VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

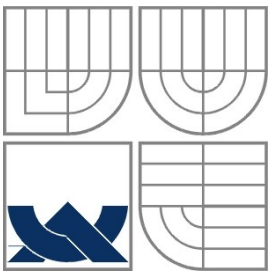
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM SIMULACE ZÁSOBNÍKOVÝCH AUTOMATŮ

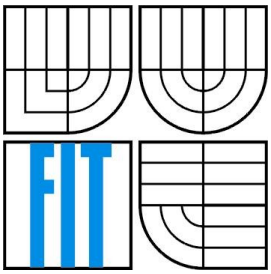
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUDMILA KUŽELOVÁ
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM SIMULACE ZÁSOBNÍKOVÝCH AUTOMATŮ

DEMONSTRATION PROGRAM OF SIMULATION OF PUSHDOWN AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUDMILA KUŽELOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN LUKÁŠ, Ph.D.

BRNO 2009

Abstrakt

Tato práce se zabývá zpracováním simulace zásobníkových a rozšířených automatů. Součástí práce je demonstrační program, který umožňuje uživateli zadat nastavení automatu a následně spustit simulaci přijetí vstupního řetězce automatem.
Vybrána je vhodná metoda prohledávání stavového prostoru, z oboru umělé inteligence – prohledávání do šířky – Breadth First Search.

Abstract

The thesis deals with working simulation of stack and extended automata. Part of the work is a demonstration application, enabling the user to enter a configuration of automaton and start simulating the proccess of accepting a input sequence.
Chosen is a fitting method of searching a state space, using the Breadth First Search method.

Klíčová slova

simulace, automaty, rozšířené, zásobníkové, BFS

Keywords

simulation, automata, extended, pushdown, BFS

Citace

Ludmila Kuželová: Demonstrační program simulace zásobníkových automatů, bakalářská práce, Brno, FIT VUT v Brně, rok 2009

Demonstrační program simulace zásobníkových automatů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Romana Lukáše, Pd.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

……
Ludmila Kuželová
Datum (17.5.2009)

Poděkování

Chtěla bych poděkovat především svému vedoucímu této bakalářské práce Ing. Romanu Lukášovi, za odborné rady, které mi v průběhu této práce poskytnul. Velké díky patří také všem blízkým, kteří mě během doby, kterou jsem strávila prací na tomto díle, podporovali a dodávali sílu.

Obsah

1 Úvod

Cílem této práce je vytvořit aplikaci, která umožní názorně předvést ukázku práce různých zásobníkových automatů. Důvodem ke vzniku této bakalářské práce je to, že teorie zásobníkových a rozšířených zásobníkových automatů je probírána v rámci výuky formálních jazyků a překladačů na fakultě. Tento program bude užitečný zejména během přednášek a cvičení k názorným ukázkám funkce zásobníkových automatů studentům. Také jim umožní samostatně navrhnout a vyzkoušet simulovat vlastní automat, nebo si prověřit běh automatů probíraných při výuce a přípravě ke zkoušce

Základním předpokladem je nastudovat teorii zásobníkových a rozšířených automatů, vybrat z oblasti umělé inteligence vhodnou metodu prohledávání stavového prostoru tak, aby bylo možné co nejlepším způsobem zjistit, zda nadefinovaný automat zpracovává vstupní řetězec. Pro vytvoření simulačního programu je proto potřeba zvolit také vhodný programovací jazyk a nastudovat vytváření uživatelských rozhraní.

Samotný program by měl umožnit simulaci různých typů zásobníkových automatů, ukládání a načítaní jejich konfigurace v souborech XML. Pro konečný výpis výsledku bude umožněno také krokování jednotlivých přechodů a následné grafické zobrazení obsahu zásobníku, stavu vstupního řetězce a aktuálního stavu. Při tvorbě této práce jsem nejvíce čerpala ze studijních skript předmětů „Formální jazyky a překladače“ a „Základy umělé inteligence“.

Rozsáhlá druhá kapitola obsahuje základní definice obohacené o názorné příklady týkající se automatů a gramatik. Podkapitoly této části se postupně zabývají základními pojmy, operacemi nad jazyky, definicemi gramatik a dalšími modely pro formální popisy jazyků. Poslední podkapitola obsahuje definice právě pro zásobníkové i rozšířené zásobníkové automaty. Dále je zde také vysvětlena teorie BFS (prohledávání do šířky) pro prohledávání stavového prostoru.

Třetí kapitola se zabývá návrhem celého programu spolu s popisem některých použitých postupů při výběru nejjednoduššího a uživatelsky příjemného ovládání této aplikace. Samotné implementaci je věnována kapitola čtvrtá. V závěru práce shrnuji dosáhnuté poznatky, možnosti dalšího vývoje aplikace i případné využití programu.

2 Základní definice

2.1 Základní pojmy

2.1.1 Definice abecedy

Abeceda je neprázdná konečná množina prvků, které nazýváme symboly.

2.1.2 Definice řetězce nad danou abecedou

Nechť Σ je abeceda. Potom:

- je řetězec nad abecedou Σ.
- Jestliže x je řetězec nad abecedou Σ, $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

Symbol ε značí *prázdný řetězec*. Prázdný řetězec je takový řetězec, který neobsahuje žádný symbol. Symbolem Σ^* budeme značit množinu všech řetězců nad abecedou Σ. Příkladem abecedy může být $\{a, b\}$, řetězcem nad touto abecedou je například *ababba*.

2.1.3 Definice délky řetězce

Nechť x je řetězec nad abecedou Σ. Délka řetězce x, $|x|$, je definována následovně:

- Pokud $x = \varepsilon$, potom $|x| = 0$.
- Pokud $x = a_1 a_2 \ldots a_n$, kde $a_i \in \Sigma$ pro všechna $i = 1, 2, \ldots n$, potom $|x| = n$.

2.1.4 Formální jazyk

Nechť je dána abeceda Σ. Potom množinu L, pro kterou platí: $L \subseteq \Sigma^*$, nazveme *formálním jazykem* nad abecedou Σ.

2.2 Operace nad jazyky

2.2.1 Definice sjednocení dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Sjednocením jazyků L_1 a L_2 rozumíme jazyk $L_1 \cup L_2$, který je definován následovně:

$$L_1 \cup L_2 = \{x: x \in L_1 \vee x \in L_2\}$$

2.2.2 Definice průniku dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Průnikem jazyků L_1 a L_2 rozumíme jazyk $L_1 \cap L_2$, který je definován následovně:

$$L_1 \cap L_2 = \{x: x \in L_1 \wedge x \in L_2\}$$

2.2.3 Definice konkatenace dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Konkatenací jazyků L_1, L_2 rozumíme jazyk $L_1.L_2$, který je definován následovně:

$$L_1.L_2 = \{xy: x \in L_1 \wedge y \in L_2\}$$

2.2.4 Definice rozdílu dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Rozdílem jazyků L_1, L_2 rozumíme jazyk $L_1 - L_2$, který je definován následovně:

$$L_1 - L_2 = \{x: x \in L_1 \wedge x \notin L_2\}$$

2.2.5 Definice doplňku jazyka

Nechť L je formální jazyk nad abecedou Σ. Doplňkem jazyka L rozumíme jazyk $\overline{L}$, který je definován následovně:

$$\overline{L} = \Sigma^* - L$$

2.2.6 Definice iterace a pozitivní iterace jazyka

Nechť L je formální jazyk nad abecedou Σ. Iterací jazyka L rozumíme jazyk L^* a pozitivní iterací jazyka L rozumíme jazyk L^+, které jsou definovány následovně:

$L^0 = \{\varepsilon\}$

$L^n = L.L^{n-1}$

$L^* = \bigcup_{n \geq 0} L^n$

$L^+ = \bigcup_{n \geq 1} L^n$

2.3 Definice gramatik

2.3.1 Gramatiky typu 0 (neomezené gramatiky)

Neomezená gramatika je gramatika, která obsahuje nejobecnější tvar pravidel. Množinu všech jazyků, které jsou generovány nějakou neomezenou gramatikou, nazveme třídou rekurzivně vyčíslitelných jazyků nebo také třídou jazyků typu 0. Formální definice je následující:

2.3.1.1 Definice neomezené gramatiky

Neomezená gramatika G je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $x \rightarrow y$, kde $x \in (N \cup T)^* N (N \cup T)^*$ a $y \in (N \cup T)^*$,
- S je počáteční nonterminální symbol.

2.3.1.2 Definice přímé derivace u neomezené gramatiky

Nechť $G = (N, T, P, S)$ je neomezená gramatika, nechť $u, v \in (N \cup T)^*$ a $p = x \rightarrow y \in P$ je pravidlo. Pak říkáme, že *uxv přímo derivuje uyv* podle pravidla p a zapisujeme $uxv \Rightarrow uyv$ $[p]$ nebo také zkráceně $uxv \Rightarrow uyv$.

2.3.1.3 Definice sekvence derivací u neomezené gramatiky

Nechť $G = (N, T, P, S)$ je neomezená gramatika.

- Nechť $u \in (N \cup T)^*$. Pak říkáme, že u derivuje *v 0-krocích* u a zapisujeme $u \Rightarrow^0 u$ $[\varepsilon]$ nebo také zkráceně $u \Rightarrow^0 u$.
- Nechť $u_0, u_1, \ldots, u_n \in (N \cup T)^*$, nechť pro všechna $i = 1, ..., n$ platí: $u_{i-1} \Rightarrow u_i$ $[p_i]$. Pak říkáme, že u_0 *derivuje v n-krocích* u_n a zapisujeme $u_0 \Rightarrow^n u_n$ $[p_1 p_2 \ldots p_n]$ nebo také zkráceně $u_0 \Rightarrow^n u_n$.
- Nechť $u \Rightarrow^n v$ $[\pi]$ pro nějaké $n \geq 1$, $u, v \in (N \cup T)^*$. Pak říkáme, že u *netriviálně derivuje* v a zapisujeme $u \Rightarrow^+ v$ $[\pi]$ nebo také zkráceně $u \Rightarrow^+ v$.
- Nechť $u \Rightarrow^n v$ $[\pi]$ pro nějaké $n \geq 0$, $u, v \in (N \cup T)^*$. Pak říkáme, že u *derivuje* v a zapisujeme $u \Rightarrow^* v$ $[\pi]$ nebo také zkráceně $u \Rightarrow^* v$.

2.3.1.4 Definice jazyka generovaného neomezenou gramatikou

Nechť $G = (N, T, P, S)$ je neomezená gramatika. Jazyk generovaný neomezenou gramatikou G (budeme jej označovat $L(G)$) je definován následovně:

$$L(G) = \{w : w \in T^* \wedge S \Rightarrow^* w\}.$$

2.3.1.5 Definice rekurzivně vyčíslitelného jazyka

Jazyk je rekurzivně vyčíslitelný právě tehdy, když je generován nějakou neomezenou gramatikou.

2.3.2 Gramatiky typu 1 (kontextové gramatiky)

Kontextová gramatika je speciální neomezená gramatika obsahující pravidla tvaru $x \rightarrow y$, pro která navíc platí, že $|x| \leq |y|$. Množinu všech jazyků, které jsou generovány nějakou kontextovou gramatikou, nazveme třídou kontextových jazyků nebo také třídou jazyků typu 1. Formální definice je následující:

2.3.2.1 Definice kontextové gramatiky

Kontextová gramatika G je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $x \rightarrow y$, kde $x \in (N \cup T)^* N (N \cup T)^*$ a $y \in (N \cup T)^*$, přičemž $|x| \leq |y|$,
- S je počáteční nonterminální symbol.

2.3.2.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného kontextovou gramatikou

Tyto definice jsou shodné s příslušnými definicemi u neomezené gramatiky.

2.3.2.3 Definice kontextového jazyka

Jazyk je kontextový právě tehdy, když je generován nějakou kontextovou gramatikou.

2.3.3 Gramatiky typu 2 (bezkontextové gramatiky)

Bezkontextová gramatika je speciální kontextová gramatika obsahující pravidla tvaru $x \rightarrow y$, pro která navíc platí, že řetězec x je pouze jeden nonterminální symbol. Množinu všech jazyků, které jsou generovány nějakou bezkontextovou gramatikou, nazveme třídou bezkontextových jazyků nebo také třídou jazyků typu 2. Formální definice je následující:

2.3.3.1 Definice bezkontextové gramatiky

Bezkontextová gramatika G je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $A \rightarrow x$, kde $A \in N$ a $x \in (N \cup T)^*$,
- S je počáteční nonterminální symbol.

Název „bezkontextová“ vychází ze skutečnosti, že nonterminál A se může přepsat na x bez ohledu na okolní *kontext.* Jazyky generované bezkontextovými gramatikami se nazývají bezkontextové.

2.3.3.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného bezkontextovou gramatikou

Tyto definice jsou shodné s příslušnými definicemi u neomezené gramatiky.

2.3.3.3 Definice bezkontextového jazyka

Jazyk je bezkontextový právě tehdy, když je generován nějakou bezkontextovou gramatikou.

2.3.3.4 Příklad

Uvažujme následující bezkontextovou gramatiku:

$G = (N, T, P, S)$, kde:

$N = \{S\}$

$T = \{x, y, z, +, -, *, /, (,)\}$,

$P = \{S \rightarrow x, S \rightarrow y, S \rightarrow z, S \rightarrow S + S, S \rightarrow S - S, S \rightarrow S * S, S \rightarrow S / S, S \rightarrow (S)\}$

Tato bezkontextová gramatika generuje jazyk všech aritmetických výrazů obsahující proměnné x, y, z.

2.3.4 Gramatiky typu 3 (Pravé lineární gramatiky)

Pravá lineární gramatika je speciální bezkontextová gramatika obsahující pravidla tvaru $A \rightarrow x$, pro která navíc platí, že řetězec x obsahuje buď samé terminální symboly a nebo terminální symboly zakončené pouze jedním nonterminálním symbolem. Množinu všech jazyků, které jsou generovány nějakou pravou lineární gramatikou, nazveme třídou regulárních jazyků nebo také třídou jazyků typu 3. Formální definice je následující:

2.3.4.1 Definice pravé lineární gramatiky

Pravá lineární gramatika G je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $A \rightarrow xB$ nebo $A \rightarrow y$, kde $A, B \in N$ a $x, y \in T^*$,
- S je počáteční nonterminální symbol.

2.3.4.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného lineární gramatikou

Tyto definice jsou shodné s příslušnými definicemi u bezkontextové gramatiky.

2.3.4.3 Příklad

Uvažujme následující bezkontextovou gramatiku:
$G = (N, T, P, S)$, kde:
$N = \{S, A\}$
$T = \{x, y\}$,
$P = \{S \rightarrow xS, S \rightarrow \varepsilon, S \rightarrow yA, A \rightarrow yS, A \rightarrow xA\}$

Tato bezkontextová gramatika generuje jazyk obsahující řetězce na abecedou $\{x, y\}$, které obsahují sudý počet „y“.

2.3.4.4 Definice regulárního jazyka

Jazyk je regulární právě tehdy, když je generován nějakou pravou lineární gramatikou.

2.3.5 Věta o Chomského hierarchii jazyků

Označme symbolem L_{RE} třídu rekurzivně vyčíslitelných jazyků, symbolem L_{CS} třídu kontextových jazyků, symbolem L_{CF} třídu bezkontextových jazyků a symbolem L_{REG} třídu regulárních jazyků. Mezi těmito třídami jazyků platí vztah:

$$L_{REG} \subset L_{CF} \subset L_{CS} \subset L_{RE}$$

2.4 Další modely pro formální popisy jazyků

2.4.1 Konečný automat

Konečný automat (*FA* z anglického *finite automaton*) je teoretický výpočetní model používaný v informatice pro studium vyčíslitelnosti a obecně formálních jazyků. Popisuje velice jednoduchý počítač, který může být v jednom z několika stavů, mezi kterými přechází na základě symbolů, které čte ze vstupu. Množina stavů je konečná (odtud název), konečný automat nemá žádnou další paměť

kromě informace o aktuálním stavu. Konečný automat je velice jednoduchý výpočetní model, dokáže rozpoznávat pouze regulární jazyky. Konečné automaty se používají pro zpracování regulárních výrazů, např. jako součást lexikálního analyzátoru v překladačích .

2.4.1.1 Definice konečného automatu

Konečný automat M je pětice $M = (Q, T, R, q_0, F)$, kde:

Q je konečná množina stavů,

T je konečná vstupní abeceda,

R je konečná množina pravidel tvaru: $pa \rightarrow q$, kde $p, q \in Q, a \in T \cup \{\varepsilon\}$,

$q_0 \in Q$ je počáteční stav,

$F \subseteq Q$ je množina koncových stavů.

2.4.1.2 Definice konfigurace konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Potom konfigurací χ konečného automatu M je řetězec: $\chi = qw$, kde $q \in Q, w \in T^*$.

2.4.1.3 Definice přechodu konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Nechť pax, qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, x \in T^*, a \in T \cup \{\varepsilon\}$. Nechť $r = pa \rightarrow q \in R$ je pravidlo. Pak říkáme, že konečný automat M provede *přechod* z konfigurace pax do qx podle pravidla r a zapisujeme $pax \vdash qx$ $[r]$ nebo také zkráceně $pax \vdash qx$.

2.4.1.4 Definice sekvence přechodů u konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat.

- Nechť χ je konfigurace. M provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi$ $[\varepsilon]$ nebo zjednodušeně $\chi \vdash^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_n$ jsou konfigurace, nechť pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \vdash \chi_i$ $[r_i]$. Pak říkáme, že M provede n přechodů z χ_0 do χ_n. zapisujeme $\chi_0 \vdash^n \chi_n$ $[r_1 r_2 \ldots r_n]$ nebo také zkráceně $\chi_0 \vdash^n \chi_n$.
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 1$, potom zapisujeme $\chi_0 \vdash^+ \chi_n$
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, potom zapisujeme $\chi_0 \vdash^* \chi_n$

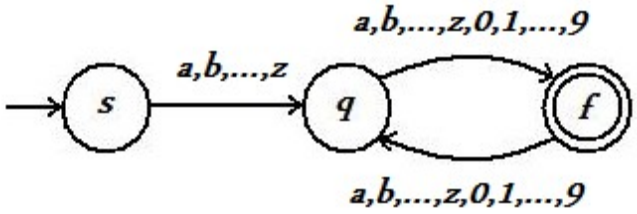
2.4.1.5 Definice jazyka přijímaným konečným automatem

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Potom jazyk přijímaný konečným automatem M, $L(M)$, je definován jako:

$$L(M) = \{w: w \in T^* \wedge q_0 w \vdash^* f \wedge f \in F\}$$

2.4.1.6 Znázornění konečného automatu + příklad

Konečný automat, který znázorňuje přijímání idenifikátorů sudé délky



2.4.2 Zásobníkový automat

Zásobníkový automat (*PDA* z anglického *pushdown automaton*) je teoretický výpočetní model používaný v informatice pro studium vyčíslitelnosti a obecně formálních jazyků. Popisuje jednoduchý počítač, který má jako pracovní paměť k dispozici pouze zásobník. Zásobníkový automat dokáže rozpoznávat bezkontextové jazyky.

2.4.2.1 Definice zásobníkového automatu

Zásobníkový automat M je sedmice $M = (Q, T, \Gamma, R, q_0, Z_0, F)$, kde:

Q je konečná množina vnitřních stavů,

T je konečná vstupní abeceda,

Γ je konečná zásobníková abeceda,

R je konečná množina pravidel tvaru: $Apa \rightarrow xq$, kde $p, q \in Q, a \in T \cup \{\varepsilon\}, A \in \Gamma, x \in \Gamma^*$

$q_0 \in Q$ je počáteční stav,

$Z_0 \in \Gamma$ je startovací symbol zásobníku,

$F \subseteq Q$ je množina koncových stavů.

2.4.2.2 Definice konfigurace zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat. Potom konfigurací χ zásobníkového automatu M je řetězec: $\chi = uqw$, kde $u \in \Gamma^*, q \in Q, w \in T^*$.

2.4.2.3 Definice přechodu zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat. Nechť $uApav$, $uxqv$ jsou dvě konfigurace zásobníkového automatu M, kde $p, q \in Q$, $v \in T^*$, $a \in T \cup \{\varepsilon\}$, $A \in \Gamma$, $u, x \in \Gamma^*$. Nechť $r = Apa \rightarrow xq \in R$ je pravidlo. Pak říkáme, že zásobníkový automat M provede *přechod* z konfigurace $uApav$ do $uxqv$ podle pravidla r a zapisujeme $uApav \vdash uxqv\ [r]$ nebo také zkráceně $uApav \vdash uxqv$.

2.4.2.4 Definice sekvence přechodů u zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat.

- Nechť χ je konfigurace. M provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi\ [\varepsilon]$ nebo zjednodušeně $\chi \vdash^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_n$ jsou konfigurace, nechť pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \vdash \chi_i\ [r_i]$. Pak říkáme, že M provede n přechodů z χ_0 do χ_n. zapisujeme $\chi_0 \vdash^n \chi_n\ [r_1 r_2 \ldots r_n]$ nebo také zkráceně $\chi_0 \vdash^n \chi_n$.
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 1$, potom zapisujeme $\chi_0 \vdash^+ \chi_n$
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, potom zapisujeme $\chi_0 \vdash^* \chi_n$

2.4.2.5 Definice jazyka přijímaným zásobníkovým automatem

Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je zásobníkový automat.

1. Jazyk přijímaný zásobníkovým automatem M přechodem do koncového stavu, značen jako $L_f(M)$, je definován:
$$L_f(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z \in \Gamma^*, f \in F\}$$
2. Jazyk přijímaný zásobníkovým automatem M s vyprázdněním zásobníku, značen jako $L_\varepsilon(M)$, je definován:
$$L_\varepsilon(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z = \varepsilon, f \in Q\}$$
3. Jazyk přijímaný zásobníkovým automatem M s přechodem do koncového stavu a s vyprázdněním zásobníku, značen jako $L_{f\varepsilon}(M)$, je definován:
$$L_{f\varepsilon}(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z = \varepsilon, f \in F\}$$

2.4.3 Rozšířený zásobníkový automat

Rozšířený zásobníkový automat je speciálním typem zásobníkového automatu, kdy rozdílem mezi těmito automaty je čtení ze zásobníku. V případě zásobníkového automatu lze z vrcholu zásobníku číst pouze jeden znak, zatímco rozšířený zásobníkový automat umožňuje čtení celého řetězce.

2.4.3.1 Definice rozšířeného zásobníkového automatu

Rozšířený zásobníkový automat M je sedmice $M = (Q, T, \Gamma, R, q_0, Z_0, F)$, kde:

Q je konečná množina vnitřních stavů,

T je konečná vstupní abeceda,

Γ je konečná zásobníková abeceda,

R je konečná množina pravidel tvaru: $Apa \rightarrow xq$, kde $p, q \in Q, a \in T \cup \{\varepsilon\}, A, x \in \Gamma^*$

$q_0 \in Q$ je počáteční stav,

$Z_0 \in \Gamma$ je startovací symbol zásobníku,

$F \subseteq Q$ je množina koncových stavů.

2.4.3.2 Definice konfigurace rozšířeného zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat. Potom konfigurací χ zásobníkového automatu M je řetězec: $\chi = uqw$, kde $u \in \Gamma^*, q \in Q, w \in T^*$.

2.4.3.3 Definice přechodu rozšířeného zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je rozšířený zásobníkový automat. Nechť $uApav$, $uxqv$ jsou dvě konfigurace rozšířeného zásobníkového automatu M, kde $p, q \in Q, v \in T^*, a \in T \cup \{\varepsilon\}, A, u, x \in \Gamma^*$. Nechť $r = Apa \rightarrow xq \in R$ je pravidlo. Pak říkáme, že rozšířený zásobníkový automat M provede *přechod* z konfigurace $uApav$ do $uxqv$ podle pravidla r a zapisujeme $uApav \vdash uxqv$ [r] nebo také zkráceně $uApav \vdash uxqv$.

2.4.3.4 Definice sekvence přechodů u rozšířeného zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat.

- Nechť χ je konfigurace. M provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi$ [ε] nebo zjednodušeně $\chi \vdash^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_n$ jsou konfigurace, nechť pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \vdash \chi_i$ [r_i]. Pak říkáme, že M provede n přechodů z χ_0 do χ_n. zapisujeme $\chi_0 \vdash^n \chi_n$ [$r_1 r_2 \ldots r_n$] nebo také zkráceně $\chi_0 \vdash^n \chi_n$.
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 1$, potom zapisujeme $\chi_0 \vdash^+ \chi_n$
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, potom zapisujeme $\chi_0 \vdash^* \chi_n$

2.4.3.5 Definice jazyka přijímaným rozšířeného zásobníkovým automatem

Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je zásobníkový automat.

- Jazyk přijímaný zásobníkovým automatem M přechodem do koncového stavu, značen jako $L_f(M)$, je definován:

$$L_f(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z \in \Gamma^*, f \in F\}$$

- Jazyk přijímaný zásobníkovým automatem M s vyprázdněním zásobníku, značen jako $L_\varepsilon(M)$, je definován:

$$L_\varepsilon(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z = \varepsilon, f \in Q\}$$

- Jazyk přijímaný zásobníkovým automatem M s přechodem do koncového stavu a s vyprázdněním zásobníku, značen jako $L_{f\varepsilon}(M)$, je definován:

$$L_{f\varepsilon}(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z = \varepsilon, f \in F\}$$

2.4.4 Metoda prohledávání stavového prostoru

Prohledávání do šířky (Breadth-first search, neboli BFS) je metoda prohledávání stavového prostoru postupným procházením uzlů grafů a jejich expandováním. Nevyužívá žádnou heuristickou analýzu, pouze prochází všechny uzly a pro každý nalezený uzel vytvoří všechny jeho následovníky. Přitom si poznamenává předchůdce jednotlivých uzlů. Díky tomuto zaznamenávání je poté vytvořen strom nejkratších cest k jednotlivým uzlům z kořene.

Z hlediska algoritmu, veškeré následovníky uzlu získané expandujícím uzlem jsou vkládané do seznamu typu fronta - *FIFO*. To znamená, že první uzel, který do fronty vstoupil, ji také první opustí. Uzly, které se dostávají do fronty a jsou právě v danou chvíli vyšetřovány na jejich následníky, jsou označeny jako *OPEN* a nakonec uzly, které již byly z fronty vybrány a nadále se s nimi nebude pracovat, jsou označeny jako *CLOSED*. Uzly ve frontě *CLOSED* již nebudou v tomto běhu algoritmu znovu prozkoumávány ani expandovány.

Na počátku algoritmu se provede inicializace každé fronty. Poté se kořenový uzel umístí jako první do fronty *OPEN*. V případě simulace zásobníkového automatu bude počátečním uzlem vstupní řetězec společně s počátečním symbolem na zásobníku a počátečním stavem. Nyní se začne generovat seznam následníků – řetězců vzniklých z původního pomocí pravidel automatu. Pro prohledávání se rozběhne cyklus, který běží dokud není fronta prázdná nebo dokud není nalezen cílový stav – uzel. Na počátku cyklu z fronty vyjmeme uzel. Pro všechny následníky tohoto uzlu budeme zjišťovat, jestli se nenalézají ani ve frontě *OPEN*, ani *CLOSED*. Pokud se nenachází v žádné frontě, uzel je expandován a jeho následníci umístěni do fronty *OPEN*. Nakonec se nastaví také pravidlo, kterým byly tyto uzly generovány. Po nastavení všech těchto vlastností je nový uzel vložen do fronty. Samotný expandovaný uzel nakonec je vyjmut z fronty *OPEN* a uzel přesunut do fronty *CLOSED*.

Pokud však narazíme na uzel, který se již nachází v seznamu *CLOSED*, potom je jen vybrán z fronty *OPEN* a není nikam přesunut. Znamená to, že je zahozen, neboť jeho uložení a následné

procházení by vedlo k pouhému opakování již prohledaných stavu od začátku. Také by to značně a naprosto zbytečně prodloužilo délku, společně se složitostí, hledání výsledné posloupnosti.

Možný by byl také postup opačný – jako kořenový, první uzel použít finální stav automatu. Zpětnou aplikací „obrácených“ verzí pravidel poté lze generovat následníky – varianty řetězců přijímaných zadaným automatem. Zajímavé by bylo porovnat časovou a paměťovou náročnost obou přístupů – takový však není účel ani prostor této práce.

Seznam *OPEN* je fronta *FIFO* (first in, first out), do které jsou ukládány postupně expandované uzly, se kterými se právě pracuje. To znamená, že ve frontě jsou uzly, které představují jakoby "vlnu" provádění algoritmu.

Všechny tyto datové struktury jsou na počátku algoritmu inicializované. *FIFO* fronta je inicializovaná jako prázdná fronta. V poli vzdáleností mají všechny prvky na počátku hodnotu nekonečno. Nekonečno je nastaveno proto, jelikož by cesta byla nekonečně daleká neboli, že neexistuje. Opačná hodnota nula je nastavena na začátku pouze uzlu počátečnímu, ze kterého bude algoritmus začínat svůj běh. Nulová vzdálenost znamená, že se jedná o totožný uzel. Pole stavů je nastaveno na počátku na hodnoty *FRESH* pro všechny uzly mimo prvního, který má stav *OPEN*. Pole předchůdců je nastaveno na *null* pro všechny uzly. Nakonec se do fronty vloží uzel *s* (startovací uzel). Velikosti jednotlivých polí jsou nastaveny na velikost shodnou s počtem uzlů v grafu.

Časová i paměťová složitost algoritmu BFS je exponenciální - je dána výrazem $O(b^{d+1})$, kde *b* je tzv. *faktor větvení* (průměrný počet bezprostředních následníků každého uzlu), a *d* je hloubka nejlepšího řešení, tj. řešení, které se nachází v nejmenší hloubce.

2.4.4.1 Příklad použití metody BFS

Metodu BFS lze demonstrovat na klasické úloze dvou džbánů, kdy k dispozici jsou dva džbány – třílitrový a pětilitrový. Lze přelít vodu z jednoho džbánu do druhého, zcela jej naplnit nebo zcela vyprázdnit. Úkolem je odměřit přesně čtyři litry.

Uvedený způsob řešení metodou BFS již uvažuje použití vylepšené verze s frontami OPEN a CLOSED. Pro přehlednost je uveden pětilitrový džbán na prvním místě dvojice, druhé místo ve dvojici znázorňuje obsah třílitrového džbánu.

Nejdříve je vytvořena fronta OPEN a CLOSED, na vrchol fronty OPEN je umístěn výchozí stav – oba džbány jsou prázdné, tedy: 0. Open: [((0,0), nil)]; Closed: [];

Nyní startuje prohledávací algoritmus. Protože máme oba džbány prázdné, jeden z nich můžeme naplnit – vznikají tedy dva různé uzly – jeden představuje plný třílitrový, jeden plný pětilitrový džbán. 1. Open: [((4,0),(0,0)),((0,3),(0,0))]; Closed: [((0,0), nil)];

Nedosáhlo se koncového stavu, proto je nadále pokračováno v algoritmu. Postupně je expandován vrchní uzel z OPEN. Jedním z následníků je i uzel (0,0) – pro představu si lze tuto situaci

vybavit jako opětovné vylití pětilitrového džbánu. Aktuální uzel je však již uložen ve frontě CLOSED, proto jej nebudeme dále generovat. Ostatní generované uzly umisťujeme na konec fronty.
2. Open: [((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0))]; Closed: [((0,0), nil),((4,0),(0,0))];

Další kroky hledání a generování jsou následující:

3. Open: [((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3))]; Closed: [((0,0), nil),((4,0),(0,0)),((0,3),(0,0))];

4.Open: [((1,3),(4,0)),((3,0),(0,3))]; Closed: [((0,0), nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0))]; U tohoto kroku si povšimněte, že není generován žádný uzel – následník. Je to tím, že dva stavy, které mohou po stavu (4,3) nastat, jsou výsledkem vyprázdnění jednoho ze džbánů – tedy buď (4,0) nebo (0,3), tyto se však již nacházejí ve frontě CLOSED.

5.Open:[((3,0),(0,3)),((1,0),(1,3))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0))];

6.Open: [((1,0),(1,3)),((3,3),(0,3))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3))];

7.Open: [((3,3),(0,3)),((0,1),(1,0))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3)),((1,0),(1,3))];

8.Open: [((0,1),(1,0)),((4,2),(3,3))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3)),((1,0),(1,3)),((3,3),(0,3))];

9.Open: [((4,2),(3,3)),((4,1),(0,1))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3)),((1,0),(1,3)),((3,3),(0,3)),((0,1),(1,0))];

10.Open: [((4,1),(0,1)),((0,2),(4,2))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3)),((1,0),(1,3)),((3,3),(0,3)),((0,1),(1,0)),((4,2),(3,3))];

11.Open: [((0,2),(4,2)),((3,2),(4,1))]; Closed: [((0,0),nil),((4,0),(0,0)),((0,3),(0,0)),((4,3),(4,0)),((1,3),(4,0)),((3,0),(0,3)),((1,0),(1,3)),((3,3),(0,3)),((0,1),(1,0)),((4,2),(3,3)),((4,1),(0,1))];

12.Open: [((3,2),(4,1))]; Jak je patrné, v předcházejícím kroku bylo nalezeno řešení, které bylo určeno na počátku jako cílové. Nyní lze snadno pomocí seznamu CLOSED nalézt cestu od tohoto konečného stavu po počáteční, kdy k aktuálnímu stavu se v seznamu CLOSED vyskytuje vždy jen jedna varianta, která obsahuje předka daného stavu. Tudíž takto zapsaná výsledná cesta je následující (0,2),(4,2),(3,3),(3,0),(0,3),(0,0). Avšak je samozřejmé, že se jako výsledek uvádí cesta od počátečního stavu po koncový, finálním výsledkem tedy je (0,0),(0,3),(3,0),(3,3),(4,2),(0,2).

3 Návrh aplikace

3.1 Výběr programovacího jazyka

Při volbě implementačního prostředí bylo důležité zvážit několik faktorů.

1. Snadnou přenositelnost programu. Program bude provozován především na fakultě, kde existují dvě různé prostředí – platforma Windows a Unix / Linux. Musí být samozřejmě použitelný i pro studenty, z nichž drtivá většina opět využívá některého z výše uvedených operačních systémů. V úvahu také připadá operační systém MacOS – java runtime existuje také ve verzi pro tento systém.
2. Přívětivé uživatelské rozhraní.
3. Dostupnost knihoven jak pro funkčnost programu, tak pro grafické zpracování výsledku.

Při výběru jsem se rozhodovala mezi grafickým vývojovým prostředím Delphi od firmy Borland, tedy jazykem Object Pascal, a mezi objektovým programovacím jazykem Java od firmy Sun. Pro tvorbu demonstračního programu jsem upřednostnila jazyk Java.

Java je nezávislá na platformě, program napsaný v tomto jazyce je spustitelný prakticky na všech operačních systémech bez nutnosti kódových úprav. Obsahuje všechny potřebné funkce pro vytvořené této aplikace, je zdarma. Neustále se vyvíjí a zdokonaluje.

3.2 Koncepce programu

Účelem tohoto programu je vytvoření aplikace, která umožní uživateli nadefinovat si vlastní automat, kdy se bude jednat o dva typy automatů – rozšířený zásobníkový a zásobníkový automat. Nejdůležitější částí bude samozřejmě spuštění simulace, její následné vyhodnocení a zobrazení výsledku na výstupu. Z těchto důvodů je nutné vyřešit následující problémy:

1. Vytvořit pro uživatele jednoduché prostředí, v němž si bude moci zadat vlastní automat.
2. Nastavení automatu se musí řídit tím, o jaký automat se ve skutečnosti jedná(zásobníkový, rozšířený zásobníkový), protože konfigurace jednotlivých automatů a přijímání pravidel se liší.
3. Umožnit ukládání a zpětné načítání vytvořených automatů – zvolení vhodného formátu XML.
4. Možnost krokování výsledku, pokud simulace skončila úspěchem.
5. Zvolení vhodné metody prohledávání stavového prostoru z umělé inteligence.
6. Vyřešení determinismu x nedeterminismu.

3.2.1 Návrh XML formátu pro ukládání a načítání automatu

Dle definic uvedených v předchozí kapitole je automat tvořen množinou stavů, vstupní a zásobníkovou abecedou, seznamem přechodových pravidel, a dalšími. Je logické, že navrhované XML by mělo odrážet toto složení. Celková struktura XML by také neměla obsahovat příliš mnoho zanoření, různých druhů elementů nebo jejich parametrů. Jedna z požadovaných vlastností je snadnost vytvoření, úprav a přehlednost ukládaných automatů. Uživatel orientující se v problematice automatů a částečně značkovacích jazyků by měl mít možnost vytvářet konfigurace automatů ručně např. textovým editorem nebo pomocí jednoduchého skriptu.

Základním předpokladem je držet se ustavených názvů. Z důvodů čitelnosti a srozumitelnosti také pro česky nemluvící uživatele budou zvoleny termíny běžně užívaného anglického názvosloví. Kořenovým elementem tedy bude samotný automat – automaton. Program v současné době počítá s možností uložení pouze jednoho automatu do jednoho souboru. Ačkoliv by mohlo být úsporou času a místa umístit definice podobných automatů do jednoho XML souboru (s výslovně uvedenými neshodnými částmi), nejedná se o nezbytnou funkčnost a struktura takového souboru může být spíše matoucí pro koncového uživatele.

3.2.2 Návrh pro nastavení typu automatu a možnosti ukončení

Jak již bylo řečeno, v programu bude možno vytvářet dva druhy automatů, kdy jejich definice se mírně liší. Přesněji se tento rozdíl týká formátu pravidel. Z toho plyne, že kontrola pravidel bude probíhat u každého z nich trochu jiným způsobem. Proto bude potřeba při spuštění simulace zjistit, o jaký automat se jedná a v tomto důsledku potom pokračovat v jejich zpracování. Uživateli bude z tohoto důvodu poskytnuta možnost, nejlépe v menu, zvolit si typ automatu, který se bude chtít simulovat. Nastavení uživatelem shledávám jako nejlepší možnost pro jasné zadání automatu. Bylo by možné nechat podle formátu pravidel zjistit si, o jaký typ automatu by s největší pravděpodobností mělo jít. Avšak toto neshledávám příliš šťastnou volbou, snadno by pak mohlo docházet k simulaci jednoho typu místo druhého.

Stejný způsob řešení bude pravděpodobně aplikován také pro volbu ukončení procházení stavů. Neboť tato volba je opravdu jen a jen přímo na zadavateli.

3.2.3 Řešení determinismu x nedeterminismu

Pokud je automat zadán deterministicky, nemělo by při vykonávání pravidel a přechodů docházet k žádným cyklům ani opakování již nalezených stavů. Implementace by se tedy měla obejít bez větších komplikací. Jelikož z jednoho stavu existuje pouze jedno jediné pravidlo, kterým přejdeme do

stavu následujícího. Dojde-li k situaci, že zde takové pravidlo není, je jasné, že zadaný řetězec není automatem přijímán.

Avšak při nedeterministicky zadaných automatech může dojít k cyklení, případně rozvíjení větví, jejichž procházení již jednou bylo započato. Neboť pro přechod z jednoho stavu může existovat více pravidel. Tomuto problému se bude muset věnovat delší úvaha při řešení.

Pokud možno, bylo by nejlepší implementovat jedno stejné řešení pro oba typy automatů. Tedy dalo by se předpokládat, že automat je vždy zadán nedeterministicky. Díky tomuto postupu bude postup řešení a vyhledaný výsledek nalezeno v obou případech správným způsobem.

3.3 Metoda prohledávání stavového prostoru

Zásobníkové automaty umožňují definovat několik různých přechodových pravidel pro každý stav. Je uvažován například následující zásobníkový automat:

$M = (Q, \Sigma, \Gamma, R, s, S, F)$

$Q = \{s, q, f\}$

$\Sigma = \{a, b\}$

$\Gamma = \{S, a, b\}$

$R = \{Ssa \rightarrow Sas, asa \rightarrow aas, asb \rightarrow q, aqb \rightarrow q, Sq \rightarrow f\}$

$F = \{f\}$

Tento deterministický automat je tedy uveden zadáním vstupního řetězce *aaaabbbb* do počáteční konfigurace *Ssaaaabbbb (počáteční symbol na zásobníku, počáteční stav a vstupní řetězec)*. Je tedy zřejmé, že pro přechod může být použito pouze jediné pravidlo - $Ssa \rightarrow Sas$. V dalších krocích lze pracovat s použitím pravidla $asa \rightarrow aas$, a to do té doby, dokud je na vstupu symbol *a*. V případě, že se na vstupu objeví symbol *b*, užitým pravidlem pro přechod se stane - $aqb \rightarrow q$. Kdy s každým symbolem *b* je ze zásobníku odebrán symbol *a*, aktuálním stavem se stává *q*. Teprve pokud je na zásobníku opět pouze počáteční symbol a na vstupu se již nenachází žádné další znaky, přejdeme do koncového stavu *f*, který značí úspěšné přijetí řetězce. Na pohled je jasné, které pravidlo je nutné použít, aby bylo možné dále generovat zadaný řetězec a průchod neskončil před dosažením požadovaného stavu. Algoritmus simulace automatu v tomto případě není složitý – stačí pouze zjistit, zda pro aktuální stav existuje pravidlo s odpovídající levou stranou, a generovat následující stav s využitím pravé strany pravidla.

Pokud ale bude zadán nedeterministický automat takový, že v jednom okamžiku bude existovat více pravidel, kdy z daného stavu bude možno přejít do více různých stavů – a to se

stejným vstupním řetězcem, situace se komplikuje. Pro ukázku je dán takovýto rozšířený zásobníkový automat:

$M = (Q, \Sigma, \Gamma, R, s, S, F)$

$Q = \{s, f\}$

$\Sigma = \{a, b\}$

$\Gamma = \{S, a, b\}$

$R = \{sa \rightarrow as, sb \rightarrow bs, s \rightarrow Cs, aCsa \rightarrow Cs, bCsb \rightarrow Cs, SCs \rightarrow f\}$

$F = \{f\}$

Automat bude nedeterministicky zpracovávat řetězec *abba*, pomocí pravidel $sa \rightarrow as$, $sb \rightarrow bs$, bude číst symboly ze vstupu a ukládat je na zásobník. Rozšířený zásobníkový automat nemusí číst žádný symbol ze zásobníku, což umožňuje pravidlo $s \rightarrow Cs$. Pro správné zpracování řetězce automat použije pravidlo $s \rightarrow Cs$ teprve po přečtení jeho poloviny, kdy vloží na zásobník symbol *C*.

Následně k ověření, zda první polovina řetězce je shodná jako druhá reverzovaná, stačí porovnávat čtené symboly ze vstupu se zásobníkem a odstraňovat je. Protože na vrcholu zásobníku je uložen symbol *C,* může být použit k porovnání pravidla $aCsa \rightarrow Cs$, $bCsb \rightarrow Cs$, v závěru je třeba zkontrolovat, zda zásobník pod symbolem *C* obsahuje již pouze startovací symbol zásobníku, je-li tomu tak, využitím pravidla $SCs \rightarrow f$ lze přejít do koncového stavu.

Zatímco člověk může prostou úvahou nebo intuicí odhalit správný postup řešení, stačí jen pohled na zadaný řetězec a je jasné, kde se nachází jeho přesná polovina, zásobníkový automat sám o sobě nic takového nedokáže. V této chvíli nastupují metody umělé inteligence. Nahrazují inteligenci lidskou a umožňují programu najít jednu nebo více cest k cíli – v tomto případě zjistit, zda lze nalézt cestu do koncového stavu, přečíst řetězec na vstupu, ověřit zda automat přijímá řetězec jazyka. Metody umělé inteligence můžeme rozdělit na informované a neinformované. Další kritéria hodnocení metod jsou časová a prostorová složitost, případně zda je metoda optimální nebo úplná (tedy vždy nalezne řešení).

Zvolenou je metoda BFS – *breadth first search*. Procházením do šířky zajistíme úplnost metody – existuje-li v daném prostoru alespoň jedno řešení (o omezeních vyplývajících z velikosti zásobníku bude pojednáno dále). Kdy jak bylo uvedeno při popisu této metody v předchozí kapitole, budu snadno ukládat do dvou front jednotlivé uzly. Bude tedy procházet všechny možné varianty vycházející z aktuálního stavu a pokud ještě na daný uzel program při prohledávání nenarazil, zařadí ho na konec fronty, kterou prochází. Kdy každý uzel bude obsahovat jak aktuální, tak předchozí stav. Tato varianta poskytne snadné vyhledání případného výsledku.

4 Praxe, použitelnost, testy

4.1 Funkčnost demonstračního programu

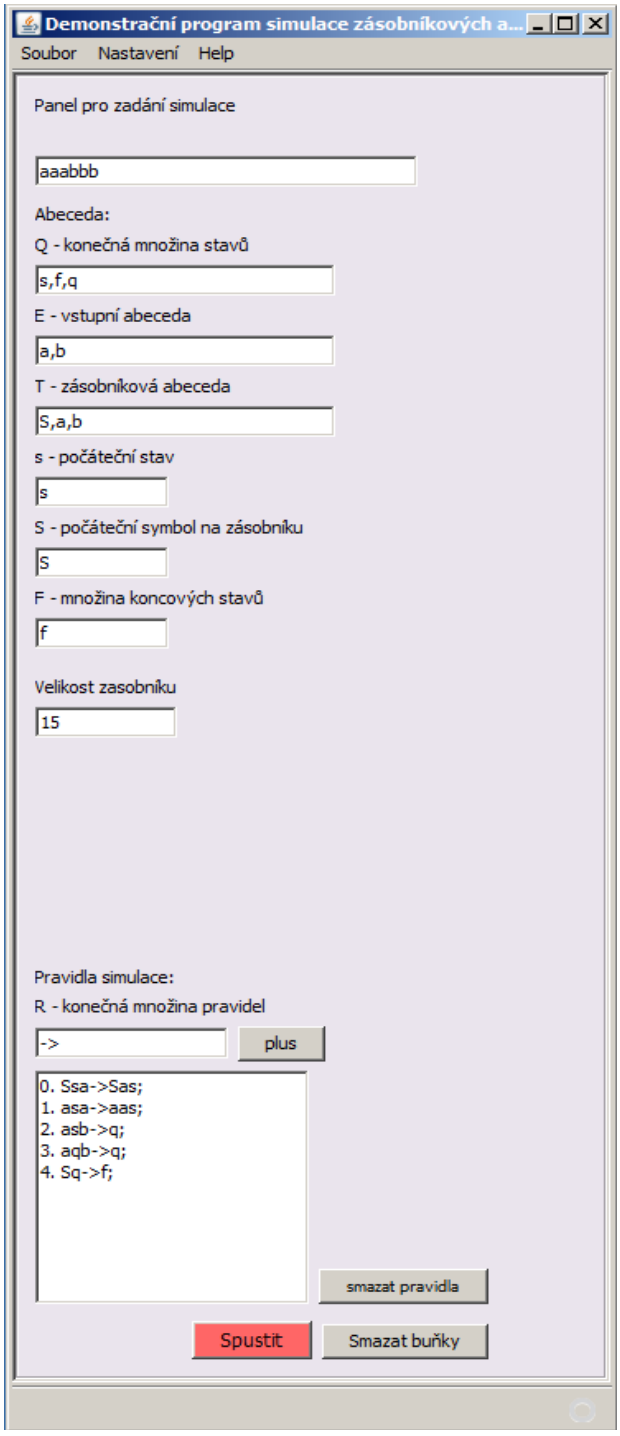
Program ke své funkčnosti vyžaduje nainstalovaný balík Java, a to nejméně ve verzi 1.5.

Po spuštění aplikace může uživatel zadat automat, nebo načíst dříve uložený z XML souboru. Po vložení automatu a spuštění simulace se otevře druhé okno, kde uživatel vidí výsledek simulace, může simulaci posouvat dopředu a dozadu, pozastavit, a vidí názorně grafický průběh práce automatu.

Hlavní okno programu je tvořeno formulářem třídy BPView. V okně jsou textová pole v panelu, který slouží pro zadání celé sedmice M=(Q,Σ,Γ,R,s,S,F), vstupního řetězce a také velikosti zásobníku. Dále jsou zde tlačítka pro úpravu a mazání pravidel, spuštění nebo zastavení simulace.

Formát vstupních údajů vychází ze stylu používaného ve cvičeních předmětu IFJ. Jednotlivé množiny (množina stavů, vstupní abeceda, zásobníková abeceda, a další) jsou zadány po znacích, oddělených čárkou. Pravidla jsou očíslována a rozdělena šipkou na levou a pravou stranu. Obě strany jsou zadány jako celé řetězce (tedy bez čárek), stejně jako ve cvičeních a opoře IFJ.

V nabídce možností se nachází několik nastavení, především přepínače mezi ukončovacími módy automatu a také



Obrázek 1: Hlavní okno programu

například typ automatu – jedná-li se o klasický zásobníkový, nebo o rozšířený automat. Nápověda k programu obsahuje stručný popis účelu programu, vysvětlení různých nabídek, vstupních polí a tlačítek.

4.1.1 Grafický výstup simulace

V jazyce Java existují metody ve třídách Graphics a Graphics2D pro kreslení grafických objektů, nicméně Java (resp. Netbean) neobsahuje žádnou komponentu pro přímý grafický výstup. Z toho důvodu je pro vykreslení jakéhokoliv tvaru nebo objektu nejdříve nutné vytvořit vlastní prostředky. Jedním ze známých a často používaných způsobů bývá vytvoření nové třídy, rozšiřující již existující třídu a objekt Jpanel. Níže popsaným způsobem vytvořím novou třídu Java2DPanel, odvozenou ze třídy Jpanel. Tím získám komponentu pro výstup, u které následně implementuji jednotlivé metody ze třídy Graphics, respektive Graphics2D. Abych mohla použít grafické objekty a metody, musím každý z nich „obalit“ do funkce v právě vytvořené třídě Java2DPanel.

V konstruktoru třídy je nejdříve zavolán konstruktor z původní třídy Jpanel – tím vytvořím kreslící plátno. V třídě simulace poté voláním veřejných metod ze třídy Java2DPanel provádím vykreslení jednotlivých částí, které byly potřeba pro vykreslení výstupu.

4.2 Automat a simulace

Definice automatu je uložena v instanci třídy classAutomat. Tuto třídu tvoří několik seznamů obsahujících záznamy typu řetězec. Instanci třídy automat jsou předávány hodnoty ze vstupních polí hlavního okna. K tomu obsahuje třída classAutomat příslušné gettery a settery – metody, pomocí kterých lze nastavovat hodnotu a zjišťovat obsah seznamů abecedy, stavů, pravidel. Při spuštění simulace je předána instance classAutomat konstruktoru třídy classSimulation.

4.2.1 Spuštění simulace

Před spuštěním simulace je nutné provést kontrolu automatu. Nejdříve se určí podle nastavení, zda se jedná o zásobníkový nebo rozšířený zásobníkový automat, dále se zjistí, jak je nastaven druh ukončení simulace (vyprázdnění zásobníku, vyprázdnění zásobníku společně s přechodem do koncového stavu, přechod do koncového stavu).

Po spuštění simulace je voláno procházení stavového prostoru metodou BFS. Tuto metodu jsem nastínila již dříve, zde se tedy budu zabývat pouze omezením a přizpůsobením této metody pro simulaci zásobníkových automatů. Omezení velikosti zásobníku především ovlivňuje délku a množství generovaných stavů. Jestliže dojde k přeplnění zásobníku během simulace, dojde

k ukončení této simulace bez nalezení výsledku, ačkoliv při dostatečné velikosti zásobníku by řešení mohlo být nalezeno. Proto je na výstup zobrazena informace o neúspěšném výsledku simulace.

Simulace je ukončena dvěma způsoby. Buď dojde ke splnění podmínky, kterou si uživatel zvolil před spuštěním jako ukončovací nebo uživatel stisknul tlačítko „Stop“ v simulačním okně. Přesnějším popisem ukončení se zabývá kapitola Testy.

4.2.2 Kontrola správnosti zadaného automatu

Funkce *updateAutomatValues* slouží ke kontrole vstupu - zda jsou nastaveny typ automatu a ukončení, velikost zásobníku, že je zadána abeceda, množina stavů, startovací stav a množina ukončovacích stavů. Ověřuje se správnost zadání pravidel – tedy jestli obsahují platné symboly a mají správnou syntaxi. Použití funkce je při spuštění simulace, při ukládání automatu a také ke kontrole při načtení ze souboru XML.

4.2.3 Vytváření XML souboru

Ukládání vytvořených automatů je prováděno do souborů ve formátu XML, ze kterých je poté umožněno také načítání. Soubor XML se skládá z hlavičky udávající verzi XML specifikace a znakovou sadu uložených dat, dále kořenového elementu a obsahuje jednotlivé elementy představující abecedu, stavový prostor společně s množinou pravidel. Například element *states* obsahuje seznam stavů automatu.

```
<?xml version="1.0" encoding="UTF-8"?>
<automaton>
<states>
<state>s</state>
<state>f</state>
</states>
<input_alphabet>
<input_symbol>a</input_symbol>
<input_symbol>b</input_symbol>
</input_alphabet>
<stack_alphabet>
<stack_symbol>S</stack_symbol>
<stack_symbol>a</stack_symbol>
<stack_symbol>b</stack_symbol>
</stack_alphabet>
<start>
<start_state>s</start_state>
</start>
<start_stack>
<start_stack_symbol>S</start_stack_symbol>
</start_stack>
<final_states>
<final_state>f</final_state>
</final_states>
<rules>
<single_rule>Ssa->Sas</single_rule>
<single_rule>Ssb->Sbs</single_rule>
<single_rule>asa->aas</single_rule>
<single_rule>bsb->bbs</single_rule>
<single_rule>bsa->s</single_rule>
<single_rule>asb->s</single_rule>
<single_rule>as->f</single_rule>
<single_rule>af->f</single_rule>
<single_rule>Sf->f</single_rule>
</rules>
<active_settings>
<set_ending_type>2</set_ending_type>
<set_automat_type>2</set_automat_type>
</active_settings>
</automaton>
```

Načítání automatu probíhá postupným čtením souboru XML s následným rekurzivním procházením jednotlivých elementů. Jakmile je načten příslušný element, jeho hodnota je uložena jako další prvek do příslušného seznamu (stavů, pravidel,...) podle toho, ve kterém elementu se nachází.

V automatu jsou uloženy stavy, abeceda, pravidla a také nastavení automatu. Toto nastavení se skládá pouze ze dvou hodnot udávajících, zda se jedná o rozšířený zásobníkový automat nebo zásobníkový automat a uživatelem zvolený způsob ukončení simulace.

Během činnosti programu je kompletní konfigurace automatu uložena v instanci třídy classAutomat. Příslušné metody (tzv. gettery a settery) umožňují zjistit, případně aktualizovat, jednotlivé seznamy – stavů, pravidel, abecedy či nastavení.

Třídě *createXML* je při volání konstruktoru předána instance třídy *classAutomat* – obsahuje abecedu, seznam stavů a pravidel automatu, dále nastavení programu – tedy způsob ukončení simulace.V metodě *createXML* je postupně čten seznam stavů, znaků abecedy, pravidel, ukládaný do odpovídajících elementů. Výsledkem je textový řetězec obsahující XML dokument s kořenovým elementem *automaton*, a childovskými elementy *states*, *rules*, atd. Příklad výsledného XML souboru je uveden na předchozí straně.

4.2.4 Čtení XML souboru

Třída *readXML* obsahuje několik funkcí pro zpracování XML souboru. Využívá SAX parser v Javě. Funkce *readXML* načte postupně rekurzivně vždy kořenový element, a poté čte postupně prvky vnořené. Jakmile narazí např. na element, který obsahuje stav, uloží jej do seznamu stavů. Takto načte všechny elementy v souboru XML, a poté převede funkcí *flushXMLtoAutomat* do textových polí v hlavním panelu programu.

4.2.5 Nápověda

V nabídce *Nápověda* jsou uvedeny základní informace ke spuštění simulace, nastavení simulace, ukládání a načítání, a také příklad automatu. Nápověda je vytvořena jako stránka HTML s textem nápovědy a odkazy do těla stránky. Takto vytvořená nápověda lze poté zobrazit v kterémkoliv prohlížeči nebo v programu podporujícím HTML. Java samotná umožňuje pomocí textového pole a využití třídy *java.net* zobrazovat HTML a v omezené míře také kaskádové styly. Html je zobrazeno pomocí metody *SetPage* kontejneru *JtextPane*, ve třídě *BPHelpBox*.

4.3 Použitá metoda BFS

Jak již bylo výše zmíněno, pro prohledávání stavového prostoru byla zvolena metoda z umělé inteligence – BFS(*breadth first search*), nebo-li metoda prohledávání do šířky.

Pro jednotlivé stavy, které se ukládají do fronty nebo následně do seznamu CLOSED, jsem zvolila formát (*nový stav, předchozí stav, číslo pravidla,které bylo použito pro přechod).* Kdy pro počáteční stav se pravidlo neukládá a jako předchůdce je uložen prázdný řetězec (nill). Viz následující ukázka obsahu fronty a seznamu CLOSED pro prvních 5 kroků rozšířeného nedeterministického zásobníkového automatu při vstupním řetězci „i+i*i“:

*fronta OPEN[Esi+i*i,nill]*

CLOSED[]

*fronta OPEN[T+Esi+i*i,Esi+i*i,3, Tsi+i*i,Esi+i*i,4]*

*CLOSED [Esi+i*i,nill]*

*fronta OPEN[Tsi+i*i,Esi+i*i,4, T+T+Esi+i*i,T+Esi+i*i,3, T+Tsi+i*i,T+Esi+i*i,4]*

*CLOSED[Esi+i*i,nill, T+Esi+i*i,Esi+i*i,3]*

*fronta OPEN[T+T+Esi+i*i,T+Esi+i*i,3,T+Tsi+i*i,T+Esi+i*i,4,i*Tsi+i*i,Tsi+i*i,5, isi+i*i,Tsi+i*i,6]*

*CLOSED [Esi+i*i,nill, T+Esi+i*i,Esi+i*i,3, Tsi+i*i,Esi+i*i,4]*

*frontaOPEN [T+Tsi+i*i,T+Esi+i*i,4,i*Tsi+i*i,Tsi+i*i,5,isi+i*i,Tsi+i*i,6, T+T+T+Esi+i*i,T+T+Esi+i*i,3, T+T+Tsi+i*i,T+T+Esi+i*i,4]*

*CLOSED [Esi+i*i,nill, T+Esi+i*i,Esi+i*i,3, Tsi+i*i,Esi+i*i,4, T+T+Esi+i*i,T+Esi+i*i,3]*

Dojde-li při průchodu stavovým prostorem program k uživatelem zvolené ukončovací podmínce, díky vytvářenému seznamu CLOSED lze snadno zpětně dohledat cesta od koncového stavu až po počáteční.

Výsledná cesta je vyhledávána následujícím způsobem. Začíná se u koncového stavu, kdy tento stav je uložen do seznamu společně s pravidlem a předchůdce tohoto stavu je vyhledán v seznamu CLOSED. Tento postup se opakuje do doby, dokud není jako předchůdce stavu nalezena hodnota „nill“, kterou obsahuje pouze počáteční stav.

Výpis na výstup je shodný s formátem výpisu v předmětu „Formální jazyky a překladače“, aby si studenti nemuseli navykat na jiné konvence než ty, které již mají z výuky zažité. Ukázka výsledku je zobrazena níže.

Ssaabbaa|-Sasabbaa[4]|-Saasbbaa[4]|-Saabsbaa[5]|-SaabCsbaa[3]|-SaaCsaa[1]|-SaCsa[2]|-SCs[2]|-f[6]

4.4 Testy

Během tvorby a také po dokončení programové části byla aplikace otestována na několika různých zadáních automatů. Jelikož existují tři typy zakončení simulace úspěchem, bylo nutné najít od každé varianty alespoň jedno zadání. Abych si mohla být jistá správným zadáním automatu, čerpala jsem příklady, jenž jsou přiloženy k programu, ze studijních opor předmětu „Formální jazyky a překladače“.

V průběhu se samozřejmě vyskytly problémy u různých zakončení, ale postupem času byly všechny nalezené nedostatky odstraněny a nyní je možnost simulovat bez problému, tak jako je požadováno v zadání.

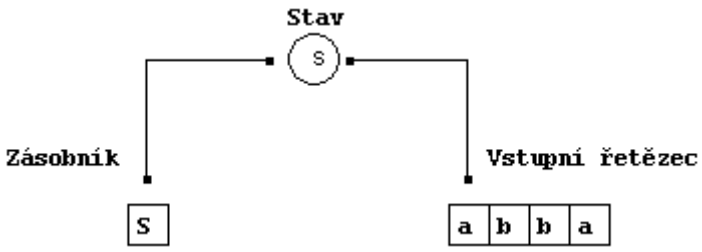
4.4.1.1 Velikost zásobníku menší než potřebná :

Simulace započne, avšak je ukončena dříve, než je možné najít správný výsledek, neboť velikost zásobníku není dostačující. Jelikož stavy, které dokázal projít, než se vyskytl problém s velikostí zásobníku, ještě nedošly stanovených podmínek pro vrácení úspěchu spolu s výsledkem, dojde k ukončení simulace a zobrazení oznámení na výstup programu.

4.4.1.2 Velikost zásobníku je dostačující nebo větší než potřebná a zadaný řetězec je přijímán daným automatem:

Simulace proběhne a na výstup je vrácena výsledná posloupnost, ve tvaru:
počáteční řetězec|-řetězec po provedení pravidla[použité pravidlo k tomuto přechodu]|-řetězec po provedení pravidla[použité pravidlo k tomuto přechodu]

Poté je možné si pomocí příslušného tlačítka daný výsledek zobrazit ve stejném okně, kdy koncepce zobrazení je následující.



4.4.1.3 Velikost zásobníku je dostatečná nebo větší, ale zadaný řetězec není přijímán daným automatem:

Výsledkem je vždy neúspěch. Avšak pro tento případ mohou existovat dvě různé varianty ukončení:

1.a Z daného stavu již není možné přejít do dalšího, jelikož neexistuje pravidlo, které by umožnilo další přechod. Potom je v okně pro výsledek vypsána informace, že simulace neproběhla úspěšně (k tomuto dochází převážně u deterministicky zadaných automatů).

1.b Pokud je automat zadán nedeterministicky, může dojít k neustálému rozvíjení jednotlivých stavů, a proto se může vyskytnout situace, kdy simulace nedochází konce. V takovém případě zbývá jen ukončit simulaci pomocí tlačítka „Stop“.

Pokud nejsou nastaveny všechny povinné údaje neboli, není-li sedmice automatu kompletní, nemůže být započata simulace, takže nemůže dojít k selhání a chybám kvůli neúplné specifikaci.

Prvotní oznámení, že nebyla zadána velikost zásobníku, byla doplněna o automatické vyplnění velikosti zásobníku podle velikosti vstupního řetězce. Přesto je uživatel opět varován, že na tuto položku pozapomněl a proto bude vyplněna automaticky, jelikož častokrát tato velikost není zdánlivě dostačující pro vykonání simulace až do konce.

5 Závěr

Při tvorbě programu jsem se seznámila s principem přesné funkčnosti jak zásobníkových tak rozšířených zásobníkových automatů z teorie formálních jazyků a použití gramatik. Odhalila jsem, jaké jsou rozdíly mezi těmito a také dalšími automaty, kdy ne všechny bylo nutné zpracovávat pro chod programu. Avšak nauka širšího spektra automatů byla vhodná pro plnohodnotné porozumění zpracovávání vstupních řetězců, které automat přijímá či nikoliv.

Využití tohoto programu by bylo možné zejména při výuce zásobníkových automatů jako demonstrační materiál k názornému zobrazení zadaných typů automatů. Protože automat je schopen nalezené řešení nejen vypsat, ale také poskytuje jednoduché vykreslení obsahu zásobníku, vstupního řetězce a aktuálního stavu krok po kroku.

Rychlost, přesnost a efektivita simulace i spolehlivé nalezení výsledku závisí na zadané velikosti zásobníku. Uživatel může ručně zadat velikost, nebo ji nechat automaticky doplnit podle velikosti vstupního řetězce (je-li spuštěna simulace bez předchozího vyplnění požadované velikosti). Při ručním zadání s kombinací špatně zvoleného vstupního řetězce může dojít k několika hodinové simulaci, kdy i po tak dlouhé době není nalezen výsledek. Na tento problém jsem nenalezla žádné řešení. Jedinou možností by byly speciální algoritmy, které by ještě před započetím simulace zjistily, zda je možné výsledek objevit a také by mohly odhadnout, jak velký zásobník by pro toto zpracování byl potřeba. Řešení pomocí algoritmů je však složité a nespadá již do této bakalářské práce.

Omezení velikostí zásobníku je jen pro usnadnění práce, program by byl samozřejmě funkční i bez tohoto opatření. Avšak při zadání špatného řetězce je díky velikosti zásobníku větší šance na ukončení simulace programem.

Výsledná práce by mohla projít ještě další řadou úprav a mohly by přibýt další funkce. Například vylepšení grafického zobrazení a další. Přesto však program splňuje stanovené zadání. Simuluje rozšířené i zásobníkové automaty, deterministické i nedeterministické, graficky zobrazuje průběh a výsledek řešení, kdy grafický výstup by se dal předpokládat za vlastní vylepšení ještě v průběhu vzniku této práce. Poskytuje uživateli lepší představu o celkové funkčnosti. Činnost programu lze během simulace zastavit, a zadání automatů je možné ukládat i načítat ve formátu xml.

Literatura

[1] Lukáš, R.: *Multigenerativní gramatické systémy*. Disertační práce FIT VUT v Brně, 2006
[2] Meduna, A.: *Automata and Languages, Theory and Applications,* 1999
[3] Meduna, A., Lukáš, R.: *Formální jazyky a překladače – studijní opora*, FIT VUT v Brně, 2006
[3] Meduna, A., Lukáš, R.: *Formální jazyky a překladače – materiály k přednášce*, FIT VUT v Brně, 2007
[4] Zbořil, F., Zbořil, F.: *Základy umělé inteligence – studijní opora*, FIT VUT v Brně, 2007

Seznam příloh

Příloha: CD/DVD se zdrojovými texty a programem