VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

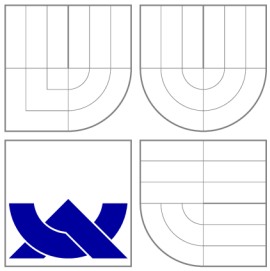
GRAFICKÁ SIMULACE ČINNOSTI KONEČNÝCH AUTOMATŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

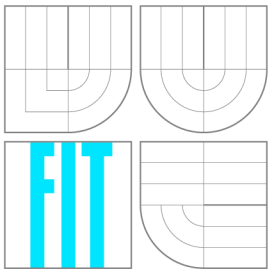
AUTOR PRÁCE
AUTHOR

ROMAN ŠRAJER

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GRAFICKÁ SIMULACE ČINNOSTI KONEČNÝCH AUTOMATŮ

GRAPHICAL SIMULATION OF FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ROMAN ŠRAJER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN ČERMÁK

BRNO 2009

Zadání bakalářské práce/8274/2008/xsraje00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2008/2009

Zadání bakalářské práce

Řešitel: **Šrajer Roman**
Obor: Informační technologie
Téma: **Grafická simulace činnosti konečných automatů**
Kategorie: Teorie informatiky

Pokyny:

1. Detailně se seznamte s principy konečných automatů a regulárních výrazů
2. Vyberte vhodný programovací jazyk pro implementaci a vhodný formát XML pro načítání a ukládání simulovaného automatu
3. Implementujte webovou aplikaci, která bude simulovat činnost KA pro libovolné regulární výrazy
4. Diskutujte výhody a nevýhody vašeho řešení a možné pokračování projektu

Literatura:

- Meduna, A.: Automata and Languages: Theory and Applications, Springer-Verlag, London, 2000.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Čermák Martin, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2008
Datum odevzdání: 20. května 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Originální licenční smlouva je přiložena ve výtisku bakalářské práce v knihovně FIT VUT Brno.

Abstrakt

Tato bakalářská práce se zabývá teorií převodu regulárního výrazu na konečný automat a zpět. Cílem praktické části je vytvořit webovou aplikaci, která tyto převody zajistí a odsimuluje automat pro vstupní řetězec. Pro převod konečného automatu na regulární výraz je použita algebraická metoda, která spočívá ve vytvoření soustavy rovnic a jejím následném vyřešení. Pro implementaci jsem zvolil jazyk Java a technologie Java Applet a Java Web Start, které umožňují přístup k aplikaci přes webové stránky. Nezbytnou součástí aplikace je možnost pracovat se třemi způsoby reprezentace konečného automatu, možnost ukládát automaty do XML a možnost zcela intuitivně vytvářet vlastní automaty.

Abstract

This Bachelor's thesis is about the theory of converting regular expression to finate state machine and vice versa. The goal of practical part is to make web application that performs these coversions and do simulation for input string. For converting automata into regular expression it is used the algebraic method which is based on making an equation system to be solved. I have chosen Java as programming language and Java Applet and Java Web Start as technologies that make possible to access the application through the web page. The possibilities like working with three types of automata views, saving atomatons into XML and intuitively creating own automatons are the important part of the application.

Klíčová slova

Regulární výraz, konečný automat, simulace, Java, Java Web Start, swing, awt

Keywords

Regular expression, finite state machine, simulation, Java, Java Web Start, swing, awt

Citace

Roman Šrajer: Grafická simulace činnosti konečných
automatů, bakalářská práce, Brno, FIT VUT v Brně, 2009

Grafická simulace činnosti konečných automatů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Čermáka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Roman Šrajer
1. května 2009

Obsah

Kapitola 1

Úvod

V této bakalářské práci se zabývám problematikou teorie regulárních jazyků se zaměřením na dva modely pro jejich popis: **regulární výrazy** a **konečné automaty**.

V kapitole 2 je z oblasti formálních jazyků shrnuto a vysvětleno několik pojmů, které jsou nezbytné pro pochopení další teorie. Jedná se o pojmy jako *abeceda*, *zřetězení*, *iterace* a podobně, přičemž po každém z nich následuje vždy názorný příklad, který by měl osvětlit někdy obtížně pochopitelné formální definice.

V kapitole 3 rozebírám samotnou teorii regulárních výrazů a konečných automatů. K oběma modelům jsou uvedeny jejich formální definice a u konečného automatu jsou navíc popsány jeho tři způsoby reprezentace. V dalších částech jsou vysvětleny algoritmy pro převod regulárního výrazu na základní konečný automat až do formy automatu minimálního. Algoritmy jsou popsány formou pseudo-kódu a ten je poté vysvětlen slovně na názorném příkladu, aby čtenář tyto převody snáze a rychleji pochopil. Převod konečného automatu na regulární výraz je proveden algebraickou metodou, kdy je vytvořena soustava rovnic, která musí být vyřešena. Zde je uveden pouze příklad vyřešení takové soustavy. Algoritmus ve formě pseudo-kódu se mi nepodařilo nalézt, takže jsem vytvořil vlastní s pomocí rekurzivního sestupu.

Úkolem praktické části práce je vytvoření webově dostupné aplikace, která provede všechny typy převodů a dokáže pro libovolný automat odsimulovat jeho činnost pro libovolný řetězec.

Pro implementaci, kterou rozebírá kapitola 4, jsem využil programovací jazyk **Java**. Jsou zde popsány jen ty nejdůležitější aspekty aplikace, jako způsob převodu regulárního výrazu na základní konečný automat a převod opačný. Dále je rozebrán formát XML pro ukládání konečných automatů a export automatu do obrázku ve formátu GIF.

Kapitola 5 popisuje způsob realizace webového přístupu k Java aplikaci. Pro toto jsem využil technologie **Java Applet** a **Java Web Start**, které jsou použitelné přímo z webové stránky tak, že Java aplikace je buď spuštěna v kontextu webového prohlížeče nebo je spuštěna ve vlastním okně podobně jako běžná aplikace, jen s omezeným přístupem jako applet.

V poslední kapitole 6 jsou diskutovány výhody a nevýhody implementované aplikace a možné pokračování vývoje.

Kapitola 2

Základní pojmy

Předtím, než přistoupíme k samotné teorii konečných automatů, je potřeba vysvětlit některé základní pojmy, které jsou nezbytné k dalšímu pochopení. Prvním z nich je pojem *množina*:

Definice 2.1. Množina je souhrn nějakých navzájem různých elementů. Důležité je, že se v ní nemohou prvky opakovat a že pořadí prvků není důležité. Prvky se zapisují do složených závorek {} oddělené čárkou, viz. příklad 2.1. Prázdná množina se zapisuje $\emptyset$.

Příklad 2.1. Množinu X čísel 1, 2 a 3 zapisujeme: $X = \{1, 2, 3\}$

Nejčastějšími operacemi nad množinami je sjednocení, průnik a rozdíl.

Definice 2.2. Nechť A a B jsou množiny. Sjednocení ($\cup$), průnik ($\cap$) a rozdíl ($-$) mezi množinami A a B je definován:

$$A \cup B = \{x : x \in A \; nebo \; x \in B\}$$
$$A \cap B = \{x : x \in A \; a \; x \in B\}$$
$$A - B = \{x : x \in A \; a \; x \notin B\}$$

Následující uvedené pojmy již souvisí s teorií formálních jazyků. Důležitými pojmy jsou: *abeceda, řetězec, mocnina, jazyk, iterace a sjednocení jazyků.*

Definice 2.3. Abeceda — je neprázdná, konečná množina elementů, které nazýváme *symboly.*

Příklad 2.2. Abeceda se často označuje řeckým písmenem Σ, takže např.: $\Sigma = \{a, b, c\}$.

Definice 2.4. Řetězec nad abecedou Σ — formálně je definován následovně:

- Nechť $a \in \Sigma$, pak a je také řetězcem nad abecedou Σ
- Nechť $a \in \Sigma$ a r je řetězec nad abecedou Σ, pak ar a ra jsou také řetězce nad abecedou Σ
- Nechť r je řetězec nad abecedou Σ. Pak řetězec ε značí prázdný řetězec a platí $r\varepsilon = \varepsilon r = r$

Definice 2.5. Délka řetězce — nechť x je řetězec nad abecedou Σ. *Délka řetězce* x (značeno $|x|$) je definována:

- Pokud $x = \varepsilon$, pak $|x| = 0$
- Pokud $x = a_1 \dots a_n$, pak $|x| = n$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \dots, n$

Příklad 2.3. *acb* je řetězec x nad abecedou Σ z příkladu 2.2, kde $|x| = 3$ (délka je 3).

Neformálně je možné říci, že řetězec nad Σ je libovolná posloupnost symbolů z abecedy Σ nebo ε. Z formální definice řetězce je možné jednoduše odvodit pojem *zřetězení*.

Definice 2.6. Zřetězení (konkatenace) řetězců — nechť x a y jsou dva řetězce nad abecedou Σ. Zřetězení x a y je řetězec xy.

Příklad 2.4. Nechť máme dva řetězce *cca* a *bca* nad abecedou Σ z příkladu 2.2. Jejich zřetězení je *ccabca*.

Definice 2.7. Mocnina řetězce — nechť x je řetězec nad abecedou Σ, pak jeho i-tá mocnina (značeno x^i, kde $i \geq 0$) je definována:

- $x^0 = \varepsilon$
- $x^i = xx^{i-1}$ pro $i \geq 1$

Neformálně je i-tá mocnina řetězce x takový řetězec, ve kterém se i-krát opakuje řetězec x. Pokud se nějaký řetězec opakuje 0-krát, jedná se o prázdný řetězec ε.

Příklad 2.5. Nechť x je řetězec z příkladu 2.3, pak platí $x^3 = acbacbacb$.

Definice 2.8. Jazyk — nechť Σ je abeceda a nechť Σ^* je množina všech řetězců nad Σ, pak každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Příklad 2.6. Pro abecedu Σ z příkladu 2.2 platí: $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, \dots\}$. Jazyky L_1 a L_2 nad Σ mohou být $L_1 = \{a, c\}$, $L_2 = \{b, ab, ac\}$.

Definice 2.9. Zřetězení (konkatenace) jazyků — nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ. Zřetězení jazyků L_1 a L_2 (značeno L_1L_2) je definováno $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$. Dále pro libovolný jazyk L platí:

- $L\{\varepsilon\} = \{\varepsilon\}L = L$
- $L\emptyset = \emptyset L = \emptyset$

Příklad 2.7. Pro jazyky z příkladu 2.6 platí $L_1L_2 = \{ab, aab, aac, cb, cab, cac\}$.

Definice 2.10. Sjednocení jazyků — nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ. Sjednocení jazyků L_1 a L_2 (značeno $L_1 \cup L_2$) je definováno $L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\}$.

Příklad 2.8. Pro jazyky z příkladu 2.6 platí $L_1 \cup L_2 = \{a, c, b, ab, ac\}$.

Definice 2.11. Mocnina jazyka — nechť L je jazyk nad abecedou Σ, pak jeho i-tá mocnina (značeno L^i, kde $i \geq 0$) je definována:

- $L^0 = \{\varepsilon\}$
- $L^i = LL^{i-1}$ pro $i \geq 1$

Příklad 2.9. Pro jazyk L_1 z příkladu 2.6 platí:

- $L_1^0 = \{\varepsilon\}$
- $L_1^1 = L_1 = \{a, c\}$
- $L_1^2 = L_1 L_1 = \{aa, ac, ca, cc\}$
- $L_1^3 = L_1 L_1^2 = \{aaa, aac, aca, acc, caa, cac, cca, ccc\}$

Definice 2.12. Iterace jazyka — nechť L je jazyk nad abecedou Σ. *Iterace* jazyka L (značeno L^*) a *pozitivní iterace* jazyka L (značeno L^+) jsou definovány:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

Příklad 2.10. Pro jazyk L_1 z příkladu 2.6 platí:

$$L_1^* = \{\varepsilon, a, c, aa, ac, ca, cc, aaa, aac, aca, acc, caa, cac, cca, ccc, \ldots\}$$

$$L_1^+ = \{a, c, aa, ac, ca, cc, aaa, aac, aca, acc, caa, cac, cca, ccc, \ldots\}$$

Jediný rozdíl mezi *iterací* a *pozitivní iterací* spočívá v tom, že *pozitivní iterace* neobsahuje prázdný řetězec ε. Oba typy je možné mezi sebou převést:

- $L^+ = LL^* = L^*L$
- $L^* = L^+ \cup \{\varepsilon\}$

Kapitola 3

Regulární výrazy a konečné automaty

Regulární výrazy a konečné automaty jsou dva formální modely, jak popsat tzv. regulární jazyky. Mezi těmito modely je možné libovolně provádět převody, aniž by se ztratila jejich vyjadřovací schopnost.

V kapitole 2 jste se dozvěděli, že *jazyk* je množina řetězců. Z toho je možné usoudit, že existují různé jazyky, které obsahují různé řetězce. Podle počtu řetězců se jazyky rozdělují na:

- konečné (obsahují konečný počet řetězců konečné délky)
- nekonečné (obsahují nekonečné množství řetězců konečné délky)

Regulární jazyk patří do kategorie *nekonečných* jazyků. Zároveň platí, že každý *konečný* jazyk je *regulární*, viz. příklad 3.1.

3.1 Regulární výraz (RV)

Definice 3.1. Regulární výraz (RV) — nechť Σ je abeceda. RV nad Σ je definován takto:

- symbol $\emptyset$ je RV, který popisuje prázdný jazyk $\{\}$
- symbol ε je RV, který popisuje jazyk $\{\varepsilon\}$
- symbol a, kde $a \in \Sigma$, popisuje jazyk $\{a\}$
- nechť r a s jsou RV, které popisují jazyky L_r a L_s, potom platí:
  - $(r.s)$ označuje jazyk $L_r L_s$ — viz. definice 2.9
  - $(r + s)$ označuje jazyk $L_r \cup L_s$ — viz. definice 2.10
  - (r^*) označuje jazyk L_r^* — viz. definice 2.12
- priorita operátorů je: $. > * > +$; operátor $.$ je možné vynechat, takže $r.s = rs$

Jedinými prostředky pro regulární výrazy jsou tři operace: *zřetězení, iterace a sjednocení.* Pro změnu priorit operátorů je možné použít kulaté závorky tak jako v běžných matematických výrazech.

Příklad 3.1. Mějme abecedu $\Sigma = \{a, b\}$ a jazyk $L = \{a^n b^n : 2 \leq n \leq 4\}$. Tento jazyk je konečný a tedy regulární, takže je možné ho popsat regulárním výrazem:
$aabb + aaabbb + aaaabbbb$

Příklad 3.2. Následující regulární výraz $(a + b)^* cc^*$ popisuje například řetězce *abbbac*, *c*, *bcc*, *bacccc*.

3.2 Konečný automat (KA)

Dalším prostředkem pro popis regulárních jazyků je konečný automat (FSM – Finite State Machine). Zde je jeho formální definice:

Definice 3.2. Konečný automat M je pětice $(Q, \Sigma, R, \mathrm{s}, F)$:

- Q je konečná množina stavů
- Σ je vstupní abeceda
- R je konečná množina pravidel ve tvaru pa $\rightarrow$ q, kde p,$q \in Q$; $a \in \Sigma \cup \{\varepsilon\}$
- s je počáteční stav, $s \in Q$
- F je množina koncových stavů, $F \subseteq Q$

Základem činnosti automatu je tzv. *vstupní páska*, která obsahuje řetězec nad Σ. Na začátku činnosti je počáteční stav automatu nastaven jako aktuální stav. V každém kroku automat odebere jeden symbol z pásky a vyhledá takové pravidlo, které má na své levé straně aktuální stav a tento symbol. Aktuálním stavem se stává stav na pravé straně pravidla. Činnost pokračuje čtením dalšího symbolu z pásky. Jakmile je přečtena celá páska, tak:

- pokud je aktuální stav koncový, řetězec, který byl na vstupní pásce, je přijímán automatem
- pokud není aktuální stav koncový, řetězec, který byl na vstupní pásce, není přijímán automatem

Při výše zmíněné činnosti ovšem mohou nastat některé nepříjemné situace:

- Pro aktuální stav a aktuální symbol neexistuje žádné pravidlo — v tomto případě se automat *zasekne* a řetězec na pásce není přijat.
- Pro aktuální stav a aktuální symbol existuje více pravidel — v tomto případě automat neví, které pravidlo použít. Z hlediska teorie se činnost automatu jako by rozdělí do všech možných větví podle použitelných pravidel.
- Z definice 3.2 je vidět, že čteným symbolem může být nejen symbol ze Σ, ale i ε. Což v praxi znamená, že automat může provést přechod z jednoho stavu do druhého bez přečtení symbolu z pásky.

Definice 3.3. Konfigurace — nechť M je konečný automat. *Konfigurace* konečného automatu M je řetězec $\chi = Q\Sigma^*$.

Konfigurace jednoduše označuje aktuální stav automatu a zbývající řetězec na vstupní pásce.

Definice 3.4. Přechod — nechť pax a qx jsou dvě *konfigurace*, kde p, $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ a $x \in \Sigma^*$. Nechť $r = pa \to q \in R$ je pravidlo. Potom M může provést *přechod* z pax do qx za použití r, zapsáno $pax \vdash qx\ [r]$ nebo jen $pax \vdash qx$

Definice 3.5. Sekvence přechodů — nechť χ je konfigurace. Automat M provede *nula* přechodů z χ do χ. Zapisujeme: $\chi \vdash^0 \chi\ [\varepsilon]$ nebo jen $\chi \vdash^0 \chi$.

Nechť $\chi_0, \ldots, \chi_n$ je sekvence přechodů pro $n \geq 1$ a $\chi_{i-1} \vdash \chi_i\ [r_i]$, $r_i \in R$ pro všechna $i = 1, \ldots, n$. Pak M provede n-přechodů z χ_0 do χ_n. Zapisujeme: $\chi_0 \vdash^n \chi_n\ [r_1 \ldots r_n]$ nebo jen $\chi_0 \vdash^n \chi_n$. Dále platí:

- Pro $n \geq 0$ je možné zapsat: $\chi_0 \vdash^* \chi_n$
- Pro $n \geq 1$ je možné zapsat: $\chi_0 \vdash^+ \chi_n$

Definice 3.6. Přijímaný jazyk — nechť M je konečný automat. Jazyk L přijímaný konečným automatem M je definován: $L(M) = \{w\text{: } w \in \Sigma^*,\ sw \vdash^* f,\ f \in F\}$

Neformálně řečeno, řetězec w je přijímán konečným automatem M, pokud existuje aspoň jedna sekvence přechodů, která končí v nějakém koncovém stavu.

Příklad 3.3. Nechť máme konečný automat podle definice 3.2, kde: $Q = \{A,B\}$, $\Sigma = \{a,b\}$, $R = \{Aa \to B,\ Ab \to A,\ Ba \to B,\ Bb \to A\}$, $s = A$, $F = \{B\}$. Řetězec $abba$ je přijímán automatem M, protože existuje sekvence přechodů $Aabba \vdash^4 B$, kde $B \in F$:

$$\begin{aligned} Aabba &\vdash Bbba && [Aa \to B] \\ Bbba &\vdash Aba && [Bb \to A] \\ Aba &\vdash Aa && [Ab \to A] \\ Aa &\vdash B && [Aa \to B] \end{aligned}$$

3.3 Reprezentace automatů

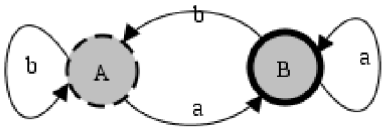
Jak je vidět z definice automatu 3.2 nebo i z příkladu 3.3, formální popis je z hlediska matematiky přesný, ale pro někoho nemusí být příliš stravitelný. Proto existují i jiné modely zobrazení, které jsou mnohem jasnější: *grafická* reprezentace a *tabulka*.

Tabulkou

Každý řádek tabulky označuje jeden stav automatu, každý sloupec označuje jeden symbol ze vstupní abecedy nebo ε. Průsečík aktuálního řádku (stavu) s aktuálním sloupcem (symbolem na pásce) protíná buňku tabulky, ve které je seznam cílových stavů. Vraťme se k příkladu 3.3. Zobrazení tohoto automatu tabulkou je na obrázku 3.1 (jedná se již o screenshot aplikace, která bude diskutována v dalších kapitolách):

	a	b
A	B	A
B	B	A

Obrázek 3.1: Automat zobrazený tabulkou



Obrázek 3.2: Automat zobrazený graficky

Graficky

Grafické zobrazení automatu je asi nejnázornější popis. Každý stav je reprezentován kolečkem se svým názvem. Koncové stavy mají zpravidla tlustší ohraničení než nekoncové. Každé pravidlo je reprezentováno šipkou ve směru přechodu. Proto, aby nebylo mezi stavy mnoho šipek, se ta pravidla, která mají stejný zdrojový a cílový stav, shlukují do jediné šipky, u které je vypsán seznam symbolů, pro které se přechod může provést.

Ohledně počátečního stavu, ten se zpravidla označuje tak, že do něj vede šipka, která nemá zdroj. Nicméně kvůli větší přehlednosti jsem v tomto projektu zvolil podle mě lepší značení a sice, že takový stav má čárkované ohraničení. Obrázek 3.2 je rovněž z implementované aplikace.

3.4 Typy automatů a jejich převody

V předešlých sekcích bylo vysvětleno, co je regulární výraz a konečný automat. Nyní tedy můžeme přejít na tu nejpodstatnější teoretickou část a sice převod regulárního výrazu na základní automat, jeho následné převody na další typy automatů a převod opačný.

3.4.1 Převod regulárního výrazu na základní konečný automat

Definice 3.7. Dva modely pro popis formálních jazyků jsou ekvivalentní (například regulární výrazy a konečné automaty), pokud popisují stejný jazyk.

Každý regulární výraz lze převést na *ekvivalentní* konečný automat. Základem je převést jednotlivé symboly výrazu na jednotlivé automaty. Nechť r je regulární výraz. Potom platí:

- $r = \emptyset$ je popsán automatem $M_\emptyset = (\{A\}, \{\}, \{\}, A, \{\})$
- $r = \varepsilon$ je popsán automatem $M_\varepsilon = (\{A\}, \{\}, \{\}, A, \{A\})$
- $r = a$ je popsán automatem $M_a = (\{A,B\}, \{a\}, \{Aa \rightarrow B\}, A, \{B\})$

Pro tvorbu složitějších automatů existují následující tři algoritmy, které korespondují se třemi možnými operátory regulárního výrazu.

Obrázek 3.3: Automaty po řadě $M_\emptyset$, M_ε a M_a

Zřetězení

Definice 3.8. Nechť máme regulární výrazy x a y a automaty $M_x = (Q_x, \Sigma_x, R_x, s_x, \{f_x\})$ a $M_y = (Q_y, \Sigma_y, R_y, s_y, \{f_y\})$, které je popisují. Potom pro regulární výraz $x.y$ existuje ekvivalentní automat $M_{x.y} = (Q_x \cup Q_y, \Sigma_x \cup \Sigma_y, R_x \cup R_y \cup \{f_x \to s_y\}, s_x, \{f_y\})$.

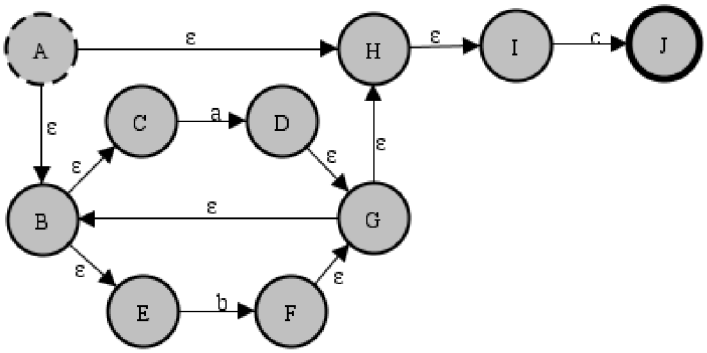
Sjednocení

Definice 3.9. Nechť máme stejné automaty jako z definice 3.8. Potom pro regulární výraz $x + y$ existuje ekvivalentní automat $M_{x+y} = (Q_x \cup Q_y \cup \{s,f\}, \Sigma_x \cup \Sigma_y, R_x \cup R_y \cup \{s \to s_x, s \to s_y, f_x \to f, f_y \to f\}, s, \{f\})$.

Iterace

Definice 3.10. Nechť máme automat M_x z definice 3.8. Potom pro regulární výraz x^* existuje ekvivalentní automat $M_{x^*} = (Q_x \cup \{s,f\}, \Sigma_x, R_x \cup \{s \to s_x, f_x \to f, s \to f, f_x \to s_x\}, s, \{f\})$.

Příklad 3.4. Máme regulární výraz $(a + b)^*c$. Základní automat, který ho popisuje, je na obrázku 3.4.



Obrázek 3.4: Základní automat pro regulární výraz $(a + b)^*c$

3.4.2 Konečný automat bez ε-pravidel

ε-pravidla jsou nežádoucí. Pokud totiž simulátor automatu na takové pravidlo narazí, neví, jestli má tento přechod provést nebo místo toho přečíst symbol z pásky a provést jiné pravidlo.

Definice 3.11. Nechť máme automat 3.2. Pro každý stav $p \in Q$ je definován ε-uzávěr(p): ε-uzávěr(p)=$\{q\colon p \vdash^* q\}$

Pro odstranění ε-pravidel je nejdříve nutné spočítat tzv. *ε-uzávěry* všech stavů automatu a s jejich pomocí aplikovat algoritmus 3.1. Takový ε-uzávěr(p) obsahuje všechny stavy, do kterých se stav p může dostat **jen** za pomocí ε-pravidel.

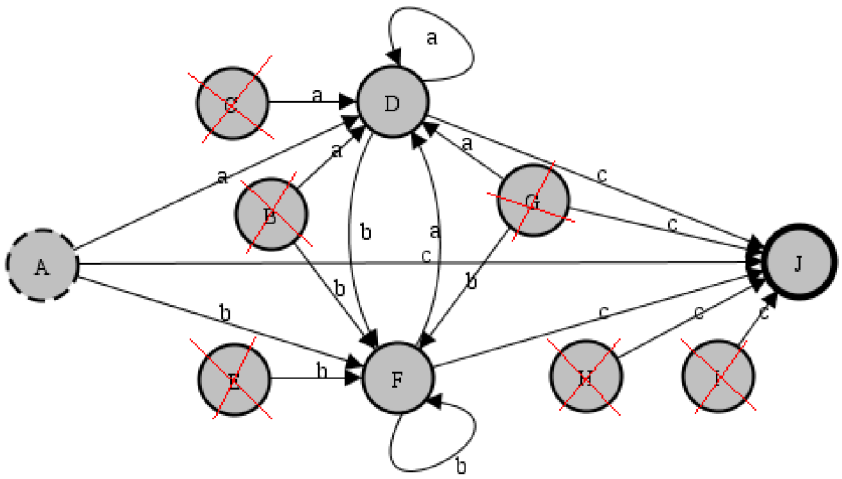
Příklad 3.5. Mějme automat 3.4. ε-uzávěry jeho stavů jsou: ε-uzávěr(A)=$\{A, B, C, E, H, I\}$, ε-uzávěr(B)=$\{B, C, E\}$, ε-uzávěr(C)=$\{C\}$, ε-uzávěr(D)=$\{B, C, D, E, G, H, I\}$, ε-uzávěr(E)=$\{E\}$, ε-uzávěr(F)=$\{B, C, E, F, G, H, I\}$, ε-uzávěr(G)=$\{B, C, E, G, H, I\}$, ε-uzávěr(H)=$\{H, I\}$, ε-uzávěr(I)=$\{I\ \}$, ε-uzávěr(J)=$\{J\}$

Vstup: KA $M = (Q, \Sigma, R, s, F)$
Výstup: KA bez ε-přechodů $M' = (Q, \Sigma, R', s, F')$

$R' = \emptyset$;
for each $p \in Q$ **do**
$\quad R' := R' \cup \{pa \to q\colon p'a \to q \in R,\ a \in \Sigma,\ p' \in \varepsilon\text{-uzávěr}(p),\ q \in Q\}$;
$F' := \{p\colon p \in Q, \varepsilon\text{-uzávěr}(p) \cap F \neq \emptyset\}$;

Algoritmus 3.1. Odstranění ε-pravidel

Po zjištění ε-uzávěrů všech stavů je možné aplikovat algoritmus 3.1. Pro automat 3.4 funguje převod takto: Například pro pravidlo $Ca \to D$ platí, že zdrojový stav C se nachází v ε-uzávěru stavu A, tudíž dojde k přidání pravidla $Aa \to D$. Pro pravidlo $Eb \to F$ platí, že stav E na levé straně pravidla je v ε-uzávěru stavu D, dojde tedy k přidání pravidla $Db \to F$. Výsledek převodu pro automat na obrázku 3.4 je na obrázku 3.5.



Obrázek 3.5: Automat bez ε-pravidel pro regulární výraz $(a + b)^*c$

Definice 3.12. Dostupné stavy — nechť máme automat M 3.2. Stav $q \in Q$ je dostupný, pokud existuje řetězec $w \in \Sigma^*$, pro který platí $sw \vdash^* q$. Jinak je stav q nedostupný.

Může se zdát, že automat je příliš složitý. Nicméně je vidět, že v automatu 3.5 se nacházejí stavy, do kterých se z počátečního stavu A nelze nikdy dostat. Takové stavy se nazývají *nedostupné* a je možné je bez obav smazat. Konkrétně v automatu 3.5 můžeme smazat stavy B,C,E,G,H,I.

Vstup: KA $M = (Q, \Sigma, R, s, F)$
Výstup: KA bez nedostupných stavů $M_t = (Q_t, \Sigma, R_t, s, F_t)$

$Q_0 = \{s\}; i := 0;$
repeat $i := i + 1;$
$\quad Q_i := Q_{i-1} \cup \{p\text{: } qa \to p \in R, a \in \Sigma, q \in Q_{i-1}\};$
until $Q_i = Q_{i-1};$
$Q_t := Q_i;$
$F_t := F \cap Q_t;$
$R_t := \{qa \to p\text{: } qa \to p \in R, q,p \in Q_t, a \in \Sigma\};$

Algoritmus 3.2. Odstranění nedostupných stavů

3.4.3 Deterministický konečný automat (DKA)

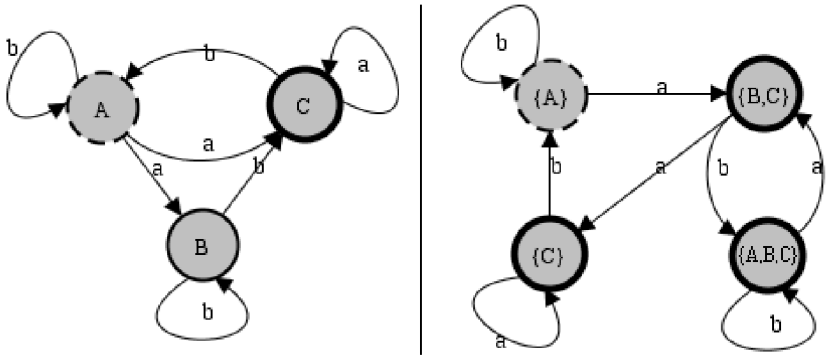
Algoritmus 3.3 se aplikuje v případě, že pro nějakou konfiguraci automatu existuje více jak jedno pravidlo, které je možné použít. Automat na obrázku 3.5 není pro účel vysvětlení vhodný, protože je již deterministický.

Determinizace používá princip slučování těch stavů, které nedeterminismus způsobují. Podmínkou je, aby automat již neobsahoval ε-pravidla. Existují 2 algoritmy. První algoritmus vytváří z množiny stavů Q potenční množinu (množina všech podmnožin bez $\emptyset$). Tento postup však generuje $2^{|Q|} - 1$ nových stavů, takže jejich počet rychle narůstá. Navíc mnohé z nich mohou být *nedostupné*, tudíž odstranitelné. Proto existuje druhý lepší algoritmus 3.3, který vytváří jen ty nové stavy, o kterých ví, že budou dostupné. Jeho výhodou není jen to, že nevytvoří nedostupné stavy z potenční množiny Q, ale odstraní i ty případné nedostupné stavy, které byly v automatu před začátkem převodu.

Vstup: KA bez ε-přechodů $M = (Q, \Sigma, R, s, F)$
Výstup: DKA bez nedostupných stavů $M_d = (Q_d, \Sigma, R_d, s_d, F_d)$

$s_d := \{s\}; Q_{new} := \{s_d\}; R_d := Q_d := F_d := \emptyset;$
repeat nechť $Q' \in Q_{new}; Q_{new} := Q_{new} - \{Q'\}; Q_d := Q_d \cup \{Q'\};$
$\quad$ **for each** $a \in \Sigma$ **do begin**
$\quad\quad Q'' := \{q : p \in Q', pa \to q \in R\};$
$\quad\quad$ **if** $Q'' \neq \emptyset$ **then** $R_d := R_d \cup \{\ Q'a \to Q''\ \};$
$\quad\quad$ **if** $Q'' \notin Q_d \cup \{\emptyset\}$ **then** $Q_{new} := Q_{new} \cup \{Q''\};$
$\quad$ **end**
$\quad$ **if** $Q' \cap F \neq \emptyset$ **then** $F_d := F_d \cup \{Q'\};$
until $Q_{new} = \emptyset;$

Algoritmus 3.3. Odstranění nedeterminismu



Obrázek 3.6: Nedeterministický automat a jeho zdeterminizovaná verze

Na obrázku 3.6 je zobrazený nedeterministický automat. Algoritmus se pokusím vysvětlit jednoduše. Vytvoříme si nový automat, ve kterém bude jen jeden stav (počáteční) s názvem $\{A\}$. Algoritmus postupně prochází stavy v novém automatu a stavy a pravidla v původním automatu.

- Stav $\{A\}$. Pravidla pro stav A a symbol a směřují do stavů B a C. Tudíž vytvořím stav s názvem $\{B, C\}$ a vytvořím pravidlo $\{A\}a \rightarrow \{B, C\}$.
  Pro stav A a symbol b vede šipka jen do stavu A, vytvořím pravidlo $\{A\}b \rightarrow \{A\}$.

- Stav $\{B, C\}$. Pro symbol a a stav B pravidlo neexistuje, pro stav C vede šipka do stavu C. Vytvořím tedy nový stav $\{C\}$ a pravidlo $\{B, C\}a \rightarrow \{C\}$.
  Pravidla pro symbol b a stav B směřují do stavů B,C. Pro symbol b a stav C šipka vede do stavu A. Vytvořím nový stav $\{A, B, C\}$ a pravidlo $\{B, C\}b \rightarrow \{A, B, C\}$.

- Stav $\{C\}$. Pro symbol a a stav C vede šipka do stavu C, vytvořím pravidlo $\{C\}a \rightarrow \{C\}$.
  Pro symbol b a stav C šipka vede jen do stavu A, vytvořím tak pravidlo $\{C\}b \rightarrow \{A\}$.

- Stav $\{A, B, C\}$. Pro symbol a a stavy A, B, C pravidla dohromady směřují do stavů B, C, vytvořím tedy pravidlo $\{A, B, C\}a \rightarrow \{B, C\}$.
  Pro symbol b a stavy A, B, C pravidla dohromady směřují do stavů A, B, C, vytvořím pravidlo $\{A, B, C\}b \rightarrow \{A, B, C\}$.

Koncovost resp. nekoncovost nových stavů je z algoritmu 3.3 jasná. Pokud je ve složeném stavu aspoň jeden stav koncový, tak bude koncový i složený stav.

Úplný deterministický konečný automat

Definice 3.13. Nechť M je konečný automat typu DKA. M je úplný, pokud pro každý $p \in Q$, $a \in \Sigma$ existuje **právě** jedno pravidlo ve tvaru $pa \rightarrow q$ pro nějaké $q \in Q$. Jinak M je neúplný.

Úplný DKA je takový automat, který se během své činnosti nemůže zaseknout. Na začátku sekce 3.2 bylo řečeno, že činnost automatu je zastavena a řetězec, který byl na pásce, není přijat. Pro zabránění zaseknutí je třeba vložit nekoncový stav simulující "past"a doplnit všechna pravidla tak, aby platila definice 3.13. Například námi vytvořený deterministický automat na obrázku 3.6 má již vytvořena všechna pravidla, past tedy není nutné vkládat a tento automat může být nazván úplným DKA.

Dobře specifikovaný konečný automat (DSKA)

Definice 3.14. Ukončující stav — nechť máme automat M 3.2 typu DKA. Stav $q \in Q$ je ukončující, pokud existuje řetězec $w \in \Sigma^*$, pro který platí: $qw \vdash^* f$, kde $f \in F$. Jinak je stav q neukončující.

Definice 3.15. Nechť automat M 3.2 je úplný DKA. Pak M je dobře specifikovaný KA (DSKA), pokud platí:

- Q nemá *nedostupné* stavy
- Q má **maximálně** 1 *neukončující* stav

Neukončující stav je takový stav, pro který neexistuje žádná sekvence přechodů do nějakého koncového stavu.

Vstup: KA $M = (Q, \Sigma, R, s, F)$
Výstup: KA bez neukončujících stavů $M_t = (Q_t, \Sigma, R_t, s, F)$

$Q_0 = F; i := 0;$
repeat $i := i + 1;$
$\quad Q_i := Q_{i-1} \cup \{q\text{: } qa \to p \in R, a \in \Sigma, p \in Q_{i-1}\};$
until $Q_i = Q_{i-1};$
$Q_t := Q_i;$
$R_t := \{qa \to p\text{: } qa \to p \in R, q,p \in Q_t, a \in \Sigma\};$

Algoritmus 3.4. Odstranění neukončujících stavů

DSKA smí obsahovat maximálně jeden neukončující stav proto, že může obsahovat stav "past", který je právě neukončující z toho důvodu, aby odebral všechny zbývající symboly z pásky, přestože je jasné, že řetězec stejně nebude přijat. Například úplný DKA na obrázku 3.6 nemá nedostupné stavy a nemá ani žádné neukončující stavy, takže tento automat může být nazván dobře specifikovaným. Dále je nutné dodat, že algoritmus 3.4 má jednu trhlinu. Neukončujícím stavem totiž může být i stav počáteční, takže je možné, že po dokončení algoritmu nebude mít nový automat M_t počáteční stav s, jak je napsáno, protože $s \notin Q_t$.

3.4.4 Minimální konečný automat

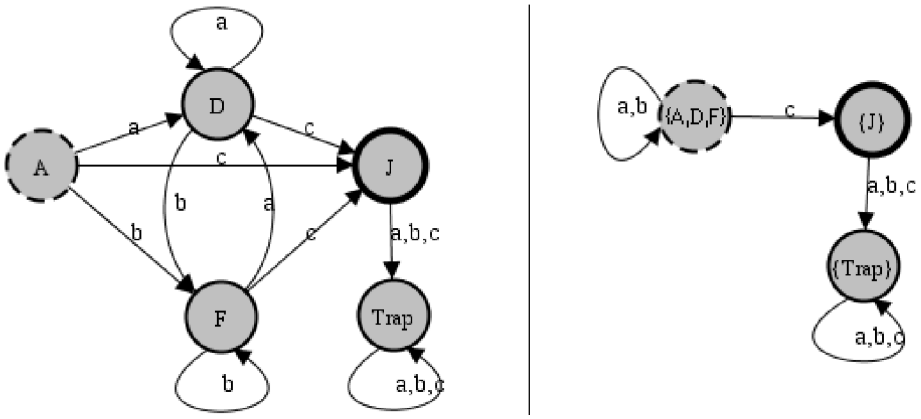
Definice 3.16. Nechť M je dobře specifikovaný automat 3.2 a nechť $p,q \in Q$, $p \neq q$. Stavy p a q jsou *rozlišitelné*, pokud existuje řetězec $w \in \Sigma^*$ takový, že: $pw \vdash^* p'$ a $qw \vdash^* q'$, kde p',$q' \in Q$ a $((p' \in F$ a $q' \notin F)$ nebo $(p' \notin F$ a $q' \in F))$.
Jinak stavy p a q jsou nerozlišitelné.

Minimalizace konečného automatu je poslední fází převodu. Algoritmus 3.5 slouží pro redukci počtů stavů tak, aby nedošlo ke změně přijímaného jazyka, a funguje tak, že slučuje ty stavy, které jsou tzv. *nerozlišitelné*. Dobré je dodat, že pro **každý** regulární jazyk existuje **právě jeden** minimální konečný automat. Nutnou podmínkou převodu je, aby vstupní automat byl typu DSKA.

Vstup: DSKA $M = (Q, \Sigma, R, s, F)$
Výstup: Minimální KA $M_m = (Q_m, \Sigma, R_m, s_m, F_m)$

$Q_m := \{\{p\colon p \in F\}, \{q\colon q \in Q - F\}\}$;
repeat
    if existuje $X \in Q_m, d \in \Sigma, X_1, X_2 \subset X$ takové, že:
    $X = X_1 \cup X_2, X_1 \cap X_2 = \emptyset$ **and**
    $\{q_1\colon p_1 \in X_1,\ p_1 d \to q_1 \in R\} \subseteq Q_1, Q_1 \in Q_m$,
    $\{q_2\colon p_2 \in X_2,\ p_2 d \to q_2 \in R\} \cap Q_1 = \emptyset$
    then
        rozštěp X na X_1 a X_2 v Q_m
until není možné provést další štěpení ;
$R_m := \{Xa \to Y\colon X, Y \in Q_m,\ pa \to q \in R,\ p \in X,\ q \in Y, a \in \Sigma\}$;
$s_m := X\colon s \in X$;
$F_m := \{X\colon X \in Q_m,\ X \cap F \neq \emptyset\}$;

Algoritmus 3.5. Minimalizace



Obrázek 3.7: Automat typu DSKA a jeho zminimalizovaná verze

Mějme automat z obrázku 3.7, který je stejný jako z obrázku 3.5, jen byly odstraněny nedostupné stavy a přidán stav *Trap*, který simuluje past. Automat je tak potřebného typu DSKA.

Algoritmus 3.5 funguje tak, že na začátku stavy rozdělí do množiny Q_m na koncové a nekoncové, protože ty jsou již triviálně rozlišitelné. Dále se bude provádět štěpení těch podmnožin, které obsahují *rozlišitelné* stavy. Algoritmus končí, pokud není možné provést další štěpení.

$$Q_m = \{\{J\}, \{A, D, F, Trap\}\}$$

Podmnožinu $\{J\}$ nemá smysl štěpit, protože obsahuje jen jeden prvek. Budeme štěpit podmnožinu $\{A, D, F, Trap\}$. Vytvoříme si seznamy 3.1 pravidel pro stavy z této podmnožiny podle jednotlivých symbolů abecedy.

Pro symbol a všechna pravidla směřují do stavů D a $Trap$, která se nacházejí ve stejné podmnožině, tedy štěpení neproběhne. Pro symbol b je situace obdobná, jen se stavy F a

a	b	c
$Aa \to D$	$Ab \to F$	$Ac \to J$
$Da \to D$	$Db \to F$	$Dc \to J$
$Fa \to D$	$Fb \to F$	$Fc \to J$
$Trapa \to Trap$	$Trapb \to Trap$	$Trapc \to Trap$

Tabulka 3.1: Štěpení podmnožiny $\{A, D, F, Trap\}$

$Trap$. Pro symbol c stavy A, D a F směřují do stavu J, což je podmnožina $\{J\}$. Avšak pro stav $Trap$ pravidlo směřuje do stavu $Trap$, což je ovšem podmnožina $\{A, D, F, Trap\}$. Dojde tedy k rozštěpení na podmnožiny $\{A, D, F\}$ a $\{Trap\}$. Nová množina Q_m vypadá následovně:

$$Q_m = \{\{J\}, \{A, D, F\}, \{Trap\}\}$$

a	b	c
$Aa \to D$	$Ab \to F$	$Ac \to J$
$Da \to D$	$Db \to F$	$Dc \to J$
$Fa \to D$	$Fb \to F$	$Fc \to J$

Tabulka 3.2: Štěpení podmnožiny $\{A, D, F\}$

Podmnožinu $\{Trap\}$ opět nemá smysl štěpit, takže na řadě je podmnožina $\{A, D, F\}$ 3.2. Pro symboly a, b a c všechna pravidla směřují do stavů po řadě D, F a J, což jsou vždy stejné podmnožiny po řadě $\{A, D, F\}$, $\{A, D, F\}$ a $\{J\}$. K rozštěpení podmnožiny $\{A, D, F\}$ tedy nedojde a celý algoritmus končí.

Nové stavy automatu tedy jsou $\{A, D, F\}$, $\{Trap\}$ a $\{J\}$. Koncové/nekoncové stavy budou takové, které jsou složeny jen ze samých koncových/nekoncových stavů původního automatu, tedy stav $\{J\}$ bude koncový, stavy $\{A, D, F\}$ a $\{Trap\}$ budou nekoncové.

Vytvoření pravidel je již snadné, protože každý původní stav složeného stavu má pravidlo, které směřuje vždy do stejného složeného stavu. Například:

$Ab \to F$ $Db \to F$ $Fb \to F$	$\{A, D, F\}b \to \{A, D, F\}$

Tabulka 3.3: Vytvoření pravidla pro symbol b a stav $\{A, D, F\}$

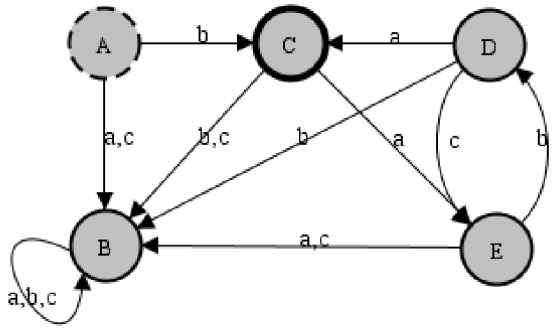
3.4.5 Převod konečného automatu na regulární výraz

Existuje několik způsobů resp. algoritmů, jak tento převod provést. Zde rozeberu princip, který je popsán v článku [2] od Janusz A. Brzozowski, který se zabývá odvozeninami regulárních výrazů. Výhodou této metody je, že vytváří celkem slušné výrazy i pro rozsáhlejší automaty (řada výjimek by se samozřejmě našla) a je použitelná pro libovolný automat. Jedná se o metodu, při které je vytvořena soustava rovnic, která musí být vyřešena. Pro každý stav konečného automatu je vytvořena jedna rovnice. Tímto dostaneme soustavu

n-rovnic ve tvaru

$$\begin{aligned} R_1 &= a_1R_1 + a_2R_2 + \dots \\ R_2 &= a_1R_1 + a_2R_2 + \dots \\ &\vdots \\ R_{n-1} &= a_1R_1 + a_2R_2 + \dots + \varepsilon \\ R_n &= a_1R_1 + a_2R_2 + \dots + \varepsilon \end{aligned}$$

kde n je počet stavů automatu a $a_1, a_2 \in \Sigma$. Je nutné ještě zdůraznit, že symbol $+$ je sjednocení a například zápis a_1R_1 označuje zřetězení a_1 a R_1. Soustava je vytvořena následovně. Do každé rovnice R_i, která koresponduje se stavem q_i, se sjednotí takové zřetězení aR_j, pokud existuje pravidlo $q_ia \to q_j$. Pokud je navíc stav q_i koncový, je ještě sjednoceno ε. Cílem je vyřešit soustavu pro rovnici, která popisuje počáteční stav automatu.



$$\begin{aligned} R_A &= aR_B + bR_C + cR_B \\ R_B &= aR_B + bR_B + cR_B \\ R_C &= aR_E + bR_B + cR_B + \epsilon \\ R_D &= aR_C + bR_B + cR_E \\ R_E &= aR_B + bR_D + cR_B \end{aligned}$$

Obrázek 3.8: Konečný automat a jeho popis soustavou rovnic

Při řešení je možné provádět vytýkání či naopak roznásobování, ale samozřejmě jen tak, aby nedošlo ke změně významu rovnice/výrazu. Pojmem roznásobení se v tomto případě nemyslí roznásobení ve smyslu součinu, ale ve smyslu zřetězení, kde pořadí symbolů je zásadní. Totéž platí pro vytýkání.

Příklad 3.6. Základní vlastnosti regulárních výrazů, které se při úpravách hodí:

$$\begin{aligned} \varepsilon^* &= \varepsilon \\ \emptyset^* &= \varepsilon \\ a\varepsilon &= \varepsilon a = a \\ a\emptyset &= \emptyset a = \emptyset \\ a + b + \emptyset &= a + b \\ (a + b + \varepsilon)^* &= (a + b)^* \end{aligned}$$

Z principu tvorby soustavy je snad jasné, že v rovnicích mohou vzniknout smyčky. Pro takové případy existuje speciální transformace. Pokud se nám podaří dostat rovnici do tvaru 3.1, je možné tuto rovnici přímo převést na tvar 3.2.

$$X = AX + B \tag{3.1}$$

$$X = A^*B \tag{3.2}$$

Mějme automat z obrázku 3.8 a soustavu, která ho popisuje. Rovnice nejsou indexovány čísly ale přímo názvy stavů, které popisují. Našim úkolem je vyřešit rovnici R_A pro počáteční stav A. Nejlepší ale bude, když nejdříve vyřešíme rovnici R_B, protože se vyskytuje v každé další rovnici.

$$\begin{aligned} R_B &= aR_B + bR_B + cR_B \\ R_B &= (a+b+c)R_B + \emptyset \\ R_B &= (a+b+c)^*\emptyset \\ R_B &= \emptyset \end{aligned} \tag{3.3}$$

Aby bylo možné po vytknutí aplikovat transformaci v R_B, je třeba do rovnice přidat prázdný výraz $\emptyset$ ($B = \emptyset$). To si můžeme dovolit, viz. příklad 3.6. Pro vyřešení rovnice R_A je nutné vyřešit R_C, pro vyřešení R_C je potřeba vyřešit R_E a pro R_E je třeba spočítat R_D. Mezi rovnicemi R_E a R_D je smyčka (vzájemně na sebe odkazují). Do rovnice R_E dosadíme rovnice R_D a R_B, upravíme a transformujeme.

$$\begin{aligned} R_E &= aR_B + bR_D + cR_B \\ R_E &= aR_B + b(aR_C + bR_B + cR_E) + cR_B \\ R_E &= a\emptyset + b(aR_C + b\emptyset + cR_E) + c\emptyset \\ R_E &= b(aR_C + cR_E) \\ R_E &= baR_C + bcR_E \\ R_E &= bcR_E + baR_C \\ R_E &= (bc)^*baR_C \end{aligned} \tag{3.4}$$

Výsledek rovnice R_E sice obsahuje nevyřešenou rovnici R_C, to ale vůbec nevadí. Rovnici R_E a R_B dosadíme do rovnice R_C a transformujeme.

$$\begin{aligned} R_C &= aR_E + bR_B + cR_B + \varepsilon \\ R_C &= a((bc)^*baR_C) + b\emptyset + c\emptyset + \varepsilon \\ R_C &= a(bc)^*baR_C + \varepsilon \\ R_C &= (a(bc)^*ba)^*\varepsilon \\ R_C &= (a(bc)^*ba)^* \end{aligned} \tag{3.5}$$

A nakonec dosadíme rovnici R_C do cílové rovnice R_A.

$$\begin{aligned} R_A &= aR_B + bR_C + cR_B \\ R_A &= a\emptyset + b((a(bc)^*ba)^*) + c\emptyset \\ R_A &= b(a(bc)^*ba)^* \end{aligned} \tag{3.6}$$

Rovnice R_A je tak vyřešena. Konečný automat na obrázku 3.8 může být popsán regulárním výrazem $b(a(bc)^*ba)^*$.

Kapitola 4

Implementace programu

Celá aplikace je implementována v jazyce Java. Nutnou podmínkou je, aby byl projekt webově dostupný, což je diskutováno v následující kapitole 5. V této kapitole proberu způsob implementace, popis těch nejdůležitějších částí, balíčků a algoritmů.

Projekt je složen z několika balíčků, které jsou logicky uspořádány podle jejich účelu. Strom balíčků vypadá takto:

- *projekt*
  - *automat*
  - *conversion*
  - *graphics*
    - *view*
  - *syntax*

4.1 Konečný automat

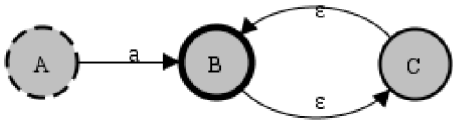
Balíček *automat* implementuje funkcionalitu konečného automatu ve třídě *Automat*. Jeho součástí jsou třídy, které definují stav automatu (třída *State*), symbol z abecedy (třída *Symbol*) a pravidlo (třída *Rule*).

Implementace třídy automatu je velmi podobná formální definici automatu. Všechny stavy a abeceda jsou uchovávány v seznamu. Třída pro symbol pouze zapouzdřuje objekt třídy *Character*, plus metody pro tisk escape sekvencí. Informace říkající, jestli je stav koncový či nekoncový, je primárně uchovávána přímo v objektu stavu, aby byla snadno dostupná. Současně s tím je rovněž udržován seznam koncových stavů. Pravidla jsou však uložena jinak. Jejich udržování ve formě seznamu je nevýhodné, například kvůli simulaci, kdy je nutné vyhledávat jen taková pravidla, která mají na své levé straně určitý stav a určitý symbol. Bylo by tak potřeba procházet seznam pravidel sekvenčně, což je neefektivní. Každé pravidlo je uloženo přímo v takovém objektu stavu, který se nachází na levé straně pravidla. Zbývající informace o pravidlu (symbol a cílový stav) je uložena do hašované mapovací tabulky, kde se mapuje symbol na seznam cílových stavů. Příkladem mohou být tato tři pravidla $Aa \rightarrow B$, $Aa \rightarrow A$, $Ab \rightarrow B$, která jsou zaznamenána v mapovací tabulce v objektu stavu A takto: $a \rightarrow \{A, B\}$, $b \rightarrow \{B\}$. Vnitřní implementace uchovávání pravidel není však pro vnější třídy viditelná. Všechny operace s pravidly vždy pracují jen s třídou *Rule*.

Test řetězce

Součástí funkcionality programu je schopnost zjištění posloupnosti přechodů z počátečního stavu pro libovolný řetězec. Pokud je řetězec přijímán daným automatem, je vrácen seznam pravidel, která je nutné aplikovat, aby automat pro daný řetězec skončil v **nějakém** koncovém stavu.

Test může být proveden pro jakýkoliv řetězec (při nalezení symbolu, který se nenachází v abecedě, je řetězec odmítnut) a pro jakýkoliv druh automatu. Algoritmus funguje tak, že z počátečního stavu prochází celý stavový prostor a skončí, jakmile najde první cestu vedoucí do koncového stavu, nebo až projde všechny možnosti. Jelikož zadaný řetězec musí mít vždy konečný počet symbolů, prohledání stavového prostoru tedy musí rovněž někdy skončit, protože při každém zanoření se vždy odebere jeden symbol z řetězce. Výjimku však mohou tvořit ε-pravidla, která jsou koncipována jako na obrázku 4.1. Pokud bychom chtěli otestovat na tento automat jakýkoliv řetězec z jazyka aaa^*, dojde k zacyklení mezi stavy B a C, aniž by byly čteny nějaké symboly. Algoritmus tak nikdy neskončí. Pro tyto případy jsem implementoval ochranu, která zabrání zacyklení v těchto smyčkách.



Obrázek 4.1: Konečný automat s problémem zacyklení

Při každém navštívení nějakého stavu se do speciální mapovací tabulky uloží aktuální index pozice v testovaném řetězci. Pokud někdy poté dojde k pokusu navštívit stav, ve kterém již algoritmus jednou byl (tedy existuje záznam v mapovací tabulce), dojde k porovnání indexu z mapovací tabulky s indexem aktuálního symbolu řetězce. Pokud platí $index_{tab} < index_{actual}$ pro $a = \varepsilon$ resp. $index_{tab} < index_{actual} + 1$ pro $a \in \Sigma$, kde a je symbol, pro který se má provést přechod, dojde k navštívení požadovaného stavu a v případě, že $a \in \Sigma$, dojde k navýšení aktuálního indexu v řetězci o 1. Pokud podmínka neplatí, požadovaný stav je ignorován.

Příklad 4.1. Mějme obrázek 4.1 a řetězec aa, který chceme otestovat. Při prohledávání stavového prostoru algoritmus dojde až do stavu C. Obsah mapovací tabulky bude $A = 0$, $B = C = 1$. Při pokusu aplikovat pravidlo $C \rightarrow B$ dojde k porovnání $1 < 1$, což neplatí, tedy cílový stav B je ignorován. Jiné možnosti ve stavovém prostoru nejsou, tedy řetězec aa bude odmítnut.

4.2 Regulární výraz na konečný automat

Pro potřeby převodu regulárního výrazu na základní konečný automat jsem implementoval balíček *syntax*, jehož účelem je zpracování regulárního výrazu na lexikální (třída *Lex*) a syntaktické (třída *Postfix*) úrovni. Pro obě zmíněné úrovně rovněž existují dvě třídy výjimek.

Lexikální analyzátor je v tomto projektu velmi jednoduchý, neboť pouze čte jednotlivé symboly z výrazu a předává je syntaktickému analyzátoru ve formě lexémů (třída *Lexem*). Analyzátor umožňuje zadávání základních escape sekvencí pro bílé znaky a pro speciální znaky regulárního výrazu. Podrobnější informace naleznete ve webové nápovědě pro ovládání

programu.

Pro syntaktickou analýzu bylo možné využít některé typy analýz, které se běžně používají, jako například precedenční nebo složitá LR analýza. Vzhledem k tomu, že regulární výraz je velmi jednoduchý, protože obsahuje jen tři operátory, levou a pravou závorku a samotné symboly, rozhodnul jsem se použít asi nejjednodušší způsob, jak vyhodnotit výraz, a sice přes postfixovou notaci. Existují tři způsoby/notace zápisu výrazů: prefixová, infixová (běžná) a postfixová.

Příklad 4.2. Mějme součet dvou čísel 2 a 5. Prefixový zápis vypadá takto: *+ 2 5*, infixový zápis takto: *2 + 5* a postfixový zápis takto: *2 5 +*.

Výhodou postfixové notace je to, že neobsahuje závorky, i když například v běžné infixové notaci být musí kvůli definici priorit. Další a nejzásadnější výhodou této notace je snadné vyhodnocení s pomocí datového modelu zasobníku. V programu je regulární výraz vždy zadáván infixovou formou, neboť je běžně používána. Tento výraz je následně převeden na postfixovou formu pomocí algoritmu 4.1.

Zpracovávej vstupní infixový výraz zleva doprava a vytvářej výstupní postfixový výraz:

1. Je-li zpracovávanou položkou operand, přidej ho na konec výstupního výrazu
2. Je-li zpracovávanou položkou levá závorka, vlož ji na vrchol zásobníku
3. Je-li zpracovávanou položkou operátor, vlož ho na vrchol zásobníku v případě, že platí jedna z následujících podmínek:
   - zásobník je prázdný
   - na vrcholu zásobníku je levá závorka
   - na vrcholu zásobníku je operátor s nižší prioritou
4. Je-li zpracovávanou položkou operátor a na vrcholu zásobníku je operátor se stejnou nebo vyšší prioritou, odstraň operátor z vrcholu zásobníku, vlož ho na konec výstupního výrazu a přejdi na bod 3
5. Je-li zpracovávanou položkou pravá závorka, odstraňuj položky z vrcholu zásobníku a vkládej je na konec výstupního výrazu, dokud nenarazíš na levou závorku
6. Není-li ze vstupního výrazu již co číst, odstraňuj položky z vrcholu zásobníku a vkládej je na konec výstupního výrazu, až zásobník zcela vyprázdníš

Algoritmus 4.1. Převod z infixové notace do postfixové

Po provedení algoritmu 4.1 je nutné nově vzniklý výraz vyhodnotit algoritmem 4.2, který je implementován ve třídě *Phase1* v balíčku *conversion*. Vyhodnocení výrazu je provedeno přesně podle algoritmu, jen v tomto případě je položkou myšlen objekt automatu. Tedy v bodu 1 algoritmu 4.2 se z daného symbolu z postfixové notace vytvoří základní miniautomat a ten se teprve vloží na vrchol zásobníku. Zpracovávání operátorů probíhá podle bodu 2, tedy slučováním automatů podle typu operace tak, jak bylo popsáno v části 3.4.1. Po dokončení zůstane na vrcholu zásobníku jeden výsledný automat, který popisuje vstupní

regulární výraz.

Příklad 4.3. Mějme infixový regulární výraz $(a+b)^*.c.(d+e)^*$. Postfixová notace tohoto výrazu je: $a\ b\ +\ ^*\ c\ .\ d\ e\ +\ ^*\ .$

Zpracovávej vstupní postfixový výraz zleva doprava:

1. Je-li zpracovávanou položkou operand, vlož ho na vrchol zásobníku
2. Je-li zpracovávanou položkou operátor, vyjmi z vrcholu zásobníku tolik operandů, *kolika*-nární operátor je.

   Pro regulární výraz platí:

   - zřetězení (.) – binární operátor (2 operandy)
   - sjednocení (+) – binární operátor (2 operandy)
   - iterace (*) – unární operátor (1 operand)

   Proveď operaci s operátorem nad danými operandy a výsledek vlož zpět na vrchol zásobníku.
3. Není-li ze vstupního výrazu již co číst, výsledek je uložen na vrcholu zásobníku

---

Algoritmus 4.2. Vyhodnocení postfixového výrazu

4.3 Ostatní typy automatů

Převody na všechny typy automatů jsou implementovány v balíčku *conversion*, jehož obsah vypadá takto:

— *conversion*
  — *Phase1*
  — *Phase2*
  — *Phase3*
  — *Phase4*
  — *ToRegExp*

Třída *Phase1* implementuje převod regulárního výrazu na základní konečný automat, který byl popsán v části 4.2. Třída *Phase2* odstraňuje z automatu ε-pravidla, třída *Phase3* odstraňuje nedeterminismus, přičemž zdrojový automat nesmí obsahovat ε-pravidla. Třída *Phase4* provádí minimalizaci, přičemž zdrojový automat musí být typu DSKA. Třída *ToRegExp* převádí konečný automat na regulární výraz, viz. část 4.4.

Všechny třídy *PhaseX* dědí z třídy *Automat*, takže disponují stejnou funkcionalitou manipulace se stavy, abecedou a pravidly popsané v části 4.1. Dále obsahují metodu *convert*, která provede příslušný typ převodu.

4.4 Konečný automat na regulární výraz

V části 3.4.5 jste se dozvěděli, že asi nejlepší způsob získání regulárního výrazu z konečného automatu je konečný automat převést na soustavu rovnic, která je poté vyřešena. Tento převod implementuje třída *ToRegExp*.

Prvním krokem převodu je získání soustavy rovnic z konečného automatu. Základem této soustavy a později i výsledného regulárního výrazu je vnitřní třída *Element*, která reprezentuje základní stavební jednotku. Každý element musí být některý z těchto typů:

- *CONCATENATION* — element popisuje zřetězení vnořených elementů (2 a více)
- *UNION* — element popisuje sjednocení vnořených elementů (2 a více)
- *ITERATION* — element popisuje iteraci **právě** jednoho vnořeného elementu
- *CHAR* — element popisuje koncový symbol regulárního výrazu

Každý element (mimo typ *CHAR*) obsahuje seznam zanořených elementů. Elementy tak vždy dohromady tvoří stromovou, rekurzivně implementovanou strukturu, která ovšem může obsahovat cyklus, protože v soustavě rovnic mohou existovat rovnice, které na sebe vzájemně odkazují.

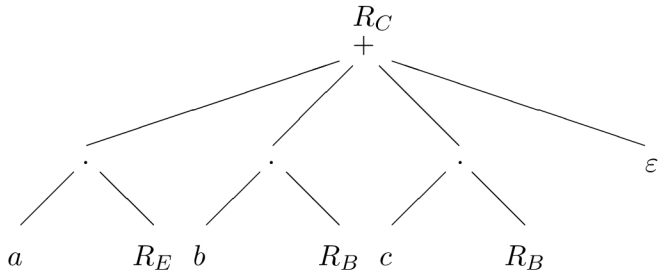
Celá soustava je vyřešena rekurzivním sestupem počínaje elementem popisujícím počáteční stav automatu, přičemž při vynořování z aktuálního elementu se provede několik typů úprav, tranformace (jen u elementu typu rovnice) a pokud je to možné, redukce resp. vymazání zbytečných elementů. Po dokončení rekurzivního sestupu je soustava vyřešena. Poté je možné pro element, který popisuje počáteční stav, zavolat metodu *toString*, která daný element vytiskne do formy řetězce.

Rekurze se při zpracování aplikuje do co nejvyšší hloubky, přičemž je ovšem nutné zabránit zacyklení. Takový stav může nastat, pokud například rovnice obsahuje odkaz samu na sebe nebo odkaz na rovnici, ve které se již program nacházel. V takovém případě se rovnice prohlásí za *nevyřešenou*, přičemž pro takovou rovnici navíc platí:

- při rekurzivním sestupu je ignorována (nedojde k zanoření)
- nevztahují se na ní úpravy a zjednodušování, protože odkaz na cizí rovnici musí zůstat nezměněn, aby později mohlo dojít k transformaci

Rovnice bude prohlášena za *vyřešenou* až poté, co jsou zpracovány všechny její vnitřní elementy.

Příklad 4.4. Mějme rovnici $R_C = aR_E + bR_B + cR_B + \varepsilon$ z obrázku 3.8. Stromová hierarchie elementů vypadá takto:



Každý uzel stromu popisuje element, který je typu zřetězení, iterace nebo sjednocení. Listy stromu popisují buď koncový symbol nebo odkaz na další rovnici soustavy.

Jakmile jsou vyhodnoceny všechny vnitřní elementy aktuálně zpracovávaného elementu, nad aktuálním elementem se provede několik druhů operací přesně v tomto pořadí:

1. Úprava elementu na základě typů vnitřních elementů či na jejich počtu
2. Vytknutí
3. Roznásobení
4. Odstranění duplicitních elementů ve sjednocení
5. Zjednodušení elementu
6. Transformace

Za zmínku stojí snad jen operace 5, která provádí některá zjednodušení, aby výsledný regulární výraz byl co nejkratší.

- Element ve tvaru $aa^* + \varepsilon + \ldots$ nebo $a^*a + \varepsilon + \ldots$ je zjednodušen na $a^* + \ldots$
- Element ve tvaru $aa^* + a + \ldots$ nebo $a^*a + a + \ldots$ je zjednodušen na $aa^* + \ldots$
- Element ve tvaru $a^* + \varepsilon + \ldots$ je zjednodušen na $a^* + \ldots$
- Element ve tvaru $(\varepsilon + \ldots)^*$ je zjednodušen na $(\ldots)^*$

Příklad 4.5. Nechť máme regulární výraz $a^*a + a + b + bb^* + \varepsilon + c$. Na základě výše uvedených úprav je možné tento výraz zjednodušit na $a^* + b^* + c$.

4.5 XML

Formát XML je používán pro ukládání konečných automatů. K dispozici jsou metody *loadXML*/*saveXML* pro načtení/uložení konečného automatu, které jsou implementovány ve třídě *AutomatStatus* v balíčku *graphics*. Pro parsování XML jsem využil vestavěné balíčky Java API.

Příklad 4.6. Nechť máme pravý konečný automat z obrázku 3.7. Část jeho popisu v XML vypadá takto:

```
<?xml version = "1.0" encoding = "utf-8"?>
<automat name = "Minimální: Nový automat">
    <sigma value = "abc"/>
    <states>
        <state name = "{A,D,F}" final = "0" posX = "60" posY = "60"/>
        <state name = "{J}" final = "1" posX = "160" posY = "60"/>
        <state name = "{Trap}" final = "0" posX = "157" posY = "163"/>
    </states>
    <start name = "{A,D,F}"/>
    <rules>
        <rule source = "{A,D,F}" char = "b" target = "{A,D,F}"/>
        <rule source = "{A,D,F}" char = "c" target = "{J}"/>
    </rules>
</automat>
```

Abeceda automatu je zadávána prostřednictvím povinného atributu *value* značky *sigma*.

Stavy jsou zadávány značkou *state*. Počet definovaných stavů může být ≥ 0. Atributy *posX* a *posY* jsou nepovinné a definují pozici stavu na ploše v grafickém zobrazení automatu. Atribut *final* udává koncovost stavu (*0 = nekoncový, 1 = koncový*, jiné hodnoty nejsou povoleny). Atribut *name* udává název stavu, jehož délka musí být minimálně 1. Více stavů se stejným názvem není povoleno.

Počáteční stav je zadán značkou *start* a povinným atributem *name*, který buď musí obsahovat název existujícího stavu nebo může být prázdný, což značí, že počáteční stav není definován. V takovém případě program zvolí jako počáteční stav ten, který je v abecedním pořadí na začátku.

Pravidla jsou definována značkou *rule*, přičemž všechny atributy jsou povinné. Atributy *source* (zdrojový stav) a *target* (cílový stav) musejí obsahovat název existujícího stavu. Atribut *char* (symbol pravidla) může být prázdný (označuje symbol ε) nebo může obsahovat **právě** jeden symbol, který ovšem musí být definován v abecedě (tedy v atributu *value* značky *sigma*). Počet pravidel smí být ≥ 0.

Pro zadání speciálních znaků jazyka XML je samozřejmě možné použít *znakové entity* (* < > apod.*), jejichž délka je 1, takže například lze zadat *char = "<"*, pokud je tento znak definován v abecedě.

4.6 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní řeší balíček *graphics*. K jeho vytvoření jsem využil vestavěné balíčky *swing* a *awt*.

4.6.1 Základní koncepce

Hlavní okno programu se skládá ze třech panelů. V horním panelu je možné vybírat otevřené automaty a způsob zobrazení automatu. V prostředním panelu se zobrazuje samotný automat takovým způsobem, jak je vybráno v horním panelu. Dolní panel slouží ke spuštění a řízení simulace otevřeného automatu. Součástí okna je i hlavní nabídka, která se skládá ze dvou roletek pro manipulaci s automaty a pro řízení akcí nad konkrétním automatem.

AutomatStatus je třída, která uchovává informace o právě otevřeném automatu. Součástí této třídy je i samotný automat (třída *Automat*), funkcionalita práce s XML (načítání, ukládání) a řízení simulace. V rámci inicializace objektu této třídy jsou vytvořeny i tři další objekty (tříd *Formal*, *Table* a *Graphic*), které definují zobrazení automatu třemi způsoby:

- formálně (třída *Formal*) — zobrazen ve formě výpisu množin podle formální definice
- tabulkou (třída *Table*) — zobrazen ve formě tabulky
- graficky (třída *Graphic*) — zobrazen formou koleček a šipek

Všechny výše uvedené třídy se nacházejí v balíčku *view* a jsou odvozeny z abstraktní třídy *View*, která definuje společné důležité rysy zobrazení automatu. Na základě této struktury je teoreticky možné přidat další způsoby zobrazení jen tak, že bude vytvořena nová třída zděděná z *View* a jen lehce upraven obsah třídy *AutomatStatus* a *TopPanel* tak, aby s nově vytvořenou třídou spolupracovaly.

4.6.2 Grafické zobrazení

Ze všech třech módů zobrazení je nejprospěšnější ten grafický, neboť jako jediný disponuje funkcionalitou modifikace automatu přímo, tedy přidávávání/mazání stavů/symbolů/pravidel. Učinil jsem tak proto, že tyto operace je v tomto módu snadné implementovat. Bylo žádoucí, aby program byl pro ovládání jednoduchý a neobsahoval příliš mnoho ovládacích prvků. Všechny výše uvedené operace jsou prováděny pomocí popup menu pravým tlačítkem myši resp. pomocí *Drag&Drop* (táhni a pusť), což by se v ostatních režimech těžko provádělo.

V tomto módu bylo nutné vyřešit rozložení stavů na ploše. Při vytváření všech druhů automatů z regulárního výrazu totiž neexistuje žádná informace o souřadnicích stavů, takže je nutné je **nějak** spočítat. V aktuální verzi programu jsou stavy rozloženy do virtuální mřížky, která v ideálním případě tvoří čtverec.

4.6.3 Uložení do GIF souboru

Součástí funkcionality programu je i možnost uložit automat do obrázku ve formátu GIF (Graphics Interchange Format). Pro toto jsem využil volně šířitelnou třídu *AnimatedGifEncoder* [4], která řeší i tvorbu animace.

Uložení automatu může proběhnout dvěma způsoby. V případě, že není spuštěna simulace, do souboru je uloženo aktuální zobrazení automatu. Pokud je simulace zapnuta, do souboru jsou uloženy všechny snímky od počátku simulace až do **aktuálního** stavu ve formě animace s rychlostí 1 snímek za 1 vteřinu.

Celá činnost je zajištěna tak, že při každém kroku simulace (tlačítko >>> v dolním panelu) je vytvořen jeden snímek jak automatu tak i dolního panelu. Všechny snímky jsou úkládány do dvou seznamů a jsou připraveny k použití. Při vypnutí simulace se oba seznamy zruší. Tento způsob je jednoduchý, ale zahrnuje v sobě ten problém, že jsou uchovávány všechny snímky automaticky a v případě, že uživatel nehodlá animaci uložit, i zbytečně.

Z výše uvedených důvodů a také proto, že Java využívá automatizovanou správu paměti (Garbage Collecting), která není příliš efektivní, se může stát, že animaci nebude možné uložit kvůli nedostatku paměti. Řešením je zmenšení velikosti plochy, kterou automat zabírá, a také neprovádět simulace o velkém počtu kroků.

Kapitola 5

Webová dostupnost

Podmínkou bakalářského projektu je jeho webová dostupnost, čímž je myšlen přístup k aplikaci přes webový prohlížeč. Nejlépe použitelné jsou technologie *Java Applet* (část 5.1) a Java Web Start (část 5.2). Obě jsou přístupné přes webové stránky generované PHP skriptem *index.php*.

Adresářová a souborová struktura potřebná pro správný webový chod vypadá takto:

- *root*
  - *classes*
    - *javax*
    - *libraries*
    - *projekt*
  - *index.php*
  - *projekt.jar*
  - *schema.xml*
  - *launch.jnlp*

classes je adresář s přeloženým projektem, který byl získán při překladu (viz. příloha). Nutně musí obsahovat složky *javax*, *libraries*, *projekt*. Soubor *schema.xml* jen popisuje formát ukládání konečných automatů do XML a při webovém přístupu nemá žádný význam.

5.1 Java Applet

Java Applet je Java program, který nemá vlastní okno a běží v kontextu webového prohlížeče. Do html stránky je možné ho vložit značkou **applet**. Pro výše uvedenou adresářovou strukturu má značka tvar:

```
<applet name = "applet" codebase = "classes" code = "projekt/MainApplet.class"
        archive = "../projekt.jar" width = "800" height = "550">
</applet>
```

Důležité je, že zatímco pro běžné spuštění je hlavní třídou *Main*, u appletu je spouštěcí třídou *MainApplet*, která dědí z třídy *JApplet*. Zásadní rozdíl spočívá v tom, že tato třída neobsahuje statickou metodu *main* (entry point), ale obsahuje čtyři metody

- **init** — je zavolána vždy, když je požadováno spuštění appletu

- **start** — je zavolána ihned po metodě **start** a také vždy, když se uživatel vrátí na stránku s appletem, kterou předtím opustil
- **stop** — je zavolána vždy, když uživatel opustí stránku s appletem
- **destroy** — je zavolána v případě, že uživatel zaktualizuje stránku (jsou zavolány obě metody **init** a **start**) nebo zavře webový prohlížeč

které řídí životní cyklus appletu. V podstatě jedinou povinnou metodou je **init**, do které je vložen kód spouštějící grafické uživatelské rozhraní. Ostatní metody nemusí být definovány.

Atribut **archive** značky **applet** je nepovinný. Ovšem v tomto případě je žádoucí, aby byl zadán. Java appletům není za normálních okolností umožněn přístup k souborovému systému počítače, samozřejmě kvůli bezpečnosti. V případě, že atribut **archive** je zadán a odkazuje na existující JAR soubor, který je digitálně podepsán, appletu může být tento přístup povolen. Uživatel je potom dotázán prohlížečem, zda-li tomuto podpisu důvěřuje. Pokud ano, applet může k souborovému systému přistupovat. Uživatel tak má tu možnost načítat/ukládat konečné automaty z/do XML.

5.2 Java Web Start

Java Web Start je technologie, která umožňuje spouštět Java aplikace přímo z webové stránky pomocí odkazu. Základem je speciální soubor s příponou JNLP (Java Network Launcher Protocol). Soubor je definován v XML a obsahuje informace o tom, odkud aplikaci stáhnout a jak ji spustit. JNLP *launch.jnlp* použité v projektu vypadá takto:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<jnlp codebase = "http://www.stud.fit.vutbr.cz/~xsraje00/" href = "launch.jnlp" spec = "1.0+">
    <information>
        <title>Graficka simulace cinnosti konecnych automatu</title>
        <vendor>Roman Srajer</vendor>
        <homepage href = "http://www.stud.fit.vutbr.cz/~xsraje00/" />
        <description>Graficka simulace cinnosti konecnych automatu</description>
        <description kind = "short">Graficka simulace cinnosti konecnych automatu</description>
    </information>

    <resources>
        <jar eager = "true" href = "projekt.jar" main = "true" />
    </resources>

    <application-desc main-class = "projekt.Main">
    </application-desc>
</jnlp>
```

Atribut **codebase** značky **jnlp** udává URL, odkud budou načítány všechny potřebné soubory, a **href** udává název JNLP souboru. Značka **information** obsahuje popis autora, název programu a další informace, které budou uživateli prezentovány při spouštění programu. Značka **resources** obsahuje informace o zdrojích programu. V tomto případě obsahuje jediný zdroj, a sice soubor *projekt.jar*. Atribut **main="true"** říká, že se jedná o zdroj, který obsahuje spouštěcí třídu. Atribut **eager="true"** říká, že zdroj musí být stažen před spuštěním programu. Značka **application-desc** určuje konkrétní způsob, jak program spustit. Atribut **main-class** nastavuje hlavní spouštěcí třídu, která obsahuje statickou metodu *main*. V této značce může být zanořeno libovolné množství značek tvaru

```
<argument>arg1</argument>
```

které vkládájí do metody *main* argumenty.

Při tomto formátu JNLP bude program spuštěn v omezeném módu podobně jako Java Applet, což znamená, že implicitně nemůže přistupovat k souborovému systému. Ovšem s použitím balíčku *javax.jnlp* je možné uživateli přístup povolit pomocí speciálních dialogů pro otevírání souborů, kdy je vždy před jejich zobrazením dotázán na povolení akce.

Pokud není pro uživatele žádoucí, aby byl dotazován **vždy** při pokusu číst nebo zapisovat, řešením je do značky **jnlp** přidat značku

```
<security>
    <all-permissions/>
</security>
```

která říká, že spouštěný program bude mít všechna přístupová práva. V takovém případě je uživatel dotázán ještě před samotným spuštěním, jestli programu důvěřuje.

Pro správný chod Java Web Start je nutné mít tuto technologii nainstalovanou. Pro automatické spouštění **přímo** z html stránky je dále potřeba, aby webový server znal JNLP, tedy aby v html odpovědi posílal MIME typ:

```
application/x-java-jnlp-file
```

Pokud je vrácen tento MIME typ, webový prohlížeč již ví, co s takovým souborem udělat (stáhnout ho a spustit). V opačném případě ho pouze zobrazí nebo jen stáhne. V takovém případě je možné stažený JNLP soubor spustit ručně příkazem:

```
javaws launch.jnlp
```

Kapitola 6

Závěr

Posláním této práce bylo prezentovat teorii převodu regulárního výrazu na všechny typy automatů a naopak. V této závěrečné kapitole bych shrnul celou práci a diskutoval výhody, nevýhody a možná pokračování projektu.

Mezi výhody řešení lze určitě zařadit snadné vytváření konečných automatů z regulárních výrazů a rovněž možnost snadno zjistit regulární výraz, který je velmi často ve srozumitelné formě, i když se některé výjimky najdou. Dalším plusem je rovněž možnost snadno vytvářet nové automaty a to zcela intuitivně. Možnost simulace je samozřejmou součástí funkcionality. Do řešení však byla vložena i možnost simulace nedeterministického automatu, kdy před simulací dojde nejprve k vyhledání cesty do koncového stavu a až poté se provede simulace nalezené cesty. Aplikace je přístupná přes web, což je vždy přínosem, neboť není nutné provádět žádné instalace nebo konfigurace.

Mezi nevýhody, které by bylo vhodné dořešit v dalších verzích, bych zařadil neeexistenci jazykových mutací. V aktuální verzi jsou všechny řetězce vloženy do třídy *Strings* a přístupné přes statické metody. Řešením by mohl být export do XML, kde každá mutace by byla udržována v jednom souboru. Bylo by tak možné dynamicky přidávat nové jazyky bez nutného opětovného překládání projektu. Další nevýhodou je ne příliš ideálně vytvořený export simulace automatu do GIF obrázku, kdy dochází ke zbytečnému plýtvání pamětí. Nedostatkem trpí i algoritmus vytýkání při převodu konečného automatu na regulární výraz. Mějme například tento výraz a jeho následné zjednodušení:

$$baa^* + b + c = b(aa^* + \varepsilon) + c = ba^* + c$$

Program neaplikuje toto zjednodušení, protože nedokáže provést vytknutí symbolu b. Symbol b se totiž nenachází ve všech částech počátečního sjednocení, ale jen ve dvou prvních.

Do dalších verzí projektu by bylo možné zařadit další typ simulace, kterou běžně využívají lexikální analyzátory a jejich generátory, jako například program *Flex*. Klasicky, pokud není možné provést přechod pro daný symbol na vstupní pásce, automat se zasekne. Tato situace může být vyřešena následovně. Pokud je stav, ve kterém došlo k zaseknutí, koncový, aktuálně přečtený řetězec je přijat a zbylé symboly jsou čteny již z počátečního stavu. Pokud je stav nekoncový, přečtený řetězec je zahozen resp. prohlášen za neplatný a čtení dalších symbolů probíhá opět z počátečního stavu. Dalším možným rozšířením by mohlo být lepší rozmísťování stavů automatu v grafické zobrazení. Mezi možné způsoby realizace lze zařadit například evoluční algoritmy, nicméně pro implementaci tohoto rozšíření již nezbylo dostatečné množství času.

Literatura

[1] Flex: Rychlý lexikální analyzátor. URL http://flex.sourceforge.net.

[2] Brzozowski, J. A.: Derivatives of regular expressions. *Journal of the ACM*, ročník 11, 1964: s. 481–494.

[3] Honzík, J. M.: Předmět Algoritmy, FIT VUT Brno.
URL http://www.fit.vutbr.cz/study/courses/IAL/public/.

[4] Kevin Weiner, F. S.: Knihovna `AnimatedGifEncoder`.
URL www.java2s.com/Code/Java/2D-Graphics-GUI/AnimatedGifEncoder.htm, 2003.

[5] Meduna, A.: Předmět Formální jazyky a překladače, FIT VUT Brno.
URL http://www.fit.vutbr.cz/study/courses/IFJ/public/.

[6] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, 2000, iSBN 18-523-3074-0.

[7] Neumann, C.: Converting Deterministic Finite Automata to Regular Expressions.
URL http://neumannhaus.com/christoph/papers/2005-03-16.DFA_to_RegEx.pdf, 2005.

[8] Prokhorenko, O. . A.: Creating a Trusted Applet with Local File System Access Rights. URL http://www.developer.com/java/data/article.php/3303561.

[9] Team, J.: Tutorial : XML generation with JAVA.
URL http://www.javazoom.net/services/newsletter/xmlgeneration.html.

[10] Zakhour, S.: *Java 6*. Computer Press, 2007, iSBN 978-80-251-1575-6.

[11] Zukowski, J.: Deploying Software with JNLP and Java Web Start.
URL http://java.sun.com/developer/technicalArticles/Programming/jnlp/index.html, 2002.

Seznam příloh

- Příloha A — Přiložené CD
- Příloha B — Překlad

Příloha A

Přiložené CD

Přiložené CD obsahuje kompletní zdrojové kódy Java aplikace (adresář *BPprojekt*) a zdrojové kódy této dokumentace v systému LaTeX(adresář *tex*).

Adresář *compiled* obsahuje přeloženou dokumentaci (soubor *projekt.pdf*), slajdy použité při obhajobě semestrálního projektu (soubor *prezentace.pdf*), přeloženou aplikaci ve formě JAR archivu (soubor *projekt.jar*), příklad uložení konečného automatu prezentovaného v dokumentaci v části 4.5 (soubor *doc.xml*) a knihovnu JNLP (soubor *jnlp.jar*).

Adresář *web* obsahuje potřebné soubory pro funkčnost webové prezentace (viz. kapitola 5). Nápověda k ovládání aplikace je součástí webové prezentace pod záložkou *Nápověda*.

Příloha B

Překlad

Vzhledem k tomu, že Java je interpretovaný jazyk, je nutné mít na počítači nainstalovanou interpretující vrstvu, tedy Java Virtual machine (JVM). Dále musí být k dispozici programy *javac*, *java*, *jar*, *keytool*, *jarsigner*.

Pro správný překlad budeme postupovat následovně. Přejdeme do hlavního adresáře projektu *BPprojekt*. V něm vytvoříme adresář *classes*, do kterého budou umístěny přeložené soubory **.class*. Překlad projektu provedeme příkazem:

```
javac -d classes -encoding windows-1250 projekt/*.java
```

Pokud se překlad podařil, v adresáři *classes* by měly být složky *projekt* a *libraries*. Dále zkopírujeme složku *javax*, která obsahuje **již přeloženou** knihovnu JNLP, viz. kapitola 5. Důvod je ten, že v některých verzích JVM není tato knihovna obsažena a program by nebylo možné spustit (problémy především na systému Linux):

```
cp -r javax classes   (pro systém Linux)
xcopy /S /i javax classes\javax   (pro systém Windows)
```

Dalším krokem je vytvoření souboru JAR z přeložených souborů. Přejdeme do adresáře *classes* a v něm vytvoříme soubor, který bude obsahovat název hlavní spouštěcí třídy (třída, která obsahuje statickou metodu *main*):

```
echo Main-Class: projekt.Main> Manifest
```

Soubor *Manifest* využijeme při vytvoření archivu. Program *jar* tak může do výsledného archivu vložit informaci, jak daný archiv spustit:

```
jar cvmf Manifest projekt.jar javax libraries projekt
```

Pokud se vše podařilo, nově vytvořený soubor *projekt.jar* je možné spustit jen prostým dvojklikem nad jeho ikonou (především systém Windows) nebo příkazem:

```
java -jar projekt.jar
```

Dále je vhodné soubor *projekt.jar* digitálně podepsat. Důvod byl popsán v kapitole 5. Nejdříve je nutné vygenerovat klíč programem *keytool*:

```
keytool -genkey -keyalg rsa -alias klic
```

Tento program ukládá všechny klíče do lokálního uložiště (*keystore*). Pokud program spouštíte poprvé, budete nejprve vyzváni na zadání nového hesla k tomuto uložišti. Pokud je již uložiště vytvořeno, musíte k němu toto heslo znát. Pokud jej neznáte, jediným řešením je uložiště smazat a vytvořit nové. Uložiště se obvykle ukládá do souboru s názvem *.keystore* ve složce uživatele (ve Windows `c:\Documents and Settings\<user>\.keystore`). Stačí tak tento soubor smazat a provést výše uvedený příkaz *keytool*. Po vygenerování klíče provedeme samotné podepsání souboru *projekt.jar*:

```
jarsigner projekt.jar klic
```

Všechny výše popsané zdlouhavé kroky je možné v prostředí *NetBeans IDE 6.5* provést mnohem rychleji.

Po spuštění prostředí vytvoříme nový projekt *File→New Project*. V levém okně vybereme kategorii *Java* a v pravém okně vybereme *Java Project with Existing Sources*. V dalším okně zvolíme název a umístění projektu. V dalším okně vyhledáme složku s projektem *BPprojekt*. Prostředí nás bude varovat, že se ve složce nacházejí soubory **.class*, hlášku budeme **ignorovat** a dokončíme průvodce.

Aby bylo aplikaci možné přeložit, je nutné v okně *Projects* do uzlu *Libraries* přiložit knihovnu *jnlp.jar*, kterou si můžeme vytvořit sami programem *jar* ze složky *javax*:

```
jar cvf jnlp.jar javax
```

Přejdeme do nastavení projektu, *File→Project Propeties*. V uzlu *Sources* nastavíme kódování na **windows-1250**. V uzlu *Packaging* zvolíme vytvoření *JAR* archivu popřípadě i s kompresí. V uzlu *Run* nastavíme spouštěcí třídu projektu *projekt.Main*. V uzlu *Web Start* (viz. kapitola 5) tuto technologii povolíme a zvolíme *Self–signed*, aby archiv byl digitálně podepsán.

Nyní již stačí přes *Run→Clean and Build Main Project* projekt přeložit.