť existuje rodina jazyků $R = \{L_1 : \{01, 23, 45\}, L_2 : \{67, 890\}, L_3 : \{89, ab\}\}$ poté použití unikátní verze operace nad touto rodinou bude vypadat následovně:

$$\begin{aligned} R^{'} = UniqueInterlacement(R) = \{&L_{1,2} : \{0617, 2637, 4657\}, L_{2,1} : \{6071, 6273, 6475\}, \\ &L_{1,3} : \{0819, 2839, 4859, 0a1b, 2a3b, 4a5b\}, ..., \\ &\{L_{3,2} : \{5697, a6b7\}\} \end{aligned}$$

2.13 Sekvenční Mazání (Sequential Deletion)

Sekvenční mazání je operace vzdáleně podobná sekvenčnímu vkládání a to tak, že bychom ji mohly nazvat přesným opakem. Takto si ji však můžeme nazvat pouze neformálně, nemůžeme u této operace spoléhat na $SequentialDeletion(SequentialInsertion(L)) = L$(viz příklad). Více je tato operace popsána v [6] na straně 55-70. Tato operace je lehce popsatelná rovnicí 2.22.

$$SequentialDeletion(K, L) = \{w | w = xz; xyz \in K \wedge y \in L\} \tag{2.22}$$

2.13.1 Vlastnosti

Všechny vlastnosti si dopodrobna může čtenář přečíst ve výše uvedené knize, jako sumarizaci se zde však hodí podotknouti, že operace je uzavřená nad regulárními jazyky.

2.13.2 Příklad

Ukázka toho, že $SequentialDeletion(SequentialInsertion(K, L), L) \neq K$**:**
Mějme jazyky $K = \{abc\}$ a $L = \{a\}$. Použijeme-li sekvenční vkládání, dostáváme $K_2 = SequentialInsertion(K, L) = \{aabc, abac, abca\}$. Pokud následně použijeme sekvenční mazání, dostáváme $K_3 = SequentialDeletion(K_2, L) = \{abc, bac, bca\}$ což se zcela zřetelně nerovná K.

2.14 Plné zakázání abecedy (Full Alphabet deletion)

Tato operace je velmi výrazně inspirovaná Paralelním mazáním [6] strany 55-70, a je zcela aplikovatelná použitím Paralelního mazání, kde bychom místo abecedy použily jazyk, kde každý řetězec je výlučně jeden symbol. Rozdíl této operace s je však v její aplikovatelnosti na konečné automaty (viz 2.14.1), případně v její implementovatelnosti [TODO reference]. Tato operace je opět lehce popsatelná rovnicí

$$FAD(K,\Sigma_2) = \{ \\ u_1u_2...u_ku_{k+1}|k \geq 1, u_i \in \Sigma^*, 1 \leq i \leq k+1 \wedge \\ \exists v_i \in \Sigma_2, 1 \leq i \leq k : u = u_1v_1...u_kv_ku_{k+1}v_{k+1} \\ kde\{u_i\} \notin \Sigma_2 \\ \} \tag{2.23}$$

2.14.1 Vlastnosti

Vzhledem k tomu, že tato operace se dá aplikovat pomocí paralelního mazání, tak víme že stejně jako paralelní mazání bude uzavřená nad regulárními jazyky. Výhodou této operace je však to, že oproti paralelnímu mazání, je ta, že není ani zdaleka tak složitý. Paralelní mazaní samo o sobě vyžaduje velkou režii, a aplikovatelnost nad automatem je velice složitá, kdežto u Plného mazání abecedy, je aplikace primitivní, neb nám pouze stačí projít všechna pravidla automatu ze kterého odstraňujeme abecedu a nahradit odstraňované symboly za ϵ.

2.14.2 Příklad

2.15 Pop

Operace Pop, de facto operace reverzní ke konkatenaci. Tuto operaci si rozdělíme na $LPop(K,L)$ a $RPop(K,L)$. Toto rozdělení provádíme z toho důvodu, že konkatenovat můžeme z obou stran, a tedy bychom rádi z obou stran i odebírali, což musíme nějak specifikovat. Obě operace pop jsou každá, vlastním způsobem omezená operace Sekvenční mazání. Obě operace si můžeme definovat následujícím způsobem:

$$LPop(K,L) = \{y|xy|inK \wedge x \in L\} \tag{2.24}$$

$$RPop(K,L) = \{x|xy|inK \wedge x \in L\} \tag{2.25}$$

2.15.1 Vlastnosti

Theorem 5 (Uzavřenost lPop nad regulárními jazyky)**.** *Libovolné dva regulární jazyky, jsou uzavřené nad operací lPop.*

Důkaz 5. Nechť existují dva jazyky L_1 a L_2, nad abecedou Σ a nechť existuje automat $M = \{Q, \Sigma, R, s, F\}$ přijímající jazyk L_1. Pro každé dva stavy $s, q \in Q$ nechť existuje:

$$L_{s,q} = \{w|sw \longrightarrow^* q \in M; w \in \Sigma\}$$

Poté uvažujme automat:

$$M^{'} = \{Q, \Sigma \cup \{\#\}, R^{'}, s, F\}$$

kde

$$R^{'} = R \cup \{s\# \longleftarrow q|s, q \in Q \wedge L_2 \cap L_{s,q} \neq \emptyset\}$$

kde # je nový symbol nepatřící do abecedy Σ.

Následně můžeme tvrdit že:

$$LPop(L_1, L_2) = h(L(M^{'} \cap \Sigma^{*}\#\Sigma^{*}))$$

Následně důkaz pokračuje, jako důkaz Sekvenčního Mazání, viz [6] strany 60-61, kde $A^{'} = M^{'}$. □

Theorem 6 (Uzavřenost rPop nad regulárními jazyky)**.** *Libovolné dva regulární jazyky, jsou uzavřené nad operací rPop.*

Důkaz 5. Nechť existují dva jazyky L_1 a L_2, nad abecedou Σ a nechť existuje automat $M = \{Q, \Sigma, R, s, F\}$ přijímající jazyk L_1. Pro každé dva stavy $q \in Q \wedge f \in F$ nechť existuje:

$$L_{q,f} = \{w | qw \longrightarrow^{*} f \in M; w \in \Sigma\}$$

Poté uvažujme automat:

$$M^{'} = \{Q, \Sigma \cup \{\#\}, R^{'}, s, F\}$$

kde

$$R^{'} = R \cup \{s\# \longleftarrow q | s, q \in Q \wedge L_2 \cap L_{q,f}\}$$

kde # je nový symbol nepatřící do abecedy Σ.

Následně můžeme tvrdit že:

$$LPop(L_1, L_2) = h(L(M^{'} \cap \Sigma^{*}\#\Sigma^{*}))$$

Následně důkaz pokračuje, jako důkaz Sekvenčního Mazání, viz [6] strany 60-61, kde $A^{'} = M^{'}$. □

Při používání této operace, je důležité mít na paměti, že i-když o ní můžeme uvažovat jako o opaku konkatenace, tedy $RPop(Concatenation(K, L), L) = K$, tak $Concatenation(RPop(K, L), L)$ K. A to protože, do *Pop* operací nepřispívají všechny řetězce jazyka K, nýbrž pouze ty, ze kterých můžeme něco smazat.

2.15.2 Příklad

Ukázka použití LPop a RPop:
Nechť existují dva jazyky, $K = \{aba, ab, c\}$ a $L = \{a\}$.

Použití operace *LPop* nám generuje jazyk $K^{'} = LPop(K, L) = \{ba, b\}$ a použití operace *RPop* nám generuje jazyk $K^{''} = RPop(K, L) = \{ab\}$

Ukázka RPop(Concatenation(K, L), L) = K:
Uvažujme stejné jazyky jako v předchozím příklade, konkatenace $Concatenation(K, L)$ nám generuje jazyk $K^{'} = \{abaa, aba, ca\}$. Následné použití RPop nám generuje opět jazyk K, $RPop(K^{'}, L) = K$.

Ukázka Concatenation(RPop(K, L), L) ≠ K:
Opět uvažujme stejné jazyky jako v předchozích příkladech, použití operace $RPop(K, L)$ nám generuje jazyk $K^{'} = ab$, což se zcela zřetelně nerovná K

Kapitola 3

Implementace ekosystému pro implementaci operací

3.1 Požadavky na implementaci

Vytvořit jednoduché prostředí, nad kterým bude možné zpracovávat operace nad konečnými automaty.

Prostředí musí splňovat následující požadavky:

1. Musí umět zpracovávat konečné automaty.
   (a) Přijmutí konečného automatu
   (b) Vypsání konečného automatu
   (c) Aplikování základních operací nad konečnými automaty neměnící přijímaný jazyk. (odstranění nekoncových stavů, nedeterminismu a pod.)
2. Musí být možné tyto automaty rozšířit/přidat nový typ automatů
3. Musí být možné vytvářet operace pro práci nad těmito automaty. (Implementace operací)
4. Musí být možné přidávat nové typy automatů.
5. Musí být možné na toto prostředí navázat nebo ho používat i v jiných pracích.

3.1.1 Rozšíření

Přidat zpracování zásobníkového automatu, na němž se ukáže rozšiřitelnost ekosystému o další automaty.

3.2 Jazyk a balíčky implementace

Jako jazyk implementace byl zvolený jazyk JavaScript. Tento jazyk byl zvolen z důvodu, že je podporován obrovskou internetovou komunitou a především knihovna(prostředí) napsané v tomto jazyce půjde použít nejen na serverech a prohlížečích, ale také i na mobilních a desktopových zařízeních.

Pro zpřehlednění čitelnosti kódu je použita verze EcmaScript 2015 [todo teference] a výše a pro typovou kontrolu rozšíření Flow [todo reference] od Facebooku [todo reference].

3.2.1 Balíčky použité pro běh (produkční prostředí)

- Lodash [1]

3.2.2 Balíčky použité pro vývoj

- Flow [2]
  - flow-bin [3]: spustitelný program, použitý pro statickou typovou kontrolu projektu
- Babel [12]
  - babel-cli [7]: spustitelný program použitý pro překlad zdrojových kódů v v novější verzi jazyka JavaScript do starší verze jazyka.
  - babel-register-cli [11]: překládá jazyk obdobně jako babel-cli, avšak až za běhu programu. Dalo by se říci že rozšiřuje interpret programu.
  - babel-plugin-transform-object-rest-spread [8]: babel balíček pro rozšíření syntaxe jazyka
  - babel-preset-es2015 [9]: babel balíček pro překlad z EcmaScript 2015 do starších verzí.
  - babel-preset-flow [10]: babel balíček rozšiřující syntaxi o datové typy pro Flow

Zde by mohla vzniknout otázka, proč používat Flow a Babel?

Flow je v implementaci použito jak je výše zmíněno pro typovou kontrolu. Není nijak Nutné používat flow v dalších rozšířeních projektu, avšak věřím že typovost dodá programu více spolehlivosti a také výrazně urychluje vývoj už tím, že zamezuje "runtime"chybám vzniklým právě při nekompatibilních typech u netypovaných jazyků. Výrazným soupeřem Flow je TypeScript, který je stejně vhodný. Zde výběr Flow jako typové nadstavby byl tvolen především pro svou kompatibilitu s Reactem[todo reference], což může zjednodušit případný vývoj grafického prostředí v budoucí projektech.

Babel je v implementaci použit tak jak definují balíčky ve výše uvedeném seznamu. Nepoužití Babel by zamezovalo použití Flow a kód napsaný v EcmaScript 2015 by byl omezený pouze na nejnovější verze interpretů a především kdokoliv kdo by chtěl na tomto kódu stavět by musel být obeznámen s EcmaScript 2015, kdežto takto stačí pouze znalost tradičního (slangově oldschool) JavaScriptu, nebo naopak je možné použít zcela jiný přístup, například beztřídní.

Upozornění: Ke konci dubna 2018 přešel Babel na novouverzi 7.0.0-beta.46. Vzhledem k faktu, že se nejedná o plné uvolnění Babel 7 a že tato práce je psána před vypuštěním Babel 7, jsou výše uvedené Babel knihovny závislé na verzi 6.*. Tento fakt nemá sebemenší vliv na implementaci, ani případné používání praktické části této práce v jiných projektech. Pouze je potřeba aby čtenář věděl, že výše uvedené knihovny jsou nyní "depracated"a tedy při navazujících pracích je doporučováno používat Babel 7, který má však jiný ekosystém. (Nebude-li však navazující práce používat Babel vůbec, tento fakt ji nijak neovlivní, jelikož práce obsahuje i překompilovanou verzi JavaScriptových souborů.)

3.2.3 Testovací prostředí

Pro testovací prostředí byl vybrán framework AVA [13] a pro zjištění pokrytí kódu je používán Istanbul Code Coverage[4], respektive jeho verzi pro příkazovou řádku "nyc"[5]

3.3 Vstupně výstupní formát automatu

Formát jazyka bude nejlépe vysvětlen jeho popisným příkladem viz 3.2, kde <cokoliv>, není součástí formátu, nýbrž popis hodnoty. Důležité je pouze zmínit, že formát je v zápisu JSON, jeho struktura je odvozena od množinového zápisu automatu, a uvedený příklad popisuje povinné hodnoty pro konečný automat a je možné ho jakkoliv rozšířit, bez porušen kompatibility, při zachování stávající struktury

```
{

    "states": [
        {name: <unikátní název stavu (řetězec)>},
        {name:  <unikátní název stavu (řetězec)>
        ],
    "alphabet": [<symbol abecedy (řetězec)>,...],
    "rules": [
        {
            "from": {"state": {"name": <název existujícího stavu (řetězec)>},},
            "to": {"state": {"name": <název existujícího stavu (řetězec)>}},
            "symbol": <existující symbol abecedy (řetězec)>,
        },
        {
            "from": {"state": {"name": <název existujícího stavu (řetězec)>},},
            "to": {"state": {"name": <název existujícího stavu (řetězec)>}},
            "symbol": <existující symbol abecedy (řetězec)>,
        }
    ],
    "finalStates": [{name: <název existujícího stavu (řetězec)>}],
    "initialState": {name: <název existujícího stavu (řetězec)>}
}
```

Výpis 3.1: Přijímaný formát konečného automatu

V rámci rozšíření, pro zásobníkové automaty, je tento formát obohacen o následující klíče:

```
{
  "rules": [
    {
      "from": {
        "state": {"name": <název existujícího stavu (řetězec)>},
        "stackTop": <symbol co musí být na vrcholu zásobníku (řetězec délky 1)>
      },
      "to": {
        "state": {"name": <symoy které budou na vrcholu zásobníku (řetězec délky 0-2)>}
      },
      "symbol": <existující symbol abecedy (řetězec)>,
    },
  ],
  "initialStackSymbol": <(řetězec o délce 1)>,
  "stackAlphabet": [<symbol zásobníkové abecedy(řetězec o délce 1)>,...]
}
```

Výpis 3.2: Rozšíření pro zásobníkový automat

Zde je důležité, dát si velký pozor na to, že symbol zásobníkové abecedy musí být řetězec délky jedna.

3.4 Implementace

Samotná implementace je provedena třídně objektově, kde každá součást automatu a automat sám je objekt, obdobně jako je tomu u formátu výše. Nejednoduší popsání bude pomocí třídní reprezentace viz 3.3, obdobně jako výše popsaný formát.

```
class Automata = {
    states: { <název stavu (řetězec)>: State };
    alphabet: Alphabet;
    rules: Rule[]; //pole Rule
    initialState: State;
    finalStates: { <název stavu (řetězec)>: State };
}

class State = {
    name: <(řetězec)>;
    isInitial: boolean;
    isFinal: boolean;
    isNonterminating: boolean;
}

class Alphabet = <Pole s pouze unikátními prvky (pole)>

class Rule = {
    from: {state:State};
    to: {state:State};
    symbol: string;
}
```

Výpis 3.3: Třídní hierarchie implementace

3.5 Implementace libovolné operace nad tímto ekosystémem

Každá operace jenž je možné provést nad daným automatem je pro tento typ automatu pomocí implementovatelná. Tato kapitola se bude věnovat implementace hypotetické operace.

Představme si tedy unární operaci *allowEmpty* jejímž cílem je pouze donutit automat aby přijímal prázdný řetězec. (viz 3.4 a 3.5)

```
//@flow
import FA from "../Automata/FA/FA.js";

/**
* Generuje automat přijímající prázdný řetězec
* @param {FA} automata
*/
export default function allowEmpty(automata:FA):FA {
  let newAutomata = automata.copy();
  newAutomata.initialState.isFinal = true;
  return newAutomata;
}
```

Výpis 3.4: Ukázka implementace operace za použití EcmaScript 2015 a Flow

```
var FA = require("../Automata/FA/FA.js").default;

```

```
/**
* Generuje automat přijímající prázdný řetězec
* @param {FA} automata
*/
function allowEmpty(automata) {
  var newAutomata = automata.copy();
  newAutomata.initialState.isFinal = true;
  return newAutomata;
}
window.exports = {default:allowEmpty};
```

Výpis 3.5: Ukázka implementace operace pomocí standardního JavaScriptu

K použití operace pak už pouze stačí, vytvořit automat a zavolat tuto operaci tak, že automat je jejím argumentem. (příklad viz 3.6)

```
import FA from "../Automata/FA/FA.js";
import allowEmpty from 'allowEmpty.js';
let automata = new FA(/*...*/);
let newAutomata = allowEmpty(automata);
-----------------------------------------------------------------------
var FA = require("../Automata/FA/FA.js").default;
var allowEmpty = require('allowEmpty.js').default;
var automata = new FA(/*...*/);
var newAutomata = allowEmpty(automata);
```

Výpis 3.6: ukázka použití operace

3.6 Zprovoznění

Pro zprovoznění práce je nutné mít nainstalováno následující:

1. *node* verze 9.11.1 a vyšší https://nodejs.org/en/download/
2. *npm* verze 5.8.0 a vyšší https://www.npmjs.com/get-npm,

Práce jako taková může běžet na serveru EVA, bez potřeby jakýchkoliv doplňků.

Zprovoznění pro použití:

1. Získání kódů bakalářké práce
2. cd {havní složka práce}
3. npm install –production

Zprovoznění pro úpravu:

1. Získání kódů bakalářké práce
2. cd {havní složka práce}
3. npm install

Spuštění testů: Zprovozníme pro úpravu, můžeme použít i zprovoznění pro použití, ale pak je potřeba: *npm install ava*
Máme li zprovozněno pouze pro použití s AVA, můžeme spustit testy pomocí *npm test*, Máme li zprovozněno pro úpravu, můžeme použít nasledující příkazy:

1. *npm test* #spustí testy
2. *npm run build* #vygeneruje novou složku **dist** dle změn v **src**
3. *npm run testbuild* #spustí build a následně testy
4. *npm run testES6* #spustí testy nad src
5. *npm run coverage* #spustí testy s "code coverage"
6. *npm run lcovCoverage* #spustí testy s "code coverage", které generují *coverage/lcov-report/index.html*, což je hezky proklikatelná verze "code coverage"

Kapitola 4

Použití operací nad automaty

Spojíme znalosti z 2 a 3 a podíváme se na použití dříve uvedených operací nad automaty.

Ukázky kódů v této kapitole budou pro lepší čitelnost přebírat syntaxi z reálného kódu. Kapitola se bude soustředit především na sémantickou část a proto budou některé části kódu vynechány. Případně může být vynechána celá implementace, pokud se bude jednat o jednoduchou operaci. Opravdu zanícený čtenář se může vždy podívat do skutečné implementace, jejíž poloha bude uvedena u každého příkladu. Dále v této kapitole už nebudeme používat standardní JavaScript, nýbrž pouze EcmaScript 2015 s Flow a to z důvodů, že by jinak tato kapitola zbytečně nabyla na velikosti.

4.1 Sjednocení (Union)

Návod jak provést tuto operaci je předveden v předmětu IFJ [TODO reference] a proto se podíváme pouze na její implementaci.

4.1.1 Implementace

V této operaci si ukážeme, nejen jak provést sjednocení, ale jak ho provést nad několika typy automatů. Proto, ikdyž se jedná o relativně jednoduchou operaci, ukážeme si zde její kód.

```
1  {
2  //@flow
3  /*Import ptřebných věcí*/
4
5  /**
6  * Sjednocení dvou Automatu
7  * @param {Automata} left
8  * @param  {Automata} right
9  * @return Automata
10 */
11 export default function union(left: (Automata | PA | FA), right: (Automata | PA | FA)) {
12   //Převedeme si automat na jeho serializovatelnou reprezentaci
13   let plainLeft: T_AnyPlainAutomata = toPlainLeft(left),
14       plainRight: T_AnyPlainAutomata = toPlainRight(right);
15
16   // suffix pro nové stavy
17   let suffix = State.randomName();
18
19
```

```
20    //sestrojíme automat
21    let plainUnion:T_AnyPlainAutomata = {
22        states: [{name: 's' + suffix}, /*puvodní stavy obou automatu*/],
23        alphabet: [./*sjednocení abeced*/],
24        finalStates: [/*sjednocení koncových stavu*/],
25        initialState: {name: 's' + suffix}
26    };
27
28    // wrapper pro předávání parametru
29    let paFun = /*funkce pro zásobníkový automat*/;
30    // wrapper pro předávání parametru
31    let faFun = /*funkce pro konečný automat*/;
32
33    // Nastavíme jaká funkce se má volat pro kterou kombinaci automatu
34    //v tomto případě, pokud je jeden z automatu Zásobníkový, používáme funkci additionalPA
35    return overload(
36      [
37        {parameters: [{value: left, type: FA}, {value: right, type: PA}], func: paFun},
38        {parameters: [{value: left, type: PA}, {value: right, type: FA}], func: paFun},
39        {parameters: [{value: left, type: PA}, {value: right, type: PA}], func: paFun},
40        {parameters: [{value: left, type: FA}, {value: right, type: FA}], func: faFun},
41      ]
42    );
43  }
```

Výpis 4.1: Ukázka implementace operace Union (*src/operations/union.js*)

4.2 Průnik (Intersection)

4.2.1 Provedení nad automatem

Průnik dvou Automatů je jednoduchá, avšak dosti zdlouhavá operace. Její aplikaci si předvedmě na příkladu. Mějme dva automaty :

$M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$

Průnik poté provedeme následovně:

$$
\begin{aligned}
Intersection(M, N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cap \Sigma_N \\
& R = \{(q_M, q_N)\alpha \longrightarrow (q_M\alpha, q_N\alpha); \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.1}
$$

4.2.2 Implementace

Implementace zde pouze opisuje postup vytvoření automatu. (Soubor *src/operations/intersectionFA.js*)

4.3 Doplněk (Complement)

4.3.1 Provedení nad automatem

Provedení této operace nad automatem vyžaduje aby měl automat takzvaný uklízecí stav (trap state). Uklízecí stav je takový stav, který je nekoncový a jsou do něj odvedeny přechody, které nemohou vést do jiných stavů. Tento stav je dobře popsán v Dobře specifikovaném automatu [TODO reference IFJ]

4.3.2 Implementace

Implementace této operace je ve své podstatě velice jednoduchá, jediné co je potřeba je uvědomit si teorii uvedenou výše, tedy že musíme mít uklízecí stav. Poté pouze uděláme z ukončujících stavů neukončující a naopak.
(*src/operations/complementFA.js*)

4.4 Rozdíl (Difference)

4.4.1 Provedení nad automatem

Zde pouze sledujeme myšlenku ze sekce 2.4 a používáme již existující operace.

4.4.2 Implementace

(*src/operations/differenceFA.js*)

4.5 Rozdílné sjednocení (Different Union)

4.5.1 Provedení nad automatem

Hlavní podmínkou provedení této operace je porovnat dva jazyky. Jsme-li schopni rozhodnout o jejich rovnosti, poté je uzavřenost a celková proveditelnost operace ponechána jen proveditelnosti sjednocení. Dva jazyky můžeme považovat za schodné, pokud rozdíl těchto jazyků generuje vždy prázdnou množinu v libovolném pořadí použití. Zde je však vidět, že ne všechny typy jazyků lze porovnávat, jelikož ne všechny typy jazyků jsou uzavřeny nad operací rozdíl. Dva automaty považujeme za shodné, pokud přijímají tentýž jazyk.

4.5.2 Implementace

V implementaci si musíme dávat pozor na porovnávání přijímaných jazyků, avšak pokud se budeme držet teorie a použijeme dříve implementovanou operaci rozdílu. je implementace pouze otázka dvou podmínek. (*src/operations/differentUnionFA.js*)

4.6 Operace "Rozdílné"(Operation Different)

4.6.1 Provedení nad automatem

Provedení je stejné jako nad jazyky, viz 2.6, tedy použijeme již existující operace.

4.6.2 Implementace

(*src/operations/operationDifferentFA.js*)

4.7 Konkatenace (Concatenation)

Návod jak provést tuto operaci je předveden v předmětu IFJ [TODO reference] a její implementace je s ní totožná.

4.7.1 Implementace

(*src/operations/concatenationFA.js*)

4.8 Unikátní Konkatenace (Konkatenace)

Tato operace lze provést pokud víme jak funguje konkatenace(4.7) a rozdíl(4.4) . Následně můžeme operaci implementovat použitím vzorce 2.11

4.9 Implementace

(*src/operations/uniqueConcatenationFA.js*)

4.10 Předpony (Prefixes)

4.10.1 Provedení nad automatem

Zde je potřeba pouze si uvědomit co je to prefix řetězce. Je prázdý řetězec, první znak, první dva znaky a tak dále, až po poslední znak včetně vice [TODO reference IFJ]. A tedy potřebujeme jen donutit automat aby přijímal všechny řetězce, které mohou vest k řetězci jenž by byl přijat původním automatem. Najdeme si tedy všechny ukončující stavy, uděláme z nich stavy koncové a máme hotovo.

4.10.2 Implementace

```
1  //@flow
2  /*Import potřebných závislostí*/
3
4  /**
5  * Generuhe automat přijímající prefixy zadaného automatu
6  * @param {FA} automata
7  * @return {FA}
8  */
9  export default function prefixes(automata: FA): ?FA {
10  //vytvoříme nový automat ze starého
11  let resAutomata = automata.clone();
12  resAutomata.removeTrapStates(); //odstraníme uklízecí stavy
13
14   //všechny stavy se stávají koncovými
15  for(let state of objectValues(resAutomata.states)) {
16    state.setAsFinal();
17  }
```

```
18    resAutomata.finalStates = _.clone(resAutomata.states);
19
20     //pro úhlednost přidáme uklízecí stav
21     resAutomata.ensureOneTrapState();
22
23     return resAutomata;
24   }
25
```

Výpis 4.2: Ukázka implementace Předpon (*src/operations/prefixesFA.js*)

4.11 (Shuffle)

4.11.1 Provedení nad automatem

Aplikace se z počátku může zdát složitá. Uvědomíme-li si však, že můžeme používat složení automatů v kartézském součinu pro sledování cesty v kterém automatu se pohybujeme, zjistíme, že stačí provést pouze následující:
Mějme dva automaty :
$M = \{Q_M, \Sigma_M, \delta_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, \delta_N, s_N, F_N\}$
Promíchání poté provedeme následovně:

$$
\begin{aligned}
Shuffle(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& \delta = \{ \\
& \qquad (q_M, q_N)\alpha \longrightarrow (q_M\alpha, q_N) \vee (q_M, q_N\alpha); \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
& \}
\end{aligned}
\tag{4.2}
$$

4.11.2 Implementace

Jak je výše uvedeno v zápisu automatu, promíchání probíhá tak, že se posouváme v dle pravidla automatu M, nebo automatu N. Viz následující ukázka kódu, kde si ukážeme generování pravidel pro Shuffle.

```
1  /**Anotace*/
2  function createRules(
3    left: FA,
4    right: FA,
5    newStates: { [key: string]: MergedState }
6  ): Rule[] {
7    let newRules = [];
8    //pro každý nový stav generujeme pravidla
9    for (let newState: MergedState of objectValues(newStates)) {
10     //generujeme pravidla z levého automatu
11     let leftRules = /*Filtrovaná pravidla levého automatu, pro tento stav*/;
12     for (let rule: Rule of leftRules) {
```

```
13        newRules.push(new Rule({
14          from: {state: newState},
15          symbol: rule.symbol,
16          to: {state: newStates[MergedState.createName(rule.to.state,   newState.oldRight)]}
17        }));
18      }
19
20      //generujeme pravidla z pravého automatu
21      /*Obdobně jako pro lévé*/
22        ...
23          to: {state: newStates[MergedState.createName(newState.oldLeft, rule.to.state)]}
24        ...
25    }
26    return newRules;
27  }
```

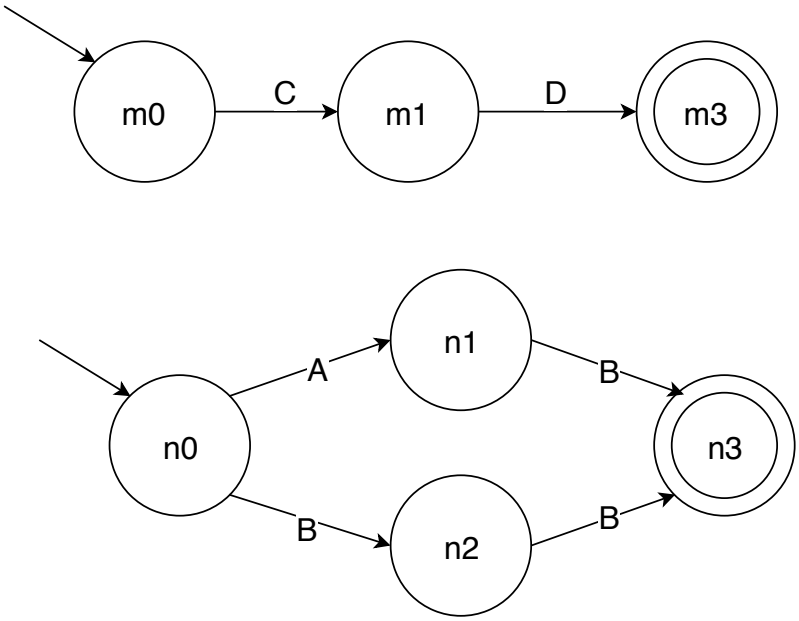
Výpis 4.3: Ukázka generování přechodů pro shuffle (*src/operations/shuffleFA.js*)

4.12 Sekvenční Vložení (Sequential Insertion)

4.12.1 Provedení nad automatem

Vzhledem k faktu, že tato operace je podobná operaci shuffle, můžeme nad ní uvažovat velice podobně. V této operaci je však rozdíl, že musíme vložit řetězec z druhého jazyka nerozdělený. Nejlépe ukážeme na příkladu:

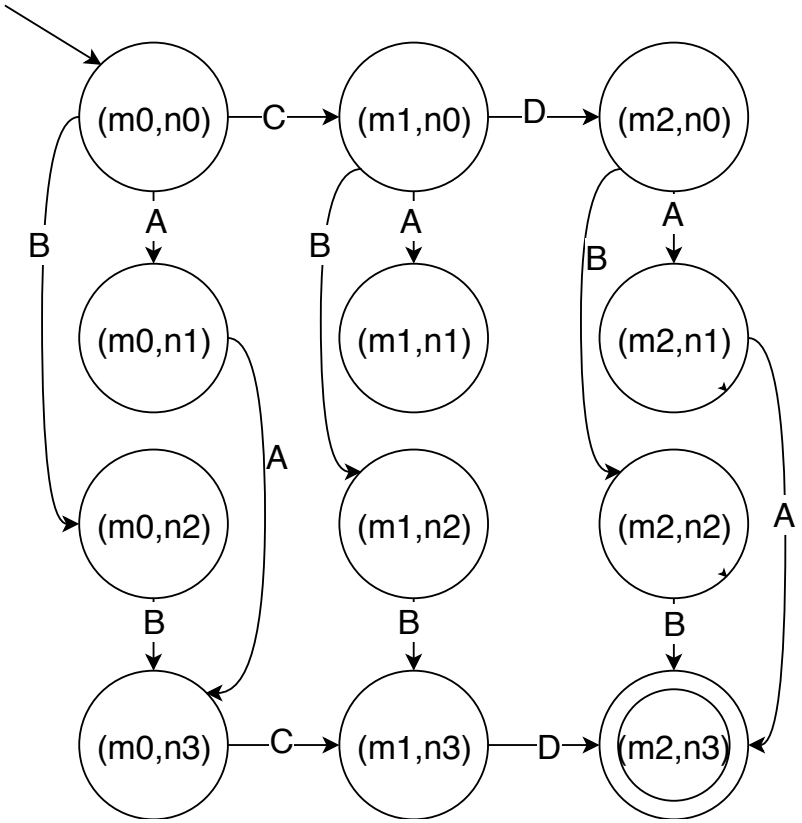
Mějme dva jazyky $K(M) = \{CD\}$ a $L(N) = \{AA, BB\}$ a automaty kterými jsou definovány.(4.1)



Obrázek 4.1: Příklad automatů pro Sequential Insertion

Nyní nad těmito jazyky budeme chtít provést sekvenční vložení N do M, tedy $SequentialInsertion(M, N)$. Zde je si potřeba uvědomit, že můžeme provést kartézský součin k tomu abychom sledovali ve kterém jsme právě automatu. Pro jednodušší pochopení

si představme, že automat M je osa X a automat N je osa Y. Sekvenční vložení nám pak říká, že musíme řetězec přijímaný automatem N můžeme vložit kamkoliv do řetězce přijímaného automatem M. Tedy se můžeme pohybovat přechody po ose X libovolně, ale ve chvíli, kdy se přesuneme po ose Y, musíme po této ose dojít až na konec řetězce přijímaného automatem N. (viz. 4.2)



Obrázek 4.2: Výsledek po použití operace Sequential Insertion

Tento příklad si můžeme zobecnit následujícím způsobem, mějme dva automaty : $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$
Sekvenční vložení N do M provedeme následovně:

$$
\begin{aligned}
Insertion(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& \delta = \{(q_M, q_N)\alpha \longrightarrow \begin{cases} (q_M\alpha, q_N) & if \quad q_N = s \vee q_N \in F_N \\ (q_M, q_N\alpha) & else \end{cases}; \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.3}
$$

4.12.2 Implementace

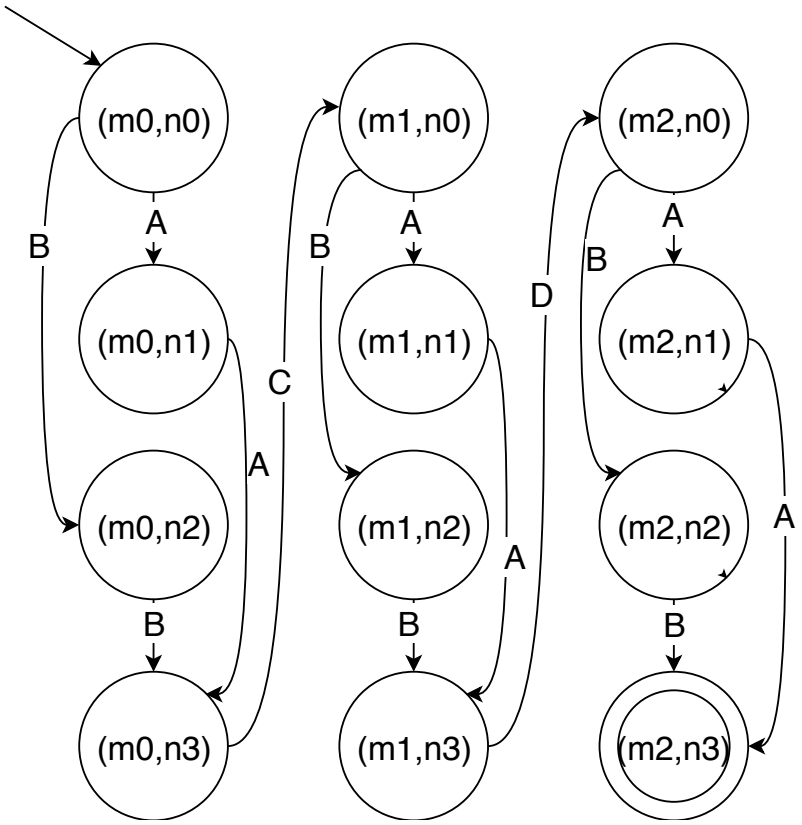
Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/sequentialInsertionFA.js*)

4.13 Paralelní vkládání (Paralel Insertion)

4.13.1 Provedení nad automatem

Provedení paralelního vkládání je velice podobné vkládání sekvenčnímu, nejednodušší bude opět si uvést příklad. Mějme tedy opět dva jazyky $K(M) = \{CD\}$ a $L(N) = \{AA, BB\}$ a automaty kterými jsou definovány, viz výše uvedený příklad pro sekvenční vkládání(4.1);

Rozdíl je zde v tom, že při paralelním vkládání, musíme vždy po zpracování znaku z řetězce jazyka $K(M)$, zpracovat celý řetězec z jazyka $L(N)$. Abychom použili analogii na na osy X a Y, tak vždy když se chceme posunout po ose X, musíme se posunout po ose Y až do konečného stavu. (viz 4.3)



Obrázek 4.3: Automat generovaný operací ParalelInsertion(N,L)

Tento příklad si můžeme zobecnit následujícím způsobem, mějme dva automaty : $M = \{Q_M, \Sigma_M, \delta_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, \delta_N, s_N, F_N\}$

Paralelní vkládání poté můžeme zobecnit následovně:

$$
\begin{aligned}
ParalelInsertion(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& \delta = \{(q_M, q_N)\alpha \longrightarrow \begin{cases} (q_M\alpha, s_N) & if \quad q_N \in F_N \\ (q_M, q_N\alpha) & else \end{cases}; \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.4}
$$

4.13.2 Implementace

Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/parallelInsertionFA.js*)

4.14 Protkání (Interlacement)

V této operaci postupujeme tak, že vytváříme automat přesně tak jak je automat vytvořen v důkazu.

4.14.1 Implementace

Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/interlacementFA.js*)

4.15 Sekvenční mazání (Sequential Deletion)

[TODO, implementovat, poté popsat]

4.16 Zakázání abecedy (Full Alphabet deletion)

[TODO, implementovat, poté popsat] [Ale v podstatě se jedná o homomorfismus alpha -> epsilon]

4.17 (Pop)

[TODO, implementovat, poté popsat] [Ale v podstatě se jedná o homomorfismus alpha -> epsilon]

Kapitola 5

Závěr

TODO

Literatura

[1] Dalton, J.-D.; aj.: *Lodash*. Duben 2018, [Online; verze 4.17.10 a vyšší; navštíveno 03.05.2018].
URL https://lodash.com/

[2] Facebook: *Flow*. Duben 2018, [Online; navštíveno 03.05.2018].
URL https://flow.org/

[3] Facebook: *flow-bin*. Duben 2018, [Online; verze 0.71.0 a vyšší; navštíveno 03.05.2018].
URL https://github.com/flowtype/flow-bin

[4] Istanbuljs tým: *Istanbul*. Apr 2018.
URL https://istanbul.js.org/,note=

[5] Istanbuljs tým: *nyc*. Apr 2018, [Online; verze 11.7.1 a vyšší ; navštíveno 03.05.2018].
URL https://github.com/istanbuljs/nyc/

[6] Kari, L.: *On Insertion and Deletion in Formal Languages*. TODO, 1999, ISBN TODO.

[7] Tým Babel: *babel-cli*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL https://github.com/babel/babel/tree/master/packages/babel-cli

[8] Tým Babel: *babel-plugin-transform-object-rest-spread*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL
https://www.npmjs.com/package/babel-plugin-transform-object-rest-spread

[9] Tým Babel: *babel-preset-es2015*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL
https://github.com/babel/babel/tree/master/packages/babel-preset-es2015

[10] Tým Babel: *babel-preset-flow*. Říjen 2017, [Online; verze 6.23.0; navštíveno 03.05.2018].
URL
https://github.com/babel/babel/tree/master/packages/babel-preset-flow

[11] Tým Babel: *babel-register-cli*. Srpen 2017, [Online; verze 5.0.0; navštíveno 03.05.2018].
URL https://github.com/babel/babel/tree/master/packages/babel-cli

[12] Tým Babel: *Babel*. Duben 2018, [Online; verze 6.26.3; navštíveno 03.05.2018].
URL https://flow.org/

[13] Wubben, M.; Sorhus, S.; Demedes, V.: *AVA*. Duben 2017, [Online; verze 1.0.0-beta.3; navštíveno 03.05.2018].
URL https://github.com/avajs/ava