VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

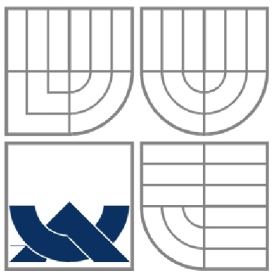
PŘEVODY MEZI REGULÁRNÍMI GRAMATIKAMI, REGULÁRNÍMI VÝRAZY A KONEČNÝMI AUTOMATY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

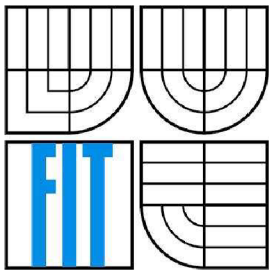
AUTOR PRÁCE
AUTHOR

Bc. MICHAL PODHORSKÝ

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEVODY MEZI REGULÁRNÍMI GRAMATIKAMI, REGULÁRNÍMI VÝRAZY A KONEČNÝMI AUTOMATY

MUTUAL TRANSFORMATIONS OF REGULAR GRAMMARS, REGULAR EXPRESSIONS AND FINITE AUTOMATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL PODHORSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. TOMÁŠ MASOPUST

BRNO 2007

Zadání diplomové práce

Řešitel: **Podhorský Michal, Bc.**

Obor: Informační systémy

Téma: **Převody mezi regulárními gramatikami, regulárními výrazy a konečnými automaty**

Kategorie: Teorie informatiky

Pokyny:

1. Seznamte se s modely, které se používají v moderní teorii jazyků.
2. Na bázi těchto modelů navrhněte a implementujte webovou aplikaci, která bude uskutečňovat převody mezi reg. gramatikami, regulárními výrazy a konečnými automaty s tím, že konečný automat bude navíc minimalizován. Navíc v případě nepříliš velkého konečného automatu bude též požadován grafický výstup. Na výstupu tedy bude: regulární gramatika, regulární výraz a konečný automat odpovídající zadané gramatice či výrazu a minimalizovaný konečný automat (včetně použitých abeced atd.)
3. Výše uvedené implementujte v jazyce Java/JSP a navrhněte vlastní metodu implementace grafického výstupu konečného automatu.
4. Studujte užití této implementace.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

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

Žára, J., Beneš, B., Sochor, J., Felkel, P.: Moderní počítačová grafika, Computer Press, 2005.

Vedoucí: **Masopust Tomáš, Mgr.**, UIFS FIT VUT

Datum zadání: 28. února 2006

Datum odevzdávání: 22. května 2007

Licenční smlouva

Licenční smlouva je uložena v archívu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Práce popisuje modely moderní teorie jazyků – konečné automaty, regulární gramatiky a regulární výrazy. Nad těmito modely je implementována webová aplikace, která provádí převody mezi jednotlivými modely, konečné automaty jsou navíc graficky zobrazeny.

Klíčová slova

Teorie jazyků, konečné automaty, regulární gramatiky, regulární výrazy, webová aplikace, Java, Java EE, JavaServer Pages, Apache Tomcat.

Abstract

This work describes models of modern language theory – finite automata, regular grammars and regular expressions. A web application converting among these models is implemented.

Keywords

Formal Language Theory, Finite Automaton, Regular Grammar, Regular Expression, Web Application, Java, Java EE, JavaServer Pages, Apache Tomcat.

Citace

Michal Podhorský: Převody mezi regulárními gramatikami, regulárními výrazy a konečnými automaty, diplomová práce, Brno, FIT VUT v Brně, 2007

Převody mezi regulárními gramatikami, regulárními výrazy a konečnými automaty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Mgr. Tomáše Masopusta.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Podhorský
14.5.2007

Obsah

1 Úvod

Webové aplikace jsou v současnosti velice často používanou formou, jak zpřístupnit různé projekty co největšímu počtu uživatelů. Jediným požadavkem na straně klienta je připojení k internetu a internetový prohlížeč, který je součástí všech operačních systémů. Celá aplikace se nachází na serveru, který ji klientům zprostředkovává ve formátu HTML stránek. Stránky jsou vytvářeny dynamicky jako reakce na vstup od uživatele, což vytváří interaktivitu známou z desktopových aplikací.

Konečné automaty, regulární gramatiky a regulární výrazy jsou základními prvky teorie formálních jazyků. V informatice nacházejí širokého uplatnění, například při zpracování textu – což nabízí i vytvořená aplikace.

Úvodní kapitola formálně popisuje jednotlivé modely – regulární gramatiky, konečné automaty a regulární výrazy, u konečných automatů především jejich rozšíření. Třetí kapitola dokončuje teoretickou část, obsahuje algoritmy pro vzájemnou převoditelnost mezi jednotlivými modely, což zároveň dokazuje jejich ekvivalenci z hlediska popisné síly.

Ve čtvrté kapitole se nachází přehled technologií, použitých při implementaci. Zároveň jsou uvedeny i jednotlivé verze, tedy požadavky na software pro instalaci a spuštění aplikace.

Pátá kapitola obsahuje samotnou implementaci aplikace. První část je věnována aplikační vrstvě. Ta zahrnuje reprezentaci definovaných formálních modelů, algoritmy pro jejich vzájemné převody a vytváření grafických výstupů konečných automatů. Jednotlivé definice modelů i algoritmy jsou zde podrobně popsány, včetně složitostí u algoritmů pro převod modelů. Samostatná část je věnována grafickému výstupu konečných automatů a srovnání vytvořených metod grafického výstupu. Nad aplikační vrstvou je vytvořena prezentační vrstva, popsaná v druhé části páté kapitoly. Definuje grafické rozhraní pro prohlížeč, které zprostředkovává práci s implementovanými modely a zobrazuje výsledky převodů mezi modely.

V rámci Semestrálního projektu jsem se seznámil s modely moderní teorie jazyků, část definující modely, nutná pro implementaci aplikace, byla ve zkráceném rozsahu převedena do diplomové práce. Zároveň jsem navrhl dvojvrstvý model aplikace a strukturu tříd v aplikační vrstvě. Tato část není v diplomové práci uvedena – návrh byl dodržen, pouze jsem jej rozšířil o některé další funkce, které vyplynuly z používání aplikace. Kompletní struktura tříd a vazby mezi třídami jsou součástí popisu aplikační vrstvy.

2 Modely teorie formálních jazyků

V této kapitole jsou uvedeny základní pojmy a definovány jednotlivé modely teorie formálních jazyků – gramatiky, konečné automaty a regulární výrazy. U konečných automatů jsou popsány i způsoby grafické reprezentace, které jsou implementovány ve vytvořené webové aplikaci. Dále jsou uvedena i rozšíření konečných automatů (vhodná pro převody mezi modely) a ukázána jejich vzájemná ekvivalence.

Definice 2.1 Abeceda je libovolná konečná množina Σ, její prvky se nazývají znaky (případně také písmena nebo symboly) abecedy.

Definice 2.2 Slovo (též řetězec) nad abecedou Σ je libovolná konečná posloupnost znaků této abecedy.

Příklad 2.1 *aabb* je slovo nad abecedou $\{a, b\}$.

Délka slova v se značí $|v|$. Prázdné posloupnosti znaků odpovídá tzv. prázdné slovo, označované ε, které má nulovou délku. Množinu všech slov nad abecedou Σ značíme Σ*, množinu všech neprázdných slov Σ+.

Platí:

$$\{a\}^* = \{\varepsilon, a, aa, aaa, aaaa, ...\}$$
$$\{a\}^+ = \{a\}^* \setminus \{\varepsilon\}$$
$$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$$

Definice 2.3 Na každá dvě slova u, v lze aplikovat binární operaci zřetězení, označovanou „.“, která je definována předpisem $u.v = uv$.

Příklad 2.2 Zřetězením slov *abc* a *ba* vznikne slovo *abcba*.

Operace zřetězení je asociativní, tj. $u.(v.w) = (u.v).w$ pro libovolná slova u, v, w. Dále ε se chová jako jednotkový prvek, tj. $u.\varepsilon = \varepsilon.u = u$ pro libovolné slovo u. Znak „.“ v zápisu zřetězení se obvykle vynechává.

Definice 2.4 Jazyk nad abecedou Σ je libovolná množina slov nad Σ (jazyky nad Σ jsou tedy právě podmnožiny Σ*).

Příklad 2.3 {10, 1, 011101} je jazyk nad abecedou {0, 1}, prázdná množina je jazyk nad libovolnou abecedou.

2.1 Gramatiky

Definice 2.5 Gramatika G je čtveřice (N, Σ, P, S), kde

- N je neprázdná konečná množina neterminálních symbolů (neterminálů)
- Σ je konečná množina terminálních symbolů (terminálů) taková, že $N \cap \Sigma = \emptyset$. Sjednocením N a Σ obdržíme množinu všech symbolů gramatiky, kterou obvykle označujeme symbolem V.
- $P \subseteq V^*NV^* \times V^*$ je konečná množina pravidel. Pravidlo (α, β) obvykle zapisujeme ve tvaru $\alpha \rightarrow \beta$ („α přepiš na β“).
- $S \in N$ je speciální počáteční neterminál (nazývaný také kořen gramatiky).

Definice 2.6 Binární relace $\Rightarrow_G$ přímého odvození na množině V^* gramatiky $G = (N, \Sigma, P, S)$ je definována: $\gamma \Rightarrow_G \delta$ právě když existuje pravidlo $\alpha \rightarrow \beta \in P$ a slova $\eta, \rho \in V^*$ taková, že $\gamma = \eta\alpha\rho$ a $\delta = \eta\beta\rho$.

Definice 2.7 Relace odvození v k krocích (k-násobné složení relace $\Rightarrow_G$) je definována pro každé $k \in \mathbb{N}_0$ induktivně:

- $\overset{0}{\Rightarrow}_G$ je identická relace
- $\overset{k+1}{\Rightarrow}_G = \overset{k}{\Rightarrow}_G \circ \Rightarrow_G$

Definice 2.8 Relace odvození $\overset{*}{\Rightarrow}_G$ (reflexivní a tranzitivní uzávěr $\Rightarrow_G$) je definována předpisem

- $\overset{*}{\Rightarrow}_G = \bigcup_{i=0}^{\infty} \overset{i}{\Rightarrow}_G$

Index G se u uvedených relací obvykle vynechává, pokud je z kontextu patrné, o kterou gramatiku se jedná.

Prvky množiny V*, které lze odvodit z počátečního neterminálu, se nazývají větnými formami gramatiky G, tedy $\alpha \in V^*$ je větná forma právě když $S \Rightarrow^* \alpha$. Větná forma, která neobsahuje žádné neterminály, se nazývá věta.

Definice 2.9 Množina všech vět tvoří jazyk generovaný gramatikou G, označovaný jako $L(G)$:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Definice 2.10 Gramatiky G_1 a G_2 se nazývají jazykově ekvivalentní, právě když generují tentýž jazyk, tj. $L(G_1) = L(G_2)$.

Příklad 2.4 Gramatika $G = (\{S, A, B\}, \{a, b\}, \{S \rightarrow ABS, S \rightarrow \varepsilon, AB \rightarrow BA, BA \rightarrow AB, A \rightarrow a, B \rightarrow b\}, S)$. Větnou formou G je např. *AaBAbBS*, větou např. *abab*. Jazyk $L(G)$ generovaný gramatikou: $L(G) = \{u \in \{a, b\}^* \mid \#_a(u) = \#_b(u)\}$.

Konvence značení

- Kořen gramatiky se obvykle značí symbolem S.
- Neterminální symboly jsou označovány velkými písmeny (obvykle ze začátku) latinské abecedy (A, B, C, ...).
- Terminální symboly se obvykle značí malými písmeny ze začátku latinské abecedy (a, b, …).
- Řetězce, které jsou složeny výhradně z terminálních symbolů (tzv. terminální řetězce) se obvykle značí malými písmeny z konce latinské abecedy (…, x, y, z).
- Řetězce, které mohou být složené z terminálních i neterminálních symbolů, jako např. větné formy, se značí malými řeckými písmeny (α, β, γ, ...).
- Pravidla $\alpha \to \beta$, $\alpha \to \gamma$ se stejnou levou stranou se často zapisují stručněji jako $\alpha \to \beta \mid \gamma$.

Lingvista Noam Chomsky rozdělil gramatiky do čtyř skupin (typů) na základě různých omezení na tvar pravidel. Toto rozdělení odpovídá popisné síle gramatik.

Definice 2.11 Gramatika se nazývá regulární, jestliže každé její pravidlo je tvaru $A \to aB$ nebo $A \to a$ s eventuální výjimkou pravidla $S \to \varepsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.
Definice 2.12 Jazyk L je regulární, pokud existuje regulární gramatika G taková, že L(G) = L.

2.2 Konečné automaty

Definice 2.13 Konečný automat (Finite Automaton, FA) $\mathcal{M}$ je pětice $(Q, \Sigma, \delta, q_0, F)$, kde

- Q je neprázdná konečná množina stavů.
- Σ je konečná množina vstupních symbolů, nazývaná také vstupní abeceda.
- $\delta : Q \times \Sigma \to Q$ je parciální přechodová funkce.
- $q_0 \in Q$ je počáteční stav.
- $F \subseteq Q$ je množina koncových stavů.

Definice 2.14 Rozšířená přechodová funkce $\hat{\delta} : Q \times \Sigma^* \to Q$ je definovaná induktivně vzhledem k délce slova ze Σ^*:

- $\hat{\delta}(q, \varepsilon) = q$ pro každý stav $q \in Q$.
- $\hat{\delta}(q, wa) = \begin{cases} \delta(\hat{\delta}(q, w), a) & \text{je - li } \hat{\delta}(q, w) \text{ i } \delta(\hat{\delta}(q, w), a) \text{ definováno} \\ \perp & \text{jinak} \end{cases}$

Symbol $\perp$ značí, že funkce není definovaná. Zápis $\hat{\delta}(q, w) = p$ znamená, že automat $\mathcal{M}$ přejde ze stavu q pod slovem w (tj. postupným čtením w znak po znaku zleva doprava) do stavu p.

Definice 2.15 Jazyk přijímaný (akceptovaný) konečným automatem $\mathcal{M}$, označovaný $L(\mathcal{M})$, je tvořen právě všemi takovými slovy, pod kterými automat přejde z počátečního stavu do některého z koncových stavů:

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Úplná definice konkrétního automatu musí zahrnovat popis všech složek pětice z definice, jak je uvedeno v příkladu 2.5.

Příklad 2.5 Konečný automat $\mathcal{M} = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$, přechodová funkce δ je určena:

$\delta(q_0, a) = q_1$ $\quad$ $\delta(q_0, b) = q_2$

$\delta(q_1, a) = q_2$ $\quad$ $\delta(q_1, b) = q_0$

$\delta(q_2, a) = q_0$ $\quad$ $\delta(q_2, b) = q_1$

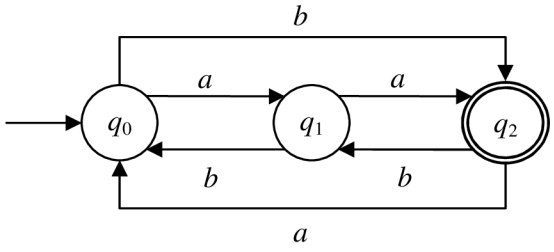
Konečný automat lze však reprezentovat i přehlednější formou, např. pomocí tabulky přechodové funkce.

Příklad 2.6 Přechodová funkce δ konečného automatu $\mathcal{M}$ reprezentovaná pomocí tabulky:

		a	b
$\rightarrow$	q_0	q_1	q_2
	q_1	q_2	q_0
$\leftarrow$	q_2	q_0	q_1

Stavy automatu jsou vypsány v záhlaví řádků, vstupní symboly v záhlaví sloupců, přechodová funkce je určena obsahem vnitřních polí tabulky (pokud je pro některé dvojice nedefinována, uvádí se v příslušném místě tabulky znak „–"), počáteční stav je označen znakem → a koncové stavy znakem ←.

Konečný automat lze reprezentovat i graficky pomocí přechodového grafu. Automat $\mathcal{M}$ je zobrazen na obrázku 2.1.



Obrázek 2.1: Přechodový graf konečného automatu

Stavy odpovídají uzlům, přechodová funkce je znázorněna ohodnocenými hranami, vstupní abeceda je tvořena symboly, kterými jsou hrany ohodnoceny, počáteční stav je označen šipkou a koncové stavy jsou dvojitě zakroužkovány.

Ke každému konečnému automatu $\mathcal{M}$ existuje ekvivalentní konečný automat $\mathcal{M}'$ s totální přechodovou funkcí (algoritmus 2.1).

Převod parciální na totální přechodovou funkci konečného automatu.

Vstup: Konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ s parciální přechodovou funkcí.
Výstup: Ekvivalentní konečný automat $\mathcal{M}'$ s totální přechodovou funkcí.

$Q' := Q$; $\delta' := \delta$;
$q_N \notin Q$;
for all $t \in \Sigma$
    $\delta'(q_N, t) = q_N$;
end for
for all $q \in Q$
    for all t $\in \Sigma$
        if $\delta(q, t) = \perp$ then
            $\delta'(q, t) = q_N$;
            $Q' := Q \cup q_N$;
        end if
    end for
end for
$\mathcal{M}' := (Q', \Sigma, \delta', q_0, F)$;

Algoritmus 2.1: Převod parciální na totální přechodovou funkci konečného automatu.

Ke každému konečnému automatu $\mathcal{M}$ existuje ekvivalentní konečný automat $\mathcal{M}'$ bez nedosažitelných stavů (algoritmus 2.2).

Eliminace nedosažitelných stavů konečného automatu.

Vstup: Konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.
Výstup: Ekvivalentní konečný automat $\mathcal{M}'$ bez nedosažitelných stavů.

$i := 0$; $S_i := \emptyset$;
repeat
    $S_{i+1} := S_i \cup \{q_0\} \cup \{q \mid \exists p \in S_i, a \in \Sigma : \delta(p, a) = q\}$;
    $i := i + 1$;
until $S_i = S_{i-1}$
$Q' := S_i$;
$\mathcal{M}' := (Q', \Sigma, \delta/Q', q_0, F \cap Q')$;

Algoritmus 2.2: Eliminace nedosažitelných stavů konečného automatu.

Konečné automaty nacházejí velmi široké uplatnění v technické praxi. Z hlediska efektivity a nákladnosti implementace je důležité, aby počet stavů byl pokud možno co nejmenší. Přirozeným

problémem je proto konstrukce minimálního automatu (tj. automatu s nejmenším počtem stavů), který rozpoznává daný regulární jazyk L.

Minimalizace konečného automatu.

Vstup: Konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů s totální přechodovou funkcí.

Výstup: Redukt $\mathcal{M}/\equiv$.

$i := 0$;
$\equiv_0 := \{(p, q) \mid p \in F \Leftrightarrow q \in F\}$;
repeat
    $\equiv_{i+1} := \{(p, q) \mid p \equiv_i q \wedge \forall a \in \Sigma : \delta(p, a) \equiv_i \delta(q, a)\}$;
    $i := i + 1$;
until $\equiv_i = \equiv_{i-1}$
$\equiv := \equiv_i$;
$\mathcal{M}/\equiv := (Q/\equiv, \Sigma, \eta, |q_0|, F/\equiv)$;

Algoritmus 2.3: Minimalizace konečného automatu.

Příklad 2.7 Je dán konečný automat $\mathcal{M}$, zadaný tabulkou 2.1. Při konstrukci minimálního automatu je nejprve třeba odstranit nedosažitelné stavy a zúplnit přechodovou funkci.

	$\mathcal{M}$	a	b
→	1	2	-
	2	3	4
←	3	6	5
	4	3	2
←	5	6	3
←	6	2	-
	7	6	1

Tabulka 2.1: Konečný automat $\mathcal{M}$.

Odstraněním nedosažitelného stavu 7 a přidáním nového stavu N za účelem zúplnění přechodové funkce získáme automat $\mathcal{M}'$, uvedený v tabulce 2.2.

	$\mathcal{M}$	a	b
→	1	2	N
	2	3	4
←	3	6	5
	4	3	2
←	5	6	3
←	6	2	N
	N	N	N

Tabulka 2.2: Konečný automat $\mathcal{M}'$ bez nedosažitelných stavů s úplnou přechodovou funkcí.

Konstrukci relací $\equiv_i$ lze provést v tabulce přechodové funkce sdružením řádků odpovídající stavům, které jsou v relaci $\equiv_i$ a jednotlivé skupiny (třídy rozkladu) se označí římskými číslicemi.

Každé políčko se doplní číslem třídy, do níž patří stav $\delta(q, x)$. Třídy rozkladu $\equiv_{i+1}$ se získají rozdělením existujících skupin – v rámci každé skupiny se sdruží stavy, které mají řádky vyplněné stejným způsobem.

	$\equiv_0$	a	b
I	1	I	I
	2	II	I
	4	II	I
	N	I	I
II	3	II	II
	5	II	II
	6	I	I

	$\equiv_1$	a	b
I	1	II	I
	N	I	I
II	2	III	II
	4	III	II
III	3	IV	III
	5	IV	III
IV	6	II	I

	$\equiv_2$	a	b
I	1	III	II
II	N	II	II
III	2	IV	III
	4	IV	III
IV	3	V	IV
	5	V	IV
V	6	III	II

Tabulka 2.3: Konstrukce relací $\equiv_i$.

Relace $\equiv$ je v tomto případě rovna relaci $\equiv_2$. Výsledný minimální automat je uveden v tabulce 2.4.

	$\mathcal{M}/\equiv$	a	b
→	I	III	II
	II	II	II
	III	IV	III
←	IV	V	IV
←	V	III	II

Tabulka 2.4: Minimalizovaný konečný automat $\mathcal{M}/\equiv$.

Definice 2.16 Nedeterministický konečný automat (NFA) $\mathcal{M}$ je pětice $(Q, \Sigma, \delta, q_0, F)$, kde

- Q je neprázdná konečná množina stavů.
- Σ je konečná množina vstupních symbolů, nazývaná také vstupní abeceda.
- $\delta : Q \times \Sigma \to 2^Q$ je přechodová funkce.
- $q_0 \in Q$ je počáteční stav.
- $F \subseteq Q$ je množina koncových stavů.

Konečné automaty s parciální přechodovou funkcí se často označují jako deterministické (DFA) a lze na ně pohlížet jako na speciální případ nedeterministických automatů.

Pro každý nedeterministický konečný automat $\mathcal{M}$ existuje ekvivalentní deterministický konečný automat $\mathcal{M}'$. (algoritmus 2.4).

Model nedeterministického konečného automatu je možné dále rozšířit o tzv. ε-kroky. Automat pak může svůj stav za určitých okolností změnit samovolně, tj. bez přečtení vstupního symbolu. Tato schopnost je formálně popsána pomocí ε-kroků, které jsou na přechodových grafech reprezentovány hranami, jejichž návěštím je prázdné slovo. Přes tyto hrany může automat během výpočtu na slově w

měnit svůj stav bez toho, aby ze vstupu cokoliv přečetl – mezi přečtením dvou po sobě následujících symbolů z w může provést libovolné konečné množství ε-přechodů.

Transformace nedeterministického konečného automatu na ekvivalentní deterministický konečný automat.

Vstup: Nedeterministický konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.
Výstup: Ekvivalentní deterministický konečný automat $\mathcal{M}' = (Q', \Sigma, \delta', \{q_0\}, F')$ bez nedosažitelných stavů s totální přechodovou funkcí.

$Q' := \{\{q_0\}\}$; $\delta' := \emptyset$; $F' := \emptyset$; $Done := \emptyset$;
while $(Q' - Done) \neq \emptyset$ do
    M := libovolný prvek množiny $Q' - Done$;
    if $M \cap F \neq \emptyset$ then
        $F' := F \cup \{M\}$;
    end if
    for all $a \in \Sigma$ do
        $N := \bigcup_{p \in M} \delta(p, a)$;
        $Q' := Q' \cup \{N\}$;
        $\delta' := \delta' \cup \{((M, a), N)\}$;
    end for
    $Done := Done \cup \{M\}$;
end while
$\mathcal{M}' := (Q', \Sigma, \delta', |q_0|, F')$;

Algoritmus 2.4: Transformace nedeterministického konečného automatu na ekvivalentní deterministický konečný automat.

Definice 2.17 Nedeterministický konečný automat s ε-kroky $\mathcal{M}$ je pětice $(Q, \Sigma, \delta, q_0, F)$, kde

- Q je neprázdná konečná množina stavů.
- Σ je konečná množina vstupních symbolů, nazývaná také vstupní abeceda.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ je přechodová funkce.
- $q_0 \in Q$ je počáteční stav.
- $F \subseteq Q$ je množina koncových stavů.

Ke každému nedeterministickému konečnému automatu s ε-kroky existuje ekvivalentní nedeterministický konečný automat (bez ε-kroků), jak ukazuje algoritmus 2.5.

Transformace nedeterministického konečného automatu s ε-kroky na ekvivalentní nedeterministický konečný automat bez ε-kroků.

Vstup: Nedeterministický konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ s ε-kroky.
Výstup: Ekvivalentní nedeterministický konečný automat $\mathcal{M}' = (Q, \Sigma \setminus \{\varepsilon\}, \delta', q_0, F')$ bez ε-kroků.

$\delta' := \emptyset$; $F' := F$;
for all $q \in Q$
    $S_0 = \{q\}$;
    $i := 1$;
    repeat
        $S_i := S_{i-1} \cup \{r \mid \exists s \in S_{i-1} : \delta(s, \varepsilon) = r\}$;
        $i := i + 1$;
    until $S_i = S_{i-1}$
    $D_\varepsilon(q) = S_i$;
end for
for all $q \in Q$
    for all $t \in \Sigma \setminus \{\varepsilon\}$
        $S := \{r \mid \exists s \in D_\varepsilon(q) : \delta(s, t) = r\}$;
        $T := \emptyset$;
        for all $r \in S$
            $T := T \cup D_\varepsilon(r)$;
        end for
        $\delta'(q, t) = T$;
    end for
    if $\delta(q, \varepsilon) \in F$ then
        $F' := F' \cup q$;
    end if
end for
$\mathcal{M}' := (Q, \Sigma \setminus \{\varepsilon\}, \delta', q_0, F')$;

Algoritmus 2.5: Transformace nedeterministického konečného automatu s ε-kroky na ekvivalentní nedeterministický konečný automat bez ε-kroků.

2.3 Regulární výrazy

Regulární výrazy jsou další formalismus, který je užitečným prostředkem pro popis (specifikaci) a návrh konečných automatů.

Definice 2.18 Třída regulárních jazyků nad abecedou Σ, označovaná $R(\Sigma)$, je definována induktivně:

1. $\emptyset$, $\{\varepsilon\}$, $\{a\}$ pro každé $a \in \Sigma$ je regulární jazyk nad Σ.
2. Jsou-li L_1, L_2, L regulární jazyky nad Σ, jsou také $L_1.L_2$, $L_1 \cup L_2$ a L^* regulární jazyky nad Σ.
3. Každý regulární jazyk vznikne po konečném počtu aplikací kroků 1 a 2.

Definice 2.19 Množina regulárních výrazů (regular expressions) nad abecedou Σ, označovaná *RE*(Σ), je definována induktivně:

1. Ø, ε, a pro každé $a \in \Sigma$ jsou regulární výrazy nad Σ (základní regulární výrazy).
2. Jsou-li E, F regulární výrazy nad Σ, jsou také $(E.F)$, $(E + F)$ a $(E)^*$ regulární výrazy nad Σ.
3. Každý regulární výraz vznikne po konečném počtu aplikací kroků 1-2.

V regulárních výrazech se mohou vyskytovat také kulaté závorky jako metasymboly, které pomáhají vymezit rozsah operátorů. Aby se jejich použití omezilo na minimum, zavádí se konvence týkající se priority operátorů – největší prioritu má „*“, pak „.“ a nakonec „+“, přičemž „nadbytečné“ závorky lze vypouštět. Výraz $a + b.c^*$ tedy odpovídá výrazu $(a + (b.(c)^*))$.

Každý regulární výraz E nad abecedou Σ popisuje jazyk $L(E)$ nad abecedou Σ podle těchto pravidel:

$$\begin{aligned}
\mathrm{L}(\varepsilon) &\stackrel{\text{def}}{=} \{\varepsilon\} \\
\mathrm{L}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{L}(a) &\stackrel{\text{def}}{=} \{a\} \text{ pro každé } a \in \Sigma \\
\mathrm{L}(E.F) &\stackrel{\text{def}}{=} \mathrm{L}(E).\mathrm{L}(F) \\
\mathrm{L}(E + F) &\stackrel{\text{def}}{=} \mathrm{L}(E) \cup \mathrm{L}(F) \\
\mathrm{L}(E^*) &\stackrel{\text{def}}{=} \mathrm{L}(E)^*
\end{aligned}$$

3 Ekvivalence konečných automatů, regulárních výrazů a gramatik

Všechny uvedené modely – konečné automaty, regulární výrazy a regulární gramatiky jsou vzájemně převoditelné, což dokazují následující algoritmy.

Pro každý konečný automat $\mathcal{M}$ existuje regulární gramatika G taková, že $L(\mathcal{M}) = L(G)$, jak ukazuje algoritmus 3.1.

Transformace nedeterministického konečného automatu na ekvivalentní regulární gramatiku.

Vstup: Nedeterministický konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.
Výstup: Regulární gramatika $G = (N, \Sigma, P, S)$.

$N' := \{ \overline{q} \mid q \in Q \}$;
P' je nejmenší množinou pravidel splňující

- pokud $p \in \delta(q,a)$, je $\overline{q} \rightarrow a\overline{p}$ pravidlo v P'
- pokud $p \in \delta(q,a)$ a $p \in F$, je $\overline{q} \rightarrow a$ pravidlo v P'

Je-li $q_0 \in F$, pak $\varepsilon \in L(\mathcal{M})$, položíme:
$N = N' \cup \{S\}$, kde $S \notin Q$
$P = P' \cup \{S \rightarrow \varepsilon\} \cup \{S \rightarrow \alpha \mid \overline{q}_0 \rightarrow \alpha\}$
jinak:
$N = N'$; $P = P'$; $S = \overline{q}_0$;

Algoritmus 3.1: Transformace nedeterministického konečného automatu na ekvivalentní regulární gramatiku.

Ke každé regulární gramatice G existuje nedeterministický konečný automat $\mathcal{M}$ takový, že $L(G) = L(\mathcal{M})$, jak ukazuje algoritmus 3.2.

Transformace regulární gramatiky na ekvivalentní nedeterministický konečný automat.

Vstup: Regulární gramatika $G = (N, \Sigma, P, S)$.
Výstup: Nedeterministický konečný automat $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$.

$Q = \{ \overline{A} \mid A \in N \} \cup \{q_F\}$, $q_F \notin N$; $\overline{q}_0 = \overline{S}$;
δ je nejmenší funkce $Q \times \Sigma \rightarrow 2^Q$ splňující

- pokud $A \rightarrow aB$ je pravidlo v P, pak $\overline{B} \in \delta(\overline{A}, a)$
- pokud $A \rightarrow a$ je pravidlo v P, kde $a \neq \varepsilon$, pak $q_F \in \delta(\overline{A}, a)$
- $F = \begin{cases} \{\overline{S}, q_F\} & \text{pokud } S \rightarrow \varepsilon \text{ je pravidlo v } P \\ \{q_F\} & \text{jinak} \end{cases}$

Algoritmus 3.2: Transformace regulární gramatiky na ekvivalentní nedeterministický konečný automat.

Definice 3.1 Regulární přechodový graf $\mathcal{M}$ je pětice $(\{q_0, q_F\}, \Sigma, \delta, q_0, q_F)$, kde

- $\{q_0, q_F\}$ množina stavů.
- Σ je vstupní abeceda.
- $\delta : Q \times Q \rightarrow RE(\Sigma)$ je parciální přechodová funkce.
- q_0 je počáteční stav.
- q_F je koncový stavů.

Regulární přechodové grafy představují zobecnění automatů s ε-kroky – hrany mohou být ohodnoceny i libovolným regulárním výrazem nad Σ.

Pro libovolný regulární přechodový graf M existuje ekvivalentní NFA M' s ε-kroky, což dokazuje algoritmus 3.3.

Transformace regulárního přechodového grafu na ekvivalentní nedeterministický konečný automat.

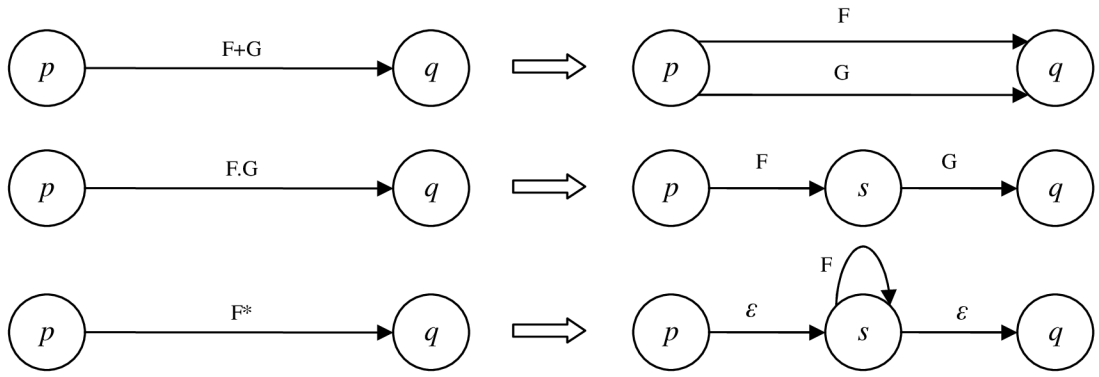
Vstup: Regulární přechodový graf $\mathcal{M} = (\{q_0, q_F\}, \Sigma, \delta, q_0, q_F)$.
Výstup: Nedeterministický konečný automat $\mathcal{M}' = (Q, \Sigma, \delta', q_0, q_F)$.

$Q = \{ \overline{A} \mid A \in N\} \cup \{q_F\}, q_F \notin N$;
$\overline{q}_0 = \overline{S}$;

1. Odstraň všechny hrany, které jsou ohodnoceny symbolem Ø.
2. Vyber libovolnou hranu $p \xrightarrow{E} q$, kde $E \notin \Sigma \cup \{\varepsilon\}$, odstraň ji a proveď:
   - pokud $E = F + G$, přidej hrany $p \xrightarrow{F} q$ a $p \xrightarrow{G} q$
   - pokud $E = F.G$, přidej ke Q nový stav s a hrany $p \xrightarrow{F} s$ a $s \xrightarrow{G} q$
   - pokud $E = F^*$, přidej ke Q nový stav s a hrany $p \xrightarrow{\varepsilon} s$, $s \xrightarrow{F} s$ a $s \xrightarrow{\varepsilon} q$

   Tyto dva kroky se provádí tak dlouho, dokud přechodový graf obsahuje alespoň jednu hranu ohodnocenou symbolem, který nepatří do $\Sigma \cup \{\varepsilon\}$.

Algoritmus 3.3: Transformace regulárního přechodového grafu na ekvivalentní nedeterministický konečný automat.



Obrázek 3.1: Pravidla pro transformaci regulárního přechodového grafu na ekvivalentní nedeterministický konečný automat s ε-kroky.

.

4 Použité technologie

V tomto přehledu jsou uvedeny všechny softwarové technologie, využité při vývoji aplikace. Pokud existuje více verzí, je vždy uvedeno, která byla použita.

Java

Java je objektově orientovaný jazyk vytvořený firmou Sun Microsystems. V současnosti se jedná o jeden z nejpoužívanějších programovacích jazyků. Vděčí za to svým vlastnostem – jednoduchý, robustní, bezpečný, dynamický a elegantní jazyk, který je nezávislý na platformě. Java běží na abstraktně definovaném virtuálním stroji (Java Virtual Machine), který je v cílovém systému emulován. Toto prostředí pro spouštění aplikací se nazývá Java Runtime Environment (JRE). Je dostupné pro většinu systémů a je zdarma ke stažení ze stránek firmy Sun.

Firma Sun vytvořila tři platformy, které se liší podle zaměření:

- Java Platform, Micro Edition (Java ME) – určeno pro zařízení s malým výkonem (mobilní telefony, PDA).
- Java Platform, Standard Edition (Java SE) – určeno především pro desktop.
- Java Platform, Enterprise Edition (Java EE) – určeno pro velké distribuované systémy a Internetové prostředí.

Při vývoji aplikace byla použita Java EE 5.

Javadoc

Javadoc je softwarový nástroj společnosti Sun Microsystems, který umožňuje vytvářet dokumentaci ve formátu HTML z dokumentačních komentářů uvedených ve zdrojových souborech jazyka Java. V dokumentačním komentáři je možné použít veškeré HTML tagy a využít tak všech možností jazyka HTML (vhodné např. pro formátování částí zdrojového kódu, které v dokumentaci uvádíme). Do vygenerované dokumentace se zároveň automaticky doplní odkazy na Java Core API.

Java Servlet a JavaServer Pages

JavaServer Pages jsou součástí Java 2 Platform, Enterprise Edition (Java EE), průmyslového standardu pro vývoj serverových aplikací v jazyku Java.

JavaServer Pages (JSP) jsou nadstavbou nad Java Servlets. Servlety slouží k nízkoúrovňové obsluze HTTP protokolu na straně serveru. Vytvářet webové aplikace pouze pomocí servletů je ale velmi nepraktické, např. každý výstup HTML je nutné generovat voláním metody.

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.Calendar;
import java.text.DateFormat;


public class ExampleServlet extends HttpServlet
{
    protected void doGet( HttpServletRequest request,
                          HttpServletResponse response )
            throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter( );
        out.println("<html>");
        out.println(" <head>");
        out.println("  <title>Example Page</title>");
        out.println(" </head>");
        out.println(" <body>");
        out.println(
            DateFormat.getDateInstance( DateFormat.LONG ).format(
                  Calendar.getInstance().getTime()
            )
        );
        out.println(" </body>");
        out.println("</html>");
    }
}
```

Zdrojový kód 4.1: Příklad jednoduchého servletu.

Poznámka: Termínem servlet se v obecnějším významu označují i běžné třídy (přeložené .class soubory), které jsou součástí webové aplikace.

Technologie JavaServer Pages má za úkol zjednodušit práci se servlety a ulehčit tak vytváření webových aplikací. JSP stránka je textový soubor obsahující výstup pro prohlížeč (např. HTML), do tohoto výstupu je vložen kód v jazyce Java. Předchozí servlet lze přepsat na JSP stránku takto:

```
<%@ page import="java.util.Calendar" %>
<%@ page import="java.text.DateFormat" %>
<%@ page contentType="text/html;charset=UTF-8" %>
<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
<%=
DateFormat.getDateInstance( DateFormat.LONG ).format(
            Calendar.getInstance().getTime()
      )
%>
  </body>
</html>
```

Zdrojový kód 4.2: Příklad jednoduché JSP stránky.

JSP stránky jsou ve webové aplikaci umístěné přímo jako textové soubory, s příponou .jsp. Při prvním požadavku na zobrazení jsou servletovým kontejnerem převedeny na servlet (.java) a přeloženy (.class). Po jakékoliv další změně .jsp souboru jej servletový kontejner opět převede na servlet a znovu přeloží. Každé další volání JSP znamená již jen provedení hlavní metody přeloženého servletu. To znamená značné urychlení a podstatný rozdíl proti interpretovaným jazykům (např. PHP), které jsou zpracovávány při každém požadavku.

Servlety vzniklé z JSP stránek jsou tedy mapovány na URL původního textového souboru a v prohlížeči se lze na ně přímo odkazovat.

Apache Tomcat

Tomcat je oficiální referenční implementace technologií Java Servlet a JavaServer Pages. Lze jej používat jako samostatný server nebo integrovat do serveru Apache.

Specifikace Java Servlet/JavaServer Pages	Verze Apache Tomcat
2.5/2.1	6.0.10
2.4/2.0	5.5.23
2.3/1.2	4.1.36
2.2/1.1	3.3.2

Tabulka 4.1: Přehled verzí Apache Tomcat a implementovaných specifikací Java Servlet/JSP.

Specifikace Java Servlet a JavaServer Pages jsou vyvíjeny společností Sun Microsystems.

Při vývoji a testování aplikace byl použit Apache Tomcat 5.5 s podporou specifikace Java Servlet/JavaServer Pages 2.4/2.0.

Apache Ant

Ant (Another Neat Tool) je konfigurovatelný, flexibilní a plně přenositelný sestavovací systém napsaný v Javě. Usnadňuje kompilaci a sestavování programů v Javě, především rozsáhlých.

Pro popis sestavovacího procesu se využívá soubor ve formátu XML, který je zpravidla pojmenován build.xml. V jednom sestavovacím souboru může být i více projektů. Každý projekt se skládá z několika cílů sestavování (targets). Jednotlivé cíle lze vázat do závislostí – typická aplikace v Javě má většinou cíle jako prepare, compile, clean, dist, install a javadoc, cíl dist může být závislý na cílech clean, prepare, compile a javadoc.

HyperText Markup Language

HyperText Markup Language (HTML) je značkovací jazyk pro hypertext, užívaný pro vytváření webových stránek. Je charakterizován množinou značek a jejich atributů. Mezi značky se uzavírají části textu dokumentu a tím se určuje význam obsaženého textu.

Část dokumentu uzavřená mezi značkami tvoří tzv. element dokumentu. Součástí obsahu elementu mohou být další vnořené elementy. Atributy jsou doplňující informace, které upřesňují vlastnosti elementu.

Prezentační část aplikace je vytvořena v HTML 4.01.

Cascading Style Sheets

Cascading Style Sheets (CSS, tabulky kaskádových stylů) jsou jazyk pro popis způsobu zobrazení dokumentů napsaných v jazycích HTML, XHTML nebo XML. Mají za úkol oddělit vzhled dokumentu od jeho struktury a obsahu.

JavaScript

JavaScript je multiplatformní, objektově orientovaný skriptovací jazyk vytvořený společností Netscape. Slovo Java je součástí názvu pouze z marketingových důvodů. Nejčastěji se využívá jako interpretovaný programovací jazyk, vkládaný přímo do HTML kódu, který se spouští na straně klienta až po načtení celého dokumentu. Většina prohlížečů umožňuje zakázat spouštění JavaScriptu, dle statistik[1] však tuto možnost využívá jen několik procent uživatelů. Pomocí JavaScriptu lze přistupovat ke struktuře a jednotlivým elementům HTML dokumentu, elementy přidávat nebo odebírat a měnit jejich vlastnosti.

[1] http://www.w3schools.com/browsers/browsers_stats.asp

5 Implementace

Vytvořená webová aplikace se skládá ze dvou částí – aplikační a prezentační vrstvy. Aplikační vrstva implementuje definované modely a grafický výstup, je celá tvořena pouze třídami jazyka Java a lze ji použít zcela samostatně. Při návrhu a následně i implementaci byl dodržován objektový přístup, kód je přehledný a dobře rozšiřitelný, algoritmy jsou naprogramovány s ohledem na maximální efektivitu.

Prezentační vrstva představuje nadstavbu nad aplikační vrstvou. Realizuje práci s modely v prostředí internetového prohlížeče. Je tvořena dynamickými stránkami vytvořenými technologií JavaServer Pages a vyžaduje tedy pro svůj běh server s nainstalovaným servletovým kontejnerem. Aby se uživateli usnadnilo vytváření jednotlivých modelů, implementoval jsem pro každý model JavaScriptovou aplikaci, která dovoluje vytvořit celý model bez odeslání požadavku na server.

Výsledná webová aplikace je funkční bez omezení ve všech hlavních prohlížečích – Internet Explorer, Opera, Mozilla FireFox. Příjemný vzhled a jednoduché a intuitivní ovládání se snaží nabídnout maximální uživatelské pohodlí.

Všechny zdrojové soubory byly vytvořený v kódování UTF-8, což kromě bezproblémového použití češtiny ve všech textech (např. dokumentaci) znamenalo především snadné použití znaků užívaných při definici modelů.

Sestavení aplikace

K sestavení aplikace byl vytvořen soubor build.xml určený pro nástroj Ant. Definuje projekt „common“ a v něm několik cílů pro jednotlivé části aplikace.

Název cíle	**Závisí na**	**Popis**
clean	-	Odstranění všech vytvořených souborů
compile	-	Přeložení aplikační vrstvy.
all	project	Hlavní cíl.
src	-	Vytvoření archívu (.zip) se zdrojovými kódy.
javadoc	-	Vytvoření archívu (.zip) s dokumentací vygenerovanou nástrojem Javadoc.
war	-	Vytvoření webového archívu (.war) obsahující soubory prezentační vrstvy a knihovny Graphviz.
xpodho05	-	Vytvoření webového archívu (.war) obsahující soubory prezentační vrstvy včetně dokumentace a knihovny Graphviz.
project	xpodho05, war, src, javadoc	Vytvoření archívu (.zip) obsahujícího všechny součásti aplikace.

Tabulka 5.1: Přehled cílů pro nástroj Ant v souboru build.xml vytvořené aplikace.

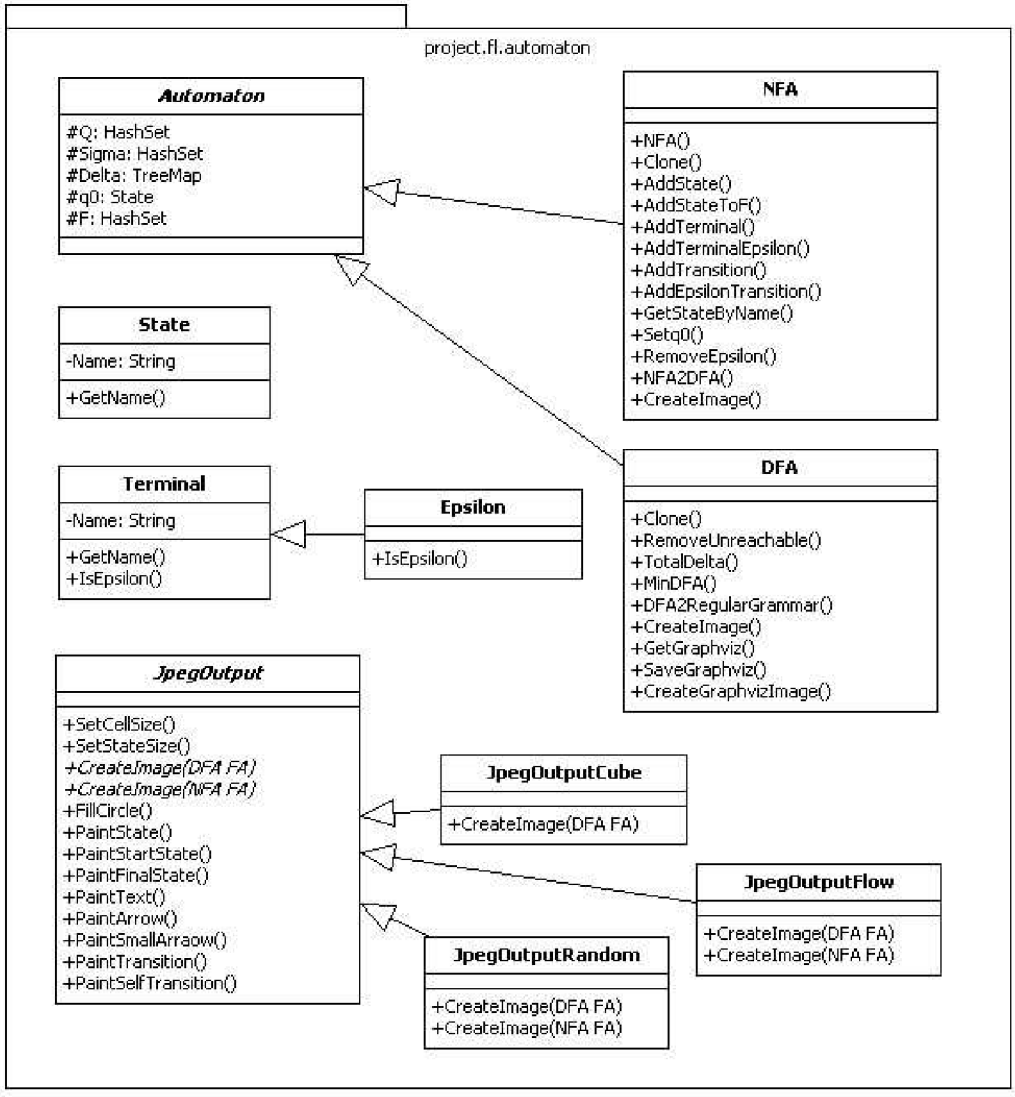
Vytvoření hlavního cíle (celé aplikace) je pak již velice jednoduché, zvláště při použití IDE s podporou Antu.

5.1 Aplikační vrstva

Aplikační vrstva je tvořena třídami jazyka Java, s kterými lze pracovat i nezávisle na vytvořené prezentační vrstvě v JSP. Všechny modely například nabízí metodu pro výpis na konzoli. Třídy jsou logicky rozděleny do tří balíků (package), které reprezentují automaty, regulární gramatiky a regulární výrazy.

5.1.1 Package project.fl.automaton

Balík project.fl.automaton obsahuje třídy pro deterministické a nedeterministické konečné automaty a třídy pro grafický výstup.



Obrázek 5.1: Diagram tříd v balíku project.fl.automaton.

5.1.1.1 Popis tříd a metod

Jsou uvedeny pouze metody, které jsou zásadní pro použití a práci s aplikací. U ostatních má jejich název dostatečně vypovídající schopnost a podrobnosti jsou uvedeny v dokumentaci (programátorská dokumentace vytvořená nástrojem Javadoc).

Třída Automaton

Abstraktní třída Automaton reprezentuje prostřednictvím svých atributů konečný automat. Třídy NFA pro nedeterministický konečný automat a DFA pro deterministický konečný automat jsou odvozeny od třídy Automaton.

Třída NFA

Třída NFA reprezentuje nedeterministický konečný automat. Poskytuje metody pro odstranění ε-kroků a převod na deterministický konečný automat.

Metoda	**Popis**
NFA()	Konstruuje nový nedeterministický konečný automat.
RemoveEpsilon()	Odstraní z přechodové funkce automatu ε přechody.
NFA2DFA()	Převod nedeterministického konečného automatu na deterministický konečný automat.
CreateImage()	Vytvoření grafického výstupu do formátu JPEG zvolenou metodou (lze použít pouze třídu JpegOutputFlow reprezentující Tokovou metodu).

Tabulka 5.2: Popis metod ve třídě NFA.

Třída DFA

Třída DFA reprezentuje deterministický konečný automat. Poskytuje metody pro odstranění nedosažitelných stavů, zúplnění přechodové funkce, převod na minimální konečný automat, převod na regulární gramatiku a výstup do grafického souboru.

Metoda	**Popis**
RemoveUnreachable()	Odstranění nedosažitelných stavů.
TotalDelta()	Zúplnění přechodové funkce automatu.
MinDFA()	Vytvoří minimální deterministický konečný automat.
DFA2RegularGrammar()	Převede konečný automat na regulární gramatiku.
CreateGraphvizImage()	Vytvoří grafický výstup automatu do formátu PNG pomocí knihovny Graphviz.
CreateImage()	Vytvoření grafického výstupu do formátu JPEG zvolenou metodou.

Tabulka 5.3: Popis metod ve třídě DFA.

Třída State

Třída State reprezentuje stav konečného automatu.

Třída Terminal

Třída Terminal reprezentuje terminální symbol.

Třída Epsilon

Třída Epsilon reprezentuje terminální symbol ε.

Třída Nonterminal

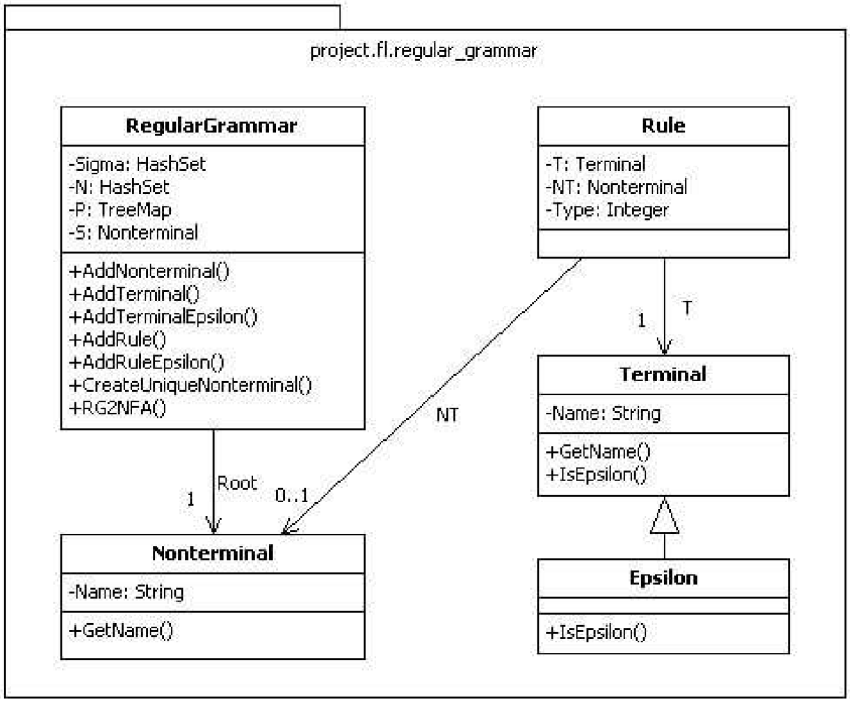
Třída Nonterminal reprezentuje neterminální symbol.

Třída JpegOutput

Abstraktní třída JpegOutput poskytuje funkce pro vykreslení konečných automatů (stavy, hrany). Předepisuje funkce CreateImage pro deterministické a nedeterministické konečné automaty, které musí implementovat třídy reprezentující vykreslovací metody (JpegOutputCube, JpegOutputFlow, JpegOutputRandom).

5.1.2 Package project.fl.regular_grammar

Balík project.fl.regular_grammar obsahuje třídy pro vytváření regulárních gramatik a převod na nedeterministický konečný automat.



Obrázek 5.2: Diagram tříd v balíku project.fl.regular_grammar.

5.1.2.1 Popis tříd a metod

Je uvedena pouze metoda pro převod regulární gramatiky na nedeterministický konečný automat, ostatní metody jsou pouze pomocné.

Třída Terminal

Třída Terminal reprezentuje terminální symbol.

Třída Epsilon

Třída Epsilon reprezentuje terminální symbol ε.

Třída Nonterminal

Třída Nonterminal reprezentuje neterminální symbol.

Třída Rule

Třída Rule reprezentuje pravidlo gramatiky vztažené k neterminálnímu symbolu gramatiky.

Třída RegularGrammar

Třída RegularGrammar reprezentuje regulární gramatiku. Poskytuje metodu pro převod na konečný automat.

Metoda	Popis
RG2NFA()	Převede regulární gramatiku na nedeterministický konečný automat.

Tabulka 5.4: Popis metod ve třídě RegularGrammar.

5.1.3 Package project.fl.regular_expression

Balík project.fl.regular_expression obsahuje třídy pro vytváření regulárních výrazů a pro převod regulárního výrazu na konečný automat.

5.1.3.1 Popis tříd a metod

Jsou uvedeny pouze metody třídy RegExp, které jsou použity při zpracování regulárního výrazu. Ostatní metody jsou pouze pomocné.

Třída RegExpParseException

Třída RegExpParseException reprezentuje výjimku při zpracování regulárního výrazu – chybné uzávorkování nebo operátor na pozici, kde není očekáván.

Třída RegExpTree

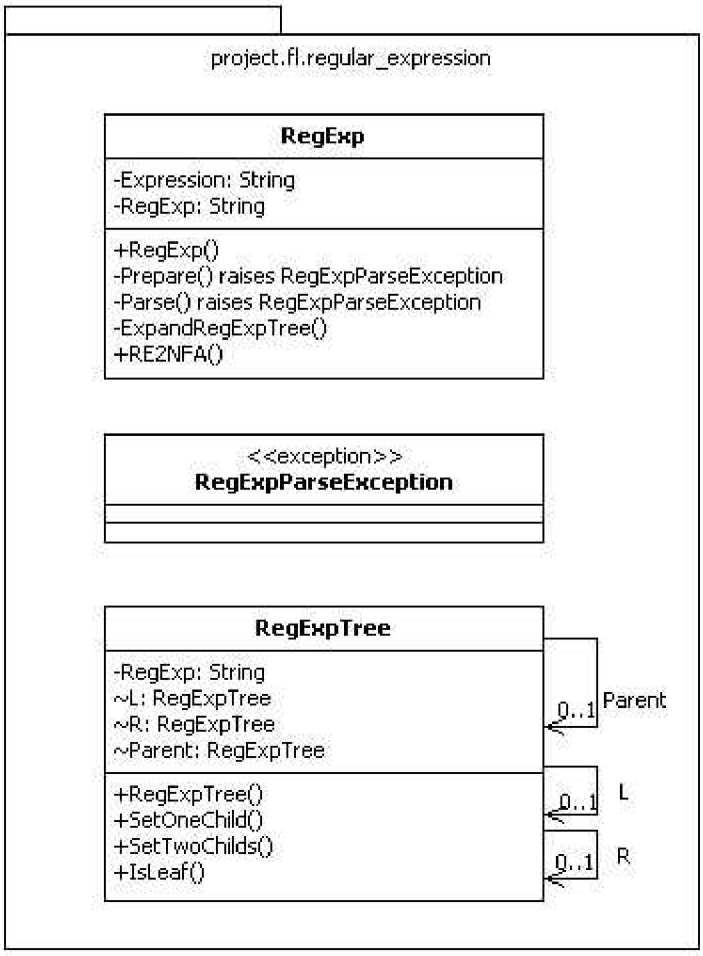
Třída RegExpTree reprezentuje (binární) strom, který je použit při převodu regulárního výrazu na konečný automat. Strom může mít jeden (levý podstrom) nebo dva (levý i pravý podstrom) potomky, případně žádný (a jedná se tedy o list).

Třída RegExp

Třída RegExp reprezentuje regulární výraz. Poskytuje metody pro parsování zadaného regulárního výrazu ve formě řetězce a převod na (nedeterministický) konečný automat.

Metoda	Popis
RegExp()	Konstruuje nový výraz a zkontroluje jej na výskyt závorek.
Parse()	Parsuje zadaný výraz a vytváří strom reprezentující regulární výraz.
RE2NFA()	Provede parsování zadaného výrazu, pokud se jedná o regulární výraz, z vytvořeného stromu sestaví nedeterministický konečný automat s ε kroky.

Tabulka 5.5: Popis metod třídy RegExp.



Obrázek 5.3: Diagram tříd v balíku project.fl.regular_expression.

5.1.4 Package project.graphics

Balík project.graphics obsahuje dvě pomocné třídy pro práci s vektory v dvourozměrném prostoru.

5.1.5 Reprezentace konečného automatu

Množina stavů Q, množina terminálních symbolů Sigma, množina koncových stavů F i počáteční stav q_0 odpovídají množinám, resp. stavu, z definice a jsou ve třídách DFA i NFA identické. Konstrukce přechodové funkce Delta je ve třídách odlišná. Každý stav odkazuje na množinu všech terminálů – v případě DFA každý terminál odkazuje přímo na stav z Q, v případě NFA odkazuje na množinu stavů z Q. Výhodou tohoto řešení je rychlost přístupu k jednotlivým přechodům mezi stavy, což je nejčastěji používaná operace.

Příklad reprezentace deterministického konečného automatu

$\mathcal{M} = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$ je deterministický konečný automat, kde přechodová funkce δ je určena:

$\delta(q_0, a) = q_1$ $\quad$ $\delta(q_0, b) = q_2$

$\delta(q_1, a) = q_2$ $\quad$ $\delta(q_1, b) = q_0$

$\delta(q_2, a) = q_0$ $\quad$ $\delta(q_2, b) = q_1$

Pak odpovídající instance třídy DFA je zobrazena na obrázku 5.4.

Q : $\{q_0, q_1, q_2\}$
Sigma : $\{a, b\}$
q_0 : q_0
F : $\{q_2\}$
Delta :

q_0	→	a	→	q_1
		b	→	q_2
q_1	→	a	→	q_2
		b	→	q_0
q_2	→	a	→	q_0
		b	→	q_1

Obrázek 5.4: Reprezentace deterministického konečného automatu.

Příklad reprezentace nedeterministického konečného automatu

$\mathcal{M} = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$ je nedeterministický konečný automat, kde přechodová funkce δ je určena:

$\delta(q_0, a) = \{q_1\}$ $\quad$ $\delta(q_0, b) = \{q_2\}$

$\delta(q_1, a) = \{q_0, q_2\}$ $\quad$ $\delta(q_1, b) = \{\}$

$\delta(q_2, a) = \{q_0, q_1, q_2\}$ $\quad$ $\delta(q_2, b) = \{q_0, q_1\}$

Pak odpovídající instance třídy NFA je zobrazena na obrázku 5.5.

Q	:	$\{q_0, q_1, q_2\}$			
Sigma	:	$\{a, b\}$			
q_0	:	q_0			
F	:	$\{q_2\}$			
Delta	:	q_0	$\rightarrow$	$a \rightarrow \{q_1\}$	
				$b \rightarrow \{q_2\}$	
		q_1	$\rightarrow$	$a \rightarrow \{q_0, q_2\}$	
				$b \rightarrow \{\}$	
		q_2	$\rightarrow$	$a \rightarrow \{q_0, q_1, q_2\}$	
				$b \rightarrow \{q_0, q_1\}$	

Obrázek 5.5: Reprezentace nedeterministického konečného automatu.

5.1.6 Reprezentace regulární gramatiky

Množina neterminálních symbolů N, množina terminálních symbolů Sigma i kořen gramatiky S odpovídají množinám, resp. neterminálu, z definice. Množina pravidel P je tvořena neterminály, které se vyskytují na levé straně pravidel gramatiky, každý neterminál odkazuje na množinu větných forem, na které lze tento neterminál přepsat.

Příklad reprezentace regulární gramatiky

Regulární gramatika $G = (\{S, A, B\}, \{a, b\}, P, S)$, kde $P = \{S \rightarrow A, S \rightarrow \varepsilon, A \rightarrow B, A \rightarrow a, B \rightarrow b, B \rightarrow bB\}$.

Pak odpovídající instance třídy RegularGrammar je zobrazena na obrázku 5.6.

N	:	$\{A, B, S\}$		
Sigma	:	$\{a, b\ \varepsilon\}$		
S	:	S		
P	:	S	$\rightarrow$	$\{A, \varepsilon\}$
		A	$\rightarrow$	$\{B, a\}$
		B	$\rightarrow$	$\{b, bB\}$

Obrázek 5.6: Reprezentace regulární gramatiky.

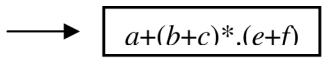
5.1.7 Reprezentace regulárního výrazu

Regulární výraz je zadán formou textového řetězce. Z tohoto řetězce jsou nejprve odstraněny bílé znaky a jiné znaky než operátory („.“, „+“, „*“) a písmena, která odpovídají terminálním symbolům. V dalším kroku jsou odstraněny operátory, které následují vícekrát po sobě (např. „*a**b*“ je upraveno na „*a*b*“). V posledním kroku se kontroluje počet závorek ve výrazu, zda počet otvíracích závorek v celém výrazu je roven počtu zavíracích, a zároveň zda počet zavíracích závorek na žádné pozici nepřevyšuje počet otvíracích závorek. Tato kontrola výrazně zjednodušuje následující parsování, zejména v případě ošetřeného správného uzávorkování.

Nad tímto upraveným textovým řetězcem se provádí parsování, při kterém se vytváří strom reprezentující regulární výraz. Kořen stromu představuje celý vstupní řetězec, ve všech vnitřních uzlech se nachází operátory a v listech terminální symboly.

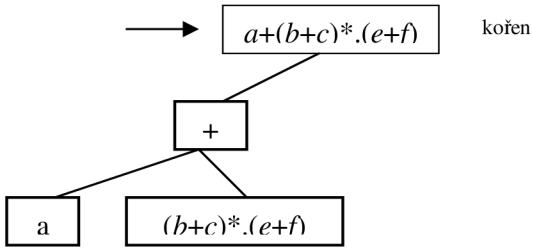
Příklad konstrukce stromu je prezentován na obrázku 5.7.

1.



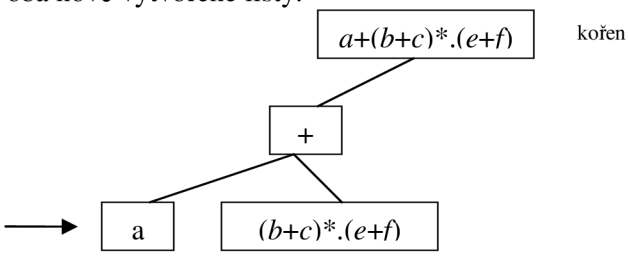
Prvním načteným symbolem je „a“, což není operátor, takže je zapamatován jako první operand.

2.



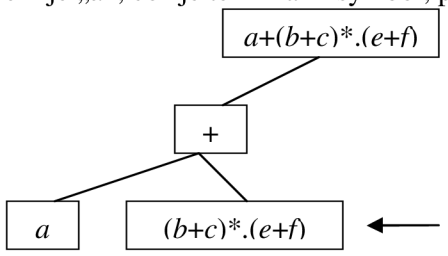
Druhým načteným symbolem je operátor „+“. Druhý operand je zbývající část výrazu „(*b*+*c*)*.(*e*+*f*)“. Vytvoří se nový strom, který se připojí jako levý potomek. Kořenem se stane operátor „+“, potomky operandy „*a*“ a „(*b*+*c*)*.(*e*+*f*)“.
Parsují se oba nově vytvořené listy.

3.



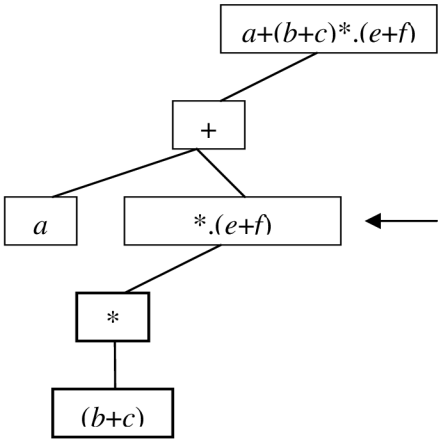
Načteným symbolem je „*a*“, což je terminální symbol, parsování je ukončeno.

4.



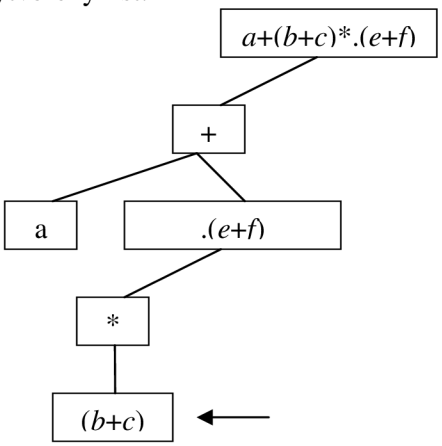
Prvním načteným symbolem je „(*b*+*c*)“, je zapamatován jako první operand.

5.



Druhým načteným symbolem je operátor „*“. Vytvoří se nový strom, který se připojí jako levý potomek. Kořenem se stane operátor „*“, potomkem je operand „(*b*+*c*)“.
Parsuje se nově vytvořený list.

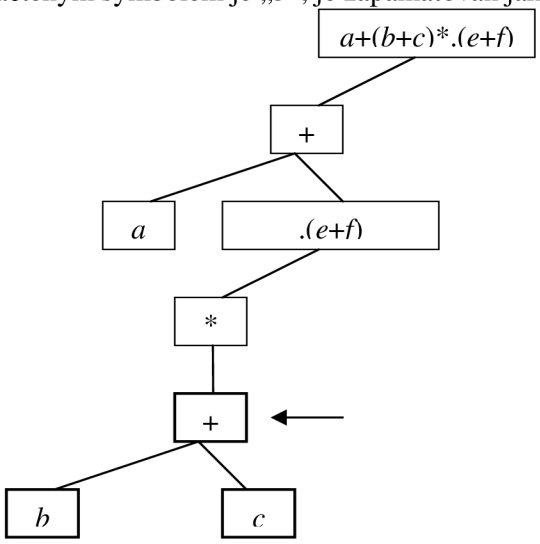
6.



Prvním načteným symbolem je „(“. Odpovídající uzavírající závorka je posledním symbolem, závorky jsou odstraněny a pokračuje se nad tímto řetězcem.
Parsuje se opět vytvořený list.

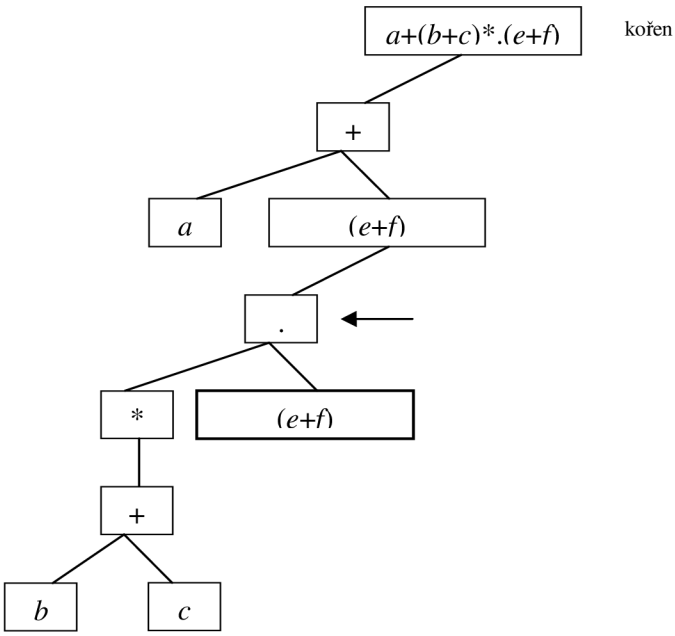
7. Prvním načteným symbolem je „*b*“, je zapamatován jako první operand.

8.



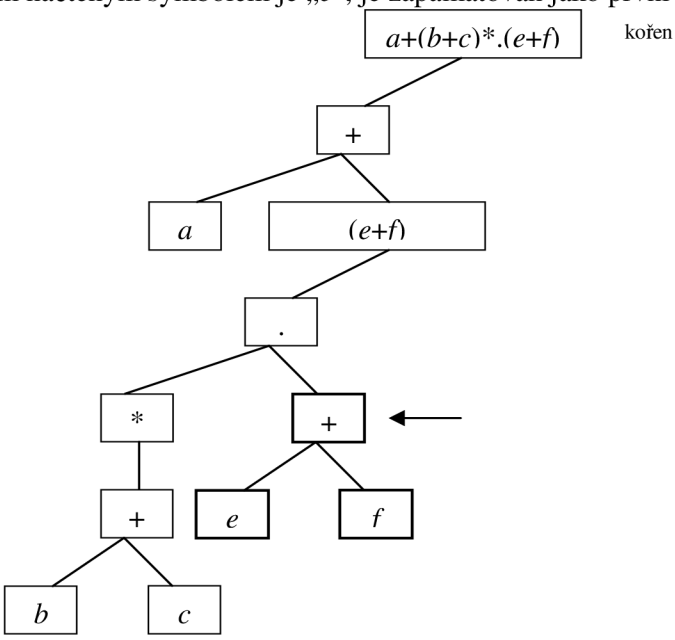
Druhým načteným symbolem je operátor „+“. Vytvoří se nový strom, který se připojí jako levý potomek. Kořenem se stane operátor „+“, potomky jsou operandy „*b*“ a „*c*“.
Konec parsovaného řetězce.

9.



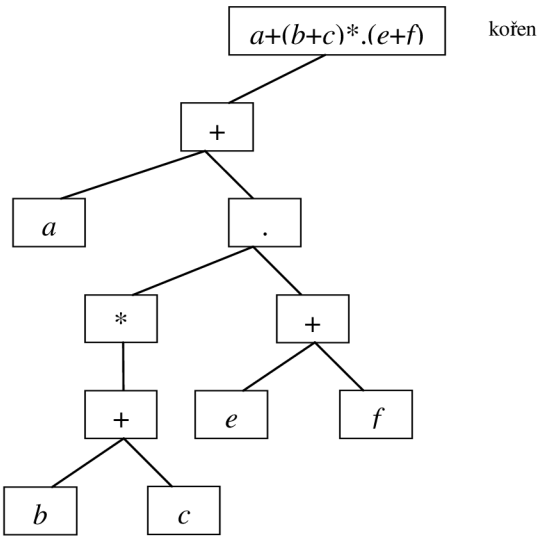
Třetím načteným symbolem je operátor „.“. Vytvoří se nový strom, který se připojí jako levý potomek. Kořenem se stane operátor „.“, levým potomkem je levý podstrom, pravým potomkem je operand „(*e*+*f*)“.
Parsuje se nově vytvořený list.

10. Prvním načteným symbolem je „(“. Odpovídající uzavírající závorka je posledním symbolem, závorky jsou odstraněny a pokračuje se nad tímto řetězcem.
Parsuje se opět vytvořený list.
11. Prvním načteným symbolem je „*e*“, je zapamatován jako první operand.
12.



Druhým načteným symbolem je operátor „+“. Vytvoří se nový strom, kořenem se stane operátor „+“, potomky jsou operandy „*e*“ a „*f*“.
Konec parsovaného řetězce.

13.



Parsovaný řetězec „(*b*+*c*)*.(*e*+*f*)" byl dočten do konce, kořen podstromu se odstraní.

14. Konec řetězce „*a*+(*b*+*c*)*.(*e*+*f*)", parsování proběhlo úspěšně, výsledný strom reprezentuje regulární výraz.

Obrázek 5.7: Příklad konstrukce stromu pro textový řetězec „a+(b+c)*.(e+f)"

Parsování zcela přirozeně zachovává prioritu operátorů, jak je uvedeno v kapitole 2.3. Nejvyšší prioritu má operátor „*", dále „." a nejnižší prioritu má operátor „+". Chování je v zadaném řetězci možné změnit správným uzávorkováním, priorita závorek je nadřazena prioritě operátorů.

5.1.8 Grafický výstup

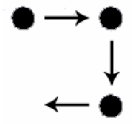
Grafický výstup konečného automatu je realizován do souborového formátu JPEG s maximální kvalitou. Vytvořil jsem tři rozdílné metody grafického výstupu pro deterministické konečné automaty, pro nedeterministické konečné automaty je možné použít pouze jednu z nich. Každá metoda vytváří grafický výstup jiným postupem, výsledek je tedy vždy závislý na vstupních datech.

Všechny metody grafického výstupu jsou použitelné až do počtu několika desítek stavů automatu. Přehlednost závisí především na počtu přechodů a počtu terminálních symbolů, s jejich vzrůstajícím počtem se přehlednost snižuje. Výhodou je však především obrovská rychlost,

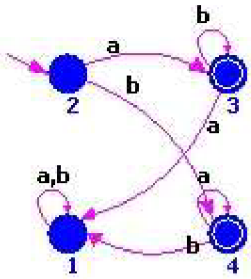
Grafický výstup byl implementován s důrazem na vysokou rychlost, která je u webové aplikace velmi důležitá. Všechny algoritmy provádí pouze jediný průchod přes množinu stavů automatu a ihned je vykreslují, včetně všech přechodů z aktuálně vykreslovaného stavu.

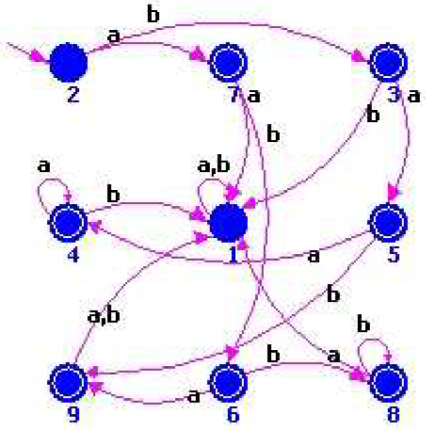
5.1.8.1 JpegOutputCube

Grafický výstup je umístěn do boxu, který má na začátku velikost čtvercové mřížky o rozměrech počet stavů × počet stavů. Počáteční stav je umístěn do levého horního rohu, každý dosud nevykreslený stav se umístí vpravo od předchozího stavu. Takto se postupuje až k pravému kraji, po jeho dosažení se posune v mřížce o řádek dolů a pokračuje se zpět k levému kraji.



Obrázek 5.8: Umístění stavů v modelu Kostkový.





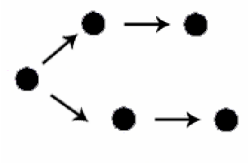
Obrázek 5.9, 5.10: Příklady grafického výstupu konečného automatu vytvořené metodou Kostkový.

Kostkový model je určen pouze pro deterministické konečné automaty, pro nedeterministické konečné automaty není implementován. Podává velmi dobré výsledky především při nízkém počtu terminálních symbolů.

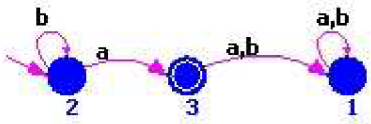
5.1.8.2 JpegOutputFlow

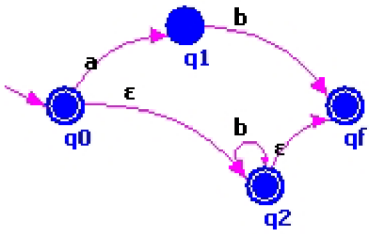
Grafický výstup se umísťuje do sloupců od levého okraje k pravému. Do prvního sloupce se umístí počáteční stav, do druhého sloupce stavy, do kterých existují přechody z počátečního stavu. Takto se postupuje se všemi ještě neumístěnými stavy.

Stavy v jednom sloupci nejsou umístěny na stejné x-ové souřadnici, ale jsou vždy posunuty o konstantní vzdálenost od středu sloupce – střídavě vlevo a vpravo od středu sloupce. Tím nedochází k překrývání přechodů mezi stavy v jednom sloupci a zlepší se přehlednost modelu.



Obrázek 5.11: Umístění stavů v modelu Tokový.





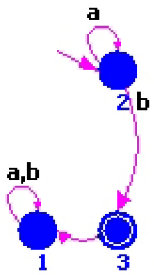
Obrázek 5.12, 5.13: Příklady grafického výstupu konečného automatu vytvořené metodou Tokový.

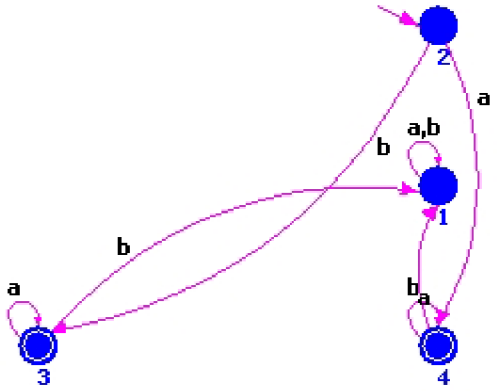
Tokový model je určen pro deterministické i nedeterministické konečné automaty. Nejlepších výsledků dosahuje u nedeterministických automatů převedených z regulárních výrazů. U deterministických konečných automatů není tento způsob rozložení stavů tak častý a dosahuje podobných výsledků jako Kostkový model.

5.1.8.3 JpegOutputRandom

Grafický výstup se umísťuje náhodně do boxu, který má na začátku velikost čtvercové mřížky o rozměrech počet stavů × počet stavů.

Náhodný model je nejjednodušším řešením grafického výstupu. Podává dobré výsledky jen v malém počtu případů, většina výsledků je ovlivněna špatným umístěním stavů – dochází ke shlukování stavů v jedné části obrazu nebo k rozptýlení stavů po celé ploše obrazu. Lepších výsledků by bylo možné dosáhnout implementací vhodného generátoru náhodných čísel.



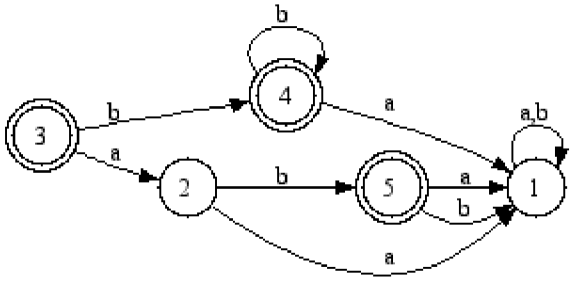


Obrázek 5.14, 5.15: Příklady grafického výstupu konečného automatu vytvořené metodou Náhodný.

5.1.8.4 Graphviz

Graphviz je open source software[1] pro vizualizaci grafů a diagramů. Podporuje mnoho vizuálních nastavení – pro barvy, fonty nebo tvary objektů. Obsahuje několik předdefinovaných stylů, styl „finite_state_machine“ je přímo určen pro generování konečných automatů.

[1] http://www.graphviz.org/



Obrázek 5.16: Příklad grafického výstupu konečného automatu vytvořeného knihovnou Graphviz.

Grafy jsou generovány z externích datových zdrojů. Po vytvoření datového zdroje, zpravidla textového souboru, je tento datový zdroj interpretován a vytvořen výstup ve formátu zadaném parametry. Datové zdroje se vytváří v jednoduchém jazyku.

```
digraph finite_state_machine
{
     rankdir=LR;
     size="5,5"
     node [shape = doublecircle]; 3 5 2;
     node [shape = circle];
     1 -> 1 [ label = "a,b" ];
     2 -> 1 [ label = "a" ];
     2 -> 4 [ label = "b" ];
     3 -> 6 [ label = "a" ];
     3 -> 5 [ label = "b" ];
     4 -> 2 [ label = "a" ];
     4 -> 1 [ label = "b" ];
     5 -> 1 [ label = "a" ];
     5 -> 1 [ label = "b" ];
     6 -> 1 [ label = "a" ];
}
```

Zdrojový kód 5.1: Příklad datového zdroje pro konečný automat v jazyku DOT.

Postup generování grafického výstupu

Metoda CreateGraphvizImage() deterministického konečného automatu.

1. Volání metody SaveGraphviz() – uložení textového souboru s vygenerovaným datovým zdrojem pro daný deterministický konečný automat.
2. Vytvoření externího procesu „dot.exe“ knihovny Graphviz, kterému je zadán jako parametr soubor vytvořený v kroku 1.
3. Spuštění externího procesu a čekání na jeho skončení. Aplikace dot zpracuje vytvořený datový zdroj a uloží grafický výstup do souboru.

Vygenerovaný grafický výstup se následně zobrazí v zobrazení výsledku. Čekání na dokončení aplikace dot je nutné, jinak by grafický výstup mohl být vygenerován až po zpracování webové stránky prohlížečem a grafický výstup by uživateli nebyl zobrazen.

Grafický model generovaný knihovnou Graphviz je implementován pouze pro deterministické konečné automaty, generování výstupu pro nedeterministické konečné automaty lze doimplementovat

velice snadno. Výsledky výstupu Graphviz lze považovat za referenční a výsledky výstupu implementovaných grafických metod s nimi porovnávat.

Ani jedna z implementovaných grafických metod nezahrnuje logiku pro kontrolu křížení přechodů nebo překrývání jednotlivých objektů (stavů, přechodů, popisů), výsledek tedy nebude nikdy dosahovat kvality referenčního výsledku generovaného knihovnou Graphviz.

5.1.8.5 Ukládání obrázků v cache prohlížečů

Grafický výstup se ukládá do souborů, tyto soubory mají stále stejný název (např. „nfa_flow.jpeg“ pro nedeterministický konečný automat vytvořený modelem Tokový). Internetové prohlížeče si ukládají obrázky do lokální cache. Aby se uživateli zobrazil vždy poslední vygenerovaný obrázek, jsou všechny obrázky v HTML kódu opatřeny časovým razítkem.

```
<img src="nfa_flow.jpeg?time=<% out.println(TimeInMillis); %>">
<img src="nfa_flow.jpeg?time=1178023512109">
```

Zdrojový kód 5.2: Obrázek v HTML kódu opatřený časovým razítkem.

5.1.9 Transformace nedeterministického konečného automatu s ε-kroky na ekvivalentní nedeterministický konečný automat bez ε-kroků

Algoritmus se skládá ze dvou částí – v prvním kroku jsou pro všechny stavy vytvořeny ε-uzávěry, v druhém kroku se s využitím těchto ε-uzávěrů vytváří přechodová funkce bez ε-kroků.

```
function HashSet EpsilonClosure ( HashSet Start )
{
      int Size = Start.size( );
      pro všechny stavy v Start
      {
            přidej množinu stavů, do kterých existuje přechod pod ε,
            do Start
      }
      if ( Start.size( ) > Size )
      {
            Start = EpsilonClosure( Start );
      }
      return Start;
}
```

Zdrojový kód 5.3: Výpočet ε-uzávěru pro stav automatu v pseudo-Java kódu.

Funkce EpsilonClosure vytváří rekurzívně množinu stavů, do kterých lze přejít z počátečního stavu pomocí ε-kroků. Funkce není z časového hlediska implementována optimálně, při opětovném rekurzivním volání se prochází celá množina stavů – přidáním druhého parametru, ve kterém by se předávaly nově přidané stavy a procházely by se pouze ty, by se zlepšila časová složitost (při zvýšení paměťových nároků).

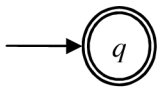
```
NFA RE = new NFA( );
přidej do RE všechny terminální symboly mimo ε
přidej do RE.Q a RE.F všechny stavy, uprav q0
pro všechny q ∈ Q spočítej
      EpsilonClosures = EpsilonClosure( new HashSet( q ) )
pokud EpsilonClosures( q0 ) ∩ F je neprázdný, přidej q0 do RE.F
pro všechny stavy q ∈ Q
{
      pro všechny přechody z q
      {
            t = terminální symbol přechodu
            pokud je t ε, pokračuj
            pokud je některý stav v EpsilonClosures( q ) koncový,
            přidej q do RE.F
            všechny stavy, do kterých existuje přechod z množiny stavů
            EpsilonClosures( q ) pod t, přidej do množiny M
            pro všechny stavy s ∈ M přidej EpsilonClosures( s )
            do množiny M'
            M = M ∪ M'
            přidej do přechodové funkce RE přechod M = δ(q, t)
      }
}
```

Zdrojový kód 5.4: Odstranění ε-kroků v nedeterministickém konečném automatu v pseudo-Java kódu.

Při vytváření přechodové funkce se prochází všechny stavy automatu a přidávají se nové přechody, které nahrazují původní *ε*-kroky. Pro každý přechod se nejprve vytvoří množina stavů M, do kterých lze přejít provedením *ε*-kroků (EpsilonClosure) a následně přes terminální symbol tohoto přechodu. Do této množiny M jsou pak přidány stavy, do nichž lze přejít *ε*-kroky (*ε*-uzávěry stavů z množiny M). Pro všechny stavy z této množiny je přidán do přechodové funkce automatu nový přechod.

5.1.10 Převod deterministického konečného automatu na regulární gramatiku

Algoritmus jsem implementoval přesně dle uvedeného formálního algoritmu. Při testování se však ukázalo, že algoritmus obsahuje chybu – pokud je počáteční stav automatu zároveň koncovým ($\varepsilon \in L(\mathcal{M})$), je nutné přidat nový kořen gramatiky a pro tento nový kořen gramatiky vytvořit nové pravidlo přepisující jej na *ε* a přidat mu všechna pravidla jako měl původní kořen gramatiky. Pokud však Q obsahuje pouze jediný stav (automat má pouze počáteční stav, který je zároveň koncovým), došlo by k vytvoření gramatiky, která má kořen přepisující se na *ε* a další neterminál, pro který neexistuje žádné pravidlo. V tom případě je nutné, aby se nevytvářel nový kořen gramatiky, pouze se přidá pravidlo přepisující kořen gramatiky na *ε*.

Vstupní automat: $\mathcal{M} = (\{q\}, \Sigma, \delta, q, \{q\})$	
Regulární gramatika dle formálního algoritmu: $G = (\{q, S\}, \Sigma, P, S)$	P: $S \rightarrow \varepsilon$ $q \rightarrow$
Regulární gramatika dle upraveného algoritmu: $G = (\{q\}, \Sigma, P, q)$	P: $q \rightarrow \varepsilon$

Tabulka 5.6: Převedená regulární gramatika dle formálního a implementovaného algoritmu.

5.1.11 Minimalizace deterministického konečného automatu

Minimalizace deterministického konečného automatu (s totální přechodovou funkcí) se provádí postupným vytvářením relací $\equiv_i$. Algoritmus je inicializován vytvořením relace $\equiv_0$, tedy rozdělením stavů na koncové a nekoncové. Každá relace je určena množinou skupin, kdy v každé skupině jsou nerozlišitelné stavy. V každé následující relaci se provádí kontrola, zda stavy ve skupině jsou stále nerozlišitelné. V případě, že skupina obsahuje rozlišitelné stavy, dochází k rozdělení. Pokud se neprovedlo žádné rozdělení, cyklus končí, byla nalezena relace $\equiv$. Tato relace je převedena na výsledný minimální automat.

```
if ( Q.size( ) == F.size( ) )
      return this.Clone( );

MinRelation MR0 = new MinRelation( 0 );
StateGroup NonFStates = new StateGroup( 1 );
StateGroup FStates = new StateGroup( 2 );
všechny stavy z F přidej do FStates, ostatní do NonFStates

MinRelation Current = MR0;
MinRelation Next = new MinRelation( );
while ( Current.CreateNextMinRelation( this.Delta, Next ) )
{
      Current = Next;
      Next = new MinRelation( );
}
převeď Current na DFA;

function CreateNextMinRelation ( TreeMap Delta, MinRelation Next )
{
      pro každou StateGroup v relaci vygeneruj příslušnosti stavů
      v této StateGroup

      pro každou StateGroup v relaci proveď rozdělení, pokud některý
      stav v této StateGroup má jinou příslušnost než ostatní

      return bylo provedeno rozdělení
}
```

Zdrojový kód 5.5: Minimalizace deterministického konečného automatu pseudo-Java kódu.

Tímto způsobem vytváření minimálního automatu je možné sledovat jednotlivé kroky a prezentovat tak uživateli konstrukce relací $\equiv_i$. Doimplementování této funkcionality je snadné, stačí vyjít z kódu použitého pro převod relace $\equiv$ na výsledný minimální automat.

5.1.11.1 Složitost algoritmu

Složitost algoritmu je určena úvodními kroky (průchod přes všechny stavy + vytvoření tříd), hlavní částí algoritmu – vytváření relací, a převodem relace na konečný automat. Vytvoření tříd se provede v konstantním čase c, průchod přes všechny stavy má časovou složitost $|Q|$.

Funkce GenerateSignatures ve třídě StateGroup (vygenerování příslušností stavů) prochází přes všechny své stavy a pro tyto stavy prochází všechny přechody. Složitost je $|\{q \mid q \in \text{StateGroup}\}| \times |\Sigma|$.

Funkce Divide ve třídě StateGroup (rozdělení, pokud má některý prvek v StateGroup jinou příslušnost) prochází přes všechny své stavy a má tedy složitost $|\{q \mid q \in \text{StateGroup}\}|$.

Funkce CreateNextMinRelation prochází dvakrát přes všechny StateGroup, v prvním průchodu se volá funkce GenerateSignatures, v druhém průchodu funkce Divide. Složitost je:

$|\mathcal{M}/\equiv_i| \times (|\{q \mid q \in \text{StateGroup}\}| \times |\Sigma|) + |\mathcal{M}/\equiv_i| \times |\{q \mid q \in \text{StateGroup}\}| = |Q| \times |\Sigma| + |Q|$

Převod na konečný automat je realizován průchodem přes všechny StateGroup, z každé je vybrán jeden stav a pro tento stav jsou do výsledného minimálního automatu přidány všechny přechody. Složitost je tedy $|\mathcal{M}/\equiv_i| \times |\Sigma|$.

Poznámka: V uvedeném značení představuje $|\{q \mid q \in \text{StateGroup}\}|$ počet stavů ve skupině (StateGroup), $|\mathcal{M}/\equiv_i|$ značí počet StateGroup v příslušné relaci, $|\Sigma|$ a $|Q|$ jsou kardinality množin terminálních symbolů a stavů konečného automatu.

Určení počtu průchodů cyklem while při vytváření relací závisí na vstupním automatu. V nejlepším případě jsou všechny stavy automatu nerozlišitelné a cyklus bude ukončen po jediném průchodu. V nejhorším případě jsou všechny stavy automatu rozlišitelné a při každém průchodu bude provedeno právě jedno rozdělení, počet průchodů se bude rovnat počtu stavů ($|Q|$). Pokud se v každé relaci provede dělení u ½ skupin, skončí algoritmus po log $|Q|$ průchodech cyklem.

Celkem tedy dostáváme:

$c + |Q| + |\Sigma| \times |Q| \times |Q| + |\mathcal{M}/\equiv_i| \times |\Sigma| = O(|\Sigma| \times |Q| \times |Q|)$

Časová složitost vytvořeného algoritmu je $O(|\Sigma| \times |Q| \times |Q|)$, zároveň v závislosti na vstupním automatu se může blížit optimální hodnotě uvedené u Hopcroftova algoritmu.

5.1.11.2 Optimální algoritmus

Hopcroftův algoritmus vytváří minimální automat z deterministického konečného automatu. Časová složitost algoritmu je $O(|\Sigma| \times |Q| \log |Q|)$.

Algoritmus využívá následující funkce:

- Funkce ADD má dva argumenty a přidává novou podmnožinu do množiny podmnožin.
- Funkce EXTRACT je výběrová funkce.

- Funkce SPLIT má tři argumenty, rozděluje druhý argument (podmnožinu) na dvě podmnožiny podle rozdělovače (první argument) a přechodů (třetí argument).

$$(B \cap T^{-1}(C, a), B \cap {}^{c}T^{-1}(C, a)) = \text{SPLIT}(B, C, a)$$

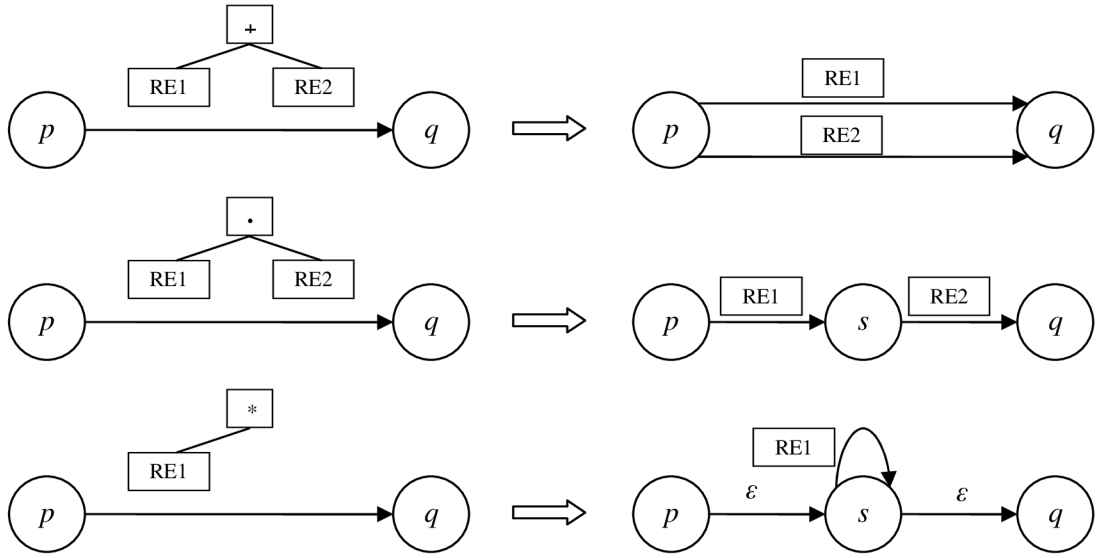
Minimalizace konečného automatu.

```
L := Ø;
if |F| < |Q/F| then
     C0 := Q/F; C1 := F; ADD(C1, L)
else
     C1 := Q/F; C0 := F; ADD(C1, L)
end if
P := {C0, C1};
while L ≠ Ø do
     C = EXTRACT(L)
     for all a ∈ Σ do
            for all B ∈ P do
                  (B', B'') = SPLIT(B, C, a)
                  if |B'| < |B''| then
                        B := B''; ADD(B', P); ADD(B', L);
                  else
                        B := B'; ADD(B'', P); ADD(B'', L);
                  end if
            end for
     end for
end while
```

Algoritmus 5.1: Hopcroftův algoritmus pro minimalizaci deterministického konečného automatu.

5.1.12 Převod regulárního výrazu na konečný automat

Převod regulárního výrazu ze stromu získaného parsováním textového řetězce na konečný automat je realizován podle tří pravidel pro každý operátor, která jsou prezentována na obrázku 5.17.



Obrázek 5.17: Pravidla pro převod regulárního výrazu na konečný automat.

Algoritmus prochází vytvořený strom od kořene směrem k listům (do hloubky). Listy představují terminální symboly, jsou přidány do Σ a do přechodové funkce je přidán přechod mezi stavy pod tímto terminálním symbolem. Vnitřní uzly stromu představují operátory, provede se akce dle uvedených pravidel.

```
RE_Tree je strom získaný parsováním, reprezentující regulární výraz.
Vytvoř nový NFA FA;
Vytvoř nový počáteční stav q0 a nový koncový stav qF pro FA;
ExpandRegExpTree( q0, qF, RE_Tree, FA );

function ExpandRegExpTree ( State S1, State S2, RegExpTree T, NFA FA )
{
      // Pokud je strom listem
      if ( T.IsLeaf( ) )
      {
            přidej terminál T.RegExp do FA.Sigma;
            přidej přechod δ(S1, T.RegExp) = S2 do FA.Delta;
      }
      // jinak obsahuje operátor
      else
      {
            switch ( T.RegExp )
            {
                  case +:
                        ExpandRegExpTree( S1, S2, T.L, FA );
                        ExpandRegExpTree( S1, S2, T.R, FA );
                  case .:
                        vytvoř nový stav S a přidej jej do FA.Q;
                        ExpandRegExpTree( S1, S, T.L, FA );
                        ExpandRegExpTree( S, S2, T.R, FA );
                  case *:
                        vytvoř nový stav S a přidej jej do FA.Q;
                        přidej přechod δ(S1, ε) = S do FA.Delta;
                        přidej přechod δ(S, ε) = S2 do FA.Delta;
                        ExpandRegExpTree( S, S, T.L, FA );
            }
      }
}
```

Zdrojový kód 5.6: Převod regulárního výrazu na konečný automat v pseudo-Java kódu.

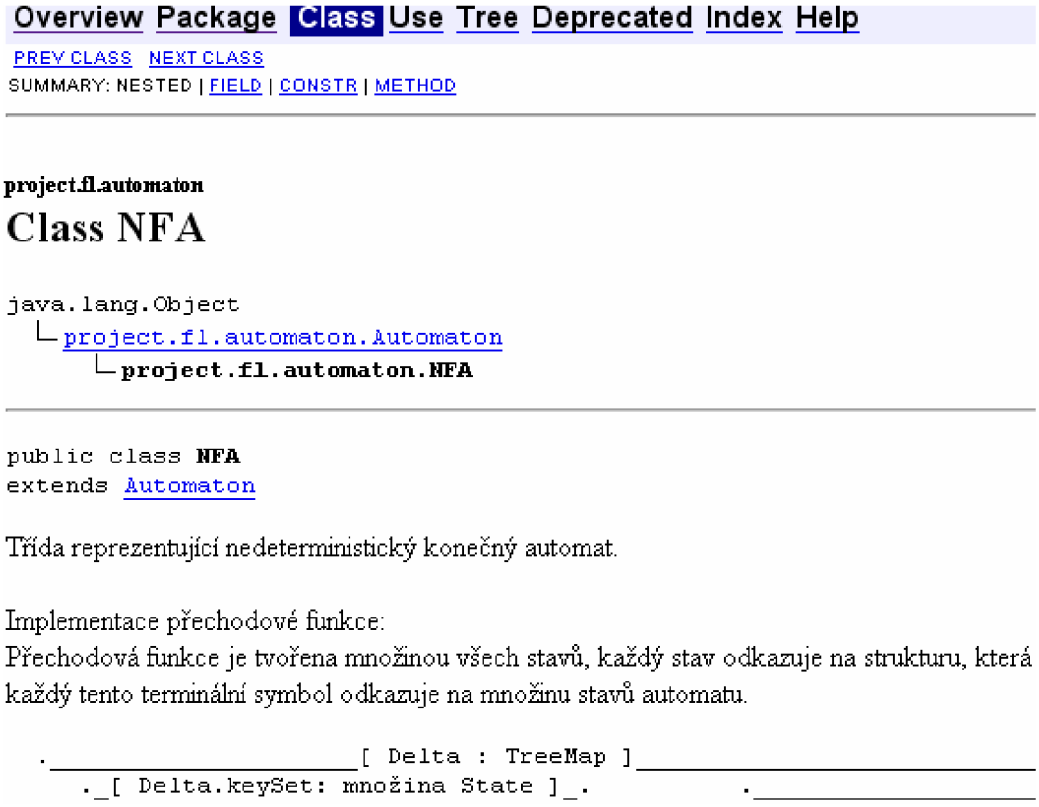
Časová složitost algoritmu je rovna počtu uzlů vstupního stromu – funkce ExpandRegExpTree je provedena právě jednou pro každý uzel, její časová složitost je konstantní.

5.1.13 Ostatní algoritmy

Ostatní algoritmy jsou jednoduché (např. odstranění nedosažitelných stavů je založeno na stejném principu jako uvedená funkce EpsilonClosure – rekurzívní přidávání stavů do množiny dosažitelných) a jsou implementovány přesně dle uvedených formálních algoritmů. Jejich zdrojové kódy jsou součástí přílohy.

5.1.14 Dokumentace

Dokumentace je zpracována nástrojem Javadoc a je přímo dostupná z vytvořené webové aplikace. Obsahuje popis všech balíků, které tvoří aplikační vrstvu, všech tříd z těchto balíků, a všech veřejných (public) metod.tříd (samozřejmě včetně parametrů). Při popisu tříd jsem dbal na maximální detailnost, především u popisů automatů a jejich atributů, u všech modelů jsou zároveň uvedeny příklady použití (pro programátory), které v oficiální dokumentaci často chybějí.


Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | CONSTR | METHOD

project.fl.automaton

Class NFA

java.lang.Object
  └ project.fl.automaton.Automaton
    └ project.fl.automaton.NFA

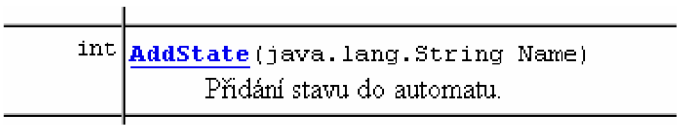
public class NFA
extends Automaton

Třída reprezentující nedeterministický konečný automat.

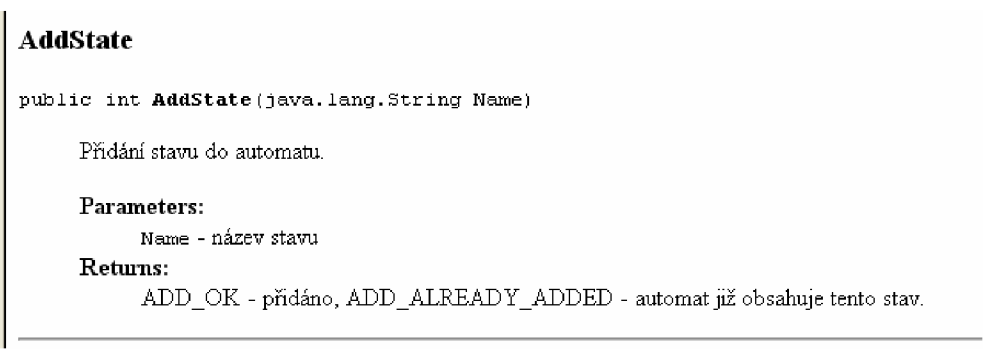
Implementace přechodové funkce:
Přechodová funkce je tvořena množinou všech stavů, každý stav odkazuje na strukturu, která každý tento terminální symbol odkazuje na množinu stavů automatu.

._____[Delta : TreeMap]_____
._[Delta.keySet: množina State]_. .____

Obrázek 5.18: Popis třídy NFA vygenerovaný nástrojem Javadoc.


int AddState(java.lang.String Name)
Přidání stavu do automatu.

Obrázek 5.19: Popis metody AddState ve třídě NFA vygenerovaný nástrojem Javadoc.


AddState

public int AddState(java.lang.String Name)

Přidání stavu do automatu.

Parameters:
Name - název stavu
Returns:
ADD_OK - přidáno, ADD_ALREADY_ADDED - automat již obsahuje tento stav.

Obrázek 5.20: Detailní popis metody AddState ve třídě NFA vygenerovaný nástrojem Javadoc.

5.2 Prezentační vrstva

Prezentační vrstva je tvořena stránkami v JSP, které generují dynamický obsah v HTML, aplikacemi v JavaScriptu pro interakci s uživatelem na straně klienta a grafickým výstupem samotné webové aplikace.

5.2.1 Struktura

Kořenový adresář webové aplikace obsahuje všechny JSP stránky (.jsp) včetně vkládaných souborů obsahujících části HTML kódu (.html), JavaScriptové aplikace (.js), kaskádové styly (.css), vygenerované grafické výstupy a sedm podadresářů. Adresář „download“ obsahuje všechny soubory, které webová aplikace nabízí ke stažení. Adresář „graphviz“ obsahuje aplikaci „dot.exe“ a příslušné knihovny pro generování grafického výstupu knihovnou Graphviz. V adresářích „img“ a „help“ se nachází obrázky používané ve webové aplikaci a nápovědě. Adresář „javadoc“ obsahuje dokumentaci vygenerovanou nástrojem Javadoc. Adresáře „META-INF“ a „WEB-INF“ slouží k popisu webové aplikace, její struktury, a obsahují především přeložené třídy aplikační vrstvy.

5.2.1.1 Popis souborů v kořenovém adresáři

index.jsp

Soubor index.jsp představuje vstupní bod webové aplikace. Podle parametrů předaných metodou GET předává řízení ostatním JSP stránkám. Pokud není žádný parametr zadán, je zobrazena vstupní stránka s informacemi o projektu.

functions.menu.jsp

Soubor functions.menu.jsp obsahuje dvě funkce, které vkládají do výstupního HTML kódu hlavní menu a pravé menu.

functions.project.fl.automaton.jsp

Soubor functions.project.fl.automaton.jsp obsahuje funkce, které vkládají do výstupního HTML kódu formátovaný textový výstup konečného automatu. Tyto funkce jsou použity vždy při zobrazení výsledků, jejich úpravou lze změnit způsob zobrazení konečných automatů v celé webové aplikaci.

functions.project.fl.regular_grammar.jsp

Soubor functions.project.fl.regular_grammar.jsp obsahuje funkce, které vkládají do výstupního HTML kódu formátovaný textový výstup regulární gramatiky. Platí pro ně totéž, jako u konečných automatů – jejich úpravou lze změnit způsob zobrazení regulárních gramatik v celé webové aplikaci.

main.jsp

Soubor main.jsp zobrazuje vstupní stránku s informacemi o projektu. Samotný kód HTML se nachází v souboru main.html.

help.jsp

Soubor help.jsp zobrazuje nápovědu k webové aplikaci, zahrnující popis aplikace, ovládání a popis implementovaných algoritmů. Samotný kód HTML pro nápovědu se nachází v souboru help.html.

re.jsp

Soubor re.jsp implementuje rozhraní pro regulární výrazy.

rg.jsp

Soubor rg.jsp implementuje rozhraní pro regulární gramatiky. Pro usnadnění zadávání gramatiky je využit klientský JavaScript, který se nachází v souboru rg.js.

fa.jsp

Soubor fa.jsp implementuje rozhraní pro konečné automaty. Pro usnadnění zadávání automatu je využit klientský JavaScript, který se nachází v souboru fa.js.

style.css

Soubor style.css obsahuje definici kaskádových stylů pro celou webovou aplikaci.

javascript.js

Soubor javascript.js provádí kontrolu, zda je klientský JavaScript v prohlížeči povolen.

5.2.2 Vzhled a ovládání

Celá webová aplikace je koncipována s jednoduchým a příjemným vzhledem, který má za úkol nerušit uživatele a nestrhávat jeho pozornost od toho důležitého, tedy práce se samotnými modely. Rozhraní je uživatelsky přívětivé, využívá pouze základní formulářové prvky, navíc je doplněno aplikacemi v JavaScriptu, aby zadávání modelů bylo výrazně rychlejší a příjemnější.

Hned první odkaz v pravém menu odkazuje na nápovědu, kde lze nalézt podrobnosti k ovládání celé aplikace i mnoho dalších důležitých informací, uživateli je tedy v případě problémů vždy nabídnuta rychlá pomoc.

5.2.2.1 Menu

Menu, umístěné v horní části obrazovky, umožňuje rychlou navigaci mezi jednotlivými částmi aplikace. Jedná se o úvodní stránku a jednotlivé modely. Část, ve které se uživatel právě nachází, je zvýrazněna.

Obrázek 5.21: Menu webové aplikace pro rychlý přechod mezi jednotlivými modely.

5.2.2.2 Pravé menu

Menu, umístěné v pravé části obrazovky, obsahuje odkazy na nápovědu, dokumenty ke stažení a informaci o klientském JavaScriptu.

Nápověda je rozdělena do několika částí, uživateli je zde prezentován popis celé aplikace, kompletní ovládání aplikace včetně doprovodných obrázků, v závěru jsou uvedeny jednotlivé algoritmy převodů mezi modely, které aplikace implementuje.

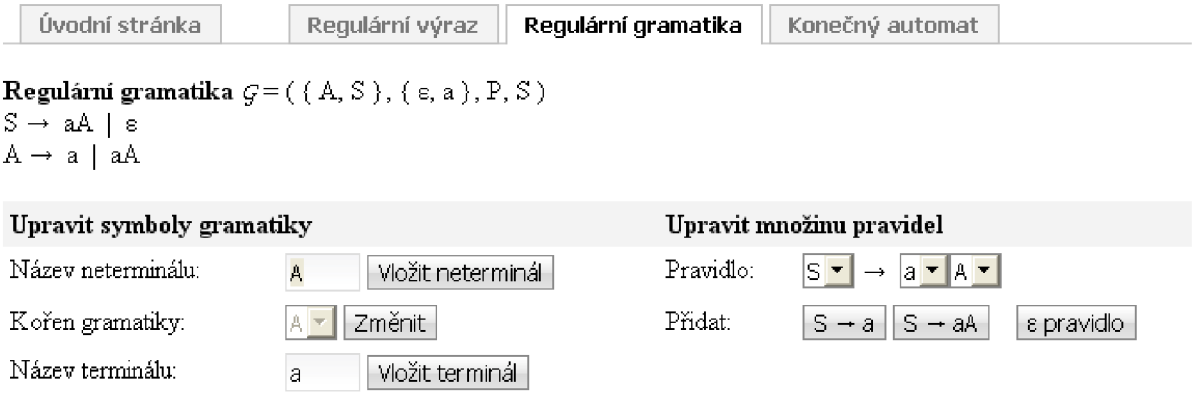
5.2.2.3 Regulární výraz

Regulární výraz lze vložit přímo zapsáním do vkládacího pole nebo s využitím připravených tlačítek. Tlačítka jsou rozdělena do čtyř skupin - operátory, levá a pravá závorka, symboly abecedy (ze kterých lze skládat terminální symboly), symbol ε (lze zadat jako „epsilon“).

Obrázek 5.22: Zadávání regulárního výrazu.

5.2.2.4 Regulární gramatika

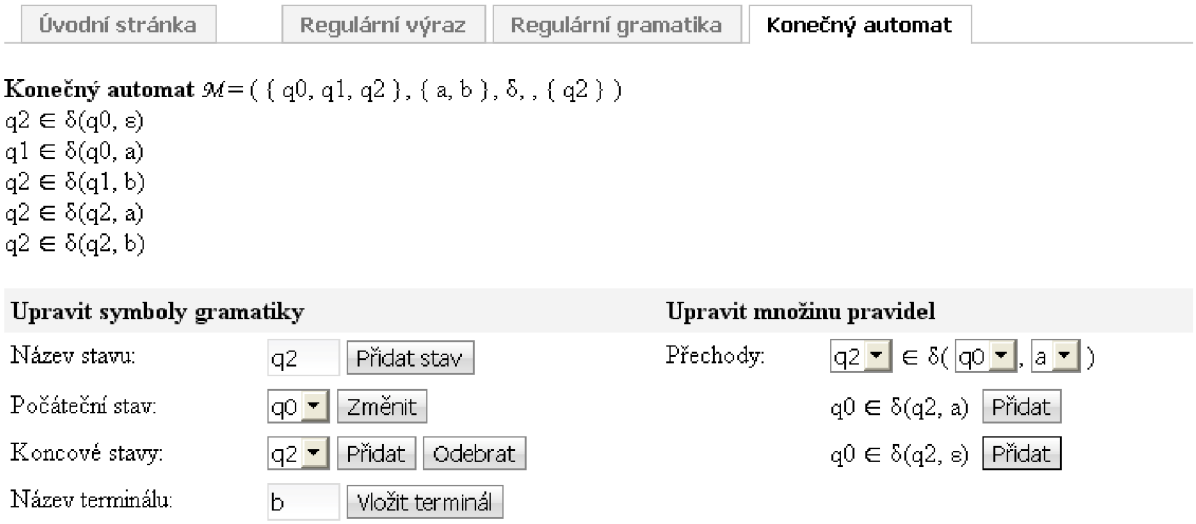
Regulární gramatika se zadává postupným vkládáním jednotlivých symbolů gramatiky a pravidel gramatiky. Každá provedená změna se ihned projeví v zobrazení gramatiky.



Obrázek 5.23: Zadávání regulární gramatiky.

5.2.2.5 Konečný automat

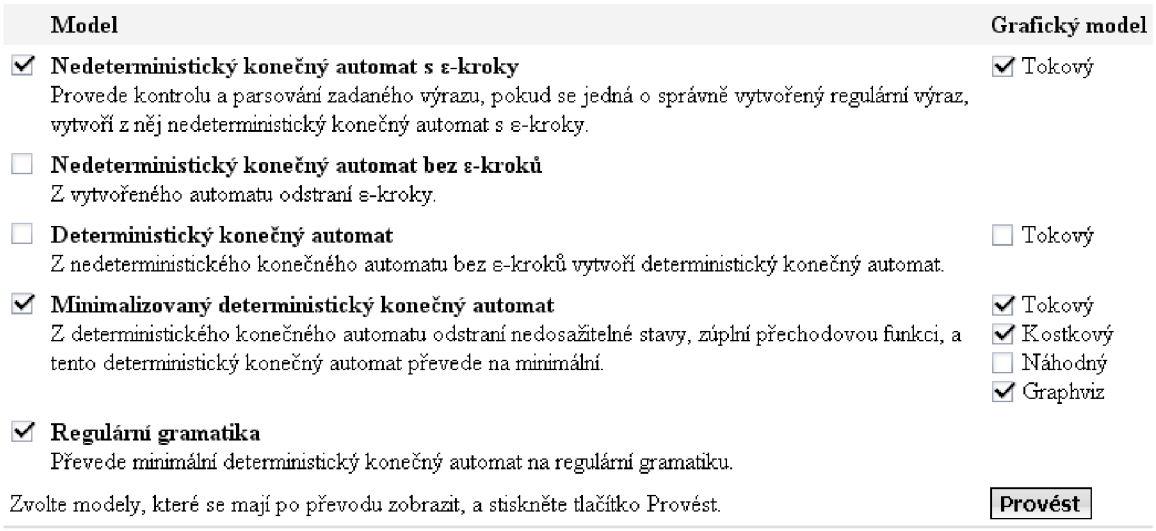
Konečný automat se zadává postupným vkládáním stavů a terminálních symbolů a následně přechodových pravidel. Každá provedená změna se ihned projeví v zobrazení automatu.



Obrázek 5.24: Zadávání konečného automatu.

5.2.2.6 Výběr výstupních modelů

U každého modelu se nachází výběr výstupních modelů a u konečných automatů lze vybrat i grafické modely. Výběr výstupních modelů se u každého modelu drobně liší (např. nepřevádí se regulární gramatika na regulární gramatiku).

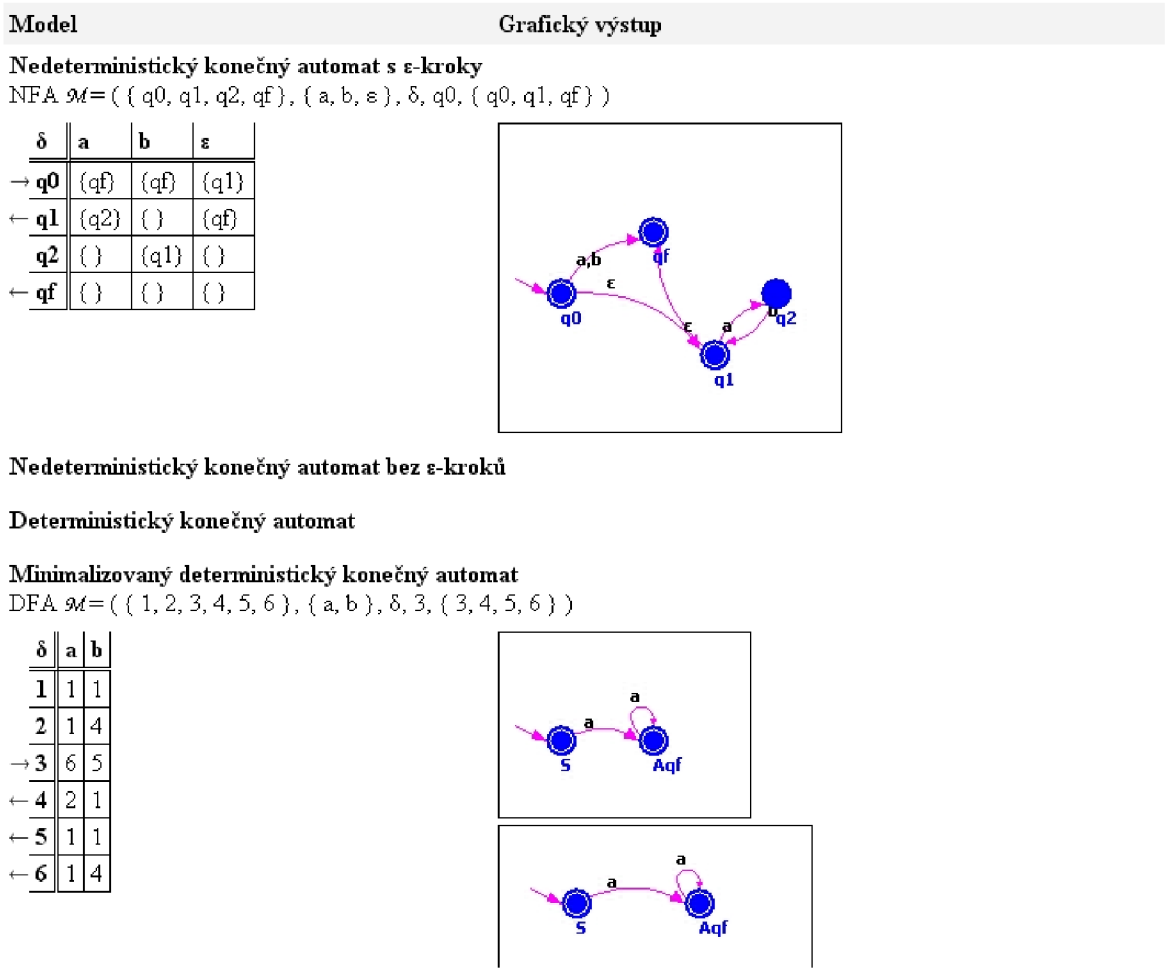


Obrázek 5.25: Výběr výstupních modelů.

Lze vybrat všechny modely, pro každý vybraný model se po převodu zobrazí detailní výpis. Zároveň s každým vybraným konečným automatem se zobrazí každý vybraný grafický model.

5.2.2.7 Zobrazení výsledku

Pro každý vybraný model (Výběr výstupních modelů) se zobrazí detailní výpis.



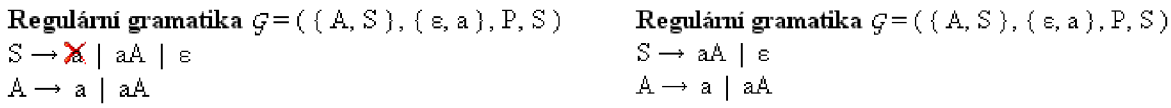
Obrázek 5.26: Zobrazení výsledku – zvolené modely a grafická reprezentace konečných automatů.

5.2.3 JavaScriptové aplikace

JavaScriptové aplikace jsou určeny pro uživatelsky přívětivější zadávání jednotlivých modelů. Při použití pouze servletů by bylo nutné při každé změně modelu zasílat data na server a posílat zpět ze serveru odpověď. S využitím JavaScriptu je možné vše realizovat na straně klienta a na server odeslat až celý vytvořený model. Nutným omezením pro uživatele je vyžadovaný klientský JavaScript. Dle statistik nepoužívá při přístupu na internetové stránky klientský JavaScript (pouze) několik procent uživatelů. Na úvodní stránce je proto toto vše zmíněno, navíc vzhledem k zaměření aplikace se předpokládá, že bude využívána především uživateli se znalostí informačních technologií, kteří si umí vše nastavit.

5.2.3.1 Regulární gramatika

JavaScriptová aplikace u regulární gramatiky umožňuje zadávat terminální i neterminální symboly gramatiky, měnit kořen gramatiky a zadávat a odebírat pravidla gramatiky. Všechny prováděné operace se ihned objeví v náhledu gramatiky.

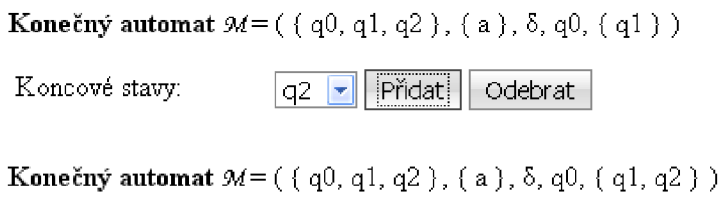


Obrázek 5.27, 5.28: Kliknutím na pravidlo (výběr je indikován červeným křížkem) se pravidlo odebere z gramatiky, změna se ihned projeví v náhledu gramatiky.

Na klientské straně lze efektivně řešit i různá omezení, plynoucí např. z definice. Jelikož regulární gramatika nesmí obsahovat pravidlo $S \rightarrow \varepsilon$, pokud se S vyskytuje na pravé straně některého pravidla, lze jednoduše omezit, aby uživatel nemohl takové pravidlo vložit.

5.2.3.2 Konečný automat

JavaScriptová aplikace u konečného automatu umožňuje zadávat stavy, měnit počáteční stav, měnit množinu koncových stavů, zadávat terminální symboly a zadávat a odebírat přechody přechodové funkce. Je tedy možné kompletně zadat celý nedeterministický konečný automat.



Obrázek 5.29: Přidání koncového stavu k automatu, změna se ihned projeví v náhledu automatu.

5.2.3.3 Indikace klientského JavaScriptu

V pravém menu se nachází indikace, zda je v prohlížeči povolen klientský JavaScript. Menu je vždy generováno (a tedy i zobrazeno v prohlížeči) s textem „Vypnutý“ a obrázkem v červeném provedení („vypnuto“). Za uzavíracím elementem menu je do HTML kódu vložen zdrojový kód JavaScriptu, který změní text na „Zapnutý“ a změní i obrázek na „zapnuto“. Tento JavaScriptový kód je proveden pouze v případě, že klientský JavaScript je v prohlížeči povolen. Pokud povolen není, kód se neprovede a zůstává původní text „Vypnutý“.

Obrázek 5.30, 5.31: Textový i grafický indikátor, zda je v prohlížeči povolen klientský JavaScript.

6 Závěr

Nastudoval jsem modely, které se používají v moderní teorii jazyků – konečné automaty, regulární gramatiky a regulární výrazy, a zpracoval část podstatnou pro vypracování projektu (formální popis jednotlivých modelů, převody mezi modely). Nad těmito modely jsem navrhl webovou aplikaci. Webová aplikace je dvouvrstvá, skládá se z aplikační a prezentační vrstvy. Aplikační vrstva realizuje jednotlivé modely formou servletů jazyka Java. Prezentační vrstva je tvořena dynamickými stránkami JSP, které tyto servlety využívají, pracují s nimi a zobrazují je uživateli.

Navrženou aplikaci jsem implementoval, přičemž původní návrh byl rozšířen o funkce, které nejsou součástí zadání. Zároveň bylo nutné některé formální algoritmy upravit, protože jejich přesná implementace obsahovala chyby.

Součástí aplikační vrstvy je grafický výstup konečných automatů. Vytvořil jsem tři odlišné zobrazovací metody, z nichž jedna je určena i pro nedeterministické konečné automaty. Do webové aplikace jsem navíc zahrnul nástroj Graphviz pro generování grafů, který slouží pro porovnávání vytvořených grafických výstupů.

Projekt je dále možné rozšířit o prezentaci kroků při vytváření minimálního automatu, v implementaci je již tato funkcionalita zahrnuta. Do webové aplikace by bylo vhodné zahrnout nastavení atributů pro grafický výstup, které je taktéž implementováno a není využito.

V rámci diplomové práce jsem implementoval webovou aplikaci, která plně obsáhla rozsah zadání. Nabízí se využití například pro studijní účely.

Literatura

[1] Meduna, Alexander. *Automata and Languages: Theory and Applications*. London: Springer Verlag, 2000. ISBN 1-85233-074-0.

[2] Černá, Ivana, Křetínský, Mojmír, Kučera, Antonín. *Automaty a formální jazyky I.* Brno: Masarykova univerzita, Fakulta informatiky, 2002. Učební text, verze 1.3.

[3] Domaratzki, Michael, Okhotin, Alexander, Salomaa, Kai, Yu, Sheng. *Implementation and Application of Automata: 9th International Conference CIAA 2004.* Springer, 2005. ISBN 3540243186.

[4] Berstel, Jean, Carton, Olivier. *On the complexity of Hopcroft's state minimization algorithm.* Konference CIAA, 2004. Dostupný na URL: <http://www.liafa.jussieu.fr/~carton/Publications/Hopcroft/Article/hopcroft.pdf>.

[5] Baclet Manuel, Pagetti, Claire. *Around Hopcroft's Algorithm.* Research Report LSV-06-12, 2006. Dostupný na URL: <http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2006-12.pdf>.

[6] Eckel, Bruce. *Myslíme v jazyku Java, knihovna programátora*. Praha: Grada Publishing, 2001. ISBN 80-247-9010-6.

[7] Eckel, Bruce. *Myslíme v jazyku Java, knihovna zkušeného programátora*. Praha: Grada Publishing, 2001. ISBN 80-247-0027-1.

[8] Burd, Barry. *JSP: JavaServer Pages Podrobný průvodce.* Computer Press, 2003. ISBN 80-7226-804-X.

[9] Hall, Marty. *JAVA servlety a stránky JSP.* Neocortex, 2001. ISBN 80-86330-06-0.

[10] Branický, Marek. *Java Servlets*. Interval.cz [online]. 2003. Dostupný na URL: <http://interval.cz/serialy/java-servlets/>.

[11] Kosek, Jiří. *HTML – tvorba dokonalých stránek: Podrobný průvodce.* 1. vyd. Praha: Grada Publishing, 1998. ISBN 80-7169-608-0.

[12] Žára, Jiří, Beneš, Bedřich, Sochor, Jiří, Felkel, Petr. *Moderní počítačová grafika*. Computer Press, 2005. ISBN 80-251-0454-0.

[13] The Apache Software Foundation. *Apache Tomcat*. Dostupný na URL: <http://tomcat.apache.org/>.

[14] Sun Microsystems Inc. *Java Technology*. Dostupný na URL: <http://java.sun.com/>.

[15] Sun Microsystems Inc. *Java Platform, Enterprise Edition.* Dostupný na URL: <http://java.sun.com/javaee/>.

[16] Sun Microsystems Inc. *JavaServer Pages Technology.* Dostupný na URL: <http://java.sun.com/products/jsp/>.

[17] Sun Microsystems Inc. *Java Servlet Technology*. Dostupný na URL: <http://java.sun.com/products/servlet/>.

[18] Sun Microsystems Inc. *Javadoc Tool.* Dostupný na URL: <http://java.sun.com/j2se/javadoc/>.

[19] The Apache Software Foundation. *Apache Ant*. Dostupný na URL: <http://ant.apache.org/>.

[20] W3C. *HTML 4.01 Specification.* Dostupný na URL: <http://www.w3.org/TR/html4/>.

[21] W3C. *Cascading Style Sheets.* Dostupný na URL: <http://www.w3.org/Style/CSS/>.

[22] Mozilla Developer Center. *About JavaScript.* Dostupný na URL: <http://developer.mozilla.org/en/docs/About_JavaScript>.

Seznam příloh

Příloha A

CD

Na přiloženém kompaktním disku je umístěna elektronická podoba této diplomové práce ve formátu DOC (Microsoft Word 2003) a PDF (Portable Document Format 1.4).

Součásti aplikace

Jednotlivé části vytvořené aplikace jsou uloženy do následujících podadresářů:

`src`

Jsou zde umístěny veškeré zdrojové kódy – aplikační vrstvy (třídy jazyka Java) a prezentační vrstvy (stránky JSP, HTML, kaskádové styly, JavaScriptové aplikace, obrázky, knihovny pro Graphviz).

`classes`

Přeložené třídy aplikační vrstvy.

`javadoc`

Dokumentace vygenerovaná nástrojem Javadoc.

`install`

Soubory pro instalaci aplikace – xpodho05.war, webapp.war, javadoc.zip, project.zip.

Příloha B

Instalace

Instalace webové aplikace je velice jednoduchá. V instalačním adresáři na CD se nachází soubor xpodho05.war – webový archív, který obsahuje všechny soubory (včetně adresářové struktury) nutné pro běh webové aplikace (JSP, HTML, obrázky, knihovny). V případě serveru Tomcat jej stačí umístit do podadresáře tomcat/webapps, server jej sám zpracuje, rozbalí a vytvoří podadresář xpodho05. Pak již je možné spustit aplikaci v prohlížeči zadáním URL. Pokud máme Tomcat nainstalován na lokálním počítači, může být URL v závislosti na nastavení serveru např.:

http://localhost:8080/xpodho05/index.jsp

U jiných JSP kontejnerů (Sun Application Server) může být adresář pro nahrání webového archívu jiný, případně je nutné provést další kroky.

Pokud mají být součástí webové aplikace i soubory pro stažení, nahrajeme ostatní soubory z adresáře install (webapp.war, javadoc.zip, project.zip) do adresáře download v hlavním adresáři webové aplikace.